# Sequential and Parallel Algorithms for the Generalized Maximum Subarray Problem

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy in Computer Science
in the
University of Canterbury
by
Sung Eun Bae

| | |
|---|---|
| Prof. Peter Eades, University of Sydney | Examiner |
| Prof. Kun-Mao Chao, National Taiwan University | Examiner |
| Prof. Tadao Takaoka, University of Canterbury | Supervisor |
| Dr. R. Mukundan, University of Canterbury | Co-Supervisor |

University of Canterbury
2007

This thesis is dedicated to
Dr. Quae Chae (1944–1997), my father-in-law
Dr. Jae-Chul Joh (1967–2004), my departed cousin

# Abstract

The maximum subarray problem (MSP) involves selection of a segment of consecutive array elements that has the largest possible sum over all other segments in a given array. The efficient algorithms for the MSP and related problems are expected to contribute to various applications in genomic sequence analysis, data mining or in computer vision etc.

The MSP is a conceptually simple problem, and several linear time optimal algorithms for 1D version of the problem are already known. For 2D version, the currently known upper bounds are cubic or near-cubic time.

For the wider applications, it would be interesting if multiple maximum subarrays are computed instead of just one, which motivates the work in the first half of the thesis. The generalized problem of $K$-maximum subarray involves finding $K$ segments of the largest sum in sorted order. Two subcategories of the problem can be defined, which are $K$-overlapping maximum subarray problem ($K$-OMSP), and $K$-disjoint maximum subarray problem ($K$-DMSP). Studies on the $K$-OMSP have not been undertaken previously, hence the thesis explores various techniques to speed up the computation, and several new algorithms. The first algorithm for the 1D problem is of $O(Kn)$ time, and increasingly efficient algorithms of $O(K^2 + n \log K)$ time, $O((n+K) \log K)$ time and $O(n+K \log \min(K,n))$ time are presented. Considerations on extending these results to higher dimensions are made, which contributes to establishing $O(n^3)$ time for 2D version of the problem where $K$ is bounded by a certain range.

Ruzzo and Tompa studied the problem of *all maximal scoring subsequences*, whose definition is almost identical to that of the $K$-DMSP with a few subtle differences. Despite slight differences, their linear time algorithm is readily capable of computing the 1D $K$-DMSP, but it is not easily extended to higher dimensions. This observation motivates a new algorithm based on the tournament data structure, which is of $O(n + K \log \min(K,n))$ worst-case time. The extended version of the new algorithm is capable of processing a 2D problem in $O(n^3 + \min(K,n) \cdot n^2 \log \min(K,n))$ time, that is $O(n^3)$ for $K \leq \frac{n}{\log n}$.

For the 2D MSP, the cubic time sequential computation is still expensive for practical purposes considering potential applications in computer vision and data mining. The second half of the thesis investigates a speed-up option through parallel computation. Previous parallel algorithms for the 2D MSP have huge demand for hardware resources, or their target parallel computation models are in the realm of pure theoretics. A nice compromise between speed and cost can be realized through utilizing a mesh topology. Two mesh algorithms for the 2D MSP with $O(n)$ running time that require a network of size $O(n^2)$ are designed and analyzed, and various techniques are considered

to maximize the practicality to their full potential.

# Table of Contents

# List of Algorithms

# Acknowledgments

When I came to New Zealand after spending a year at Yonsei university, Korea, I was just a lazy engineering student who almost failed the first year programming course.

Initially I planned to get some English training for a year and go back to Korea to finish my degree there. Many things happened since then. Years later, I found myself writing a Ph.D thesis at Canterbury, ironically, in computer science.

I believe it was the "inspiration" that made me stay at Canterbury until now. In a COSC329 lecture, Prof. Tadao Takaoka once said, half-jokingly, "If you make Mesh-Floyd (all pairs shortest paths algorithm) run in $3n$ steps, I'll give you a master degree". With little background knowledge, I boldly tried numerous attempts to achieve this goal. Certainly, it was not as easy as it looked and I had to give up in the end. However, when I started postgraduate studies with Prof. Takaoka, I realized the technique I learned from that experience could be applied to another problem, the maximum subarray problem, which eventually has become my research topic.

My English is simply not good enough to find a right word to express my deepest gratitude to Prof. Takaoka for guiding me to the enjoyment of computer science. In the months of research, programming, and writing that went into this thesis, his encouragement and guidance have been more important than he is likely to realize. My co-supervisor, R. Mukundan, also deserves special thanks. His expertise in graphics always motivated me to consider visually appealing presentations of "dry" theories. I also thank Prof. Krzysztof Pawlikowski for his occasional encouragements.

I can not neglect to mention all the staffs at Canterprise for helping patenting some of research outcomes and Mr. Steve Weddell for his expertise and support to develop a hardware prototype. Mr. Phil Holland, has been a good friend as well as an excellent technician who took care of many incidents. I will miss a cup of green tea and cigarette with him, which always saved me from frustration.

My wife, Eun Jin, has always been virtually involved in this project. Without her support in all aspects of life, this project could have never been completed.

While I was studying full-time, my mother-in-law has always shown her strong belief in me. I thank her for taking me as her beloved daughter's husband. I also hope my departed father-in-law is pleased to see my attempt to emulate him.

I acknowledge my brother and parents. Sung-ha has always been my best friend. My parents brought me up and instilled in me the inquisitive nature and sound appetite for learning something new. I cannot imagine how many things they had to sacrifice to immigrate to a new country. In the end, it was their big, agonizing decision that enabled me to come this far. Without what they had given to their son, this thesis could have not come to existence.

Finally, I would like to express thanks to Prof. Peter Eades and Prof. Kun-Mao Chao for their constructive comments. Prof. Jingsen Chen and Fredrik Bengtsson also deserve special thanks for their on-going contribution to this line of research.

# Chapter 1

# Introduction

## 1.1 History of Maximum Subarray Problem

In 1977, Ulf Grenander at Brown University encountered a problem in pattern recognition where he needed to find the maximum sum over all rectangular regions of a given $m \times n$ array of real numbers [45] *. This rectangular region of the maximum sum, or *maximum subarray*, was to be used as the maximum likelihood estimator of a certain kind of pattern in a digitized picture.

**Example 1.1.1.** When we are given a two-dimensional array $a[1..m][1..n]$, suppose the upper-left corner has coordinates (1,1). The maximum subarray in the following example is the array portion $a[3..4][5..6]$ surrounded by inner brackets, whose sum is 15.

$$a = \left[\begin{array}{rrrr} -1 & 2 & -3 & 5 \\ 2 & -4 & -6 & -8 \\ 3 & -2 & 9 & -9 \\ 1 & -3 & 5 & -7 \end{array}\;\begin{array}{c} \begin{array}{rr} -4 & -8 \\ 2 & -5 \end{array} \\ \left[\begin{array}{rr} -1 & 10 \\ 8 & -2 \end{array}\right] \end{array}\;\begin{array}{rr} 3 & -3 \\ 4 & 1 \\ -5 & 2 \\ 2 & -6 \end{array}\right]$$

To solve this *maximum subarray problem* (MSP), Grenander devised an $O(n^6)$ time algorithm for an array of size $n \times n$, and found his algorithm was prohibitively slow. He simplified this problem to one-dimension (1D) to gain insight into the structure.

* This is sometimes asked in a job interview at GOOGLE. Retrieved 19 December, 2006 from `http://alien.dowling.edu/~rohit/wiki/index.php/Google_Interview_Questions`

The input is an array of $n$ real numbers; the output is the maximum sum found in any *contiguous* subarray of the input. For instance, if the input array $a$ is $\{31, -41, 59, 26, -53, 58, 97, -93, -23, 84\}$, where the first element has index 1, the program returns the sum of $a[3..7]$,187.

He obtained $O(n^3)$ time for the one-dimensional version and consulted with Michael Shamos. Shamos and Jon Bentley improved the complexity to $O(n^2)$ and a week later, devised an $O(n \log n)$ time algorithm. Two weeks after this result, Shamos described the problem and its history at a seminar attended by Jay Kadane, who immediately gave a linear time algorithm [18]. Bentley also challenged audience with the problem at another seminar, and Gries responded with a similar linear time algorithm [46].

While Grenander eventually abandoned the approach to the pattern matching problem based on the maximum subarray, due to the computational expense of all known algorithms, the two-dimensional (2D) version of this problem was found to be solved in $O(n^3)$ time by extending Kadane's algorithm [19]. Smith also presented $O(n)$ time for the one-dimension and $O(n^3)$ time for the two-dimension based on divide-and-conquer [84].

Many attempts have been made to speed up thereafter. Most earlier efforts have been laid upon development of parallel algorithms. Wen [103] presented a parallel algorithm for the one-dimensional version running in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM (Exclusive Read, Exclusive Write Parallel Random Access Machine) and a similar result is given by Perumalla and Deo [77]. Qiu and Akl used interconnection networks of size $p$, and achieved achieved $O(n/p + \log p)$ time for the one-dimension and $O(\log n)$ time with $O(n^3/\log n)$ processors for the two-dimension [79].

In the realm of sequential algorithms, $O(n)$ time for the 1D version is proved to be optimal. $O(n^3)$ time for the 2D MSP has remained to be the best-known upper bound until Tamaki and Tokuyama devised an algorithm achieving sub-cubic time of $O\left(n^3 \left(\log\log n/\log n\right)^{1/2}\right)$ [94]. They adopted divide-and-conquer technique, and applied the fastest known distance matrix multiplication (DMM) algorithm by Takaoka [87]. Takaoka later simplified the algorithm [89] and recently presented even faster DMM algorithms that are readily applicable to the MSP [90].

## 1.2 Applications

### 1.2.1 Genomic Sequence Analysis

With the rapid expansion of genomic data, sequence analysis has been an important part of bio-informatics research. An important line of research in sequence analysis is to locate biologically meaningful segments.

One example is a prediction of the membrane topology of a protein. Proteins are huge molecules made up of large numbers of amino acids, picked out from a selection of 20 different kinds. In Appendix A, the list of 20 amino acids and an example of a protein sequence is given.

Virtually all proteins of all species on earth can be viewed as a long chain of amino acid residues, or a sequence over a 20-letter alphabet. In most cases, the actual three-dimensional location of every atom of a protein molecule can be accurately determined by this linear sequence, which in turn, determines the protein's biological function within a living organism. In broad terms, it is safe to state that all the information about the protein's biological function is contained in the linear sequence [64].

Given that, a computer-based sequence analysis is becoming increasingly popular. Especially, computerized algorithms designed for structure prediction of a protein sequence can be an invaluable tool for biologists to quickly understand the protein's function, which is the first step towards the development of antibodies, drugs.

Identification of *transmembrane domains* in the protein sequence is one of the important tasks to understand the structure of a protein or the membrane topology. Each amino acid residue has different degree of water fearing, or *hydrophobicity*. Kyte and Doolittle [70] experimentally determined hydrophobicity index of the 20 amino acids ranging from -4.5(least hydrophobic) to +4.5(most hydrophobic), which is also given in Appendix A. A hydrophobic domain of the protein resides in the oily core of the membrane, which is the plane of the surrounding lipid bi-layer. On the other hand, *hydrophilic* (least hydrophobic, or water-loving) domains protrude into the watery environment inside and outside the cell. This is known as *hydrophobic effect*, one of the bonding forces inside the molecule.

Transmembrane domains typically contain hydrophobic residues. When the hydropathy index by Kyte and Doolittle is assigned to each residue in the protein sequence, the transmembrane domains would appear as segments with high total scores.

Karlin and Brendel [61] used Kyte-Doolittle hydropathy index to predict the transmembrane domains in human $\beta_2$-adrenergic receptor and observed that high-scoring segments correspond to the known transmembrane domains.

If we are interested in the segment with the highest total score, the problem is equivalent to the MSP. Particularly, the approach based on the MSP is ideal in a sense that it does not assume a predefined size of the solution. A conventional algorithm suggested by [70], on the contrary, involved a predefined segment size and often required manual adjustment to the *window size* until a satisfactory result would be obtained.

This score-based technique has been applied to other similar problems in genomic sequence analysis, which include identification of

(A). conserved segments [21, 86, 48],

(B). GC-rich regions [29, 47]

(C). tandem repeats [100]

(D). low complexity filter [2]

(E). DNA binding domains [61]

(F). regions of high charge [62, 61, 22]

While the basic concept of finding a highest scoring subsequence is still effective, slightly modified versions of the MSP are better suited to some applications above.

For example, a single protein sequence usually contains multiple transmembrane domains, thus the MSP that finds the single highest scoring subsequence may not be very useful. As each transmembrane domain corresponds to one of high scoring segments that are disjoint from one another, an extended version of the MSP that finds *multiple-disjoint maximum subarrays* is more suited to this application. The statistical significance of multiple disjoint high scoring segments was highlighted in [3] and Ruzzo and Tompa [82] developed a linear time algorithm that finds all high scoring segments.

A statistical analysis showed that the minimal and maximal length for all-$\alpha$ and all-$\beta$ membrane proteins lies in the interval 15-36 and 6-23 residues [58]. Thus, depending on a specific task, better prediction can be obtained when the *length constraint*s are imposed on the MSP, such that the highest scoring subsequence should be of length at least $L$ and at most $U$. For the problem of maximum subarray with the length constraints, efficient algorithms with $O(n)$ time were developed by Lin et al.[74], Fan et al. [32] and Chen and Chao [66]. Fan et al. applied their algorithm to rice chromosome 1 and identified a previously unknown very long repeat structure. Fariselli et al. [33] also studied the same problem to locate transmembrane subsequences. Their algorithm is combined with conventional methods, such as HMM (Hidden Markov Model) [65], neural network and TMHMM (Transmembrane HMM) method, and obtained the accuracy of 73%, which is almost twice more accurate than conventional methods alone.

In the process of conserved segment detection, the assigned scores may be all positive. If the maximum subarray, the subsequence of highest total score, is computed, the entire array will be reported erroneously as the solution, a conserved region. One way to resolve this issue is to adjust the score of each residue by subtracting a positive anchor value. If an appropriate anchor value is not easy to determine, a slightly different problem may be considered, such that the maximum *average* subarray will be computed instead. It should be noted that, the maximum average subarray needs to have a length constraint. When there is no constraint, often a single element, the maximum element, will be reported as the maximum average, which is of little interest. Huang gave the first non-trivial $O(nL)$ time algorithm for this maximum average (or density) problem, where $L$ is the minimum length constraint [53]. Improved algorithms with $O(n \log L)$ time and $O(n)$ time were reported thereafter by Lin et al. [74] and Goldwasser et al. [43]. The latter result, in particular, is optimal and considers both the minimum length $L$ and the maximum length $U$ constraints.

### 1.2.2 Computer Vision

Ulf Grenander's idea in late 70's, even if it was not quite materialized, proposes some interesting applications in computer vision.

A bitmap image is basically a two-dimensional array where each cell, or pixel, represents the color value based on RGB standard. If the brightest portion inside the image is to be found, for example, we first build a two-dimensional array where each cell of this array represents the brightness score of each pixel in the corresponding location of the bitmap image.

Brightness is formally defined to be (R+G+B)/3; however this does not correspond well to human color perception. For the brightness that corresponds best to human perception, *luminance* is used for most graphical applications. Popular method for obtaining the luminance ($Y$) adopted by all three television standards (NTSC, PAL, SECAM) is based on the weighted sum of R,G,B values, such as,

$$Y = 0.30R + 0.59G + 0.11B.$$

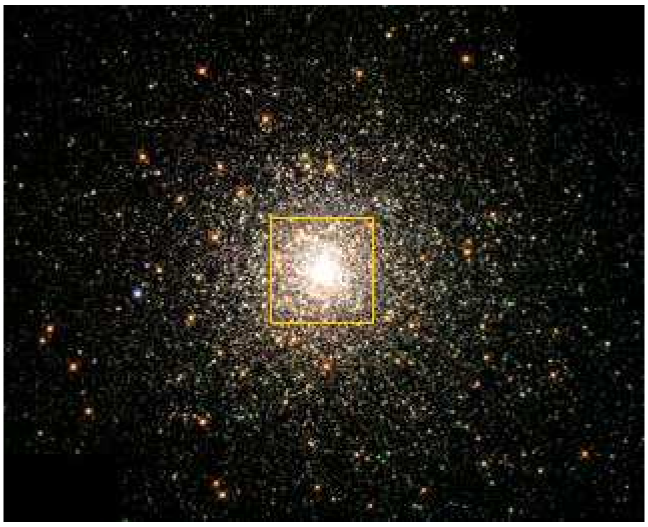
Here, weightings 0.30, 0.59, and 0.11 are chosen to most closely match the sensitivity of the eye to red, green, and blue [55].

Certainly, all the values assigned to each cell are non-negative and the maximum subarray is then the whole array, which is of no interest. Before computing the maximum subarray, it is therefore essential to normalize each cell by subtracting a positive anchor value, Usually, the overall mean or median pixel value may be a good anchor value. We may then compute the maximum subarray, which corresponds to the brightest area in the image.

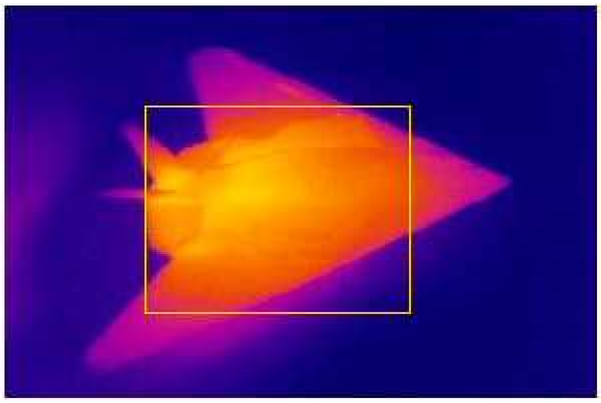
This approach may be applicable to other various tasks with a different scoring scheme. One example may be locating the warmest part in the thermo-graphic image. In the thermo-graphic image, the brightest (warmest) parts of the image are usually colored white, intermediate temperatures reds and yellows, and the dimmest (coolest) parts blue. We can assign a score to each pixel following this color scale.

Two examples shown in Figure 1.1 were obtained by running an algorithm for the 2D MSP on the bitmap images.

Even if the upper bound for the 2D MSP has been reduced to sub-cubic



(a) Brightest region in an astronomy image



(b) Warmest region in an infrared image

Figure 1.1: Contents retrieval from graphics

[94, 89], it is still close to cubic time, and the computational efficiency still remains to be the major challenge for the MSP-based graphical applications. Indeed, performing pixel-wise operation by a near-cubic time algorithm can be time consuming. It suggests, however, that the framework based on the MSP will be still useful in conjunction with recent developments in the computer vision or at least, will provide the area of computer vision with some general insights. For example, a recent technique for real-time object detection by Viola and Jones [99] adopted a new image representation called an *integral image* and incorporated machine learning techniques to speed up the

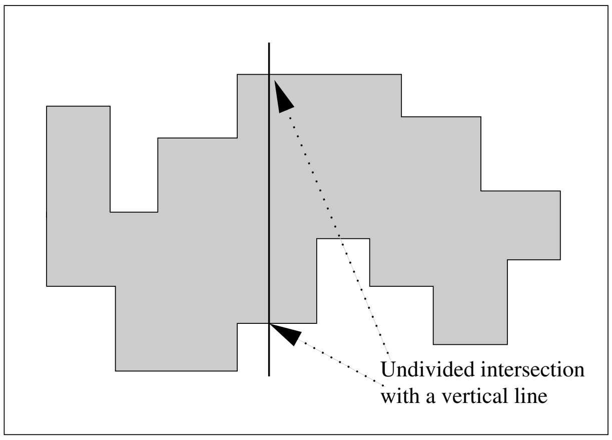


Figure 1.2: A connected $x$-monotone region

computation. Here, the integral image is basically a *prefix sum*, the central algorithmic concept extensively used in computing the MSP. The definition of the prefix sum and its application to the MSP will be fully addressed in Chapter 2.

Recently, Weddell and Langford [101] proposed an efficient centroid estimation technique suitable for Shack-Hartmann wavefront sensors based on the FPGA implementation of the mesh algorithm for the 2D MSP. The design of the mesh algorithm they used is a part of the research for this thesis that will be presented in Chapters 8 and 9.

In some occasions, detection of a non-rectangular region may be more desirable than a rectangular one. Fukuda et al.[41] generalized an algorithm for the segmentation problem by Asano et al.[5] and showed that the maximum sum contained in a connected *$x$-monotone* region can be computed in $O(n^2)$ time. A region is called $x$-monotone if its intersection with any vertical line is undivided, as shown in Figure 1.2. Yoda et al.[104] showed that the maximum sum contained in a *rectilinear convex* region can be computed in $O(n^3)$ time, where a rectilinear convex region is both $x$-monotone and $y$-monotone. However, if we want the maximum sum in a random-shaped connected region, the problem is NP-complete [41]. Both [41] and [104] concern the data mining applications, but their results are readily applicable to the computer vision.

### 1.2.3 Data mining

The MSP has an instant application in data mining when numerical attributes are involved. General description on the MSP in the context of data mining techniques is given by Takaoka [93], which we summarize below.

Suppose we wish to identify some rule that can say if a customer buys item $X$, he/she is likely to buy item $Y$. The likelihood, denoted by $X \rightarrow Y$ and called an *association rule*, is measured by the formula of confidence, such that,

$$conf(X \rightarrow Y) = \frac{support(X, Y)}{support(X)}$$

$X$ is called the *antecedent* and $Y$, the *consequence*. Here, $support(X, Y)$ is the number of transactions that include both $X$ and $Y$, and $support(X)$ is that for $X$ alone.

Let us consider a supermarket example shown in Appendix B. We have two tables, where the first shows each customer's purchase transactions and the second maintains customers' personal data. In the example, all of the three customers who bought "cheese" also bought "ham" and thus $conf(ham \rightarrow cheese) = 3/3 = 1$.

The same principle can be applied if we wish to discover rules with numerical attributes, such as *age* or *income* etc. This problem was originated by Srikant and Agrawal [85].

Suppose we wish to see if a particular age group tends to buy a certain product, for example, "ham". We have two numerical attributes of customers, their *age* and their purchase amount for "ham".

In the example, let us use the condition $age < 40$ for the antecedent. Then $conf(age < 40 \rightarrow ham) = 1$. If we set the range to $age \leq 50$ however, the confidence becomes $3/5$. Thus the most cost-efficient advertising outcome would be obtained if the advertisement for "ham" is targeted at customers younger than 40.

This problem can be formalized by the MSP. Suppose we have a 1D array $a$ of size 6, such that $a[i]$ is the number of customers whose age is $10i \leq age < 10(i+1)$, and bought "ham". Then we have $a = \{0, 1, 2, 0, 0, 0\}$, which is normalized by subtracting the mean $\frac{1}{2}$, such that $a = \{-\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, -\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}\}$. The array portion $a[2..3]$ is the maximum subarray, which suggests that customers in their twenties and thirties are promising groups for purchasing "ham".

If we use two numerical attributes for the antecedent, the problem is then equivalent to the 2D MSP. Aforementioned work by Fukuda et al. [41] described the 2D MSP in the context of two-dimensional association rules[†].

Fukuda and Takaoka [40] recently studied the association between air pollution level measured by particulate matter (PM) and various demographic attributes based on this technique. They successfully identified the association between age groups and specific range of $[PM_{10}]$ levels[‡], and variations of such an association depending on season and gender group.

## 1.3 Scope of this thesis

In this thesis, we review the MSP and various techniques previously designed to compute the problem. Further background information relating to this research area is given in Chapter 2.

In Part I, we study a generalized problem of $K$ maximum subarrays. We aim to develop efficient algorithms for both 1D and 2D versions of $K$-maximum subarray problem ($K$-MSP). The $K$-MSP has two sub-categories, where each solution is strictly disjoint from others or overlapping is allowed. We define both problems and design efficient solutions for them.

In Part II, we focus on developing efficient, yet practical, parallel algorithms for the 2D MSP. Various enhancements to the new mesh algorithms will be discussed, and extended mesh algorithms for the two $K$-MSPs will be also described.

While the research on the MSP is partly *motivated* by potential application areas, genomic sequence analysis, computer vision and data mining, it is not application-*oriented*. The main objective of the thesis is mostly confined to the development of efficient algorithms for the MSP and the $K$-MSPs, and the considerations for potential application are kept minimal. This decision is deliberate.

[†] This paper also applies [5] and [104] to find two-dimensional association rules contained in a connected $x$-monotone and in a rectilinear convex region.

[‡] Particulate matter with diameter less than $10\mu m$

The MSP, indeed, is a conceptually simple problem. In most advanced applications, it would run in conjunction with other techniques, or slight change to the problem definition will be necessary. Recent examples include the MSP with length constraints and the maximum average subarray problem.

The range of techniques presented in this thesis and efficient algorithms for the original MSP and the generalized $K$-MSP are generic, and it is expected that they can give some insight into other similar problems, or variations of the MSP that have not yet emerged.

Early versions of this research were published. These publication are listed as References [6, 7, 8, 9, 10, 12, 11].

# Chapter 2

# Background Information

## 2.1 Maximum Subarray in 1D Array

### 2.1.1 Problem Definition

We give a formal definition of the maximum subarray problem (MSP) for 1D. Let MAX($L$) be the operation that selects the maximum element in a list $L$.

**Definition 2.1.1.** For a given array $a[1..n]$ containing positive and negative real numbers and 0, the *maximum subarray* is the consecutive array elements of the greatest sum, such that,

$$M = \mathrm{MAX}(L), \text{ where } L = \left\{ \sum_{x=i}^{j} a[x] \mid 1 \leq i \leq j \leq n \right\}$$

This problem can be computed in a number of ways. Among them, we present some earlier results to solve this problem in linear time.

In this thesis, to avoid confusion, the notations *min* and *max* in italic font are used for variable, list or array names, and MIN and MAX are used for minimum and maximum operations. We will use, however, the lowercase min for minimum operation inside $O$-notation following the convention. When we present an algorithm, we follow the `C99` comment style, given by `//` or `/*..*/`.

### 2.1.2 Iterative Algorithms

The first linear time algorithm for 1D is referred to as Kadane's algorithm after the inventor [18], which we describe in Algorithm 1.

**Algorithm 1** Kadane's algorithm

```
1: $M \leftarrow 0$, $t \leftarrow 0$
2: $i \leftarrow 1$
3: for $j \leftarrow 1$ to $n$ do
4:     $t \leftarrow t + a[j]$
5:     if $t > M$ then $M \leftarrow t$, $(x_1, x_2) \leftarrow (i, j)$
6:     if $t \leq 0$ then $t \leftarrow 0$, $i \leftarrow j + 1$// reset the accumulation
7: end for
8: output $M$, $(x_1, x_2)$
```

Algorithm 1 finds the maximum subarray $a[x_1..x_2]$ and its sum, $M$. In the following, we represent a subarray $a[i..j]$ by $(i, j)$. The algorithm accumulates a partial sum in $t$ and updates the current solution $M$ and the position $(x_1, x_2)$ when $t$ becomes greater than $M$. If the location is not needed, we may omit the update of $(x_1, x_2)$ for simplicity. If $t$ becomes non-positive, we reset the accumulation discarding tentative accumulation in $t$. This is because the maximum subarray $M$ will not start with a portion of non-positive sum*. Note that $M$ and $(x_1, x_2)$ are initialized to 0 and $(0, 0)$ respectively to allow an empty set to be the solution, if the input array is all negative. This may be optionally set to $-\infty$ to force the algorithm to obtain a non-empty set.

Here, we observe the following property.

**Lemma 2.1.2.** *At the $j$-th iteration, $t = a[i] + a[i+1] + .. + a[j]$, that is the maximum sum ending at $a[j]$,*

meaning that no subarray $(x, j)$ for $x < i$ or $x > i$ can be greater than $(i, j)$, whose sum is currently held at $t$. A simple proof by contradiction can be made, which we omit.

Another linear time iterative algorithm, which we shall describe below, is probably easier to understand.

We first describe the concept of prefix sum. The prefix sum $sum[i]$ is the sum of preceding array elements, such that $sum[i] = a[1]+..+a[i]$. Algorithm 2 computes the prefix sums $sum[1..n]$ of a 1D array $a[1..n]$ in $O(n)$ time.

* Alternatively one may reset $t$ when $t$ becomes "negative" allowing a sequence may begin with a portion of zero sum. However, we may have smaller and more focused area of the same sum by excluding such a portion.

**Algorithm 2** Prefix sum computation

```
$sum[0] \leftarrow 0$
**for** $i \leftarrow 1$ to $n$ **do** $sum[i] \leftarrow sum[i-1] + a[i]$
```

As $sum[x] = \sum_{i=1}^{x} a[i]$, the sum of $a[x..y]$ is computed by the subtraction of these prefix sums such as:

$$\sum_{i=x}^{y} a[i] = sum[y] - sum[x-1]$$

To yield the maximum sum from a 1D array, we have to find indices $x, y$ that maximize $\sum_{i=x}^{y} a[i]$. Let $min_i$ be the minimum prefix sum for an array portion $a[1..i-1]$. Then the following is obvious.

**Lemma 2.1.3.** *For all $x, y \in [1..n]$ and $x \leq y$,*

$$\begin{aligned}
&\underset{1\leq x\leq y\leq n}{\mathrm{MAX}} \left\{ \sum_{i=x}^{y} a[i] \right\} = \underset{1\leq x\leq y\leq n}{\mathrm{MAX}} \{sum[y] - sum[x-1]\} \\
&= \underset{1\leq y\leq n}{\mathrm{MAX}} \left\{ sum[y] - \underset{1\leq x\leq y}{\mathrm{MIN}} \{sum[x-1]\} \right\} \\
&= \underset{1\leq y\leq n}{\mathrm{MAX}} \{sum[y] - min_y\}
\end{aligned}$$

Based on Lemma 2.1.3, we can devise a linear time algorithm (Algorithm 3) that finds the maximum sum in a 1D array.

**Algorithm 3** Maximum Sum in a one-dimensional array

```
1: $min \leftarrow 0$ //minimum prefix sum
2: $M \leftarrow 0$ //current solution. 0 for empty subarray
3: $sum[0] \leftarrow 0$
4: **for** $i \leftarrow 1$ to $n$ **do**
5:     $sum[i] \leftarrow sum[i-1] + a[i]$
6:     $cand \leftarrow sum[i] - min$ //min=$min_i$
7:     $M \leftarrow \mathrm{MAX}\{M, cand\}$
8:     $min \leftarrow \mathrm{MIN}\{min, sum[i]\}$ //min=$min_{i+1}$
9: **end for**
10: output $M$
```

**Algorithm 4** $O(n \log n)$ time algorithm for 1D

```
procedure $MaxSum(f,t)$ begin
//Finds maximum sum in $a[f..t]$
 1: if $f = t$ then return $a[f]$ //One-element array
 2: $c \leftarrow (f + t - 1)/2$  //Halves array. $left$ is $a[f..c]$, $right$ is $a[c+1..t]$
 3: $ssum \leftarrow 0$, $M^{left}_{center} \leftarrow 0$
 4: for $i \leftarrow c$ down to $f$ do
 5:     $ssum \leftarrow ssum + a[i]$
 6:     $M^{left}_{center} \leftarrow \text{MAX}\{M^{left}_{center}, ssum\}$
 7: end for
 8: $psum \leftarrow 0$, $M^{right}_{center} \leftarrow 0$
 9: for $i \leftarrow c + 1$ to $t$ do
10:     $psum \leftarrow psum + a[i]$
11:     $M^{right}_{center} \leftarrow \text{MAX}\{M^{right}_{center}, psum\}$
12: end for
13: $M_{center} \leftarrow M^{left}_{center} + M^{right}_{center}$  //Solution for the center problem
14: $M_{left} \leftarrow MaxSum(f,c)$  //Solution for left half
15: $M_{right} \leftarrow MaxSum(c + 1,t)$  //Solution for right half
16: return $\text{MAX}\{M_{left}, M_{right}, M_{center}\}$  //Selects the maximum of three
end
```

While we accumulate $sum[i]$, the prefix sum, we also maintain $min$, the minimum of the preceding prefix sums. By subtracting $min$ from $sum[i]$, we produce a candidate for the maximum sum, which is stored in $cand$. At the end, $M$ is the maximum sum. For simplicity, we omitted details for locating the subarray corresponding to $M$.

To the author's knowledge, the first literature that presented this algorithm is attributed to Qiu and Akl [79].

### 2.1.3 Recursive Algorithms

Another linear time algorithm based on the divide-and-conquer technique can be made. We first review Algorithm 4 by Bentley [18], which provides a starting point for the linear time optimization. Algorithm 4 and its enhanced version, Algorithm 5, are given for historical reasons. Readers may skip these and start from Algorithm 6.

Algorithm 4 is based on the following principle.

**Algorithm 5** Smith's $O(n)$ time algorithm for 1D

```
procedure $MaxSum2(f,t)$ begin
//Finds maximum sum in $a[f..t]$
1: if $f = t$ then return $(a[f], a[f], a[f], a[f])$ //One-element array
2: $c \leftarrow (f + t - 1)/2$ //Halves array. $left$ is $a[f..c]$, $right$ is $a[c+1..t]$
3: $(M_{left}, maxS_{left}, maxP_{left}, total_{left}) \leftarrow MaxSum2(f,c)$
4: $(M_{right}, maxS_{right}, maxP_{right}, total_{right}) \leftarrow MaxSum2(c+1,t)$
5: $maxS \leftarrow \text{MAX}\{maxS_{right}, total_{right} + maxS_{left}\}$ //Max suffix
6: $maxP \leftarrow \text{MAX}\{maxP_{left}, total_{left} + maxP_{right}\}$ //Max prefix
7: $total \leftarrow total_{left} + total_{right}$ //Total sum of elements
8: $M_{center} \leftarrow maxS_{left} + maxP_{right}$ //Solution for the center problem
9: $M \leftarrow \text{MAX}\{M_{left}, M_{right}, M_{center}\}$
10: return $(M, maxS, maxP, total)$
end
```

**Lemma 2.1.4.** *The maximum sum M is,*

$$M = \{M_{left}, M_{right}, M_{center}\}$$

We recursively decompose the array into two halves until there remains only one element. We find the maximum sum in the left half, $M_{left}$, and the maximum sum in the right half, $M_{right}$. While $M_{left}$ and $M_{right}$ are entirely in the left half or in the right half, we also consider a maximum sum that crosses the central border, which we call $M_{center}$.

To obtain $M_{center}$, lines 3-7 find a portion of it located in the left half, $M^{left}_{center}$. Similarly, lines 8-12 computes $M^{right}_{center}$. The sum of $M^{left}_{center}$ and $M^{right}_{center}$ then makes $M_{center}$.

As the array size is halved each recursion, the depth of recursion is $O(\log n)$. At each level of recursion, $O(n)$ time is required for computing $M_{center}$. Thus this algorithm is $O(n \log n)$ time. Smith [84] pointed out that the $O(n)$ time process for computing $M_{center}$ at each level of recursion can be reduced to $O(1)$ by retrieving more information from the returned outputs of recursive calls. His $O(n)$ total time algorithm is given in Algorithm 5.

Consider a revised version $MaxSum2(f,t)$ that returns a 4-tuple ($M$, $maxS$, $maxP$,$total$) as the output, whose attributes respectively represent the maximum sum ($M$), the maximum suffix sum ($maxS$), the maximum prefix sum ($maxP$) and the total sum of array elements in $a[f..t]$ ($total$).

Assuming that $maxS_{left}$ and $maxP_{right}$ are returned by the recursive calls in line 3 and 4, we show that lines 5 and 6 correctly compute $maxS$ and $maxP$.

**Lemma 2.1.5.** *The maximum prefix sum for $a[f..t]$, $maxP$, is obtained by*

$$\text{MAX}\left\{maxP_{left}, total_{left} + maxP_{right}\right\}$$

*Proof.* Let $c = (f + t - 1)/2$,

$$\begin{aligned} maxP &= \text{MAX}\left\{maxP_{left}, total_{left} + maxP_{right}\right\} \\ &= \text{MAX}\left\{\underset{f \le x \le c}{\text{MAX}}\left\{\sum_{i=f}^{x} a[i]\right\}, \sum_{p=f}^{c} a[p] + \underset{c+1 \le x \le t}{\text{MAX}}\left\{\sum_{i=c+1}^{x} a[i]\right\}\right\} \\ &= \underset{f \le x \le t}{\text{MAX}}\left\{\sum_{i=f}^{x} a[i]\right\} \end{aligned}$$

□

The maximum suffix sum can be recursively computed in a similar manner.

To compute the maximum sum in $a[1..n]$, we call $MaxSum2(1, n)$, and obtain $M$ from the returned 4-tuple.

The time complexity $T(n)$ can be obtained from the recurrence relation,

$$T(n) = 2T(n/2) + O(1),\ T(1) = O(1),$$

hence $T(n) = O(n)$.

We can simplify the algorithm such that it would not need to compute the suffix sums. Suppose we have the prefix sums $sum[1..n]$ pre-computed by Algorithm 2. Then the following Lemma provides an alternative to compute $M_{center}$ that can be used in lieu of line 8 of Algorithm 5.

**Lemma 2.1.6.**

$$M_{center} = \underset{c+1 \le x \le t}{\text{MAX}}\left\{sum[x]\right\} - \underset{f-1 \le x \le c}{\text{MIN}}\left\{sum[x]\right\}$$

**Algorithm 6** Alternative $O(n)$ time divide-and-conquer algorithm for 1D

```
procedure $MaxSum3(f,t)$ begin
//Finds maximum sum in $a[f..t]$
1: if $f = t$ then return $(a[f], sum[f-1], sum[f])$ //One-element array
2: $c \leftarrow (f+t-1)/2$ //Halves array. $left$ is $a[f..c]$, $right$ is $a[c+1..t]$
3: $(M_{left}, minP_{left}, maxP_{left}) \leftarrow MaxSum3(f,c)$
4: $(M_{right}, minP_{left}, maxP_{right}) \leftarrow MaxSum3(c+1,t)$
5: $minP \leftarrow \text{MIN}\{minP_{left}, minP_{right}\}$ //Min prefix sum in $sum[f-1..t-1]$
6: $maxP \leftarrow \text{MAX}\{maxP_{left}, maxP_{right}\}$ //Max prefix sum in $sum[f..t]$
7: $M_{center} \leftarrow maxP_{right} - minP_{left}$ //Solution for the center problem
8: $M \leftarrow \text{MAX}\{M_{left}, M_{right}, M_{center}\}$
9: return $(M, minP, maxP)$
end
```

Including the pre-process for the prefix sum computation, Algorithm 6 takes $O(n)$ time in total. We will be using this algorithm as a framework in Chapter 4. This algorithm was introduced in the mid-year examination of COSC229, a second year algorithm course offered at the University of Canterbury in 2001 [88].

## 2.2 Maximum Subarray in 2D Array

We consider the two-dimensional (2D) version of this problem, where we are given an input array of size $m \times n$. We assume that $m \leq n$ without loss of generality.

In 1D, we have optimal linear time solutions, Algorithm 1, Algorithm 3 and Algorithm 5. These algorithms can be extended to 2D through a simple technique, which we shall refer to as *strip separation*. Alternatively, a framework specific to 2D based on distance matrix multiplication (DMM) is known. While we will be mostly using strip separation technique throughout this thesis, we describe both techniques.

Throughout this thesis, we denote a maximum subarray with a sum $M$ located at $a[g..i][h..j]$ by $M(g,h)|(i,j)$.

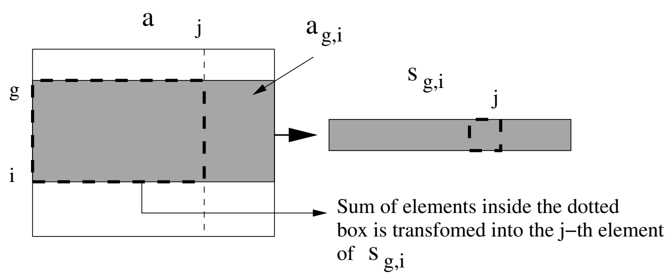

a
j
a g,i
g
i
s g,i
j
Sum of elements inside the dotted box is transfomed into the j−th element of s g,i

Figure 2.1: Separating a strip $sum_{g,i}$ from the 2D input array

### 2.2.1 Strip separation

A *strip* is a subarray of $a$ with horizontally full length, i.e. $n$, which covers multiple consecutive rows. For example, a subarray $a[g..i][1..n]$ is a strip covering row $g..i$. Let this strip be denoted by $a_{g,i}$. Likewise, in the following description, a variable associated with a strip $a_{g,i}$ is expressed with the subscript, such as $sum_{g,i}$, whose meaning will be described later. In an array $a[1..m][1..n]$, there are $m(m+1)/2$ possible combinations of $g$ and $i$, thus the same number of strips.

The essence of the strip separation technique relies on the transformation of a strip into a 1D prefix sum array. As a result, we fix $g$ and $i$, and the problem is now simply to find $h$ and $j$ such that the sum of $(g,h)|(i,j)$ could be maximized. This can be solved by any of the aforementioned algorithm for 1D. Upon completion of running an algorithm for 1D, we obtain $h$ and $j$, and $M$, which correspond to the rectangular subarray that starts at the $g$-th row, ends at the $i$-th row, i.e. $M(g,h)|(i,j)$.

We describe how to transform a strip into a 1D problem.

We first compute the prefix sums $sum[1..m][1..n]$, where $sum[i][j]$ is defined as,

$$sum[i][j] = \sum_{1 \le p \le i, 1 \le q \le j} a[p][q]$$

Actual computation of $sum[i][j]$ can be done iteratively such as,

$$sum[i][j] = sum[i-1][j] + sum[i][j-1] - sum[i-1][j-1] + a[i][j]$$

,where $sum[0][1..n] = 0$ and $sum[1..m][0] = 0$.

Now, we compute the prefix sum of a strip $a_{g,i}$, which is denoted by $sum_{g,i}$. We wish $sum_{g,i}[j]$ to be the prefix sum starting from $a[g][1]$, such that,

$$sum_{g,i}[j] = \sum_{g \leq p \leq i, 1 \leq q \leq j} a[p][q]$$

If we have pre-computed $sum[1..m][1..n]$, the prefix sum of the entire array, the computation of $sum_{g,i}[1..n]$ can be easily done by,

$$sum_{g,i}[j] = sum[i][j] - sum[g-1][j]$$

For each combination of $g, i$, we obtain $sum_{g,i}$ through this computation. We call $sum_{g,i}$, a *strip prefix* sum of $a_{g,i}$ in this thesis. Computing a strip prefix $sum_{g,i}$ from the input array is illustrated in Figure 2.1.

The pre-process of computing the prefix sum $sum[1..m][1..n]$ is done in $O(mn)$ time. A strip prefix $sum_{g,i}[1..n]$ needs $O(n)$ time to retrieve from $sum$. As there are $m(m+1)/2$ possible combinations of $g$ and $i$, computing all strip prefixes from the input array takes $O(m^2n)$ time.

Since each strip prefix sum is a 1D array, we can apply the algorithms devised for 1D problem, such as Algorithm 3, 5 or 6, to compute the maximum sum in each strip. As each maximum sum in a strip is computed in $O(n)$ time, we spend $O(m^2n)$ time to process all strips, and to select the largest one among $O(m^2)$ solutions.

Including the pre-process time, the total time for 2D version of the MSP is $O(m^2n)$.

In [19], Bentley informally described this technique to present $O(m^2n)$ time solution for the 2D MSP. In the following, we describe how the main principle of the strip separation is applied to some of algorithms for 1D.

### Extending Algorithm 3

We devise an algorithm for the 2D MSP based on Algorithm 3 to process 2D MSP.

The correctness of the presented Algorithm 7 is obvious. We separate a strip $a_{g,i}$ (for all $g$ and $i$ in $1 \leq g \leq i \leq m$) from the input array, and let

**Algorithm 7** 2D version of Algorithm 3

```
 1: /* Pre-process: Compute prefix sums */
 2: Set $sum[0][1..n] \leftarrow 0$, $sum[1..m][0] \leftarrow 0$
 3: for $i \leftarrow 1$ to $m$ do
 4:   for $j \leftarrow 1$ to $n$ do
 5:     $sum[i][j] \leftarrow sum[i-1][j] + sum[i][j-1] - sum[i-1][j-1] + a[i][j]$
 6:   end for
 7: end for
 8: $M \leftarrow 0$, $s[0] \leftarrow 0$
 9: for $g \leftarrow 1$ to $m$ do
10:   for $i \leftarrow g$ to $m$ do
11:     /* Computes 1D problem */
12:     $min \leftarrow 0$
13:     for $j \leftarrow 1$ to $n$ do
14:       $s[j] \leftarrow sum[i][j] - sum[g-1][j]$ //$s[j] = sum_{g,i}[j]$
15:       $cand \leftarrow s[j] - min$
16:       $M \leftarrow \text{MAX}\{M, cand\}$
17:       $min \leftarrow \text{MIN}\{min, s[j]\}$
18:     end for
19:   end for
20: end for
21: output $M$
```

the routine derived from Algorithm 3 process each strip. The prefix sums $sum[1..m][1..n]$ are computed during the pre-process routine.

Note that the initialization of $M$ is made by line 8 before the nested for-loop begins, and $min$ is reset by line 12 before we start to process the strip $a_{g,i}$ through the inner-most loop (lines 13–18).

We use $s$ as a container for $sum_{g,i}$. As given in Lemma 2.1.3, we subtract the minimum prefix sum from the current prefix sum to obtain a candidate, *cand*. This candidate is compared with current maximum sum and the greater is taken.

For simplicity, operations for computing the location of the maximum subarray are not shown. When the location is needed, it can be easily computed in the following way. Suppose the current minimum prefix sum, *min*, is $sum_{g,i}[j_0]$. When line 15 is executed, we know that *cand* corresponds to the sum of subarray $(g, j_0+1)|(i, j)$. If this *cand* is selected as $M$, we copy

**Algorithm 8** Alternative 2D version of Algorithm 3

```
1: $M \leftarrow 0$, $min \leftarrow 0$
2: for $g \leftarrow 1$ to $m$ do
3:    Set $s[1..n] \leftarrow 0$
4:    for $i \leftarrow g$ to $m$ do
5:       $min \leftarrow 0$
6:       for $j \leftarrow 1$ to $n$ do
7:          $r[i][j] \leftarrow r[i][j-1] + a[i][j]$ //Assume $r[i][0] = 0$
8:          $s[j] \leftarrow s[j] + r[i][j]$ //$s[j] = sum_{g,i}[j]$
9:          $cand \leftarrow s[j] - min$
10:         $M \leftarrow \text{MAX}\{M, cand\}$
11:         $min \leftarrow \text{MIN}\{min, s[j]\}$
12:      end for
13:   end for
14: end for
15: output $M$
```

this location to $(r_1, c_1)|(r_2, c_2)$.

Alternatively, we can dispose of the pre-process of prefix sums, $sum[1..m][1..n]$ given in lines 1–7. Notice that $sum_{g,i}[j]$ can be instead obtained by

$$sum_{g,i}[j] = sum_{g,i-1}[j] + r[i][j], \text{ where } r[i][j] = \sum_{q=1}^{j} a[i][q]$$

This observation leads us to a simpler algorithm, Algorithm 8.

Notice that this algorithm does not need prefix sums $sum[1..m][1..n]$. Instead, we compute $r[i][j]$ (line 7) and add it to $s[j]$ that currently contains $sum_{g,i-1}[j]$ to get $sum_{g,i}[j]$. To ensure the correct computation of $sum_{g,i}[j]$ through this scheme, line 3 resets $s[1..n]$ to 0, when the top boundary of the strip, $g$ changes.

One may notice that placing the operation that computes $r[i][j]$ at line 7 is not very efficient. Due to the outer-most loop (lines 2–14), for $i$ possible values of $g$, we unnecessarily compute the same value of $r[i][j]$ many times. Certainly, this line can be moved out of the loop, and placed at the top of the algorithm before line 1, surrounded by another double-nested loop by $i$ and $i$. Either way, there is no change in the overall time complexity. The reason for placing line 7 at its present position is to make the algorithm more transparent for converting into a mesh parallel MSP algorithm, which we will describe later in Chapter 8.

It is easy to see that the presented two versions of 2D MSP algorithms have $O(m^2n)$ time complexity.

### Extending Algorithm 1 (Kadane's Algorithm)

Algorithm 1 is not based on the prefix sums, and the concept of strip prefix sum is not directly applicable. Still, the basic idea remains intact.

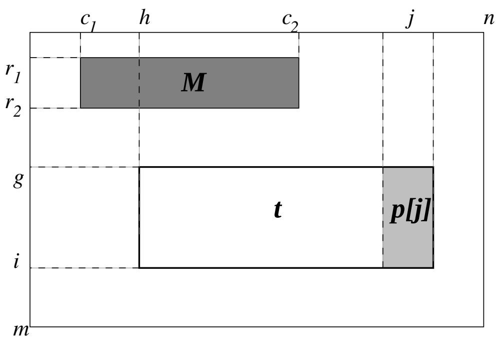


Figure 2.2: Snapshot of Algorithm 9

Consider Figure 2.2. The figure illustrates some arbitrary point of time where the current maximum subarray $M(r_1, c_1)|(r_2, c_2)$ is near top-left corner, and we are processing strip $a_{g,i}$. Here, we have a column-wise (vertical) partial sum of elements in $p[j]$, which corresponds to $\sum a[g..i][j]$. We perform Algorithm 1 on each strip $a_{g,i}$ for all pair of $g$ and $i$, and maintain $t$ to accumulate $p[h..j]$. In the figure, $t$ is the sum of the area surrounded by thick line, which is the accumulation $p[h] + ... + p[j]$. If $t$ is greater than the current maximum $M$, we replace $M$ with $t$ and update its location $(r_1, c_1)|(r_2, c_2)$. Again, if $t$ becomes non-positive, $t$ is reset and we update $h$ to $j + 1$, to start a fresh accumulation.

Similar to Lemma 2.1.2, we can derive the following.

**Corollary 2.2.1.** *For fixed i,j and g, the value of t is the maximum sum ending at $a[i][j]$ with the top boundary g.*

**Algorithm 9** 2D version of Algorithm 1 (Kadane's Algorithm)

```
1: $M \leftarrow 0$, $(r_1, c_1) \leftarrow (0,0)$, $(r_2, c_2) \leftarrow (0,0)$
2: for $g \leftarrow 1$ to $m$ do
3:    for $j \leftarrow 1$ to $n$ do $p[j] \leftarrow 0$
4:    for $i \leftarrow g$ to $m$ do
5:       $t \leftarrow 0,\ h \leftarrow 1$
6:       for $j \leftarrow 1$ to $n$ do
7:          $p[j] \leftarrow p[j] + a[i][j]$
8:          $t \leftarrow t + p[j]$
9:          if $t > M$ then $M \leftarrow t$, $(r_1, c_1) \leftarrow (g, h)$, $(r_2, c_2) \leftarrow (i, j)$
10:         if $t \le 0$ then $t \leftarrow 0$, $h \leftarrow j + 1$// reset the accumulation
11:      end for
12:   end for
13: end for
14: output $M(r_1, c_1)|(r_2, c_2)$
```

There are total of $O(m^2)$ such horizontal strips. This algorithm also computes the maximum subarray in $O(m^2n)$ time, which is cubic when $m = n$.

### 2.2.2 Distance Matrix Multiplication

Using the divide-and-conquer framework, Smith also derived a 2D algorithm from Algorithm 5, which achieves $O(m^2n)$ time. Tamaki and Tokuyama found that a divide-and-conquer framework similar to Smith's can be related to Takaoka's distance matrix multiplication(DMM) [87], and reduced the upper bound to sub-cubic time. Takaoka later gave a simplified algorithm with the same complexity [89].

Let us briefly review DMM. For two $n \times n$ matrices $A$ and $B$, the product $C = A \cdot B$ is defined by

$$c_{ij} = \underset{1 \le k \le n}{\mathrm{MIN}} \{a_{ik} + b_{kj}\} \quad (i, j = 1, ..., n) \tag{2.1}$$

The operation in the right-hand-side of (2.1) is called *distance matrix multiplication*(DMM) and $A$ and $B$ are called distance matrices in this context. The best known DMM algorithm runs in $O(n^3\sqrt{\log\log n/\log n})$ time, which is sub-cubic, due to Takaoka [87].

**Algorithm 10** Maximum subarray for two-dimensional array

```
1: If the array becomes one element, return its value.
2: Otherwise, if $m > n$, rotate the array 90 degrees.
3:  /* Now we assume $m \leq n$ */
4: Let $M_{left}$ be the solution for the left half.
5: Let $M_{right}$ be the solution for the right half.
6: Let $M_{center}$ be the solution for the center problem.
7: Let the solution be the maximum of those three.
```

Let us review the 2D MSP in this context. Consider we have the prefix sums, $sum[1..m][1..n]$ ready. The prefix sum array takes $O(mn)$ time to compute and is used throughout the whole process.

The outer framework of Takaoka's algorithm [89] is given in Algorithm 10.

In this algorithm, the center problem is to obtain an array portion that crosses over the central vertical line with maximum sum, and can be solved in the following way.

$$M_{center} = \operatorname*{MAX}_{\substack{0\leq g\leq i-1\\ 0\leq h\leq n/2-1\\ 1\leq i\leq m\\ n/2+1\leq j\leq n}} \{sum[i][j] - sum[i][l] - sum[g][j] + sum[g][h]\} \tag{2.2}$$

In the above equation, we first fix $g$ and $i$, and maximize the above by changing $h$ and $j$. Then the above problem is equivalent to maximizing the following. For $i = 1..m$ and $g = 0..i-1$,

$$M_{center}[i,g] = \operatorname*{MAX}_{\substack{0\leq h\leq n/2-1\\ n/2+1\leq j\leq n}} \{-sum[i][h] + sum[g][h] + sum[i][j] - sum[g][j]\}$$

Let $sum^*[i][j] = -sum[j][i]$. Then the above problem can further be converted into

$$\begin{aligned} M_{center}[i,g] = -&\operatorname*{MIN}_{0\leq h\leq n/2-1} \{sum[i][h] + sum^*[h][g]\} \\ +&\operatorname*{MAX}_{n/2+1\leq j\leq n} \{sum[i][j] + sum^*[j][g]\} \end{aligned} \tag{2.3}$$

Here, Tamaki and Tokuyama [94] discovered that the computation above is similar to DMM[†] . In (2.3), the first part can be computed by DMM as stated in (2.1) and the second part is done by a modified DMM with MAX-operator.

Let $S_1$ and $S_2$ be matrices whose elements at position $(i, j)$ are $sum[i, j-1]$ and $sum[i, j+n/2]$ for $i = 1..m;\ j = 1..n/2$. For an arbitrary matrix $T$, let $T^*$ be obtained by negating and transposing $T$. As the range of $g$ is $[0 .. m-1]$ in $S_1^*$ and $S_2^*$, we shift it to $[1..m]$. Then the above can be computed by

$$S_2 S_2^* - S_1 S_1^* \tag{2.4}$$

,where multiplication of $S_1$ and $S_1^*$ is computed by the MIN-version, and that of $S_2$ and $S_2^*$ is done by the MAX-version. Then subtraction of the distance products is done component-wise. Finally $M_{center}$ is computed by taking the maximum from the lower triangle of the resulting matrix.

For simplicity, we apply the algorithm on a square array of size $n \times n$, where $n$ is a power of 2. Then all parameters $m$ and $n$ appearing through recursion in Algorithm 10 are power of 2, where $m = n$ or $m = n/2$. We observe the algorithm splits the array vertically and then horizontally.

We define the work of computing the three $M_{center}$'s through this recursion of depth 2 to be the work at level 0. The algorithm will split the array horizontally and then vertically through the next recursion of depth 2. We call this level 1, etc.

Now let us analyze the time for the work at level 0. We measure the time by the number of comparisons for simplicity. Let $Tm(n)$ be the time for multiplying two $(n/2, n/2)$ matrices. The multiplication of $n \times n/2$ and $n/2 \times n$ matrices is done by 4 multiplications of size $n/2 \times n/2$, which takes $4Tm(n)$ time. Due to (2.3), $M_{center}$ takes each of MIN- and MAX-version of multiplications. Thus $M_{center}$ involving $n \times n/2$ and $n/2 \times n$ matrices requires $8Tm(n)$ time, and computing two smaller $M_{center}$'s involving $n/2 \times n/2$ matrices takes $4Tm(n)$ time.

Then each level, computing an $M_{center}$ and two smaller $M_{center}$'s accounts

[†] Their version does not use prefix sum, and is similar to Algorithm 4. Only a MAX version of DMM is used.

for $12Tm(n)$ time. We have the following recurrence for the total time $T(n)$.

$$T(1) = 0$$
$$T(n) = 4T(n/2) + 12Tm(n)$$

Takaoka showed that the following Lemma holds.

**Lemma 2.2.2.** *Let $c$ be an arbitrary constant such that $c > 0$. Suppose $Tm(n)$ satisfies the condition $Tm(n) \geq (4+c)Tm(n/2)$. Then the above $T(n)$ satisfies $T(n) \leq 12(1+4/c)Tm(n)$.*

Clearly the complexity of $O(n^3\sqrt{\log\log n/\log n})$ for $Tm(n)$ satisfies the condition of the lemma with some constant $c > 0$. Thus the maximum subarray problem can be solved in $O(n^3\sqrt{\log\log n/\log n})$ time. Since we take the maximum of several matrices component-wise in our algorithm, we need an extra term of $O(n^2)$ in the recurrence to count the number of operations. This term can be absorbed by slightly increasing 12, the coefficient of $Tm(n)$.

Suppose $n$ is not given by a power of 2. By embedding the array $a$ in an array of size $n' \times n'$ such that $n'$ is the next power of 2 and the gap is filled with 0, we can solve the original problem in the complexity of the same order.

### 2.2.3 Lower Bound for the 2D MSP

A trivial lower bound for the 2D MSP is $\Omega(mn)$ or $\Omega(n^2)$ if $m = n$. Whereas the upper bound has been improved from $O(n^3)$ to sub-cubic by Tamaki and Tokuyama [94] and Takaoka [89], a non-trivial lower bound remains open and there is a wide gap between the lower bound and the upper bound.

Remarkably, a non-trivial lower bound for the all-pairs shortest paths (APSP) is still not known [44]. Considering that the 2D MSP is closely related to the APSP, a non-trivial lower bound for the 2D MSP is expected difficult to obtain.

## 2.3 Selection

A selection problem arises when we find the $k$-th largest (or $k$-th smallest) of $n$ elements in a list. Subsets of this problem include finding the minimum, maximum, and median elements. These are also called *order statistics*. Perhaps finding the minimum or maximum in a list is one of the most frequent problems in programming. A simple iteration through all elements in the list gives the solution, meaning that it is a linear time operation in the worst case.

Finding the $k$-th largest is more difficult. Certainly, if all $n$ elements are sorted, selecting the $k$-th is trivial. Then sorting is the dominant process, which is $O(n \log n)$ time if a fast sorting algorithm such as merge sort, heap sort etc., is applied. Performing sorting is, however, wasteful when the order of all elements are not required.

Knuth [63] finds the origin of this problem goes back to Lewis Carroll's essay on tennis tournaments that was concerned about the unjust manner in which prizes were awarded in the competition. Many efforts have been made in the quest for the faster algorithms. The linear average-time algorithm was presented by Hoare [49] and Floyd and Rivest [35] developed an improved average-time algorithm that partitions around an element recursively selected from a small sample of the elements. The worst-case linear time algorithm was finally presented by Blum, Floyd, Pratt, Rivest and Tarjan [20].

The MSP is essentially a branch of the selection problem. We review two notable techniques for selecting the $k$-th largest item.

### 2.3.1 Tournament

Knuth [63] gave a survey on the history of the selection problem, the problem of finding the $k$-th largest of $n$ elements, which goes back to the essay on tennis tournaments by Rev. C. L. Dodgson (also known as Lewis Carroll). Dodgson set out to design a tournament that determines the true second- and third-best players, assuming a transitive ranking, such that if player $A$ beats player $B$, and $B$ beats $C$, $A$ would beat $C$.

A *tournament* is a complete binary tree that is a conceptual representation of the recursive computation of MAX (or MIN) operation over $n$

**Algorithm 11** Select maximum of $a[f]...a[t]$ by tournament

```
procedure tournament(node,f,t) begin
1: $node.from \leftarrow f$, $node.to \leftarrow t$
2: if $from = to$ then
3:     $node.val \leftarrow a[f]$
4: else
5:     create $left$ child node, tournament($left$,$f$,$\frac{f+t-1}{2}$)
6:     create $right$ child node, tournament($right$,$\frac{f+t+1}{2}$,$t$)
7:     $node.val \leftarrow$ MAX $(left.val, right.val)$ //MAX $\{a[f], ..a[t]\}$
8: end if
end
```

elements. To build a tournament, we prepare *root* node and run Algorithm 11 by executing *tournament*(*root*,1,*n*). When the computation completes, *root.val* is MAX $\{a[1], ..a[n]\}$.

Counting from the leaf level, the number of nodes are halved each level such that $n + n/2 + ... + 1$, the total number of nodes are $O(n)$, meaning that it takes $O(n)$ to find the maximum.

It may be argued that building a tree is not essential if we are only interested in the first maximum. However, having the tree makes it very convenient to find the next maximum. To do this, we first locate the leaf whose value advanced all the way to the *root*. We replace the value of this leaf with $-\infty$ and update each node along the path towards *root*. The next maximum is now located at *root*. This takes $O(\log n)$ time as the tree is a complete binary tree with the height $O(\log n)$. The $k$-th maximum can be found by repeating this process.

Including the time for building the tree, we spend total of $O(n + k \log n)$ time to find $k$ maximum values. Furthermore, Bengtsson and Chen found that [16],

**Lemma 2.3.1.** *For any integer $k \leq n$, $O(n + k \log n) = O(n + k \log k)$*

*Proof.* We consider the following two cases,
(1) If $\frac{n}{\log n} < k \leq n$, $O(n + k \log k)$=$O(n + k \log n)$.
(2) If $k \leq \frac{n}{\log n}$, $O(n + k \log n) = O(n)$ and $O(n + k \log k)$=$O(n)$. □

**Algorithm 12** Quicksort

```
procedure $quicksort(a, begin, end)$ begin
 1: if length of $a > 1$ then
 2:     locate a pivot $a[i]$
 3:     $p \leftarrow partition(a, begin, end, i)$
 4:     $quicksort(a, begin, p - 1)$
 5:     $quicksort(a, p, end)$
 6: end if
end
```

Note that this tournament-based selection not only selects the $k$-th maximum element, but also finds all $k$ largest elements in sorted order.

A basically same complexity can be obtained with a similar data structure, *heap*. Heap (more formally, binary heap) is a complete binary tree where a node contains a value that is greater than (or smaller than) that of its children. Building a heap is also $O(n)$ time and we repeatedly delete the root $k$ times until we obtain the $k$-th largest element. A special case, deleting the root $n$ times, results in all $n$ elements in non-increasingly sorted order, which is of course, known as *heapsort*.

### 2.3.2 Linear time selection

The need for an efficient selection algorithm can be attributed to the famous *quicksort* by Hoare [50]. We describe how the selection algorithm affects the performance of the quicksort.

The key of linear selection algorithm relies on a good partition techniques. We define a process $partition(a, begin, end, i)$ as a function that puts all elements less than $a[i]$ on the left, and others greater than or equal to $a[i]$ on the right to the pivot $a[i]$. After this rearrangement, the new location of $a[i]$ will be returned as $p$.

For the maximum efficiency, we wish the problem size of two recursive calls to be the same. To achieve this, we need to take the median of $a[begin]..a[end]$ as the pivot $a[i]$, such that $p$ will be always the mid-point between *begin* and *end*. If the median is selected in $O(n)$ time in the worst-case, the total time for *quicksort* is $O(n \log n)$ time due to the following

**Algorithm 13** Select the $k$-th largest element in $a$

```
procedure select(k,a) begin
 1: if $|a| \leq 5$ then
 2:     sort $a$ and return the $k$-th largest
 3: else
 4:     divide $a$ into $\lfloor |a|/5 \rfloor$ subarrays of 5 elements each
 5:     sort each 5-element subarray and find the median of each subarray
 6:     let $M$ be the container of all medians of the 5-element subarrays
 7:     $m \leftarrow select(\lceil |M|/2 \rceil, M)$
 8:     Partition $a$ into $(a_1, a_2, a_3)$, where
        $a_1 = \{x | x \in a, x > m\}$,
        $a_2 = \{x | x \in a, x = m\}$,
        $a_3 = \{x | x \in a, x < m\}$
 9:     if $|a_1| \geq k$ then return $select(k, a_1)$
10:     else if $|a_1| + |a_2| \geq k$ then return $m$
11:     else return $select(k - |a_1| - |a_2|, a_3)$
12: end if
end
```

recurrence relation.

$$T(1) = O(1)$$
$$T(n) = 2T(n/2) + O(n)$$

Still, in 1962, when Hoare's original paper on quicksort [50] was published, the worst-case linear time solution was not known. Hoare used his own $FIND$ algorithm [49] which selects the median in $O(n)$ average time. After 11 years, Blum, Floyd, Pratt, Rivest and Tarjan finally established $O(n)$ worst-case time solution [20]. This algorithm has been reproduced in virtually every algorithm textbook, and we follow the description given in [1].

To select the $k$-th largest element, we first partition the array $a$ into subarrays of five elements each (line 4). Each subarray is sorted and the median is selected from each subarray (line 5) to form an auxiliary array $M$ (line 6). Sorting 5 elements requires no more than 8 comparisons, thus is a constant time.

The container of medians, $M$, has only $\lfloor n/5 \rfloor$ elements, and finding its median is therefore five times faster than doing that on an array of $n$ elements.

Let the median in $M$ be $m$ (line 7). As $m$ is the median of medians of 5 elements, at least one-fourth of the elements of $a$ are less than or equal to $m$, and at least one-fourth of the elements are greater than or equal to $m$. This observation is used to determine the overall complexity shortly.

We rearrange and partition the array $a$ into $a_1$, $a_2$ and $a_3$ such that each contains elements of $a$ greater than, equal to, and less than $m$ respectively (line 8).

Now we examine the size of each partition. If $|a_1|$, the size of $a_1$, is greater than or equal to $k$, we know that the $k$-th largest element is in $a_1$. So we run the selection on $a_1$ searching for the $k$-th largest element (line 9). Otherwise, if $|a_1| + |a_2| \geq k$, one of the element in $a_2$ is the $k$-th largest. Since all elements in $a_2$ are equal to $m$, we return $m$ (line 7). The last case is where the $k$-th largest is in $a_3$. We now recursively search for the $(k - |a_1| - |a_2|)$-th largest in $a_3$ (line 11). When all levels of the recursive calls terminate, the $k$-th largest in $a$ is eventually returned.

Now we analyze this algorithm. Let $T(n)$ be the total time required to select the $k$-th largest from an array of size $n$. The size of $M$ is at most $n/5$, thus the recursive call by line 4 requires at most $T(n/5)$ time.

Above, we observed that at least 1/4 elements are less than or equal to $m$. Meaning that the size of $a_1$ will be at most $3n/4$. Similarly, the maximum size of $a_3$ will be $3n/4$. Then the recursive call at line 9 or line 11 requires at most $T(3n/4)$ time. All other operations are bounded by $O(n)$. Therefore, the following recursive relation is established.

$$\begin{aligned} &T(n) = O(1) \quad , n \leq 5 \\ &T(n) \leq T(n/5) + T(3n/4) + O(n) \quad , n > 5 \end{aligned}$$

**Lemma 2.3.2.** *$T(n) = O(n)$, therefore selecting the $k$-th largest from a set of $n$ elements takes $O(n)$ time in the worst-case.*

*Proof.* We show that $T(n) \leq cn$ for some sufficiently large constant $c$. We also pick a constant $d$ to describe the term of $O(n)$ above is bounded by $dn$.

$$\begin{aligned} T(n) &\leq c(n/5) + c(3n/4) + dn \\ &= cn + (-cn/20 + dn), \end{aligned}$$

which is at most $cn$ if

$$-cn/20 + dn \leq 0$$

Choosing any integer $c$ and $d$ satisfying $c \leq 20d$ suffices to show $T(n) \leq cn$. Therefore $T(n) = O(n)$. □

**Selecting $k$ largest elements**

When the $k$-th largest element is selected, it is easy to collect all $k$ largest elements. Suppose the array $a$ contains $n$ elements and $kthMax$ is the $k$-th maximum. If $k > n$, there is no point processing all elements. Such a selection is regarded *invalid*. We return the whole array $a$ as the solution and exit. Otherwise, the selection is *valid*. We proceed to select $kthMax$ by the linear selection algorithm [20]. We rearrange and partition the array $a$ into $a_1$, $a_2$ and $a_3$ such that each contains elements of $a$ greater than, equal to, and less than $kthMax$ respectively (line 5).

We compare each element of $a$ against $kthMax$ and partition them into $a_1, a_2$ and $a_3$, such that each contains elements greater, equal to $kthMax$ and less than $kthMax$ respectively. If $|a_1| = k$, we take $a_1$ as the solution. If there are multiple elements of the same value as the $k$-th maximum, $|a_1| < k$. We take first $k - |a_1|$ elements from $a_2$ and append them to $a_1$. $a_1$ now contains $k$ largest values as required. The total time is bounded by $O(n)$.

**Lemma 2.3.3.** *Selection of $k$ largest values from a set of $n$ elements takes $O(n)$ time*

The selected $k$ values are, however, in random order. If we require sorted order, extra time for sorting, $O(k \log k)$, is introduced to the total time complexity. This is then asymptotically equivalent to the tournament-based selection given in Section 2.3.1. Indeed, Algorithm 13, while it is regarded

**Algorithm 14** Select $k$ largest elements in array $a$

```
procedure $extract(k, a)$ begin
1: $a_1, a_2, a_3 \leftarrow \emptyset$
2: if $k > n$ then return $a$ and exit
3: $kthMax \leftarrow select(k,a)$ //run Algorithm 13
4: Partition $a$ into $(a_1, a_2, a_3)$, where
   $a_1 = \{x|x \in a, x > kthMax\}$,
   $a_2 = \{x|x \in a, x = kthMax\}$,
   $a_3 = \{x|x \in a, x < kthMax\}$
5: if $|a_1| < k$ then append first $k - |a_1|$ elements of $a_2$ to $a_1$
6: return $a_1$
end
```

as theoretical breakthrough, may involve much overhead in the implementation level and is arguably not practical at all. Hoare's $O(n)$ expected time solution, $FIND$ algorithm [49] is reputedly faster in practice.

# Part I

# $K$-Maximum Subarray Problem

## Foreword

When one knows an algorithm that finds an extreme value, such as a maximum or a minimum, it often opens a new challenge to generalize the algorithm such that it can compute the $K$-th best solution. The selection problem given in Section 2.3, addresses this issue.

Naturally, one may be tempted to find the $K$-maximum subarrays, where we find not only the first, but also the second, third,.., $K$-th. However, when we have computed the first maximum subarray, what defines the second maximum involves a little delicacy. To elaborate this issue, let us separate two properties, *sum* and *location*, of each subarray.

Let us consider the problem in 1D and let $M_1$ and $M_2$ be the first and second maximum sum respectively. Certainly, it should be ensured that $M_1 \geq M_2$.

The delicacy arises when we define the location of the second maximum subarray $(p_2, q_2)$. Should it be totally exclusive of the first subarray $(p_1, q_1)$, or is it allowed for the second to overlap the first?

The former option is more intuitive. We can exclude elements in the first maximum subarray and compute the second one from the remaining array. It follows that the second maximum subarray is totally *disjoint* from the first maximum. For general $K$, we wish to compute $K$ maximum subarrays that are disjoint from one another, hence we call this version of problem, the *$K$-disjoint maximum subarray problem($K$-DMSP).*

The latter option imposes less restriction on the location of the second maximum subarray. While the former requires $(p_1, q_1)$ and $(p_2, q_2)$ to be mutually exclusive, such that $q_1 < p_2$ or $q_2 < p_1$, the latter option only requires these subarray to be non-identical. This loose requirement potentially results in the new maximum subarray overlapping the previous ones. As overlapping solutions are acceptable, we shall refer to this version of problem, the *$K$-overlapping maximum subarray problem($K$-OMSP).*

The $K$-OMSP has never been discussed in the literature prior to the commencement of this research, while Ruzzo and Tompa [82] studied a problem similar to the $K$-DMSP.

Studies on these two problems and development of algorithms constitute major part of this research, which we will describe in Chapters 3 and 4.

# Chapter 3

# $K$-Overlapping Maximum Subarray Problem

## 3.1 Introduction

The new problem of $K$-overlapping maximum subarrays, or the *$K$-OMSP* was first presented in our preliminary paper in 2004 [7]. Bengtsson and Chen also studied the problem independently around the same time [15]. There was a slight difference in the problem definition, such that [7] produces a list of $K$ maximum subarrays in sorted order, while no particular order is assumed in [15].

Since then, a rich collection of publications addressing the problem has been accumulated [7, 15, 8, 10, 16, 26, 11][*], and the time complexity of the problem has been increasingly improved.

In this chapter, we formalize the problem definition and present the algorithms for the problem.

## 3.2 Problem Definition

### 3.2.1 Problem Definition

For a given array $a[1..n]$ containing positive and negative real numbers and 0, the maximum subarray is the consecutive array elements of the greatest sum. Let $MAX(K, L)$ be the operation that selects the $K$ largest elements in a list $L$ in non-increasing order. The definition of $K$ overlapping maximum subarrays is given as follows.

[*] References are listed chronologically based on the publication date

**Definition 3.2.1.**

$$M = MAX(K, L), \text{ where } L = \left\{ \sum_{x=i}^{j} a[x] \mid 1 \le i \le j \le n \right\}$$

Here, the $k$-th maximum subarray is stored in $M[k]$. Note that the solution set $M$ is in sorted order. In [15], the sortedness was not required, but all other literatures on this problem unanimously assumed the order.

**Example 3.2.2.** $a$ ={3, 51, -41, -57, 52, 59, -11, 93, -55, -71, 21, 21 }. For this array of size 12, total of 78($= 12(12+1)/2$) subarrays exist. Among them, the first maximum subarray is 193, $a[5]+a[6]+a[7]+a[8]$ if the first element is indexed 1. We denote this by $193(5, 8)$. When overlapping is allowed, the second and third maximum subarrays are $149(1, 8)$ and $146(2, 8)$. The 78-th maximum subarray, (or the minimum subarray) is $-126(9, 10)$.

## 3.3 $O(Kn)$ Time Algorithm

Based on Algorithm 3, let us proceed to discuss the $K$-maximum subarray problem, again for the one-dimensional case. We make it mandatory to have the solution in sorted order.

The simplest method may be producing all $n(n+1)/2$ subarrays and performing Algorithm 14 to find all $K$ maxima of them. As the result needs to be sorted, we perform a sorting on the final $K$ maxima. The total time for this method is $O(n^2 + K \log K)$. Theoretically $K$ may be as large as $n(n+1)/2$, but it is unlikely that any size greater than $n$ is needed in practice. We first introduce an algorithm for $K \le n$ and modify it for the general case.

While we had a single variable that book-keeps the minimum prefix sum in Algorithm 3, we maintain a list of $K$ minimum prefix sums, sorted in non-decreasing order. Let $min_i$ be the list of $K$ minimum prefix sums for $a[1..i-1]$ given by $\{min_i[1] \ldots, min_i[K]\}$, sorted in non-decreasing order. The initial value for $min_i$, that is $min_1$, is given by $min = \{0, \infty \ldots, \infty\}$.

We also maintain the list of candidates produced from $sum[i]$ by subtracting each element of $min_i$. The resulting list $cand_i = \{sum[i] - min_i[1], sum[i] - min_i[2] \ldots, sum[i] - min_i[K]\}$ is sorted in non-increasing order.

**Algorithm 15** $K$ maximum sums in a one-dimensional array for $1 \le K \le n$

```
1: for $k \leftarrow 1$ to $K$ do  $min[k] \leftarrow \infty$, $M[k] \leftarrow -\infty$
2: $sum[0] \leftarrow 0$, $min[1] \leftarrow 0$, $M[1] \leftarrow 0$
3: for $i \leftarrow 1$ to $n$ do
4:     $sum[i] \leftarrow sum[i-1] + a[i]$
5:     for $k \leftarrow 1$ to $K$ do  $cand[k] \leftarrow sum[i] - min[k]$
6:     $//extract(K, L)$ by Algorithm 14
7:     $M \leftarrow extract(K, merge(M, cand))$
8:     insert $sum[i]$ into $min$
9: end for
```

Let $M_i$ be the list of $K$ maximum sums for $a[1..i]$. This list is maintained in $M$ at the end of the $i$-th iteration in Algorithm 15 sorted in non-increasing order. When the algorithm ends, $M$ contains the final solution $M_n$. The merged list of two sorted sequences $L_1$ and $L_2$ are denoted by $merge(L_1, L_2)$. Note that result of merge is a sorted list. We have the following lemma.

**Lemma 3.3.1.**

$$M_{i+1} = extract(K, merge(M_i, cand_{i+1}))$$

In Algorithm 15, the list $min$ at the beginning of the $i$-th iteration represents $min_i$.

Each time a prefix sum is computed, we subtract these $K$ minima from this prefix sum, and prepare a list $cand$ of candidate $K$ maximum values. These $K$ values are merged with the current maximum sums stored in $M$, from which we choose the $K$ largest values.

After this, we insert the prefix sum to the list of $K$ minimum prefix sums for the next iteration. When a new entry is inserted, the list of $K$ minimum prefix sums has $K+1$ items. By discarding the largest one, we keep the size of this list to be fixed at $K$. Of course, if this sum is found to be greater than all current $K$ minima, no insertion is made.

We initialize the list of tentative solutions by $M = \{0, -\infty \ldots, -\infty\}$.

The line 8 in the algorithm preserves the loop-invariant from step $i$ to step $i+1$ as stated in Lemma 3.3.1. At the end, $M$ is the solution, given in the sorted order.

At each iteration, it takes $O(K)$ time for generating the candidate list, and $O(K)$ time for merging this list and the list of current maximum sums. The time for inserting a prefix sum into the list of minimum prefix sums (line 8) depends on what data structure is used.

If it is a simple array or list, the insertion takes $O(K)$ time, which establishes $O(K)$ overall time for each iteration. Using an advanced data structure makes little significance at this point due to lines 5-7 where we anyway need to spend $O(K)$ time generating the candidate list and updating the solution at each iteration.

As we need to perform $n$ iterations, the total time complexity is $O(Kn)$. When $K = 1$, this result is comparable to $O(n)$ time of Kadane's algorithm and Algorithm 3.

Note that Algorithm 15 is specifically designed for $K \leq n$. When $K > n$, this algorithm still works, but not efficiently. Considering that there are only $i$ prefix sums preceding to $sum[i]$ (if $sum[0] = 0$ is counted), maintaining $min$ of size $K > n$ is meaningless and introduces inefficiency. Note that when $K = n(n + 1)/2$, Algorithm 15 runs in $O(n^3)$ time. Even the simplest method described in the beginning of this section, does not exceed $O(n^2 \log n)$ time.

We can slightly modify Algorithm 15 to handle the general case better. Specifically, the following modification no more relies on Lemma 3.3.1. In Algorithm 16, we declare an empty set $C$, and append each candidate to $C$. Finally, we select $K$ largest candidates from $C$ by Algorithm 14 and sort them.

The total time is $O(n * \min(K, n) + K \log K)$, where the second term is due to sorting. For $K \leq n$, this time is $O(Kn)$ as $O(K \log K) < O(Kn)$ and is absorbed. The complexity is comparable to Algorithm 15. In the extreme, where $K = n(n + 1)/2$, the total time becomes $O(n^2 \log n)$ due to the dominance of the second term. The space complexity of this algorithm is $O(n * \min(K, n))$ due to the size of $C$. In terms of space, this algorithm is not as efficient as the previous one when $K \leq n$, since Algorithm 15 only needs $O(n)$ space due to $a[1..n]$ and $sum[0..n]$. The space consumed by $cand$, $min$ and $M$ are all bounded by $O(K)$.

While further refinement to this algorithm is possible, we focus on im-

**Algorithm 16** $K$ maximum sums in a one-dimensional array for $1 \leq K \leq n(n+1)/2$

```
1: $C \leftarrow \emptyset$
2: for $k \leftarrow 1$ to MIN $\{K, n\}$ do
3:     $min[k] \leftarrow \infty$
4: end for
5: $sum[0] \leftarrow 0$, $min[1] \leftarrow 0$, $M[1] \leftarrow 0$
6: for $i \leftarrow 1$ to $n$ do
7:     $sum[i] \leftarrow sum[i-1] + a[i]$
8:     for $k \leftarrow 1$ to MIN $\{K, i\}$ do  append $sum[i] - min[k]$ to $C$
9:     insert $sum[i]$ into $min$
10: end for
11: $M \leftarrow extract(K, C)$
12: sort $M$
```

proving Algorithm 15 in this chapter. When $K \leq n$, we can apply a simple sampling technique to reduce the number of candidates. In Section 3.4 and Section 3.5, we assume $K \leq n$ and give improved algorithms based on the sampling technique. In Section 3.5.4, we show how such a technique can be used for $n < K \leq n(n+1)/2$.

While these new algorithms made milestones in the history of the 1D $K$-OMSP, readers may skip to Section 3.6 for a simple, yet theoretically optimal algorithm, which is based on a newly developed algorithm for the $X + Y$ problem.

## 3.4 $O(n \log K + K^2)$ time Algorithm

Previously, we generated the list of candidates by subtracting the $K$ minimum prefix sums from each prefix sum, which results in production of $Kn$ candidates in total. $K$ maximum sums are basically selected from this pool of $Kn$ candidates. Let $A$ be the name of the array keeping such $Kn$ candidates. In this section, we discuss possible improvements to Algorithm 15. We show how to reduce the number of candidates before selecting $K$ final elements. This is achieved by avoiding the actual computation of the entire array $A$. Thus $A$ is an imaginary array.

We describe a simple solution that decreases the number of candidates from $Kn$ to $K^2$. Note that $K^2$ is considered to be smaller than $Kn$ due to the assumption $K \le n$. This solution is introduced in the preliminary paper [8] and provides a starting point for the further improved algorithm in Section 3.5.

Intuitively we may consider the total of $Kn$ candidates, $cand_i[1..K]$, $(i = 1\ldots, n)$ as elements of an imaginary two-dimensional array $A$, such that the first column of $A$ is given as $cand_1[1..K]$, and the second column is given as $cand_2[1..K]$ etc.

Since each array element is obtained by computation $cand_i[k] = sum[i] - min_i[k]$ for $k = 1..K$ and $i = 1..n$, we can formulate the following.

$$A[k][i] = cand_i[k] = sum[i] - min_i[k]$$

As $min_i$ is sorted in non-decreasing order, the produced list of candidates $cand_i$, the $i$-th column of array $A$, is sorted in non-increasing order. The first item $cand_i[1](=A[1][i])$ is the largest candidate produced from $sum[i]$.

We first produce $n$ samples of $cand_1[1]...cand_n[1]$ and let them be elements of a list $sample$.

$$sample = \{A[1][1], A[1][2]\ldots, A[1][n]\}$$

We then select the $K$-th largest value $KthSample$ by a linear time selection algorithm [20]. It is easily observed that if $sample[i]$, the largest element in the $i$-th column, is smaller than $KthSample$, no elements in the same column can become one of the final $K$ maximum sums as we already know there are at least $K$ elements not smaller than them. This is illustrated in Figure 3.1 which shows a case for $K = 8$. Elements with (O) label are greater than or equal to $KthSample$ while light shaded elements in the first row are those not included in the $K$ largest samples.

At each iteration, we check if $sample[i]$, the first element in the $i$-th column, is not smaller than $KthSample$. If so, we generate all elements in the $i$-th column. Otherwise, this column needs not produce any element. We save $O(K)$ time by skipping candidate generation in such columns. Elements under $KthSample$ are also discarded as they all are not greater than $KthSample$.

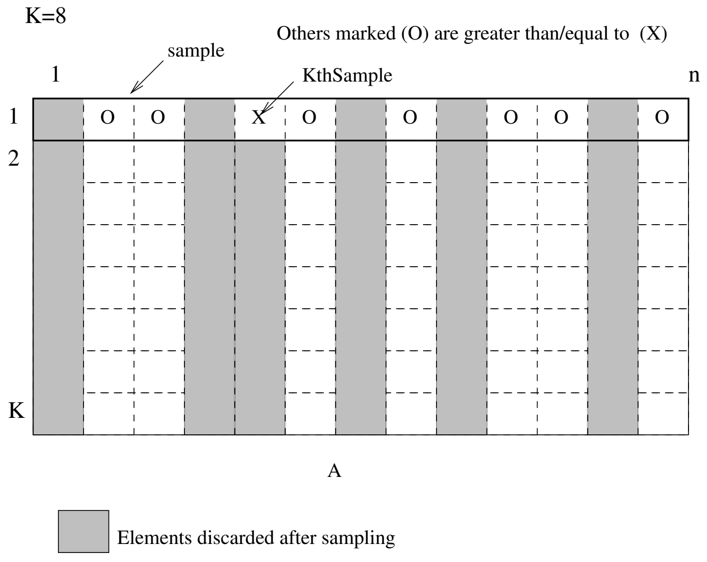

K=8
Others marked (O) are greater than/equal to (X)
sample
KthSample
1
n
1
2
K
O
X
A
Elements discarded after sampling

Figure 3.1: Selection of $K$ samples

Such an idea is implemented as Algorithm 17. We describe the details of this algorithm. At the end of the $i$-th iteration, $M$ and $min$ represent $M_i$ and $min_i$. $min$ is described as an array for description purposes, but actually organized as a tree as shown later.

### 3.4.1 Pre-process: Sampling

During the pre-process, we sequentially visit the input array $a[1..n]$ and compute the prefix sum $sum[1..n]$ in $O(n)$ time. Within this time frame, we find the minimum prefix sum ($min_i[1]$ only) for each $sum[i]$, as $min_i[1]$ is the minimum of $sum[j]$ for $1 \leq j \leq i-1$. We note that we do not need $min_i[1..K]$ for all $i \in [1..n]$ before the sampling and selection process. We only need $min_i[1]$ for $i = 1 \ldots n$. Full lists of $K$ minimum prefix sums for each $sum[i]$ are not produced during this pre-process.

The $K$-th maximum of this sample, $KthSample$, is selected by a linear time selection algorithm. Then we filter out values smaller than $KthSample$, being left with the $K$ largest samples shown as elements with (O) label in Figure 3.1. If there are multiple samples of the same value as $KthSample$, we may have more than $K$ remaining samples after filtering. As no more

**Algorithm 17** Faster algorithm for K maximum sums in a one-dimensional array

```
1: //Initialization
2: for $k \leftarrow 1$ to $K$ do  $min[k] \leftarrow \infty$, $M[k] \leftarrow -\infty$
3: $sum[0] \leftarrow 0$, $min[1] \leftarrow 0$, $M[1] \leftarrow 0$
4: //Pre-process: Sampling
5: for $i \leftarrow 1$ to $n$ do
6:    $sum[i] \leftarrow sum[i-1] + a[i]$
7:    //$sample$ for initial $K$ large values
8:    $sample[i] \leftarrow sum[i] - min[1]$
9:    if $sum[i] < min[1]$ then  $min[1] \leftarrow sum[i]$
10: end for
11: $KthSample \leftarrow$ K-th max of $sample[1..n]$
12: //Candidate Generation
13: $min[1] \leftarrow 0$
14: for $i \leftarrow 1$ to $n$ do
15:    if $sum[i] - min[1] \geq KthSample$ then
16:       //Part I
17:       for $k \leftarrow 1$ to $K$ do  $cand[k] \leftarrow sum[i] - min[k]$
18:       $M \leftarrow extract(K, merge(M, cand))$
19:    end if
20:    //Part II
21:    insert $sum[i]$ into $min$
22: end for
```

than $K$ samples are necessary, we regard these extra samples to be smaller than *KthSample*, and discard them. This is not explicitly given in the code.

### 3.4.2 Candidate Generation and Selection

Inside the outer 'for' loop, there are two parts, Part I and Part II. We consider time for each part separately.

Part I is for the generation of $cand_i$ and maintaining the tentative solution set $M$. The generation of $cand_i$, the elements in the $i$-th column of array $A$, is performed when the first element in the $i$-th column, $sample[i]$, is greater than *KthSample*. Thus Part I is performed $K-1$ times. To be precise, we can skip the generation of the elements in the column of *KthSample* as shown in Figure 3.1, but this does not improve the overall asymptotic complexity.

Now we analyze each part.

### Part I

For Part I, generating a candidate list(=a column of $A$), involves access to $min$, the list of minimum prefix sums. If a 2-3 tree is used, accessing each of $min[1]..min[K]$ costs $O(\log K)$ time. We need to access all $min[1]..min[K]$ sequentially to generate all elements in one column. The sequential reading of all leaf nodes is done in $O(K)$ time by depth-first search. The later part of this chapter, Section 3.5.2, also discusses this complexity.

The initial $O(\log K)$ search time is absorbed into $O(K)$, the time for actual generation of the $K$ candidates. The total time for Part I over $K$ iterations is therefore $O(K^2)$.

### Part II

For Part II, finding position for a new entry and actual insertion is done in $O(\log K)$ time. When there are more than $K$ items, deletion of the largest item and update of the tree costs another $O(\log K)$ time. For $n$ iterations, the total time for Part II is $O(n \log K)$.

## 3.4.3 Total Time

Using the data structure for $min$ described above, the overall time including Part I and Part II is thus $O(n \log K + K^2)$.

Let us consider the time for the initialization and the pre-process.

During the initialization, the 'for' loop sequentially sets $min[1..K]$ and $M[1..K]$ to $\infty$ and $-\infty$ respectively.

Sequential access to the leaf nodes of a 2-3 tree is done in linear worst case time as discussed in later part of this chapter, Section 3.5.2. Both $min[1..K]$ and $M[1..K]$ are set in $O(K)$ time.

The pre-process (sampling, selection and screening) is $O(n)$ time, when *KthSample* is selected by a linear time selection algorithm [20].

Times for the initialization and pre-process are absorbed into the time for Part I and Part II, making the total time $O(n \log K + K^2)$. Compared with

$O(min\{K + n\log^2 n, n\sqrt{K}\})$ time by [15], this algorithm is faster when $K \leq \sqrt{n}\log n$ and the complexity $O(n \log K + K^2)$ is even reduced to $O(n \log K)$ for smaller $K$ ($K \leq \sqrt{n \log n}$).

## 3.5 $O(n \log K)$ Time Algorithm

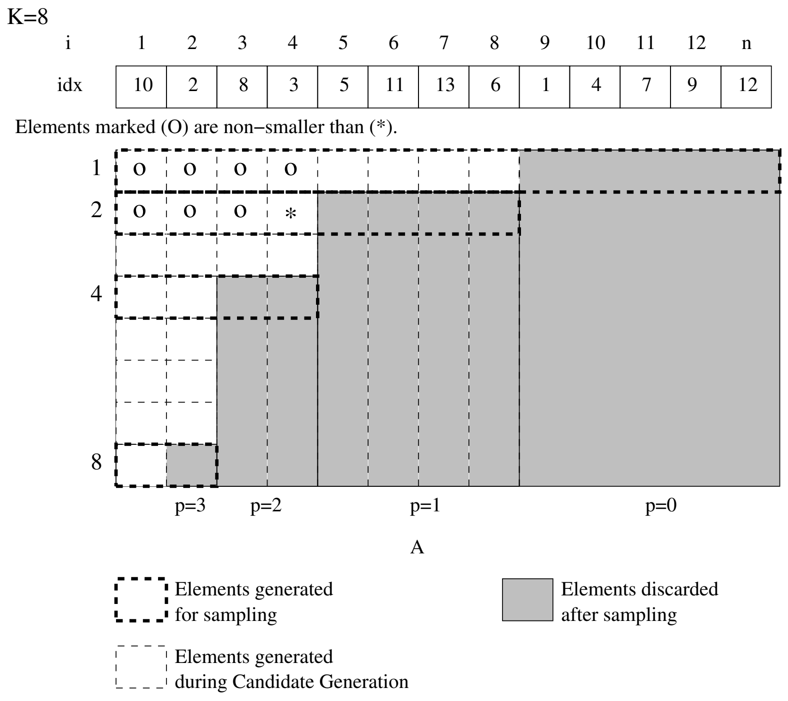
The algorithm in Section 3.4 regards a list of candidates as a column of an imaginary array $A$ of size $(K, n)$. This array has columns sorted in non-increasing order, having the first element in each column the largest.

This sortedness enabled the algorithm in Section 3.4 to discard unnecessary elements after obtaining *KthSample*. In this section, we try to extend the same idea for further improvement. In the following, the process of sampling followed by selection is simply referred to as *sampling technique*.

Frederickson and Johnson [38] present an efficient selection algorithm to find the $k$-th smallest element in an $n \times m$ array with sorted columns in $O(m + p\log(k/p))$ time for $p = min\{k, m\}$. This algorithm rapidly discards unnecessary items that are doomed to be larger than the final $k$-th smallest. Certainly, the same idea may be configured to find the $k$-th largest element. This algorithm is composed of two routines, where the first routine eliminates unnecessary items until $O(k \log k)$ items left, the second routine further reduces this to $O(k)$ remaining items. Then the $k$-th smallest can be selected directly by a linear time selection algorithm. The first routine of this solution is basically a generalized notion of the sampling technique. While the previous algorithm performs the sampling technique in the first row only, we can extend the same idea to multiple rows.

Namely, when the $K$-th largest element in the first row is selected, we rearrange the columns such that those having the first element greater than the selected value are located on the left of the selected value. Since all columns that appear on the right of the selected value have elements smaller than this $K$-th largest value, we may safely discard these columns. The area containing discarded elements is shown shaded in Figure 3.2 with 'p=0' label, meaning that this area is removed during the first iteration.

We further this idea and select the $K/2$-th largest element in the second row and rearrange the remaining $K$ columns such that columns whose second



K=8
i
1
2
3
4
5
6
7
8
9
10
11
12
n
idx
10
2
8
3
5
11
13
6
1
4
7
9
12
Elements marked (O) are non−smaller than (*).
1
2
4
8
O
*
p=3
p=2
p=1
p=0
A
Elements generated for sampling
Elements discarded after sampling
Elements generated during Candidate Generation

Figure 3.2: Sampling in row 1,2,4,8...

element is greater than the selected value are located on the left of the selected one. Then we discard half of the remaining $K$ columns that contain smaller elements. In Figure 3.2, all elements denoted by (O) are greater than or equal to the element denoted by (*). The number of these elements including (*) is $K$. No element in the shaded area with 'p=1' label can be greater than these $K$ elements. Then none can be included in the final set of $K$ maximum sums. So this shaded area is safely discarded. We continue this sampling process by doubling the row number at each iteration. On termination of this process, the number of remaining elements is significantly smaller than in the previous solution shown in Figure 3.1.

Before applying Frederickson and Johnson's solution to our problem, let us identify some difficulties.

First, their solution is applicable when such an array is already available before selection. If we have to build the array beforehand, the array construction alone already takes $O(Kn)$ time. Even a fast selection algorithm can not help. Alternatively, we may simultaneously construct the necessary portion of the array and perform the selection algorithm.

When we wish to construct the array and process the selection algorithm at the same time, we encounter another problem, which is caused by the fact that $min$ is *ephemeral*, in the sense that making a change to it destroys the old version. To clarify this situation, let us review the selection process of the $K/2$-th largest element in the second row. In the first iteration, we have sampled $n$ elements and selected $K$ largest elements in the first row. Let the selected ones be $A[1][x_1]...A[1][x_K]$ where $A[1][x_K]$ is the $K$-th largest element. We come to the second row for the second iteration. As the array is not built, there are no elements available in the second row. Before selecting the $K/2$-th largest element, we need to sample $K$ elements in this row. Each sample of $A[2][x_1]...A[2][x_K]$ is computed by coupling $sum[x_k]$ and $min_{x_k}[2]$. For $k = 1..K$,

$$A[2][x_k] = cand_{x_k}[2] = sum[x_k] - min_{x_k}[2]$$

In Algorithm 15(line 5) and Algorithm 17(line 17), $cand_i$ is produced from $min_i$ that is maintained in a single data structure $min$. At the $i$-th iteration of both algorithms, after a new prefix sum $sum[i]$ is inserted to the current $min_i$, the next version $min_{i+1}$ is created. We lose access to all previous versions $min_1..min_i$.

When the previous versions of $min$ are lost, it is impossible to produce elements in the second row by computing $sum[x_k] - min_{x_k}[2]$ $(k = 1..K)$. We therefore need a *persistent* data structure [30] to overcome this problem.

In the following, we describe Algorithm 18 that applies Frederickson and Johnson's selection algorithm to the $K$-maximum subarray problem. Specifically, we show that the array construction routine can be combined with the selection algorithm. To overcome the deficiency caused by ephemeral data structure for $min$, we use a partially persistent 2-3 tree for maintenance of the $n$-versions of sorted set $min_i[1..K], (i = 1..n)$ without spending $O(Kn)$ time and space. The detail of this data structure is discussed in the Section 3.5.2. Note that we use control variables $i$ and $k$ for row-wise and column-wise operations respectively in the following algorithm and its description.

**Algorithm 18** Algorithm for K maximum sums in a one-dimensional array with generalized sampling technique

```
1: //Initialization
2: for $k \leftarrow 1$ to $K$ do  $min_0[k] \leftarrow \infty$
3: for $i \leftarrow 1$ to $n$ do  $u[i] \leftarrow K$
4: $sum[0] \leftarrow 0$
5: //Pre-process
6: for $i \leftarrow 1$ to $n$ do
7:     insert $sum[i-1]$ into $min_{i-1}$, which creates $min_i$
8:     delete $min_i[K+1]$ //deletes from $min_i$ to keep size K
9:     $sum[i] \leftarrow sum[i-1] + a[i]$
10: end for
11: //Sampling/re-indexing
12: $q \leftarrow n$, $q' \leftarrow K$, $p \leftarrow 0$, $idx \leftarrow [1, 2, ...n]$
13: while $2^p \leq K$ do
14:     //Compute $A[2^p][1..q]$, contained in $A$
15:     for $i \leftarrow 1$ to $q$ do  $A[i] \leftarrow sum[idx[i]] - min_{idx[i]}[2^p]$
16:     $l \leftarrow q'$-th max of $A[2^p][1..q]$
17:     Partition $A$ into $(A1, A2, A3)$, where
        $A1 = \{x|x \in A, x > l\}$, $A2 = \{x|x \in A, x = l\}$, $A3 = \{x|x \in A, x < l\}$
18:     Copy prefix sum indices of elements in $(A1, A2, A3)$ to $idx[1..q]$
19:     for $i \leftarrow 1$ to $q$ do  $u[i] \leftarrow 2^p - 1$
20:     $q \leftarrow q'$, $p \leftarrow p + 1$, $q' \leftarrow \lceil K/2^p \rceil$
21: end while
22: //Candidate Generation
23: $C \leftarrow \emptyset$
24: for $i \leftarrow 1$ to $K$ do
25:     //$u[i]$: number of candidates to generate in col. $i$ of $A$
26:     for $k \leftarrow 1$ to $u[i]$ do  append $sum[idx[i]] - min_{idx[i]}[k]$ to $C$
27: end for
28: //Final Selection of $K$ maxima
29: $M \leftarrow extract(K, C)$
30: sort $M$
```

### 3.5.1 Algorithm Description

Algorithm 18 is composed of five major routines, namely, initialization, pre-process, sampling/re-indexing, candidate generation and final selection of $K$ maximum sums. We describe details of each routine.

### Initialization

We create the initial version of the minimum prefix sum, $min_0$, maintained in a partially persistent 2-3 tree. The array $u$ is prepared to indicate the number of candidates to be produced in each column, which will be used in the 'candidate generation' routine. Initially, each column is entitled to produce $K$ candidates.

### Pre-process

Over $n$ iterations, we compute the prefix sum and insert this prefix sum into $min$. We have $n$-versions of sorted sets $min_0[1..K]\ldots, min_n[1..K]$ maintained in a partially persistent 2-3 tree. We need to fix the number of leaf node to be $K$. Line 7 inserts a prefix sum $sum[i-1]$ to the $(i-1)$-th version of $min$ kept in persistent 2-3 tree, which creates $min_i$, the $i$-th version of $min$. We have $K+1$ leaf nodes in $min_i$. A deletion of the last leaf node from $min_i$ is thus needed as shown by line 8. Details of update operations to the persistent 2-3 tree is discussed in Section 3.5.2.

### Sampling/re-indexing

In our problem setting, we start with an empty array $A$ whose dimension is $K \times n$. During the routine shown by lines 12-21, we examine rows 1,2,4,8... only and generate a limited number of array elements in each row for *sampling*. Otherwise it may cost $O(Kn)$ time to generate all array elements. We use an auxiliary array $idx[1..n]$ to ease the column re-indexing. The initial setting to $idx$ is {1,2,...n}. Let us call the index $i$ of $sum[i]$ a *prefix sum index*. The value of $idx[i]$ indicates which prefix sum $sum[idx[i]]$ we use to produce the array elements in column $i$.

With $p$ being incremented by 1 at each iteration, we visit row 1,2,4,8..(=$2^p$) sequentially where we generate only $q$=$n$, $K$, $K/2$, $K/4$.. samples respectively. Such samples are shown by thick dotted lines in Figure 3.2. Line 15 shows that $q$ samples, $A[2^p][1..q]$, are computed by $sum[idx[i]] - min_{idx[i]}[1]$ for $i = 1..q$. This involves the access to different versions of $min$. The persistent data structure for $min$ enables this.

Due to the initial setting to $idx[1..n]$, all $sum[i] - min_i[1]$ for $i = 1..n$ are computed in row 1. The following line 16 performs a linear selection algorithm to find the $q'(= \lceil K/2^p \rceil)$-th largest one. For example, in row 1,2,4.., it is the $K$, $\lceil K/2 \rceil$ , $\lceil K/4 \rceil$-th largest respectively. This item is marked $l$. We rearrange the elements in this row and partition into $(A1, A2, A3)$ in a similar way to Algorithm 14 such that all items greater than $l$ are moved to the left partition $A1$, equal to $l$ to $A2$ and smaller to $A3$. Let $(A1, A2, A3)$=$\{sum[x_1] - min_{x_1}[1], sum[x_2] - min_{x_2}[1]...sum[x_q] - min_{x_q}[1]\}$. We copy the prefix sum indices $\{x_1, x_2..x_q\}$ to $idx[1..q]$.

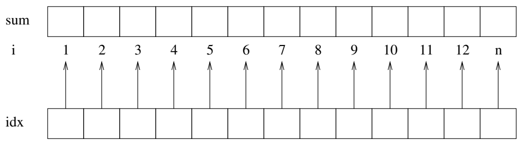
This rearrangement of $idx[1..q]$ achieves the effect of *column re-indexing* such that the prefix sum indices that produce array elements non-smaller than $l$ are stored in $idx[1..q']$ and rest indices are in $idx[q' + 1..q]$. During the 'candidate generation' routine, we use the prefix sum indices maintained in $idx$. The virtual array $A$ may be illustrated as Figure 3.2 having elements of large value concentrated around the top-left corner.

A snapshot of the update to $idx$ at each iteration is given in Figure 3.3. Note that $idx[1..K]$ at $p = 1$ correspond to the column indices marked (O) and (X) in Figure 3.1.

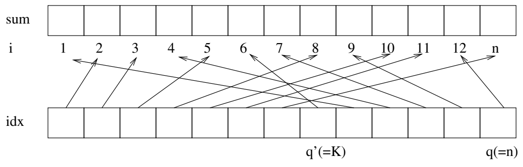
The value $l$ is the $q'(= \lceil K/2^p \rceil)$-th largest in this row. At the same time, it is the $2^p$-th largest in its column since the column is sorted in non-increasing order. Then we are assured that there are at least $K$ elements non-smaller than $l$. Apart from the first $q'$ samples in $A1$ and $A2$, rest samples are now disqualified. When they do not qualify the $K$ largest at this stage, they are never included in the final set of $K$ maxima. The prefix sum indices of such disqualified samples are kept in $idx[q' + 1..q]$ after rearrangement of $idx$. In the 'candidate generation' routine, we will not generate candidates with the prefix sum of such an index in the $2^p$-th row and below. This virtually discards unnecessary elements from the array, or, to be precise, *aborts* them from being produced.

The shaded areas labeled 'p=0,1,2,3' in Figure 3.2 represent such aborted portions. Note that the first iteration of the 'while' loop at lines 13-21 is essentially equivalent to the sampling process used in Algorithm 17.

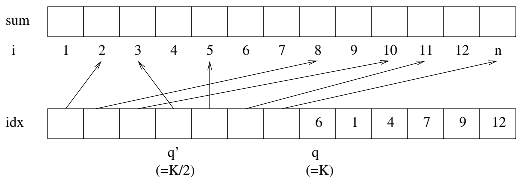
Even if the array $A$ is not pre-built, this 'abortion' technique effectively simulates elimination from the pre-built array, the basic idea of the selection



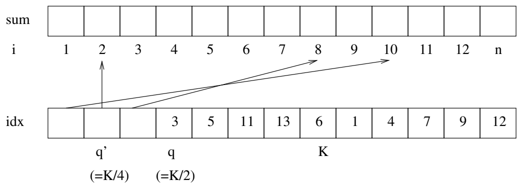
(a) Initialization



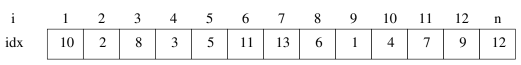
(b) p=0.



(c) p=1.



(d) p=2.



| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| idx | 10 | 2 | 8 | 3 | 5 | 11 | 13 | 6 | 1 | 4 | 7 | 9 | 12 |

(e) Final values when p=3.

Figure 3.3: Rearrangement of index array *idx* when $n = 13$, $K = 8$

algorithm by Frederickson and Johnson.

Actual generation of non-aborted candidates is done in the next subroutine starting at line 23. We update the array $u$ at line 19 to indicate how many candidates can be produced in each column of $A$. Due to the initialization at line 3, each column is entitled to produce $K$ candidates. At the $2^p$-th row, once the $q'$-th largest element $l$ is found and the column re-indexing is done, the columns of disqualified samples are not allowed to produce more than $(2^p - 1)$ candidates. When the 'while' loop at lines 13-21 terminates, we have $u[1..K] = \{8, 7, 3, 3, 1, 1, 1, 1\}$ for the example given in Figure 3.2. Further discussion is given in the next section.

Note that the update to $q$ at line 20 may be replaced with $q \leftarrow \lceil K/2^p \rceil - 1$ for further reducing the size of sample generation, which however makes no asymptotic improvement.

### Candidate generation

We now produce all the elements that survived the sampling process. The final version of $idx$ as shown in Figure 3.3 indicates the prefix sum index to be used for generation of elements. Specifically, the first column of $A$ is built with the prefix sum index $idx[1]$.

While the sampling process in Section 3.5.1 was performed in a row-wise manner, we choose to generate candidates column-wise. By column-wise computation, we visit one version of 2-3 tree and access each leaf node sequentially. Later in Section 3.5.2, it will be shown that each candidate computed in such a manner costs $O(1)$ amortized time. We can not afford the complexity incurred by row-wise computation here, since it involves an element retrieval with index from each version of 2-3 tree. Section 3.5.2 will show that each candidate computed this way needs $O(\log K)$ time.

With the array $u$ that indicates the number of candidates to produce in each column, column-wise computation is easily done by $sum[idx[i]] - min_{idx[i]}[1..u[i]]$ in column $i$. Those generated are shown in white in Figure 3.2.

We start with an empty set $C$, and append each generated element to a set $C$ at lines 23-27. There is no specific order in $C$ at the stage.

Let us determine the total number of generated elements, $|C|$. Counting them in row-wise manner is easier. We have $K$ elements in the first row, and $\lceil K/2 \rceil$ elements each in the second and the third row. In general, there are $q'(=\lceil K/2^p \rceil)$ candidates each in rows $2^p..(2^{p+1}-1)$. Note that $(2^{p+1}-2^p)\cdot q' = O(K)$. We can obtain $|C|$ by $K + 2(K/2) + 4(K/4) + ...=O(K \log K)$.

While [38] introduces further reduction techniques to reduce this number to $O(K)$, having $O(K \log K)$ remaining elements still suffices our needs. Further discussion is given in Section 3.5.3.

#### Final selection of the $K$ maximum sums

Finally, lines 29-30 describe the selection of $K$ maximum elements in $C$. We sort such final $K$ elements and obtain the sorted list of $K$ maximum subarrays.

### 3.5.2 Persistent 2-3 Tree

The choice of an appropriate data structure for the collection of the minimum prefix sums $min_i$ is essential to the algorithm.

To maintain sorted set with efficient support for insert and delete operations, a 2-3 tree provides optimal performance. The 2-3 tree is a class of search trees invented by Hopcroft [1], where every internal node has either two or three children and all leaf nodes appear on the same level. This perfectly balanced property means $O(\log n)$ time for search, insert and delete operations, where $n$ is the number of elements in the tree. An internal node having two children is called a 2-node, and one with three children is called a 3-node. Each 2-node contains two keys and a 3-node has three keys, where each key has the same value of the first key of a child node. Some authors including [1, 63] prefer to have one key in a 2-node and two keys in a 3-node, and such implementation can be used instead with no significant difference[†].

The data structure that loses its old version is called *ephemeral*. If the data structure allows access to the old versions after subsequent update operations, it is called *persistent*. Since the seminal paper of Driscoll *et al.* [30], there has been considerable development of persistent data structures

[†] Having $x$ keys in a $x$-node is intuitively more transparent.

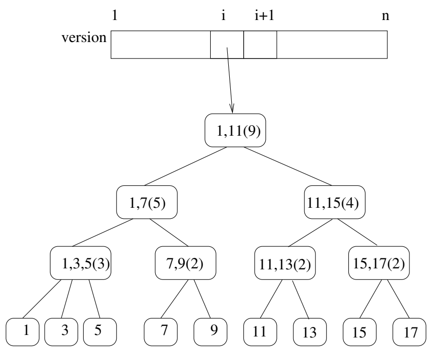


Figure 3.4: The i-th version of 2-3 tree. Leaf nodes represent sorted set in non-decreasing order. The number inside () shows the number of leaf nodes under this node. $K = 9$

[23, 14, 83, 60]. The *partially persistent* data structure allows all versions to be accessed, but only the newest version can be modified. The structure is *fully persistent* if every version can be both accessed and modified [30]. As we only modify the newest version, a partially persistent structure will be sufficient.

Combining two requirements, a partially persistent 2-3 tree is the structure of choice.

We adopt *node copying* method for making a 2-3 tree persistent. Figure 3.4 shows the $i$-th version of the 2-3 tree storing $K = 9$ elements *1,3,...17*, in non-decreasing order We have an array of size $n$, $version$, whose i-th item points to the root of the i-th version. Each internal node of the tree has an extra field storing the number of leaf nodes under the node for efficient access with an index. The details are given in Section 3.5.2.

### Update operation

When a new element *6* is inserted, we first perform a search on the i-th version for an appropriate position. We find that node *[7,9(2)]* will be the parent of this new entry, but this node will need to change its shape. We thus copy this node and add *6* to it, and change the cardinality of the copied node to 3. When this node is copied, the pointers to node *[7]* and *[9]* are also copied. Then a new copy has three children, *[6]*, *[7]* and *[9]*. We must

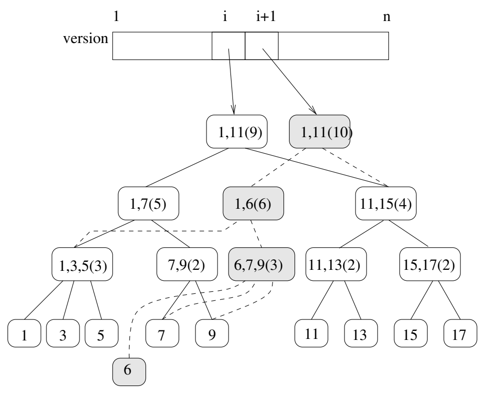


Figure 3.5: The (i+1)-th version is created when 6 is inserted.

copy and update all the nodes in the path to the root in the same manner. Newly created nodes are shaded in Gray in Figure 3.5. Finally the (i+1)-th item of *version* is arranged to point to the new copy of root node, the root of the (i+1)-th version.

After the insertion of *6*, there are 10(=$K$ + 1) leaf nodes. Since this structure is to be used for *min* which has fixed size $K$, we have to delete the leaf node with the largest value, *17*. We intend to delete the node *[17]*, but only from the (i+1)-th version. Previous versions should still have an access to the node *[17]*. Thus we only remove the pointer link to the node *[17]* from the (i+1)-th version of the tree. We first traverse from the root of the (i+1)-th version to the rightmost leaf node *[17]*. Since its parent *[15,17(2)]* will have only one child after losing *17*, we will delete this node from the (i+1)-th version too. Then the sibling node *[11,13(2)]* should adopt the orphan leaf node *[15]* as its rightmost child, which updates *[11,13(2)]* to *[11,13,15(3)]*. A similar operation is carried out for *[11,15(4)]*. As node *[1,6(6)]* has two children and is the only child of the root *[1,11(10)]*, we choose *[1,6(5)]* to become the new root of the (i+1)-th version of the tree. The node *[11,13,15(3)]* is taken as the rightmost child of *[1,6(5)]* and the root is updated to *[1,6,11(9)]*. The final shape of $min_{i+1}$ should look like Figure 3.6.

Once one version $min_i$ has $K$ elements, each insertion to $min_i$ means the next version $min_{i+1}$ has $(K + 1)$ elements. Thus each insertion should

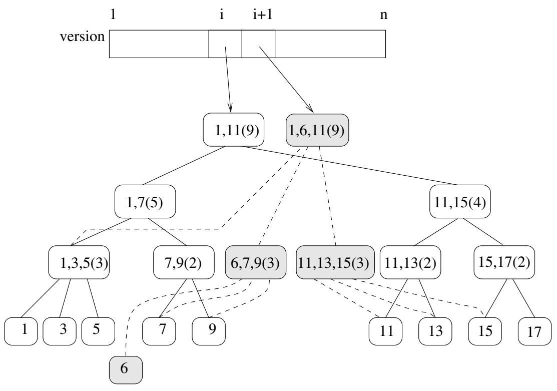


Figure 3.6: The largest item 17 is deleted from the (i+1)-th version to keep the size $K$.

be followed by a deletion of the largest element to keep the size of the next version $min_{i+1}$. As we described, each operation recursively copies nodes in the path to the root and updates them. As the height of the tree is bounded by $O(\log K)$ when we have the fixed number of leaf nodes $K$, we spend $O(\log K)$ time and $O(\log K)$ space for each insertion and deletion.

In the following, we examine the time complexity for two types of leaf node access, sequential reading of leaf nodes and random access to the $k$-th element.

### Sequential access to leaf nodes

We first examine the time for the following routine.

```
**for** $k \leftarrow 1$ to $K$ **do** print $min_i[k]$
```

If $min_i$ is contained in an array, the time is $O(K)$. Such time for $min_i$ maintained in a 2-3 tree deserves discussion.

First of all, we access the array *version* to find the root of the i-th version of the 2-3 tree that keeps $min_i$. Sequential access to all leaf node values can be done by simple depth-first search traversal. If $N$ is the number of internal nodes of a 2-3 tree that has $K$ leaf nodes, $(K-1)/2 \leq N \leq K-1$. The number of internal nodes is thus bounded by $O(K)$. Then sequential access to all $K$ leaf nodes is done in $O(K)$ time.

If a level-linked 2-3 tree [25] is used, it guarantees $O(1)$ worst case time for accessing the next item as well as $O(K)$ time for sequential reading of all items. The ordinary 2-3 tree that we described here only provides $O(1)$ amortized time for the next item access while $O(K)$ time for sequential reading is still supported.

#### Element retrieval with index from a 2-3 tree

Each internal node in a 2-3 tree maintains an attribute of the number of leaf nodes below this node. In the above, it was shown that all nodes in the path to the root update their cardinality on each insertion or deletion.

In the 2-3 tree described above, we assume the leftmost leaf has an index 1, and the rightmost leaf has an index $K$ accordingly.

When the $k$-th item needs to be retrieved, the cardinality can be utilized to find the location of this item. Suppose the root node has cardinality $c_P$ and the left, center and right child have $c_L$, $c_C$ and $c_R$ respectively such that $c_P = c_L + c_C + c_R$. To search for the $k$-th item, we first look at $c_P$ to make sure if $k \leq c_P$. If so, we try series of comparisons to find which subtree this item belongs to. If $c_L \geq k$, the $k$-th element is in the left subtree. Otherwise, we examine $c_C$ to see if $c_C \geq k - c_L$. If so, the $k$-th item is in the center subtree. Otherwise, it is in the right subtree. We perform at most 3 comparisons at each node recursively until we arrive at a leaf node following a path from the root. When there are $K$ leaf nodes in the tree, we spend $O(\log K)$ time to retrieve the $k$-th element.

### 3.5.3 Analysis

First we analyze the time for the initialization. Inside the loop, $K$ elements of $min_1$ are set sequentially. Due to Section 3.5.2, the time for initialization of $min_1[1..K]$ is $O(K)$.

The pre-process includes preparations of the prefix sums and the partially persistent 2-3 tree storing the minimum prefix sums. Each update to $min_i$ is done in $O(\log K)$ time as discussed in Section 3.5.2. The time for the pre-processing is thus $O(n \log K)$.

Let us examine the time complexity within the 'while' loop at lines 13-21. We separate the analysis when $p = 0$ and $p \geq 1$. In the latter case, we always halve the size of sample, for example, we eliminate $K/2$ samples from $K$ at $p = 1$. This is not the case when $p = 0$.

At the first iteration ($p = 0$), we sample $n$ elements $A[1][1]..A[1][n]$ at line 15. The first element of each $n$ versions of $min$ is retrieved spending $O(\log K)$ time each due to Section 3.5.2. The time spent by this line is thus $O(n \log K)$. Following routines (lines 16-18) are linear time operations on $n$ samples. These $O(n)$ times are absorbed. The time spent by line 19 is $O(n - K)$, which is absorbed too.

When $p \geq 1$, at the $p$-th iteration, line 15 generate $A[2^p][i]$ by $sum[idx[i]] - min_{idx[i]}[2^p]$ for $i = 1...q$, which are $q = \lceil K/2^{p-1} \rceil$ elements. When the loop is exited, the total number of samples generated is $K + K/2 + K/4... = O(K)$.

The generation of each sample involves access to a corresponding version of $min_{idx[1]}..min_{idx[q]}$, the persistent 2-3 tree. We first refer to the $idx[i]$-th version of $min$ and need to track down from the root to locate $min_{idx[i]}[2^p]$ taking $O(\log K)$ time each. Each iteration of 'while' loop at lines 13-21, $q$ is $K$, $\lceil K/2 \rceil$ ...etc. The time spent by line 15 throughout 'while' loop ($p \geq 1$) is then $O\left((K + K/2 + K/4...) \log K\right)$=$O\left(K \log K\right)$).

Lines 16-18 perform linear operations on $K$,$\lceil K/2 \rceil$,$\lceil K/4 \rceil$... elements at each iteration. The total time is then $O(K + K/2 + K/4 + ...) = O(K)$. Similarly, the time by line 19 is $O(K)$.

The combined time of two cases, $p = 0$ and $p \geq 1$ inside the 'while' loop gives the total time spent by the loop. It is $O(n \log K) + O(n) + O(K \log K) + O(K)$, which is summarized to $O(n \log K)$ for $K \leq n$. The operation by lines 15 is the dominant one inside the 'while' loop.

The 'for' loop starting at line 24 involves the generation of non-discarded elements in the array $A$. There are $O(K \log K)$ elements remaining as discussed in Section 3.5.1. Note that lines 24-27 involve sequential reading from the sorted set maintained by 2-3 tree. It is done in linear time as discussed in Section 3.5.2. Then total time for generating $O(K \log K)$ elements is bounded by $O(K \log K)$.

All the generated items are collected in $C$ with no specific order. We proceed to the line 29 where the $K$ largest items are selected. As there are $O(K \log K)$ elements in $C$, linear time selection algorithm spends $O(K \log K)$ time for this. The final $K$ maximum values are sorted in another $O(K \log K)$ time by line 30.

As it is assumed that $K \leq n$, the total time of this algorithm is therefore bounded by $O(n \log K)$.

**Theorem 3.5.1.** $\forall K \in [1..n]$*, the sorted list of* $K$ *maximum subarrays is computed in* $O\left(n \log K\right)$ *time*

We discuss the complexity for large $K$ in the next section, Section 3.5.4.

Note that Frederickson and Johnson's algorithm [38] offers a subsequent reduction technique that further discards elements leaving only $O(K)$ elements. We only applied their first technique which leaves $O(K \log K)$ elements. Even if their subsequent technique is applied, we will still hit $O(n \log K)$ time complexity.

As we copy paths of $O(\log K)$ length to create each version of 2-3 tree, the extra space occupied by $n$ versions of $min$ is $O(n \log K)$. It may also be noted that the 2-3 tree we described is not strictly partially persistent, as any version can be accessed for update. Since the partial persistence is adequate for the requirement, it remains to be seen if a strictly partially persistent 2-3 tree can provide better efficiency in terms of time and space.

During the pre-process at lines 6-10, the prefix sum $sum[i]$ is inserted to $min_i$ regardless of its value. If it becomes the largest after insertion, this new entry is immediately deleted by the next line. By doing so, we waste $O(\log K)$ time to get the identical tree. To avoid this, we can prepare a *last* attribute at each version of the 2-3 tree to keep the value of the rightmost leaf, the maximum item, in the tree. Before each insertion, we examine whether the value of new entry is greater than *last*. If so, we simply set a pointer to the root of the current version of 2-3 tree instead of performing insertion and deletion of the same item. Otherwise, this entry is successfully inserted and the rightmost leaf which is different from the inserted item will be deleted. Meanwhile, *last* is also updated. This gives average time improvement, but the worst-case behavior is not clear at present.

Likewise, we can prepare a $first$ attribute at each version of the 2-3 tree to maintain the value of the leftmost leaf, the minimum item. This may reduce the $O(n \log K)$ time for sampling to $O(n + K \log K)$. While this does not improve the total complexity, it leaves the pre-process being the only $O(n \log K)$ time operation. Any future improvement to the pre-process will therefore reduce the total complexity.

### 3.5.4 When $n < K \leq n(n+1)/2$

So far the description and analysis of algorithms were given with an assumption that $K \leq n$. In fact, neither Algorithm 17 nor Algorithm 18 will work for $K > n$ if no modification is made. We discuss how we can handle large $K$ in this section.

In the following, we mean $n < K \leq n(n+1)/2$ by *large* $K$ and $K \leq n$ by *small* $K$. *All* $K$ is then $1 \leq K \leq n(n+1)/2$.

For large $K$, we encounter a case where the sampling technique no more improves the complexity. For example, selection of the $K$-th largest among $n$ samples in Algorithm 17 is *invalid* as there are only $n$ $(n < K)$ elements. As previously defined in Section 2.3.2, we use a term 'valid' to describe the opposite case, a meaningful application of the sampling technique.

For Algorithm 17 to support large $K$, we make a simple modification by combining it with Algorithm 16.

Since the sampling technique in Algorithm 17 is invalid for $K > n$, we skip the pre-process and perform lines 6-12 of Algorithm 16. Such a change gives $O\left((K+n) \log \min(K,n) + \min(K,n)^2\right)$ time for all $K$. Note that $O(\log \min(K,n)) = O(\log K)$ and the complexity may be simplified accordingly. Naturally, this modified version of Algorithm 17 does not improve Algorithm 16 for large $K$.

Now we consider Algorithm 18. It is obvious that the sampling in the first row is invalid as it was in Algorithm 17. The sampling in the second row may, however, be valid depending on the value of $K$. Considering that we attempt to find the $\lceil K/2 \rceil$ largest samples in the second row, the sampling becomes valid if $K < 2n$. When a valid sampling is done in the second row, it is also valid in the 4-th and 8-th rows etc. With a small modification to the original framework, the algorithm can support large $K$ with asymptotic improvement to Algorithm 16.

For large $K$, notice that the size of each version of $min$ will be at most $n$, for the same reason explained in Section 3.3. The size of the imaginary array $A$ in Figure 3.2 is then $(n, n)$. So we first let $K' = \text{MIN}\{K, n\}$ in the beginning of Algorithm 18 and replace each appearance of $K$ at line 2,3,8,13 and 24 with $K'$.

As briefly mentioned, the sampling may be invalid for some rows near the top. We first determine $2^{p_0}$, the first row where a valid sampling can be performed. Intuitively, $2^{p_0}$ is the smallest power of 2 such that $K/2^{p_0} \le n$. Certainly, we have $p_0 = 0$ for small $K$, which justifies the first valid sampling performed in the first row(=$2^0$). In general, we can determine the value of $p_0$ for both small and large $K$ by $p_0 = \lceil \log \frac{K}{K'} \rceil$. We modify the algorithm such that the initialization of $p$ and $q'$ at line 12 is done by $p \leftarrow p_0$ and $q' \leftarrow \lceil K/2^{p_0} \rceil$. In the rows above the $2^{p_0}$-th one, we skip sampling. The 'while' loop starting at line 13 runs at most $O(\log K')$ iterations. The time for sampling /re-indexing after modification is still $O(n \log K')$. The analysis can be done in a similar way described in Section 3.5.3.

We now discuss the subroutine for candidate generation. As is in the original algorithm, we produce $u[i]$ candidates in column $i$. Initially, $u[i] = K'$ due to the modification to line 3. Let us determine the total number of candidates, $|C|$. While candidates are produced column-wise, it is easier to count $|C|$ row-wise. It is the row $2^{p_0}$ where we start to have less than $n$ candidates. In the row $1..(2^{p_0} - 1)$, we have $n$ candidates each, which are $(2^{p_0} - 1) \cdot n = O(K)$ in total. In the row $2^{p_0}$ and below, $|C|$ is counted in a similar way described in Section 3.5.1. As the logarithmic distance between row $2^{p_0}$ and $K'$ is $O(\log K' - p_0) = O(\log K'^2/K)$, we have $|C| = O(K \log K'^2/K)$ where the $O(K)$ candidates above $2^{p_0}$-th row are absorbed. It is $O(K \log n^2/K)$ for large $K$ and $O(K \log K)$ for small $K$. Note that $|C|$ for small $K$ is consistent with the earlier analysis. This also implies that if $K = O(n^2)$, $|C| = O(K)$.

Each candidate needs $O(1)$ time for generation, making this subroutine $O(K \log \frac{n^2}{K})$ time for large $K$. Including final selection and sorting, the total time we spend for large $K$ is then summarized to $O(K \log K)$.

While we analyzed the modified algorithm mostly for large $K$, this version also supports small $K$ without affecting the complexity given in Theorem 3.5.1. We end this section with the following conclusion.

**Theorem 3.5.2.** $\forall K \in [1..n(n+1)/2]$*, the sorted list of* $K$ *maximum subarrays is computed in* $O((n+K)\log K)$ *time*

## 3.6 $O(n + K \log \min(K, n))$ Time Algorithm

As discussed in Section 3.5.3, the time complexity of Algorithm 18 is determined by the pre-process, which is $O(n \log K)$ time. The rest part of the algorithm is bounded by $O(K \log K)$, and it has been suggested that the improvement to the pre-process can speed up the overall complexity for small $K$. In this algorithm, we prepared $n$ versions of 2-3 tree maintaining $min_i$ for each prefix sum $sum[i]$. Among these $n$ versions, it is easily observed that at most $K$ versions are actually used to generate candidates. For small $K$, preparing $n$ versions of 2-3 tree is wasteful, as each unnecessary version still requires $O(\log K)$ time to prepare. However, even if we identify necessary $K$ versions in advance, it is not easy to make these $K$ versions be prepared within $O(K \log K)$ time.

In this section, we discuss an alternative data structure for maintaining each version of $min$.

### 3.6.1 $X + Y$ Problem

We consider the selection of the $K$-th largest elements in a set of Cartesian sums $X + Y = \{x_i + y_j \mid x_i \in X,\ y_j \in Y\}$, where $X = \{x_1, x_2, .., x_n\}$ and $Y = \{y_1, y_2, .., y_m\}$. This problem was originated by Johnson and Mizoguchi [59], and Frederickson and Johnson [38] gave an optimal solution with $O(m + p\log(K/p))$ time, where $n \le m$ and $p = \min\{K, n\}$.

If we need all $K$ largest elements in sorted order, we can devise a simple alternative algorithm.

Let $maxX[w]$ be the $w$-th maximum in $X$. Let $X \oplus Y$ be a list of size $m$ such that, $X \oplus Y = \{y_1 + maxX[1], y_2 + maxX[1], .., y_m + maxX[1]\}$. The first largest in $X + Y$ is then of course, $M[1] = \text{MAX}\{X \oplus Y\}$. Note that we use MAX, MIN for operator to avoid confusion with list names

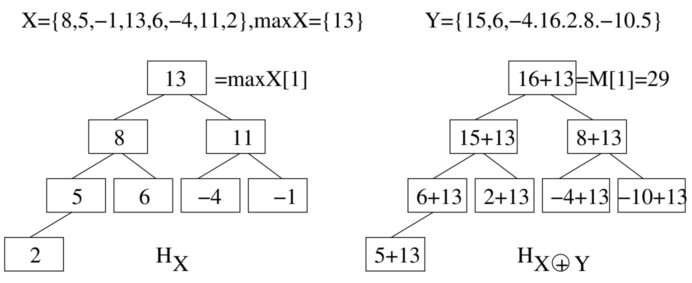


Figure 3.7: Build max-heaps $H_X$ and $H_{X \oplus Y}$ to solve $X + Y$ problem

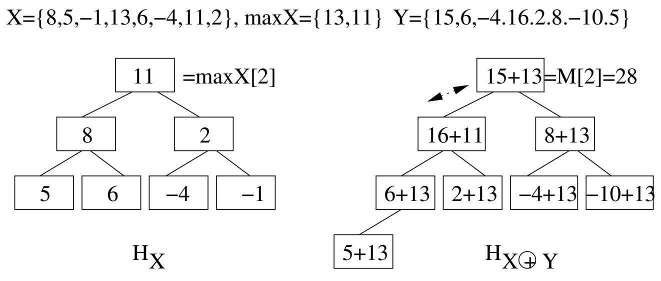


Figure 3.8: Get the next maximum in $X + Y$

containing lowercase *max* or *min*. We build a max-heap $H_X$ with $X$ and obtain MAX $\{X\}$ at the root, which we store in $maxX[1]$. The rest in $maxX$ are yet unknown. Then we build another max-heap $H_{X \oplus Y}$, where each node contains $y_j + maxX[1]$ $(j = 1..n)$. Here, the root of $H_{X \oplus Y}$ is $M[1]$ (Figure 3.7). Let $M[1]$ be $y_j + maxX[1]$ for some $j$. We update the root of $H_{X \oplus Y}$ with the next maximum obtainable with $y_j$, which is $y_j + maxX[2]$. In general, when the root containing $y_j + maxX[w]$ is updated, we check if $maxX[w+1]$ is readily available in $maxX$. Otherwise, the current root of $H_X$ is $maxX[w]$, thus we delete the root of $H_X$ and take the new root as $maxX[w + 1]$. Now we update $H_{X \oplus Y}$ and obtain $M[2]$ from the root (Figure 3.8). We repeat this process $K$ times and output $K$ largest sums in $X + Y$ in sorted order. The correctness of the algorithm is easily observed.

Building two heaps take linear time, and each subsequent maximum is found in logarithmic time. Hence, the total time is $O(m+K \log m)$. Note that the maximum value for $K$ is $mn$ in the extreme, but if $K \leq n \leq m$, we can reduce the size of $X$ and $Y$ to $K$ by leaving only the $K$ largest elements in $X$

**Algorithm 19** Computing $K$ largest elements in $X + Y$.

```
// Results are maintained in $M[1..K]$ in sorted order
1: Build max-heap $H_X$ with $x_1, x_2, .., x_n$.
2: Let $maxX[1]$ be $root(H_X)$, the root of $H_X$.
3: Build max-heap $H_{X \oplus Y}$ with $y_1+maxX[1], y_2+maxX[1], .., y_m+maxX[1]$.
4: for $k \leftarrow 1$ to $K$ do
5:     $M[k] \leftarrow root(H_{X \oplus Y})$. Output $M[k]$. Suppose $M[k]$ is $y_j + maxX[w]$.
6:     Obtain $maxX[w + 1]$, the next maximum in $X$
7:     Replace $root(H_{X \oplus Y})$ with $y_j + maxX[w + 1]$ and update $H_{X \oplus Y}$
8: end for
```

and $Y$. This can be done by the linear time selection algorithm [20] without increasing the complexity. The total time then becomes $O(m + K \log K)$, which is asymptotically equivalent to [38] plus sorting. We conclude the total time is $O(n + K \log \min(K, m))$.

This algorithm can be easily generalized to the selection in $X_1 + X_2 + .. + X_d$. We can prepare $d$ heaps in a similar manner. If $K$ is not greater than the size of any $X_i$ $(1 \le i \le d)$, we achieve $O(n + Kd \log K)$ time, where $n$ is $|X_1| + |X_2| + .. + |X_d|$.

### 3.6.2 1D $K$-OMSP

We start with the prefix sum $sum$'s of a given input $a[1..n]$, such that $sum[0] = 0$ and $sum[i] = a[1] + a[2] + .. + a[i]$. The sum of arbitrary consecutive elements, $a[i]+a[i+1]+..+a[j]$ is easily computed by $sum[j]-sum[i-1]$. We define $min_i[w]$ as the $w$-th minimum of $\{sum[0], .., sum[i-1]\}$. Let a list $Cand$ be $\{sum[1] - min_1[1], sum[2] - min_2[1], .., sum[n] - min_n[1]\}$.

The first maximum sum $M[1]$ is then MAX$\{Cand\}$. Suppose $M[1]$ is $sum[i] - min_i[1]$ for some $i$. We update this $i$-th entry in $Cand$ by replacing $min_i[1]$ with $min_i[2]$. $M[2]$ is then the new maximum of $Cand$. Similar to the $X + Y$ problem, we can build $H_{Cand}$, a max-heap with elements in $Cand$ to facilitate the maximum selection.

However, the maintenance of $min_1$,..,$min_n$ is not trivial. In $y_j+maxX[w]$, assuming that $maxX[w] = x_i$ for some $i$, we are not concerned of the position of $x_i$ in $X$. However, in $sum[i] - min_i[w]$, assuming $min_i[w] = sum[v]$ for

**Algorithm 20** Computing $K$ maximum subarrays.

```
//Results are in $M[1..K]$ in sorted order
1: for $i \leftarrow 1$ to $n$ do Compute $min_i[1]$, the minimum of $sum[0], .., sum[i-1]$
2: Build min-tournament $T_n$ with $sum[0]$, $sum[1]$,..,$sum[n-1]$.
3: Build max-heap $H_{Cand}$ with $sum[i] - min_i[1]$ for all $i = 1..n$.
4: for $k \leftarrow 1$ to $K$ do
5:     $M[k] \leftarrow root(H_{Cand})$. Output $M[k]$. Suppose $M[k]$ is $sum[i]-min_i[w]$.
6:     $min_i[w+1] \leftarrow$ the next unconsumed minimum in $\{sum[0], .., sum[i-1]\}$
7:     Replace $root(H_{Cand})$ with $sum[i] - min_i[w+1]$ and update $H_{Cand}$
8: end for
```

some $v$, the position of $v$ must be in the range of $0 \leq v \leq i-1$. Also, $maxX[w]$ is the list-wide $w$-th largest element in $X$. But $min_i[w]$ is the $w$-th minimum in the sub-list specific to $sum[i]$, i.e. $\{sum[0], .., sum[i-1]\}$. In the example above, one can easily observe that $min_4[1](= 0) \neq min_5[1](= -44)$.

To overcome the difficulty, the easiest option is to create an individual min-heap for each $min_i$ ($i = 1..n$). This is, however, too costly. Still, if we use a *persistent tournament* to maintain multiple versions of $min_i$, we can show that the followings facts hold.

**Lemma 3.6.1.** *All $min_i[1]$'s ($i = 1..n$) can be computed in $O(n)$ time.*

**Lemma 3.6.2.** *A tournament for $min_i$ can be prepared in $O(\log n)$ time.*

**Lemma 3.6.3.** *The next element in $min_i$ can be obtained in $O(\log n)$ time.*

We first present Algorithm 20 assuming that all the lemmas hold.

Lemma 3.6.1 is trivial. Line 1 runs a sequential scan on $sum[0], .., sum[i-1]$ and computes "prefix minimum" for each position $i$. Initially, we only know $min_i[1]$'s for all $i = 1..n$, but $min_i[w]$ ($w > 1$) will be found when it is needed. Now we describe the techniques to support Lemma 3.6.2 and 3.6.3.

### Creating a persistent tournament

A tournament is a binary tree that we described in Section 2.3. It is similar to a *heap* in terms of its feature and computational complexity. When

$n$ items are present, there are $O(n)$ nodes in the structure, as it is a *complete* binary tree with an exception in the leaf level, and the maximum (or minimum) can be found at the root in $O(1)$ time. The second maximum element can be retrieved by tracing the path from the root to the leaf node containing the first maximum and deleting (or replacing it with -$\infty$) the leaf node, and updating the nodes above it on the path to the root. The height of the tree is $O(\log n)$, and thus it takes $O(\log n)$ time to delete the maximum. As discussed in Section 2.3.1, retrieval of the $K$-th largest item using a tournament takes $O(n + K \log n)$ time. Indeed, a linear time selection [20] is possible. However, the tournament provides a list of $K$ largest items in order by default, while the linear time selection needs extra processing for this thereafter.

We can perform the same operation with a heap at the same cost. An advantage of a tournament over a heap is the preservation of the comparison history and the locational information relative to other participants. We can examine the tree to learn who beats who, and the original position of the final winner. Keeping such information in a heap may be difficult, if not impossible.

We build a min-tournament with $sum[0], .., sum[n-1]$. Each node contains the smaller value of two children. The overall minimum is placed at the root. Let us refer to this tournament as $T_0$. Each node also maintains the *coverage*, derived from the range of indices of prefix sums under its control. A node covering $sum[i]$,..,$sum[j]$ has a coverage $(i+1, j+1)$. Maintaining the coverage is essential to locate the $i$-th leaf in the tournament, whose detail will be given shortly. To build $T_0$, we create a root node and execute *min_prefix_tournament*(*root*, 1,$n$) presented in Algorithm 21.

To maintain $min_i$ $(i = 1..n)$, we need the $i$-th version $T_i$ that covers $sum[0], .., sum[i-1]$. However, we wish to avoid building each version of tournament from scratch. We apply the *node copying* technique used in a *persistent* data structure, which allows access to the old versions after subsequent update operations [30]. The same technique was already discussed in Section 3.5.2.

We show how to retrieve $T_i$ from $T_0$, and update $T_i$ while keeping $T_0$ and other versions intact.

**Algorithm 21** Build a tournament $T_n$ to find MIN $\{sum[f\text{-}1], .., sum[t\text{-}1]\}$

```
procedure $min\_prefix\_tournament(node, f, t)$ begin
1: $node.from \leftarrow f$, $node.to \leftarrow t$
2: if $f = t$ then
3:     $node.val \leftarrow sum[f-1]$
4: else
5:     create $left$ child node, $min\_prefix\_tournament(left, f, \frac{f+t-1}{2})$
6:     create $right$ child node, $min\_prefix\_tournament(right, \frac{f+t+1}{2}, t)$
7:     $node.val \leftarrow$ MIN $(left.val, right.val)$//MIN $\{sum[f\text{-}1], .., sum[t\text{-}1]\}$
8: end if
end
```

***Preparing the $i$-th version*** To maintain $min_i$, we prepare $T_i$, the $i$-th version of the tournament. We want the root of $T_i$ to have a coverage $(1, i)$. We visit the $i$-th leaf (containing $sum[i-1]$) in $T_0$ and traverse back towards the root. The coverage kept in each node is looked up for locating the $i$-th leaf. We make a copy of each node on the path. If the currently visiting node has a right sibling, when we copy the parent, the copied parent loses link to the right child, keeping the coverage not going above $i$. When we arrive at the root, the copied root has the coverage $(1, i)$ with the value MIN $\{sum[0], .., sum[i-1]\}$, i.e. $\min_i[1]$ . Now $T_i$ is ready for use. During the process, as we only updated the copied nodes, $T_0$ is kept intact. Note that most nodes in $T_i$ are recycled from $T_0$, and belong to $T_i$ as well as $T_0$. Only those copied nodes are *version specific* to $T_i$. The retrieval of $T_i$ from $T_0$ takes $O(\log n)$ time, proving Lemma 3.6.2. Detailed routine is given in Algorithm 22. To retrieve $T_8$ from $T_0$ as shown in Figure 3.9, we first copy the root of $T_0$ and call *make_Ti*(8,*root*), where *root* is the copied root.

***Updating the $i$-th version*** Suppose we have discovered $min_i[1..w]$, and now wish to find $min_i[w+1]$. If $w = 1$, $T_i$ is not available yet, so we retrieve it from $T_0$ as described above. Otherwise, the current root of $T_i$ is $min_i[w]$. Let $min_i[w] = sum[x]$ for $x < i$. We traverse from the root of $T_i$ to the $(x+1)$-th leaf that contains $sum[x]$. The coverage kept in each node determines where the $(x+1)$-th leaf is located. We replace this leaf with $\infty$, and update the rest nodes on the path to the root. We are allowed to update

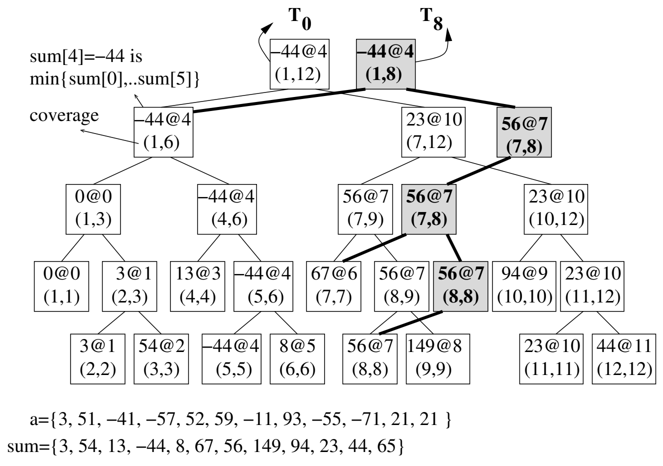


Figure 3.9: Retrieving $T_i$ with $T_0$ kept intact. $T_0$ is built with the input in Example 3.2.2. $T_i$ is the tournament that maintains $min_i$. Here, the root of $T_8$ is $min_8[1] = -44$, that is $sum[4]$

a node version specific to $T_i$. Otherwise, we make a copy and update it. This routine again is $O(\log n)$ time, proving Lemma 3.6.3. Figure 3.10 shows the result of updating $T_8$, where we delete $min_8[1]$, that is $sum[4]$. This is done by executing *update_Ti*$(root,4)$ in Algorithm 23, where *root* is a copy of the root of $T_8$. Notice that the update only affects $T_8$.

### 3.6.3 Analysis of Algorithm 20

Lines 1,2 and 3 are linear time. Building a tournament is recursively done. Line 5 is $O(1)$ time. Line 6 involves at most two $O(\log n)$ time operations. If $T_i$ is already available, we simply access this. Otherwise, we retrieve it from $T_0$ spending $O(\log n)$ time. Line 7 involves the max-heap maintaining $Cand$, and takes $O(\log n)$ time to return each of $M[k]$ and update the heap. Altogether, the total time is $O(n + K \log n)$. Note that $K$ can be $\frac{n(n+1)}{2}$ in the extreme, and this algorithm can work with any $K$. Bengtsson and Chen [16] observed that $O(n + K \log n) = O(n + K \log K)$, hence the total time is $O(n + K \log \min(K, n))$, which matches the recent results by Bengtsson and Chen [16] and Cheng et. al [26].

A trivial lower bound for computing $K$ maximum sums in an array of

**Algorithm 22** Retrieve a tournament $T_i$ from $T_0$

```
/*$node$ is a copied node. $left$ and $right$ are children of $node$*/
procedure $make\_Ti(i, node)$ begin
1: if $node.from = node.to$ then
2:    return //this $node$ is the $i$-th leaf. no need to copy
3: else
4:    /* this $node$ is an internal node */
5:    if $left.to \geq i$ then
6:       //$i$-th leaf is in the left subtree.
7:       $left \leftarrow$ copy of $left$
8:       $make\_Ti(i,left)$
9:       $node.to \leftarrow left.to$ //this node doesn't need right subtree anymore
10:      delete the link to $right$
11:   else
12:      //$i$-th leaf is in the right subtree.
13:      $right \leftarrow$ copy of $right$
14:      $make\_Ti(i,right)$
15:      $node.to \leftarrow right.to$
16:   end if
17:   //update $node.val$
18:   $node.val \leftarrow \text{MIN}\{left.val, right.val\}$ //$right.val$ is $\infty$ if $right$ is null
19: end if
end
```

size $n$ without requiring sorted order is $\Omega(n + K)$, if $K \leq n$. However, as we want to output $K$ maximum sums in non-decreasing order, we have a better lower bound.

**Lemma 3.6.4.** *The lower bound for the 1D $K$-OMSP is $\Omega(n + K \log K)$ for $K \leq n$.*

*Proof.* We prepare an adversary sequence of randomly ordered $K$ distinct values $a_1$,$a_2$,..$a_K$ in the array $a[1..n]$ and assume that there are at least one $-\infty$ between each two values. The rest elements are all $-\infty$, such that $a[1..n]$ looks like,

$$-\infty, .., -\infty, a_1, .., -\infty, .., a_2, ...., -\infty, .., a_K, -\infty, .., -\infty$$

This way, the $K$ maximum subarrays are $\{a_1\}$,$\{a_2\}$,..$\{a_K\}$, all made of

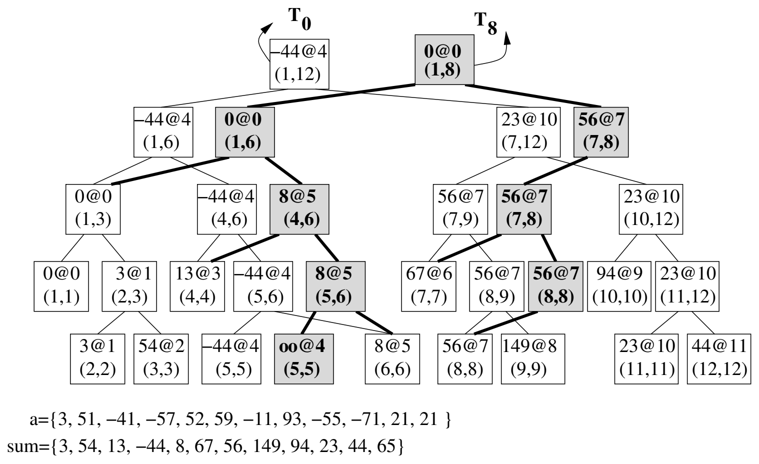


Figure 3.10: Updating $T_i$. Here, we update $T_8$ by replacing $sum[4] = -44$ with $\infty$. The new root of $T_8$ is $min_8[2] = 0$

**Algorithm 23** Delete $sum[i]$ from a tournament

```
/* $i$ is the index of the prefix sum to be deleted */
procedure $update\_Ti(node, i)$ begin
 1: if $node.from = node.to$ then
 2:    //this is the leaf node maintaining $sum[i]$
 3:    $node.val \leftarrow \infty$
 4: else
 5:    if $left.from \leq i+1 \leq left.to$ then
 6:       //$(i+1)$-th leaf is in the left subtree
 7:       $left \leftarrow$ copy of $left$ //skip if $left$ is already a copy
 8:       $update\_Ti(left, i)$
 9:    else if $right$ is not null then
10:       //$(i+1)$-th leaf is in the right subtree
11:       $right \leftarrow$ copy of $right$ //skip if $right$ is already a copy
12:       $update\_Ti(right, i)$
13:    end if
14:    //update $node.val$
15:    $node.val \leftarrow$ MIN $\{left.val, right.val\}$ //$right.val$ is $\infty$ if $right$ is null
16: end if
end
```

a single element. Now, computing the $K$ maximum sums in non-increasing order becomes equivalent to selecting $K$ largest elements in sorted order, meaning that the problem reduces to two known problems, selection of the $K$ largest elements, and sorting them. The lower bound of selecting the $K$ largest element is $\Omega(n+K)$, and that of sorting $K$ random numbers is $\Omega(K \log K)$ [63][‡]. Let $T$ be the total time. Then, $T \geq \Omega(n+K) + \Omega(K \log K) \geq \Omega(n + K \log K)$. $\square$

This lower bound was shown by Bengtsson and Chen [16] and Cheng et. al [26]. Note that this lower bound matches the upper bound $O(n + K \log \min(K, n))$ for $K \leq n$. Therefore,

**Theorem 3.6.5.** *Algorithm 20 is optimal for $K \leq n$.*

While independently devised, Algorithm 19 for $X+Y$ problem is almost identical to the technique presented in the recent result by Bengtsson and Chen [16], except Algorithm 19 uses a simple heap. They derived an algorithm for the 1D $K$-OMSP from their version of Algorithm 19. Consequently, their algorithm and Algorithm 20 are very similar in terms of the structure and the complexity.

Still, the use of a persistent data structure is unique in Algorithm 20 and it only requires to maintain prefix sums, whereas [16] requires to maintain both prefix and suffix sums. As [16] appeared earlier, the contribution of Algorithm 20 is however little. Nevertheless, Algorithm 20 is included in this thesis for the completeness as well as the following reasons. Firstly, its framework using a combination of a heap and a tournament can be generalized to devise an efficient algorithm for higher dimensions, which Bengtsson and Chen overlooked in [16]. Secondly, it demonstrates the effectiveness of the tournament for maintaining order as well as the locational information, which will serve as a vital tool for computing the $K$-disjoint maximum subarray problem covered in the next chapter.

## 3.7 Algorithms for Two Dimensions

The simplest algorithm for 2D may be based on strip-wise computation as described in Section 2.2.1. We showed that each strip is regarded as a 1D

[‡] Section 5.3 in Volume 3 in particular

problem, which can be computed by an algorithm for 1D. For an input array of size $n \times n$, there are $O(n^2)$ strips, and it is easy to see that we need $O(n^2 T_{1D}(n))$ time to compute $K$ maximum sums in a 2D array, where $T_{1D}(n)$ is the time complexity of an algorithm for 1D. For example, in our preliminary work [7], we extended $O(Kn)$ time algorithm for 1D (Algorithm 15) to get $O(Kn^3)$ time algorithm for 2D.

In this section, we present more efficient algorithms for 2D, achieving $O(n^3)$ time for certain range of $K$. Certainly, we can readily derive $O(n^3)$ time algorithm by performing strip-wise computation with Algorithm 20. It is $O(n^3 + Kn^2 \log \min(K, n))$ time, that is $O(n^3)$ for $K \leq \frac{n}{\log n}$.

Our objective is to make an algorithm more tolerant of the value of $K$, such that we get $O(n^3)$ time for wider value of $K$. We introduce two approaches based on the sampling technique and two-level heap. The former is applied to Algorithm 18 or Algorithm 20 and achieves $O(n^3)$ time for $K \leq \sqrt{\frac{n^3}{\log n}}$, while the latter specifically applies Algorithm 20 to get $O(n^3)$ time for $K \leq \frac{n^3}{\log n}$.

### 3.7.1 Sampling in Two Dimensions

If we perform the $O((n + K) \log K)$ solution (Algorithm 18), we get $K$ maximum sums from each strip and regard them as candidates. We have total of $O(Kn^2)$ candidates. From this set of candidates, we can select the final $K$ maximum subarrays using Algorithm 14. The total time for the 2D $K$-OMSP is then $O(n^2(n + K) \log K)$.

Using the sampling technique described in Section 3.4, we can reduce this complexity.

If $K \leq n(n+1)/2$, among $n(n+1)/2$ such portions, there are at least $n(n+1)/2 - K$ portions whose own $K$ maximum subarrays are totally excluded from the final solution set. We identify such portions and prevent them from producing useless candidates.

With each strip prefix $s_{g,i}[1..n]$, we compute the maximum sum by Algorithm 3 in the strip. We get $O(n^2)$ 'samples', where each sample is computed in $O(n)$ time. We spend $O(n^3)$ time for this.

Among these $O(n^2)$ samples, we choose the $K$-th maximum by the linear time selection algorithm. By doing so, we filter out 'unnecessary' portions and leave only $K$ portions that may produce meaningful candidates for the final solution.

We apply the algorithm for 1D on these selected strips. Performing $O((n+K)\log K)$ time solution (Algorithm 18) $K$ times, $O(K(n+K)\log K)$ time is spent.

As each portion produces $K$ candidates, there are total of $K^2$ candidates produced by $K$ portions. Again, by applying Algorithm 14, we select the $K$ largest values. The time for this is $O(K^2)$. Finally sorting on the $K$ final values takes $O(K\log K)$ time. The total time for two dimensions is therefore $O(n^3 + K(n+K)\log K + K^2 + K\log K)$. For $1 \leq K \leq n(n+1)/2$, it is $O(n^3 + K^2\log K)$ time. This is cubic time if $K \leq n^{1.5}/\sqrt{\log n}$.

We may use Algorithm 20 in lieu of Algorithm 18, which results in $O(n^3 + K(n + K\log n) + K^2 + K\log K) = O(n^3 + K^2\log n)$ time. This also runs in cubic time for $K \leq n^{1.5}/\sqrt{\log n}$.

In the following, we present another technique for more efficient algorithm for 2D. The sampling technique is, however, still useful in practice when it is combined with the new technique that will be described in the next section. More details on the performance benchmarks will be given in Chapter 5.

### 3.7.2 Selection in a Two-level Heap

Based on the sampling technique, we reduce the number of candidates produced from $Kn(n+1)/2$ to $K^2$. While this is a sharp reduction, it is still a large number when we are, of course, only interested in the $K$ largest candidates.

The sampling-based technique is simple, but the strip-wise computation based on this approach has little *global control*, meaning that we "blindly" produce $K$ candidates in each strip, which results in generation of $K^2$ candidates in total. The situation is similar to Algorithm 17, depicted by Figure 3.1, where we still need to generate $K$ candidates in each column.

Certainly, the ideal situation is where we know how many candidates to produce in each strip. While such information may not be available in advance, if we extend the framework used in Algorithm 20, we can introduce some degree of global control, which results in further reduction in total number of candidates.

First, we spend $O(m^2n)$ time to obtain $sum_{g,i}[1..n]$'s for all pairs of $g, i$ ($g \leq i$). Each $sum_{g,i}$ is processed by lines 1-3 of Algorithm 20. As a result, there are $O(m^2)$ max-heaps, $H_{Cand_{g,i}}$. We collect all $M_{g,i}[1]$'s, the root of each $H_{Cand_{g,i}}$ into a list $2Cand$ and build another max-heap $H_{2Cand}$. This heap contains $O(m^2)$ nodes. The first maximum sum in 2D, $M[1]$, is located at the root of $H_{2Cand}$. Suppose $M[1]$ is $M_{g,i}[1]$. To get $M[2]$, we obtain $M_{g,i}[2]$ after updating $H_{Cand_{g,i}}$. Then we replace the root of $H_{2Cand}$ with $M_{g,i}[2]$ and update this heap, whose new root returns $M[2]$. This is repeated $K$ times.

Each update operation to $H_{2Cand}$ is $O(\log m)$ time. Finding the next maximum sum involves an update to one $H_{Cand_{g,i}}$ followed by an update to $H_{2Cand}$. As $m \leq n$ is assumed, each subsequent maximum sum is found in $O(\log \min(K, n))$ time. The total time is $O(m^2n + K \log \min(K, n))$, which is cubic time if $m = n$ and $K \leq n^3/\log n$, matching the previous result [26].

## 3.8 Extending to Higher Dimensions

In general, in a $d$-dimensional array of size $n \times \cdots \times n$, there are $O(n^{2d-2})$ 1D problems. The same idea used to compute the 2D problem is readily applicable to $d$-dimensions. We build $O(n^{2d-2})$ max-heaps for 1D problems, spending $O(n^{2d-1})$ time, and prepare an extra max-heap $H_{dCand}$ that maintains the maximum sum of each 1D problem.

Here, $H_{dCand}$ has $O(n^{2d-2})$ nodes, but the sampling technique in Section 3.7.1 . We can opt to maintain the $K$ largest values only in $H_{dCand}$. The height of $H_{dCand}$ is then bounded by $O(\log \min)K, n^d))$. Each subsequent maximum sum of the $d$-D problem is found by $O(\log n)$ time update to the 1D-level max-heap and $O(\log \min(K, n^d))$ time update to $H_{dCand}$, that is, in short, $O(\log \min(K, n^d))$ time.

The total time for $K$ maximum sums is then $O(n^{2d-1} + K \log \min(K, n^d))$. While $K = O(n^{2d})$ in extreme, it is simply $O(n^{2d-1})$ time for $K \leq \frac{n^{2d-1}}{\log \min(K, n^d)}$.

## 3.9 *K*-OMSP with the Length Constraints

Through a simple modification to Algorithm 18, we can compute $K$ maximum subarrays with the length constraints $L$ and $U$ $(1 \leq L \leq U \leq n)$, such that the length of $K$ maximum subarrays would be between $L$ and $U$.

Let MIN$(K, l)$ be the operation that selects $K$ minimum elements in a set $l$ in non-decreasing sorted order. In the original form, each prefix sum $sum[i]$ is associated with a minimum prefix sums $min_i$, that is,

$$min_i = \text{MIN}(K, \{sum[j] \mid 0 \leq j \leq i-1\})$$

If we devise a way to maintain $min'_i$, such that,

$$min'_i = \text{MIN}(K, \{sum[j] \mid i-U \leq j \leq i-L\}),^{\S}$$

every candidate $cand_i[k]$ (for $1 \leq k \leq K$), computed by $sum[i] - min'_i[k]$ would be of length between $L$ and $U$.

Let us now describe how $min'_i$ can be obtained.

In the original algorithm,

$$min_{i-L+1} = \text{MIN}(K, \{sum[j] \mid 0 \leq j \leq i-L\}).$$

Hence, if the $(i-L+1)$-th version of the persistent 2-3 tree, $min_{i-L+1}$, is used as $min'_i$, each candidate $cand_i[k]$ would satisfy the lower length constraint $L$.

In the following, we will assume that $i \geq L$. Otherwise, $sum[i]$ will only produce a candidate shorter than $L$.

To satisfy the upper length constraint $U$ as well, we need to do more than just taking $min_{i-L+1}$ as $min'_i$. We should make sure that "earlier" prefix sums, including $sum[0]$,..,$sum[i-U-1]$ would not be existent in $min'_i$, such that $min'_i$ would not have more than $U-L+1$ entries. So if we wish to use $min_{i-L+1}$ of the original algorithm to express $min'_i$, these earlier prefix sums should be removed from $min_{i-L+1}$. If $i \leq U$, however, such a care is not necessary since the number of elements kept in $min'_i = min_{i-L+1}$ can never

§ To be precise, both $i-U$ and $i-L$ should not be negative. We regard them as 0 if $i-U<0$ or $i-L<0$.

exceed $U - L + 1$.

Notice that $min'_U$ includes all $sum[0]$,..$sum[U - L]$. The next version is obtained by updating the current one, so when $min'_{U+1}$ is derived from $min'_U$ by inserting $sum[U - L + 1]$, we drop the earliest prefix sum $sum[0]$. Basically, for $i > U$, when $min'_i$ is prepared, we drop $sum[i - U - 1]$, the earliest prefix sum.

Locating the earliest prefix sum can be done in $O(1)$ time if we maintain a pointer from each prefix sum $sum[i]$ to its representative node in the 2-3 tree when we insert $sum[i]$ into the 2-3 tree.

In the following analysis, let $D = U - L$. We only consider $L \leq i \leq n$, since we do not need to maintain 2-3 tree for $i < L$.

Let us count the number of elements in $min'_i$. If $K \leq D$, $min'_i$ will have $K$ elements, and $D$ elements otherwise.

If $K \leq D$, the pre-processing time is $O(n + (n - L) \log K)$ as earlier versions of the 2-3 tree, $min'_1$,..,$min_{L-1}$', do not need to be built. This contributes to $O(n + (n - L) \log K + K \log K)$ total time, as the rest of the algorithm is still bounded by $O(K \log K)$ time.

Similarly, if $K > D$, the pre-processing part of Algorithm 18 is $O(n + (n - L) \log D)$ time, and the rest is $O(K \log K)$ time. Total time is then $O(n + (n - L) \log D + K \log K)$.

Combining two cases, we have total of $O(n + (n - L) \log \min(K, D) + K \log K)$ time.

Algorithm 20 can be similarly modified to consider the length constraints.

For each prefix sum, $sum[i]$ for $L \leq i \leq n$, we need to find a matching first minimum prefix sum $min'_i[1]$. Fan et al. [32] showed that all minimum prefix sums satisfying the length constraints, $min'_i[1]$ in this context, can be obtained in $O(n)$ time. One may also consider using *Range Maximum-Sum Segment Query* (RMSQ) technique by Chen and Chao [66], but it may be more than necessary.

The summary of the technique for finding $min'_i[1]$ by Fan et al. [32] will be explained shortly in Section 3.9.1.

Incorporating this technique, we can obtain the maximum subarray ending at $i$, starting somewhere between $[i - U + 1, i - L + 1]$. Let $Cand$ be $\{sum[L] - min'_L[1], sum[L+1] - min'_{L+1}[1], .., sum[n] - min'_n[1]\}$. We build a max heap $H_{Cand}$ consisting of these $(n-L)$ elements and find the maximum sum located at the root.

The subsequent maximum sum can be searched by replacing the root $sum[i] - min'_i[1]$ with $sum[i] - min'_i[2]$ and updating $H_{Cand}$.

We also build a min-tournament $T_0$ with $sum[0],..sum[n-L]$. Note that $sum[n-L+1],..sum[n]$ can be omitted as none of these will be a minimum prefix sum satisfying the length constraint $L$. When $min'_i[k]$ for $k \geq 2$ is required, we can retrieve the $i$-th version $T'_i$ from $T_0$ in $O(\log(n-L))$ time, such that $T'_i$ would cover $sum[i-U],..sum[i-L]$. On the other hand, in the original form of the algorithm, $T_i$ covers $sum[0],..sum[i-1]$. The root of the min-tournament $T'_i$ corresponds to $min'_i[1]$, hence $min'_i[2]$ can be obtained by removing the root spending $O(\log(n-L))$ time.

The total time for finding $K$ maximum sums meeting the length constraints is then $O(n + K\log(n-L))$, or it can be expressed as $O(n + K \log\min(K, n-L))$ due to Bengtsson and Chen [16]. This algorithm is superior to the one described earlier in general.

For a 2D array of size $m \times n$, we can make an interesting observation. Suppose that we place the height constraints $L_h$ and $U_h$, such that the height of the found maximum subarray would be between $L_h$ and $U_h$. Let us ignore the length (width) constraints for now.

Interestingly, the height constraints can be handled with no modification to the original algorithm. We simply prepare strips which are between $L_h$ and $U_h$ rows tall only. The number of strips to process is now reduced to $(U_h - L_h + 1)(2n + 2 - U_h - L_h)/2$, which is $O(nD_h)$ where $D_h = U_h - L_h$.

If $D_h$ is considerably smaller than $m$, having the height constraints would result in sharply sub-cubic time, standing at $O(mnD_h)$ for a single maximum sum.

Combining both the length and the height constraints, $K$ maximum subarrays in a 2D array can be computed in $O(mnD_h + K\log\min(K, n))$ time.

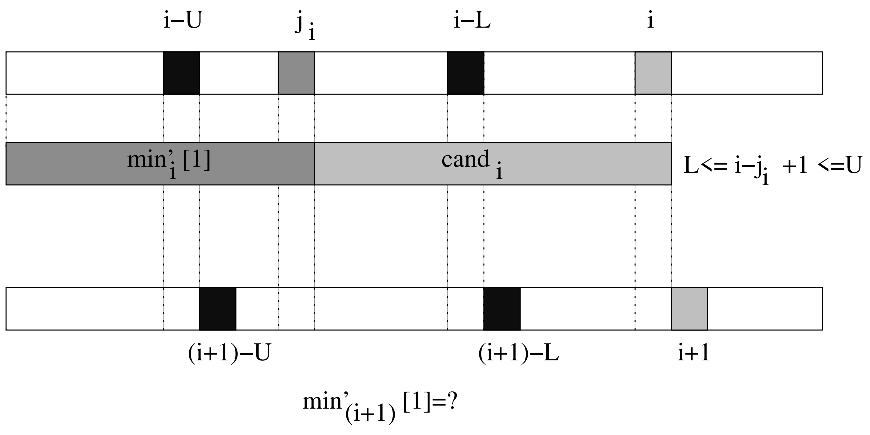


Figure 3.11: Finding the length-constraints satisfying minimum prefix sum

### 3.9.1 Finding the Length-constraints satisfying Minimum prefix sum

As shown Figure 3.11, we have $min'_i[1]$, that is $sum[j_i]$ where $i - U \leq j_i \leq i - L$, such that the length of $cand_i$, the candidate formed by $sum[i] - min'_i[1]$, would be between $L$ and $U$. In the following, we use the notation $id(X)$ to denote the index of the prefix sum $X$. For example, $id(min'_i[1])$ or $id(sum[j_i])$ is $j_i$.

From definition, we have

$$min'_i[1] = \text{MIN}\{sum[j] \mid i - U \leq j \leq i - L\},$$

and

$$min'_{i+1}[1] = \text{MIN}\{sum[j] \mid i + 1 - U \leq j \leq i + 1 - L\},$$

Then $min'_{i+1}[1]$ can be determined by the following computation, where $j_i = id(min'_i[1])$.

$$min_{i+1}[1]' = \begin{cases} \text{MIN}\{min'_i[1], sum[i+1-L]\}, & \text{if } j_i > i - U \\ \text{MIN}\{sum[j] \mid j_i + 1 \leq j \leq i + 1 - L\}, & \text{otherwise} \end{cases}$$

The first case is trivial. To compute the second case efficiently, however, we need to devise an efficient algorithm to retrieve the minimum $sum[j]$ for

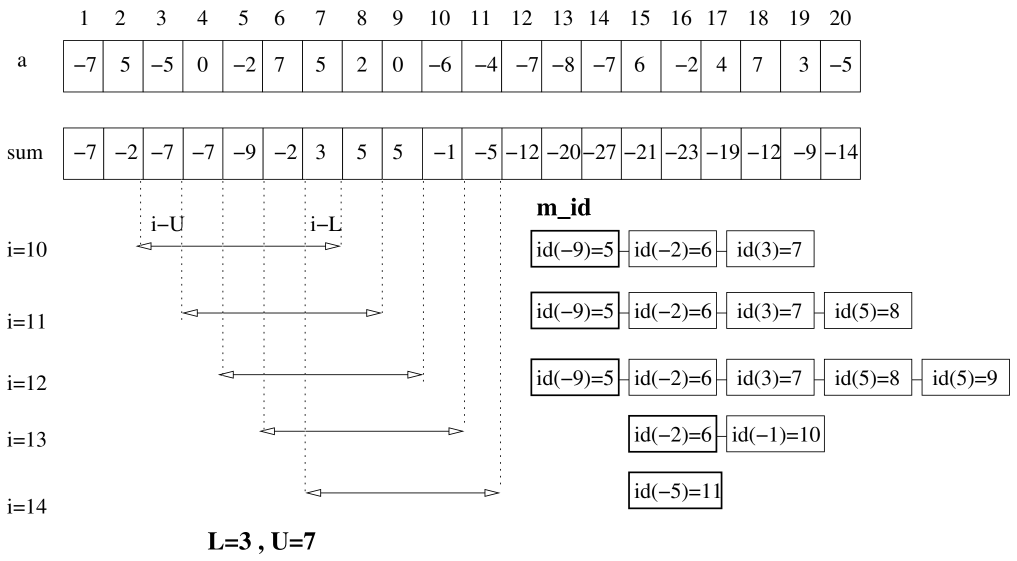


Figure 3.12: Maintaining the list $m\_id$

$j \in [j_i + 1, i + 1 - L]$. We prepare a list $m\_id$ to facilitate this. Suppose there are $p$ nodes in $m\_id$ at present, where the head of the list is denoted by $m\_id[1]$, and the subsequent nodes are referred to as $m\_id[2]$ etc. We maintain $m\_id$ such that the following properties would hold.

1. $i - U \leq m\_id[1] < m\_id[2] < .. < m\_id[p] \leq i - L$
2. $m\_id[1] = id(min'_i[1])$
3. $m\_id[q] = id(w)$ for $q > 1$, where $w = \text{MIN}\{sum[j] \mid m\_id[q-1] + 1 \leq j \leq i - L\}$

For example, when $L = 3$ and $U = 7$, $m\_id$ is maintained as shown in Figure 3.12. Note that $sum[m\_id[1]]$ always corresponds to $min'_i[1]$ for any $i$.

Algorithm 24 outlines the details for computing all $min'_i[1]$'s for $L \leq i \leq n$. This idea is attributed to Fan et al.[32], who claimed that this algorithm runs in $O(n - L)$ time. To verify this, it is crucial to find the number of iterations by the 'while'-loop at lines 5-7.

We show that the 'while'-loop performs at most $O(n - L)$ iterations throughout the entire process, and the total running time of Algorithm 24 is

**Algorithm 24** Computing all $min'_i[1]$'s $(L \leq i \leq n)$ in $O(n-L)$ time

```
1: $m\_id[0] \leftarrow 0$
2: $p \leftarrow 0$
3: for $i \leftarrow L$ to $n$ do
4:    $q \leftarrow p$ //$p$: length of $m\_id$
5:    while $q > 0$ and $sum[i-L] < sum[m\_id[q]]$ do
6:       $q \leftarrow q - 1$ //Search backward
7:    end while
8:    $q \leftarrow q+1$ //Found the left-most node where $sum[i-L] \leq sum[m\_id[q]]$
9:    $m\_id[q] \leftarrow i - L$ //Update $m\_id[q]$
10:   $p \leftarrow q$ //Length of $m\_id$ is now $q$
11:   if $m\_id[1] < i - U$ then
12:      Drop $m\_id[1]$. The next node is now referred to as $m\_id[1]$
13:      $p \leftarrow p - 1$ //Length of $m\_id$ is reduced by 1
14:   end if
15:   $min'_i[1] \leftarrow sum[m\_id[1]]$ //Found $min'_i[1]$
16: end for
```

indeed $O(n-L)$.

During each iteration of the 'for' loop at lines 3-16, if the 'while' loop is not executed at all, the length of $m\_id$ may increase by 1, but its length is limited by $(U-L+1)$ due to lines 11-14. This increase may happen at most $(n-L+1)$ times. On the other hand, each iteration of the 'while'-loop decreases the length of $m\_id$ by 1. Note that the initial length of $m\_id$ is 0.

Let us define $d(i)$ as the length of $m\_id$ at the end of the $i$-th iteration of the 'for'-loop and $\Delta d(i)$ as the change of the length, $d(i) - d(i-1)$.

Suppose that out of $(n-L+1)$ total iterations of the 'for'-loop, $x$ of them enters the 'while'-loop. The remaining $(n-L+1-x)$ iterations perform $O(1)$ time operations each, and increase $d(i)$ by 1.

During the $x$ iterations, let us assume that each entrance to the 'while'-loop runs $w_1$, $w_2$,..$w_x$ iterations. Each of these $x$ entrances to the 'while'-loop spends $w_1$,$w_2$,..$w_x$ units of time each, and reduces $d(i)$ by $w_1$,$w_2$,..$w_x$.

Let us compute the total sum of $\Delta d(i)$ for $L \leq i \leq n$.

$$\sum \Delta d(i) = (n-L+1-x) - \sum_{y=1}^{x} w_y$$

Notice that,

$$\sum \Delta d(i) = (d(L)-d(L-1))+(d(L+1)-d(L))+...+(d(n)-d(n-1)) = d(L)+d(n)$$

Since $d(L) = 1$, we have

$$\sum_{y=1}^{x} w_y = (n - L + 1 - x) - (d(n) + d(L)) = n - L - x - d(n)$$

Here, notice that $1 \leq d(n) \leq U - L + 1$.

The total running time $T$ is expressed as the sum of time spent by $x$ iterations that enter the 'while'-loop and the remaining $(n - L + 1 - x)$ iterations whose running time is $O(1)$ each.

$$T = (n - L + 1 - x) + \sum_{y=1}^{x} w_y = 2(n - L - x) + 1 - d(n) \leq 2(n - L)$$

Therefore, $T = O(n - L)$.

## 3.10 Summary of Results

In this chapter, we studied $K$-overlapping maximum subarray problem for 1D and 2D cases and presented new algorithms.

For 1D case, we presented increasingly efficient algorithms, where each achieves respectively $O(Kn)$, $O(K^2 + n \log K)$, $O((n + K) \log K)$ and $O(n + K \log \min(K, n))$ time. All algorithms presented produce $K$ maximum subarrays in sorted order.

Independently, Bengtsson and Chen studied the same problem and presented $O(min\{K + n \log^2 n, n\sqrt{K}\})$ time [15] and $O(n + K \log \min(K, n))$ time algorithms. Note that in [15], the order of final $K$ maximum sums is not sorted, while sortedness is a part of requirement in [16].

This problem has been also studied by Cheng et al. [26], and they achieved the equivalent complexity of $O(n + K \log \min(K, n))$ time.

It is interesting to observe that the 1D $K$-OMSP is of the same complexity as the selection of $K$ maximum values discussed in Section 2.3.1.

For the 2D $K$-OMSP, we can compute $K$ maximum sums in cubic time for certain range of $K$. We showed that the sampling technique makes a cubic time solution for $K \leq \sqrt{\frac{n^3}{\log n}}$, and the selection in a two-level heap results in a cubic time for wider range of $K$, that is $K \leq \frac{n^3}{\log n}$. Cheng et. al [26] independently established the same result as the latter.

Extending the framework used in the latter, a simple algorithm can be also derived for the general $d$-dimensions, which takes $O(n^{2d-1}+K \log \min(K, n^d))$ time. It is $O(n^{2d-1})$ time for $K \leq \frac{n^{2d-1}}{\log \min(K,n^d)}$.

When length constraints $L$ and $U$ are imposed, such that the length of the found maximum subarray should be between $L$ and $U$, the proposed algorithms can be used with minor modifications. We showed that $K$ maximum subarrays meeting the length constraints can be computed in $O(n + K \log \min(K, n - L))$ time.

Takaoka [10] elaborated the framework based on the distance matrix multiplication, and showed that sub-cubic time is achievable for very small $K$, that is $K \leq O(\sqrt{\alpha \log n / \log \log n})$ where $0 \leq \alpha \leq 1$.

# Chapter 4

# $K$-Disjoint Maximum Subarray Problem

## 4.1 Introduction

The goal of the $K$-disjoint maximum subarray problem ($K$-DMSP) is to find $K$ maximum subarrays, which are disjoint from one another. Ruzzo and Tompa designed a linear time algorithm that finds all disjoint maximum subarrays in a 1D array [82].

To best of the author's knowledge, little study has been undertaken on this problem for higher dimensions. Particularly, an algorithm for the 2D $K$-DMSP may be used to select brightest spots in graphics, and such a technique may be also applied to motion detection and video compression.

In this chapter, we discuss the difficulty involved in extending Ruzzo and Tompa's algorithm [82] to 2D and design an alternative algorithm for one-dimension that is more flexible to extend to higher dimensions. Based on the new framework, we first present an $O(m^2n + Km^2 \log n)$ time solution for the 2D $K$-DMSP where $m \times n$ is the size of the input array. This is cubic time when $m = n$ and $K \leq n/\log n$. We also show that the upper bound converges to $O(m^2 n \log n)$ for $K > n$.

## 4.2 Problem Definition

For a given array $a[1..n]$ containing a mixture of positive and negative real numbers, the maximum subarray is the consecutive array elements of the greatest sum. The definition of $K$ disjoint maximum subarrays for one-dimension is given as follows.

**Definition 4.2.1. [Ruzzo and Tompa [82]]** The $k$-th maximum subarray is the consecutive array elements of largest possible sum that excludes elements contained in $(k-1)$ maximum subarrays

In addition, we impose sorted order on the sum of these maximum subarrays.

**Definition 4.2.2.** The $k$-th maximum sum is not greater than the $(k-1)$-th maximum sum.

It is possible for a subarray of zero sum adjacent to a subarray of positive sum to create an overlapping subarray with tied sums. To resolve this, we select the one with smaller area if there are overlapping subarrays of tied sums. Another subtle problem arises with the value of $K$. For $k < K$, it is possible that the $k$-th maximum subarray becomes non-positive. We may stop the process at this point even if the $K$-th maximum is yet to be found. A non-positive maximum subarray is essentially a single negative array element, which is trivial to find. Let $\bar{K}$ be the maximum number of positive disjoint maximum subarrays. Theoretically $1 \leq \bar{K} \leq n/2$ and is data dependent. Throughout this thesis, we assume that $K$, the number of maximum subarrays we wish to find, is not greater than $\bar{K}$.

**Example 4.2.3.** $a$ ={3,51,-41,-57,52,59,-11,93,-55,-71,21,21}. In the array $a$, the maximum subarray is 193, $a[5]+a[6]+a[7]+a[8]$ if the index of first element is 1. We denote this by $193(5,8)$. The second and third maximum subarrays are $54(1,2)$ and $42(11,12)$. The fourth is $-41(3,3)$, so $\bar{K}=3$.

A trivial solution may be based on the repeated application of Kadane's algorithm (Algorithm 1) [18, 19]. When the first maximum subarray is found in $O(n)$ time, we replace the element values within the solution with $-\infty$. The second and subsequent maximum subarrays are found by repeating this process. This is $O(Kn)$ time. Ruzzo and Tompa's algorithm [82] takes $O(n)$ time for $\bar{K}$ disjoint maximum subarrays, but requires sorting if Definition 4.2.2 needs to be met.

## 4.3 Ruzzo and Tompa's Algorithm

For a given input array $a[1..n]$, we first compute the prefix sum array $sum[1..n]$. The data structure used by this algorithm is a simple linked list of $I_1, I_2, ..I_{k-1}$, where each node is called an *interval.* A single-linked list, where a node points to the previous node, is sufficient. Each interval $I_j$ is a candidate for the final solution set of $\bar{K}$ disjoint maximum subarrays. We repeatedly merge intervals if we can have a larger sum by doing so.

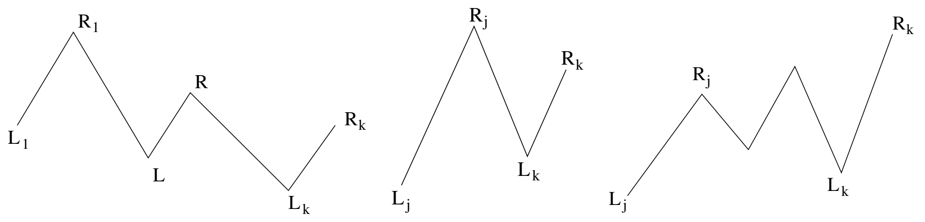
An interval $I_j$ maintains two variables, $from$ and $to$. Each value associated with $I_j$ is denoted with a subscript $j$, such as $from_j$ or $to_j$. These two variables represent a subarray $(from_j, to_j)$. The sum of contained elements can be obtained by $sum[to_j] - sum[from_j - 1]$.

We can derive two extra values $R$ and $L$ specific to each interval, such that $R_j = sum[to_j]$ and $L_j = sum[from_j - 1]$. Note that $R_j - L_j = sum[to_j] - sum[from_j - 1]$.

On termination of the algorithm, each interval in $I_1..I_{\bar{K}}$ represents each disjoint maximum subarray.

**Algorithm 25** Ruzzo, Tompa's algorithm for $\bar{K}$-unsorted disjoint maximum subarrays

```
procedure examine() begin
 1: From $k$, search for the first $j$ satisfying $L_j < L_k$
 2: if no such $j$ exists then return  //Case (a)
 3: else // $L_j < L_k$ for some $j < k$
 4:    if $R_j \geq R_k$ then return  //Case (b)
 5:    else //Case (c)
 6:       /* Merge intervals $I_j ... I_k$ into $I_j$ */
 7:       Set $R_j \leftarrow R_k$, $to_j \leftarrow to_k$ and remove $I_{j+1} ... I_k$. Set $k \leftarrow j$
 8:       examine()
 9:    end if
10: end if
end
/*main procedure*/
11: $k \leftarrow 0$
12: for $i \leftarrow 1$ to $n$ do
13:    if $a[i] > 0$ then
14:       $k \leftarrow k + 1$
15:       Create $I_k$, having $from_k \leftarrow i$, $to_k \leftarrow i$,
          $L_k \leftarrow sum[i-1]$, $R_k \leftarrow sum[i]$
16:       examine()
17:    end if
18: end for
```



(a) No $L_j < L_k$ (b) $L_j < L_k$ but $R_j \geq R_k$ (c) $L_j < L_k$ and $R_j < R_k$

Figure 4.1: Three cases in *examine*()

Inside the loop (lines 12–18) in *main* procedure of Algorithm 25, we process each *positive* array element. On occurrence of a positive element $a[i]$, a new interval $I_k$ is created such that it represents a subarray including only this element*. Then the *examine()* procedure is called. It goes through the previous nodes in the list, searching for $I_j$ whose $L$ value, $L_j$, is smaller than $L_k$ (line 1). Note that if $L_j < L_k$ and $R_j < R_k$ (Case (c)), the sum of interval $(from_j, to_k)$ is larger than any individual sum of $I_j..I_k$. Hence, $(from_j, to_k)$ is *maximal*, in a sense that it can not be lengthened or shortened without reducing its sum. In such a case, we merge intervals $I_j...I_k$ into $I_j$ and remove $I_{j+1}..I_k$ as per line 7. Once the intervals are merged, we now have $k(=j)$ intervals left in the list. We keep examining if further merge can be made (line 8). Since we have to visit all intervals in the list in the worst case, the examine procedure alone accounts for $O(n)$ time. Then the total time of Algorithm 25 is $O(n^2)$.

At the $i$-th iteration, due to line 2 of *examine*() procedure, if there is no $j$ satisfying $L_j < L_k$ (Case (a)), we know that $L_k = \text{MIN}\,\{L_0, L_1, ..L_{k-1}\}$.

Now let us denote current $k$ by $k_0$ for further discussion. Suppose certain time has passed and we have just created a node $I_k$ $(k > k_0)$. We scan the list backwards to find a $L_j$ that is smaller than $L_k$. Note that when the search ever arrives at $I_{k_0}$ and examines $L_{k_0}$, the search may stop here, as we already know that $L_{k_0}$ is smaller than all preceding $L$ values, $L_0, L_1, ..L_{k_0-1}$.

In general, once the "if" condition at line 2 is met, no future search will

* Alternatively, we can initialize $I_1$, $I_2$, etc. to be positions of consecutive positive elements without going through *examine()*. This, however, does not improve the complexity

require a visit to $I_1, ..I_{k-1}$. Hence these nodes need not to be maintained in the list. This observation allows us to modify the algorithm as follows.

```
2: if no such $j$ exists then   //All $I_1...I_{k-1}$ are maximal
       Output and remove $I_1..I_{k-1}$ from the list. Set $k \leftarrow 1$  //$I_k$ becomes $I_1$
   end if
```

Even if we proceed to line 3 by locating $j$ satisfying $L_j < L_k$, if the condition $R_j < R_k$ at line 4 is not satisfied (Case (b)), we do nothing and *examine*() terminates. We can not find any node prior to $I_k$ that can be merged with $I_k$, so $I_k$ stands as it is. In such a case, the time spent for locating $L_j$ is wasted.

However, we can keep this information for the future use. Let $j_0$ and $k_0$ be the current $j$ and $k$ for further discussion. Suppose certain time has passed and we have just created a node $I_k$ ($k > k_0$). From $I_k$, we scan the list backwards, and suppose we have found that all $L_{k_0+1}, ...L_{k-1}$ are not smaller than $L_k$ and we arrive at $I_{k_0}$. If $L_{k0}$ is still not smaller than $L_k$, we need to scan the list further backwards. However, earlier we have already found that $I_{j_0}$ is the first node whose $L$ is smaller than $L_{k_0}$. Hence we can jump from $I_k$ to $I_{j_0}$ without needing to revisit intervening nodes sequentially. If we have set an additional link from $I_{k_0}$ to $I_{j_0}$, this "jumping" or "bypassing", is achievable. Let us refer to this additional link as *bypass link*, and we replace line 1 and line 4 of Algorithm 25 by the following.

```
1: From $k$, search for the first $j$ satisfying $L_j < L_k$. Follow the bypass link
   if available
4:     if $R_j \geq R_k$ then Create a bypass link from $I_k$ to $I_j$
```

Ruzzo and Tompa pointed out that each item is examined only a bounded number of times after these modifications. While the details of it was not fully described in their original literature, an amortized analysis can be given as follows.

**Lemma 4.3.1.** *Algorithm 25 runs in $O(n)$ time.*

*Proof.* Let us define,

- $t(k)$: actual time for adding $I_k$. Counted by the number of intervals visited during the backward search.

- $a(k)$: amortized time for adding $I_k$

- $d(k)$: the total pointer distance from $I_k$ to $I_1$ after adding $I_k$

If there is a bypass link between $I_k$ and $I_j$, we consider that the distance between these two intervals is 1. For example, if there is a bypass link between $I_k$ and $I_1$, $d(k)$ is therefore 1. During the search for $L_j$ that satisfies $L_j < L_k$, the bypass link is used if available.

If $L_k$ finds $L_j$ such that $L_j < L_k$, we either merge them, or set up bypass link between $I_k$ and $I_j$. Suppose, we spent $t(k)$ time to locate such a $L_j$ by visiting $t(k)$ entries between $I_j ... I_k$. This search uses bypassing links if available.

Let $d(k)$ be the *potential*, and the change of potential, $d(k) - d(k-1)$ be denoted by $\Delta d(k)$. The following equation holds.

$$\Delta d(k) = d(k) - d(k-1) = a(k) - t(k) \tag{4.1}$$

Weiss [102] gives an intuitive *savings account* analogy. Suppose the current balance in a savings account is $d(k)$ for the $k$-th month. We budgeted $a(k)$ for monthly spending for the $k$-th month but actual expenditure was $t(k)$ during that month. If $a(k) - t(k) > 0$, we spent less than planned, and the balance in the bank account increases. Otherwise the balance is reduced. The change of balance, $\Delta d(k)$, is equal to the extra amount saved (or wasted). Hence, $\Delta d(k) = a(k) - t(k)$. The term, "potential" is very appropriate as money saved in the bank translates to the "purchasing power for the future".

Let us examine the change of potential, $\Delta d(k)$ on addition of the new interval $I_k$.

**Case (a)**: When there is no $L_j$ that satisfies $L_j < L_k$, the search needs to traverse the list all the way back to $I_1$. Hence $t(k)$, the number of intervals visited during the search, is equal to $d(k-1)$. Due to modification to line 2 of Algorithm 25, the list then outputs all the intervals $I_1 .. I_{k-1}$, leaving only

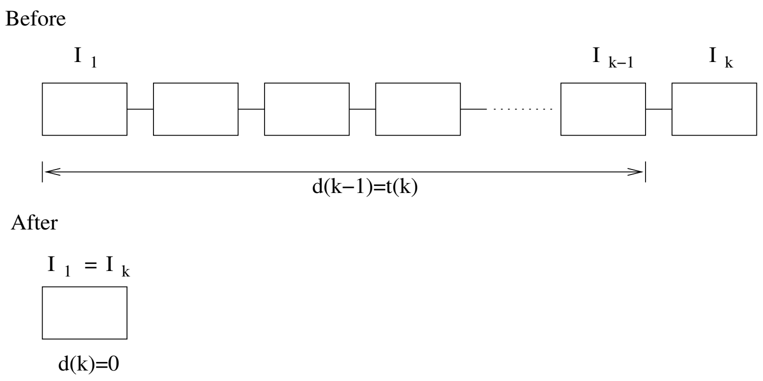


Figure 4.2: Change of the Total Pointer Distance: Case (a)

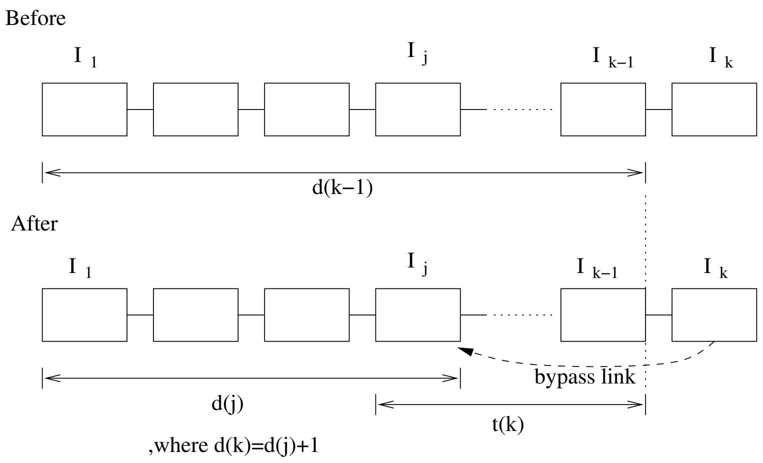


Figure 4.3: Change of the Total Pointer Distance: Case (b)

$I_k$, resulting in $d(k) = 0$. Then,

$$\Delta d(k) = d(k) - d(k-1) = -t(k)$$

**Case (b)**: When a bypass link is set up from $I_k$ to $I_j$, $t(k)$ is the pointer distance between $I_j$ and $I_{k-1}$, that is $d(k-1) - d(j) + 1$. After the bypass link is made, $d(k)$ is equal to $d(j)+1$. Since $d(k-1) = d(j) + d(k-1) - d(j)$, we have,

$$\Delta d(k) = d(k) - d(k-1) = 2 - t(k)$$

Note that $d(k) > d(k-1)$ is still possible. It happens when the previous entry has $L_{k-1} < L_k$, but $R_{k-1} \geq R_k$. Here, we add a new entry $I_k$ and set

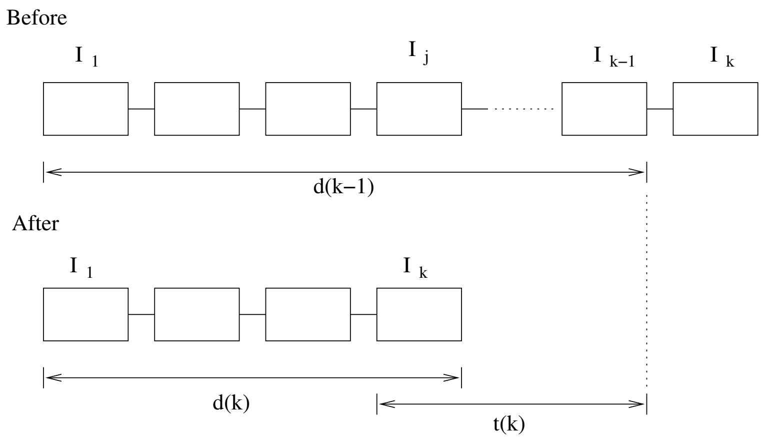


Figure 4.4: Change of the Total Pointer Distance: Case (c)

up a bypass link to $I_{k-1}$, making $d(k) = d(k-1) + 1$. As $t(k) = 1$ in such a case, $\Delta d(k) = 1$ and the above holds.

**Case (c)**: When $I_j..I_k$ are merged, $t(k)$ is the pointer distance between $I_j$ and $I_{k-1}$. After merge, $d(k)$, $d(k-1)$ and $t(k)$ satisfy the following relation.

$$\Delta d(k) = d(k) - d(k-1) = 1 - t(k)$$

Let $N$ be the number of positive element in the input array of size $n$. Due to line 13, we create $N$ intervals. From (4.1), we derived that,

$$a(k) = t(k) + \Delta d(k)$$

We compute $A$ and $T$, the total amortized and actual time for all $k = 1..N$.

$$A = T + (d(1) - d(0)) + (d(2) - d(1)) + ... + (d(N) - d(N-1))$$

, which is summarized to

$$T = A + d(N) - d(0)$$

Notice that $a(k)$ is 0,1 or 2 based on the analysis for three cases. Hence

$A \leq 2N$. The initial total pointer distance $d(0)$ is 0 and $0 \leq d(N) \leq N$. The number of positive elements, $N$, is obviously not greater than $n$. Therefore $T = O(n)$. □

Let us note that, to be precise, there are subtle differences in the problem definition. Firstly, as briefly mentioned above, Ruzzo and Tompa intend to find "all positive" maximum subarrays without specifying $K$. Taking only positive maximum subarrays into account is reasonable since a subarray with a negative sum will consist of a single negative element, which is trivial. Note that for a one-dimensional version of the problem, the total number of all maximum subarrays is bounded by $n/2$. This case arises when the input array consists of alternating positive element and negative infinity. The other extreme is where all array elements are positive. There will be only one maximum subarray in such a case, the input array itself. Thus the total number of the maximum subarrays is input-dependent. Let $\bar{K}$ stand for the total number of maximum subarrays for later discussion.

Secondly, these all maximum subarrays found by Ruzzo and Tompa's algorithm are not in a particular order. To meet Definition 4.2.2, a fast sorting algorithm may be applied after running Algorithm 25 and Algorithm 13 to select the $K$-th largest ($K \leq \bar{K}$) among $\bar{K}$ maximum sums. This results in the total of $O(n + K \log K)$ time. In the context of our problem definition, Ruzzo and Tompa's problem can be interpreted as that of "$\bar{K}$-unsorted disjoint maximum subarrays".

## 4.4 A Challenge: $K$-DMSP in Two-Dimensions

For an $(m, n)$ array, $a[1..m][1..n]$, we wish to find $K$ disjoint maximum subarrays which are in rectangular shape. We denote a subarray of sum $x$ with coordinates of top-left corner $(g, h)$ and bottom-right corner $(i, j)$ by $x(g, h)|(i, j)$. In the following example, we compute $K = 4$ disjoint maximum subarrays in the array shown in Figure 4.5.

**Example 4.4.1.** For $K = 4$, $K$ disjoint maximum subarrays are $21(3, 2)|(4, 3)$, $13(1, 4)|(2, 4)$, $7(1, 1)|(2, 1)$ and $1(4, 1)|(4, 1)$.

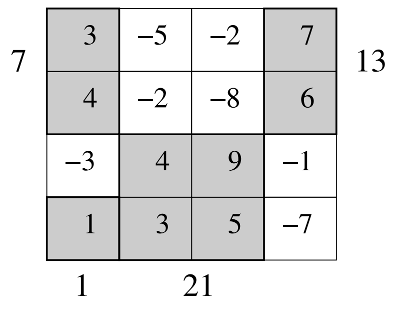


Figure 4.5: Example: $K = 4$ disjoint maximum subarrays

In this example, $\bar{K} = 4$. When $K > 4$, the subsequent subarrays will be comprised of a single negative array element such as $-1(3,4)|(3,4)$, $-2(1,3)|(1,3)$ etc.

In two-dimensions, a slight delicacy arises if there are multiple subarrays with the same sum. Consider Figure 4.6, which illustrates this issue. In the given array, there are 3 subarrays with the maximum sum 4. If $4(3,1)|(4,2)$ is taken as the first maximum subarray, the second and the third maximum subarrays would be of the sum 3. On the other hand, if $4(4,1)(4,4)$ is chosen as the first maximum subarray, the second subarray will have the sum 3, but the third will have the sum 2 etc.

Each subsequent maximum subarray is computed from the remaining portion of the original array after excluding the area occupied by the previous maximum subarrays. Note that a different decision at each round can lead to quite a different final result.

This problem does not occur in one-dimensional case, as long as we select the one with the shorter length if there is a tie. A similar tie-breaking scheme can be made available. The first tie-breaker is the area, and the one with smaller area wins. However, in the figure, both $4(3,1)|(4,2)$ and $4(4,1)|(4,4)$ have the same area and we need a more specific tie-breaker. In such a case, we may judge that the one with smaller row (or column) size wins. If the tie is still not broken, we randomly choose the winner. When such a tie-breaking scheme is applied, the maximum subarrays will be computed as Figure 4.7. Note that the proposed tie-breaking scheme will ensure that we have the consistent sequence of $K$ maximum "sums", but

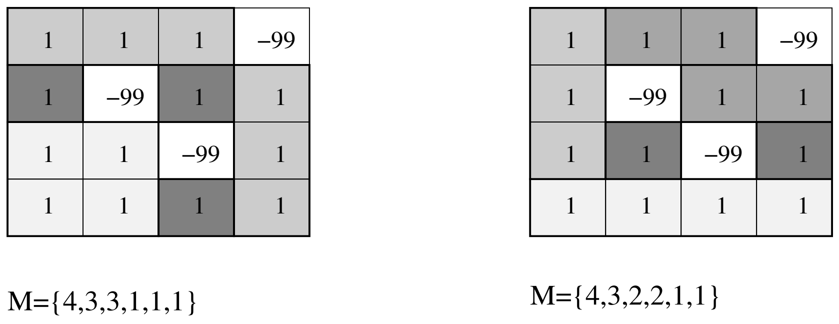


Figure 4.6: Inconsistent maximum sums with no tie-breaking scheme

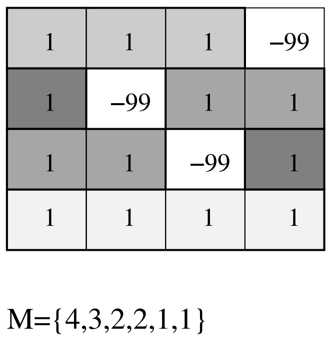


Figure 4.7: Maximum sums when a tie-breaking scheme is applied.

it does not guarantee the consistent sequence of $K$ maximum "subarrays". For example, in the given array, the third maximum sum is guaranteed to be 2, but the third maximum subarray, $M[3]$, will be randomly decided between $2(2,3)|(2,4)$ and $2(3,1)|(3,2)$. At least, we can be assured that the given array will always return the same sequence of $K=6$ maximum sums, $M=\{4,3,2,2,1,1\}$.

As is in one-dimension, a trivial solution for finding $K$ disjoint maximum subarrays can be easily made based on the repeated application of Kadane's algorithm. This is $O(Km^2n)$ time or $O(Kn^3)$ time for $m=n$. In the worst case, where $\bar{K}=n^2/2$, we have $O(n^5)$ time for $K=\bar{K}$. If we apply the sub-cubic algorithm [94, 89], whose complexity is, say, $M(n)$, we have $O(KM(n))$ time by the repeated application. Using the best known distance matrix multiplication algorithm by Takaoka [91] as an algorithmic engine for [94, 89], $M(n)=O(n^3\log\log n/\log n)$. If $K$ is small, such that $K\leq\log n/\log\log n$,

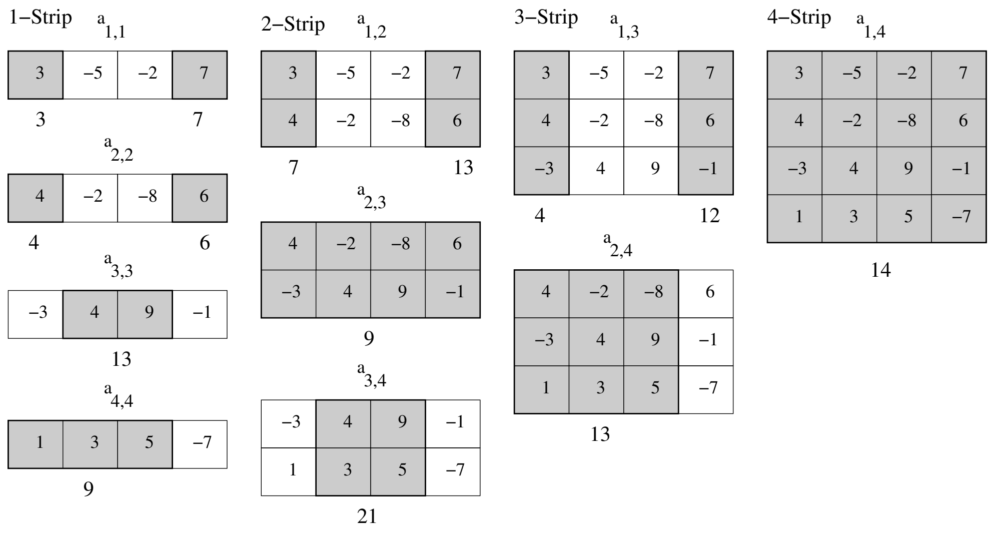


Figure 4.8: Computing maximum subarrays in each strip

$O(KM(n))$ is sub-cubic in terms of $n$.

For more efficient solution, it is natural to consider extending Ruzzo and Tompa's algorithm presented in Section 4.3. It would be tempting to use a simple technique based on strip separation such that Ruzzo and Tompa's algorithm is applied strip-wise. This idea, however, does not work. Figure 4.8 illustrates a typical pitfall. For an input array given in Figure 4.5, Algorithm 25 is applied strip-wise and identifies all disjoint maximum subarrays in each strip. Note that a maximum subarray obtained from a strip $a_{g,i}$ has a row-wise size $(i - g + 1)$ as $a_{g,i}$ produces a subarray spanning between row $g$ and row $i$. It is observed that disjoint maximum subarrays in one strip can still overlap solutions in other strips. For example, 21 in strip $a_{3,4}$ overlaps 9 in $a_{2,3}$, 13 in $a_{3,3}$, 13 in $a_{2,4}$ and 14 in $a_{1,4}$. Also notice that the fourth maximum subarray in Figure 4.5, $1(4,1)|(4,1)$, is missing in Figure 4.8. It is only included as a part of $9(4,1)|(4,3)$ in $a_{4,4}$.

This example illustrates the difficulty involved in computing the 2D $K$-DMSP strip-wise. In the 2D $K$-OMSP, the only factor that determines the final $K$ solutions was the sum of a subarray, hence it was possible to compute strip-wise and select the final $K$ maximum values from the collection of solutions without concerning of their locations.

**Algorithm 26** Maximum subarray for one-dimension

```
1: If the array becomes one element, return its value.
2: Let $M_{left}$ be the solution for the left half.
3: Let $M_{right}$ be the solution for the right half.
4: Let $M_{center}$ be the solution for the center problem.
5: $M \leftarrow \text{MAX}\{M_{left}, M_{right}, M_{center}\}$.
```

In the $K$-DMSP, however, the location of each subarray imposes an extra decision criteria. Hence, the simple approach based on strip separation and strip-wise computation is erroneous. This observation also automatically invalidates the sampling technique discussed in Section 3.7.1.

Alternatively, one may consider modifying Algorithm 25, such that it could process a 2D array as a whole without separating strips. This option, however, seems difficult to formulate. The first difficulty may involve the direction of the scanning. The scanning in the original algorithm is performed in one direction, backwards. In 2D, it will need to be performed both horizontally and vertically. It is expected to involve some difficulty to organize two directional scanning and to decide when to go left and when to go up. Secondly, when some of current solutions are merged to obtain a better solution, the shape of the merged area must be predicted. In 1D, the result of merge is guaranteed to be a legitimate subarray. In 2D, a subarray must be rectangular shaped. Resolving these two difficulties is expected to be challenging, if not impossible.

In the following section, we present another algorithm for one-dimension. This algorithm provides solid framework to extend to the two-dimensional case.

## 4.5 New Algorithm for 1D

For a one-dimensional array $a[1..n]$, we compute the prefix sum $s$ such that $sum[x] = \sum_{i=1}^{x} a[i]$. We assume $sum[0] = 0$. Note that the prefix sums once computed are used throughout recursion.

We revisit Lemma 2.1.4 and present Algorithm 26 to show the outer framework. In this algorithm, the center problem is to obtain an array portion that crosses over the central point with maximum sum, and can be solved in the following way. We assume that $n$ is power of 2 without loss of generality.

$$M_{center} = \underset{\substack{n/2<i\leq n \\ 0\leq j<n/2}}{\mathrm{MAX}} \{sum[i] - sum[j]\} = \underset{n/2<i\leq n}{\mathrm{MAX}} \{sum[i]\} - \underset{0\leq j<n/2}{\mathrm{MIN}} \{sum[j]\} \tag{4.2}$$

The recursive computation of this algorithm can be conceived as a tournament-like selection process, which we describe in the following.

### 4.5.1 Tournament

In Section 2.3.1, we described a binary tree, *tournament*, whose features are similar to that of a heap. It was regarded as a conceptual representation of a recursive routine that computes the maximum(or minimum) of a given list of elements. While actual building of a tree is not necessary when only the first maximum is computed, actual tree-building gives an effective platform to retrieve the subsequent maximum values in order.

To compute the $K$-DMSP, a tournament is again a data structure of choice. In Section 3.6.2, this data structure was employed to maintain the minimum prefix sum. For that task, it was a favored data structure, as it is particularly effective to rank items and to learn the original position of the selected item. Both needs cannot be met by other data structures, even a heap, a close relative of the tournament.

We explore a new framework based on Algorithm 6. If an actual tournament is built while the first maximum subarray is recursively computed, we could utilize the tournament as a basis to retrieve the subsequent maximum subarrays.

We build a modified version of the tournament to compute the maximum subarray problem. Each node contains attributes defined as follows.

**Definition 4.5.1.** A node contains,

- $(from,to)$: the coverage, i.e., the range of elements covered by this node.

**Algorithm 27** Build tournament for $a[f..t]$

```
procedure buildtree($node, f, t$) begin
 1: $(from, to) \leftarrow (f, t)$
 2: if $from = to$ then
 3:     $(L, M, G) \leftarrow (sum[f-1], a[f], sum[f])$ // $node$ is a leaf
 4: else // $node$ is an internal node
 5:     create two children $left$ and $right$
 6:     buildtree($left$,$f$,$\frac{f+t-1}{2}$) //build left subtree
 7:     buildtree($right$,$\frac{f+t+1}{2}$,$t$) //build right subtree
 8:     $L \leftarrow$ MIN$\{left.L, right.L\}$, $G \leftarrow$ MAX$\{left.G, right.G\}$
 9:     $M \leftarrow$ MAX$\{M_{left}, M_{center}, M_{right}\}$ where $M_{center} \leftarrow right.G - left.L$
10: end if
end
```

- $L$: the minimum prefix sum within $(from - 1, to - 1)$. Abbreviates "least".

- $G$: the maximum prefix sum within the coverage. Abbreviates "greatest".

- $M$: the maximum sum found within the coverage. Refer to Lemma 4.5.3.

We specify an attribute of node $x$ with a dot, such as $x.L$. Attributes of two children $left$ and $right$ of node $x$ are denoted by $left.L$ etc. Note that, however, we use $M_{left}$ instead of $left.M$ following the notation in Algorithm 26.

Throughout this thesis, when we use a term "tournament", it is assumed that this modified tournament is being discussed. The tournament will be simply referred to as $T$ and the root of $T$ will be denoted by $root(T)$. Conversely, the tree rooted at a node $x$ is denoted by $tree(x)$.

Based on Algorithm 26, we design Algorithm 27 that recursively builds $T$. Note that the computation of $M_{center}$ at line 9 is due to Eq. 4.2. We assume that the fraction at lines 6 and 7 are truncated to take the closest integer. After *buildtree(root,1,n)* is processed, the value of $M$ at $root(T)$ is the maximum sum in the array $a[1..n]$.

The coverage of a node, $(from, to)$ is a union of those of its two children, namely $(from, to) = (left.from, right.to)$. We let $M$ carry two indices such as $M.from$ and $M.to$ to indicate that $M$ is the sum of array elements $a[M.from]..a[M.to]$. Also $L$ and $G$ have an index $to$. If $M_{center}$ is chosen for $M$, we set $M.from = left.L.to + 1$ and $M.to = right.G.to$.

**Lemma 4.5.2.** *Algorithm 27 computes L and G correctly for any node x.*

*Proof.* (by induction)
Basis. Node $x$ is a leaf. Then $x.L = sum[from - 1]$ and $x.G = sum[to]$
Induction step. Suppose $left$ and $right$, two children of $x$ satisfy the lemma.

$$left.L = \underset{left.from-1 \le i \le left.to-1}{\text{MIN}} \{sum[i]\}$$
$$right.L = \underset{right.from-1 \le i \le right.to-1}{\text{MIN}} \{sum[i]\}$$

Since $left.to = right.from - 1$, $from = left.from$ and $to = right.to$,

$$\begin{aligned} x.L &= \text{MIN}\{left.L, right.L\} \\ &= \underset{left.from-1 \le i \le right.to-1}{\text{MIN}} \{sum[i]\} \\ &= \underset{from-1 \le i \le to-1}{\text{MIN}} \{sum[i]\} \end{aligned}$$

Proof for $x.G$ is similar. □

The following two facts are easily observed. Proofs are omitted.

**Lemma 4.5.3.** *When a node x has coverage $(from, to)$, its M is the maximum sum that can be obtained within this coverage and $a[M.from..M.to]$ is the maximum subarray.*

**Lemma 4.5.4.** *The maximum subarray of $a[1..n]$ is M at $root(T)$.*

When there is a tie during computation, such as $left.L = right.L$, we select the one that will result in $M$ with smaller physical size. For example, when $left.L = right.L$, we select $right.L$ as $M_{center}$ will have smaller physical

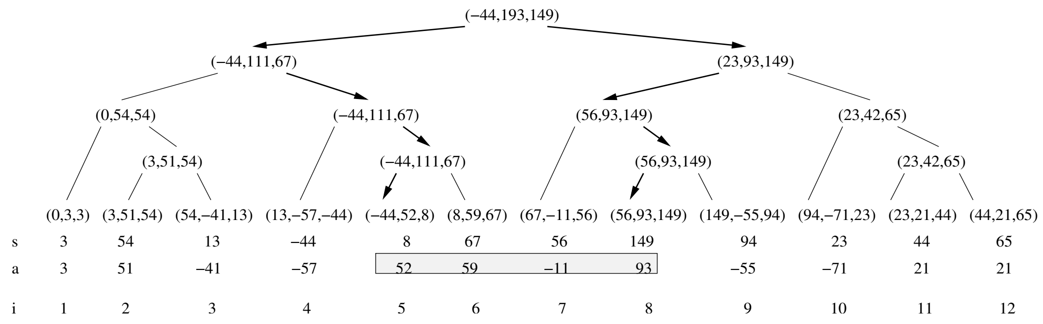


Figure 4.9: Tournament $T$

size by subtracting $right.L$. Similarly, when $left.G = right.G$, we select $left.G$. For the same reason, we choose the one with smaller physical size in computing $M = \text{MAX}\{M_{left}, M_{right}, M_{center}\}$.

Figure 4.9 shows the example in Section 4.2 computed by the tournament. Each node shows a 3-tuple of $(L, M, G)$. The value of $M$ at root, 193 represents the maximum sum. The figure omits the location ($M.from$,$M.to$) which is $(5, 8)$.

### 4.5.2 Finding the next maximum sum

We discuss how we compute the next maximum subarray that is disjoint from the previous ones.

For efficient computation, we attempt to obtain the next solution from the existing tree. This needs a careful maintenance of the tree such that the computation of the next solution will not be interfered by the previous solutions. We use a term *hole* to refer to the range that the next solution should exclude. Typically the hole is given as $(holeBegin, holeEnd)$. Note that when the $k$-th maximum subarray is to be computed, there are $(k-1)$ holes in the tree. We also use *h-node* to refer to a node whose coverage overlaps a hole. We regard an h-node is *inside* the hole when the coverage of an h-node is contained within the range of a hole. On the other hand, when the coverage of an h-node completely surrounds whole range of a hole, we regard this h-node *encompasses* the hole.

It may be tempting to delete h-nodes inside the hole and process the

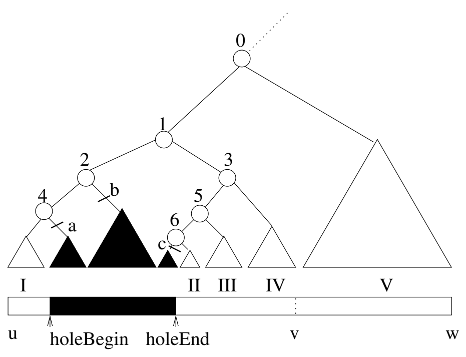


Figure 4.10: Subarray deletion

remaining tree, expecting that the root of the updated tree will provide the next maximum subarray. This simple approach, however, does not compute correctly. To illustrate this, let us consider Figure 4.10, where dark subtrees are rooted at such deleted h-nodes.

We trace the tree from the root to locate the h-nodes inside the hole and delete them. Deletion can be performed by simply removing the link $a$, $b$ and $c$. After the deletion, some nodes may have only one child. Note that an internal node should have at least one child. It is because losing both children means this node is *inside* the hole, and this node itself must have been deleted.

Nodes 2,4,6 are those having one child in the figure. When a node has one child missing, we assume that this node receives a 3-tuple $(L, M, G) = (\infty, -\infty, -\infty)$ from the missing child.

The maximum subarray in the range of $(u, v)$ is determined by node 1. We want it to be disjoint from the hole. If $M_{center}$ becomes $M$ at node 1, we have $(M.from, M.to) = (left.L.to + 1, right.G.to)$. This $M$ is a "superarray" of the hole as its range surrounds the hole. In general, an h-node encompassing the hole can "potentially" produce $M_{center}$ overlapping the hole. Node 0 is another h-node that has such potential, however, if $left.L$ comes from region II,III or IV, $M_{center}$ at node 0 can be disjoint from the hole. So we can not simply disable computing $M_{center}$ at such h-nodes. It further implies that a simple "node deletion and process" strategy fails to compute the correct next maximum subarray.

As described, an h-node encompassing the hole potentially produces $M_{center}$ being a superarray of the hole. Specifically, this problem occurs when $M_{center}$ is computed by coupling $L$ from the left to the hole and $G$ from the right to the hole. Hence those $L$ and $G$ values must be blocked from being delivered upwards.

Suppose a node $x$ is an h-node, and there are some holes in its leaf level. Let $holeBegin_l$ be the left boundary of the leftmost hole and $holeEnd_r$ be the right boundary of the rightmost hole. In the following discussion, we consider $(from, to)$ being the coverage of node $x$. $L$ and $G$ at the node $x$ should satisfy the following.

$$x.L = \underset{holeEnd_r \le i \le to-1}{\mathrm{MIN}} \{sum[i]\}$$

$$x.G = \underset{from \le i \le holeBegin_l - 1}{\mathrm{MAX}} \{sum[i]\}$$

We can establish a general definition of $L$ and $G$ that holds regardless of the type of node $x$.

**Definition 4.5.5.**

$$x.L = \underset{lb \le i \le to-1}{\mathrm{MIN}} \{sum[i]\}, \text{ where } lb = \mathrm{MAX}\{from - 1, holeEnd_r\}$$
$$x.G = \underset{from \le i \le rb}{\mathrm{MAX}} \{sum[i]\}, \text{ where } rb = \mathrm{MIN}\{to, holeBegin_l - 1\}$$

Here, $lb$ and $rb$ stand for the left boundary and right boundary respectively. If $lb \ge to$, there is no $i$ that satisfies the given range. Then $x.L = \infty$. Similarly, if $rb < from$, $x.G = -\infty$.

We now devise how we compute $L$ and $G$ satisfying the definition above. We introduce an extra Boolean value "*hole*" to each node. This attribute at a node $x$ indicates if $x$ is an h-node. Suppose a node $x$ has two children $left$ and $right$.

Let $P(x)$ be the following.

(A). If $x$ is a leaf (i.e. $from = to$), Let $left.L = sum[from - 1]$, $right.G = sum[to]$ and $x.L = left.L$, $x.G = right.G$.

Otherwise,

(B).

$$x.L = \begin{cases} right.L, & \text{if } right.hole = true \\ \text{MIN}\{left.L, right.L\}, & \text{otherwise} \end{cases}$$

(C).

$$x.G = \begin{cases} left.G, & \text{if } left.hole = true \\ \text{MAX}\{left.G, right.G\}, & \text{otherwise} \end{cases}$$

If the node $x$ is inside a hole, we set $x.L \leftarrow \infty$ or $x.G \leftarrow -\infty$.

**Lemma 4.5.6.** *$P(x)$ computes correct $x.L$ and $x.G$ for any node $x$*

*Proof.* (by induction)
BASIS. $x$ is a leaf. A leaf with $x.hole = true$ makes $x$ inside the hole. Thus we have $x.L = \infty$ and $x.G = -\infty$. When $x.hole = false$, $x.L = sum[from - 1]$ and $x.G = sum[to]$.
INDUCTION STEP. Suppose $left$ and $right$, the two children of $x$ satisfy lemma.

If $x.hole = false$, both $right.hole$ and $left.hole$ are $false$. The subtree $tree(x)$ is free of a hole and $x.L$ is computed by MIN $\{left.L, right.L\}$. This is correct due to Lemma 4.5.2.

If $x.hole = true$, $right$ or $left$ is an h-node, or both are.

If $right.hole = true$, let the rightmost hole have a range ($holeBegin$, $holeEnd$). Then,

$$x.L = right.L = \underset{holeEnd \le i \le right.to-1}{\text{MIN}} \{sum[i]\} = \underset{holeEnd \le i \le to-1}{\text{MIN}} \{sum[i]\}$$

Otherwise, i.e. $right.hole = false$, the hole should be in the left subtree. Let the rightmost hole in the left subtree have a range ($holeBegin, holeEnd$).

Then,

$$\begin{aligned} x.L &= \text{MIN}\{left.L, right.L\} \\ &= \text{MIN}\left\{ \underset{holeEnd \le i \le left.to-1}{\text{MIN}} \{sum[i]\}, \underset{right.from-1 \le i \le right.to-1}{\text{MIN}} \{sum[i]\} \right\} \\ &= \underset{holeEnd \le i \le right.to-1}{\text{MIN}} \{sum[i]\} = \underset{holeEnd \le i \le to-1}{\text{MIN}} \{sum[i]\} \end{aligned}$$

, which proves (B) of $P(x)$. (C) can be similarly proved. □

**Lemma 4.5.7.** *The maximum sum $M$ for any node $x$ is correctly computed by,*

$$M = \text{MAX}\{M_{left}, M_{center}, M_{right}\}, where\ M_{center} = right.G - left.L$$

*Proof.* (by induction)
Basis. $x$ is a leaf, where $M_{left}$ and $M_{right}$ are not available. If $x$ is not an h-node, $M = M_{center} = right.G - left.L = a[from(= to)]$. If $x$ is an h-node, $M = M_{center} = -\infty - \infty = -\infty$.
Induction step. Suppose lemma is correct for $left$ and $right$, two children of $x$. There are three possibilities of the range of $M$.

(1). The range is completely in $left$.

(2). The range is completely in $right$.

(3). Stretches over left and right subtrees.

The first two cases result in a correct computation of $M$ due to the assumption. For the third case, $M$ stretching over both subtrees implies $M = M_{center}$. When $x$ is not an h-node, neither $left$ nor $right$ is an h-node and $M_{center}$ is computed by $right.G - left.L$ correctly. Otherwise, (3a) $left$ is an h-node or (3b) $right$ is, or (3c) both are. For case (3a), we have $left.hole = true$ but not $right.hole$. Due to Definition 4.5.5,

$$left.L = \underset{holeEnd \le i \le left.to-1}{\text{MIN}} \{sum[i]\}$$

, and the resulting $M_{center}$ is disjoint from the hole. Case (3b) can be shown similarly. Note that either $left.L = \infty$ or $right.G = -\infty$ means $M_{center} = -\infty$ implying that $M_{center}$ is not taken for the final value of $M$. This is then case (1) or (2) above.

Case (3c) implies the hole does not stretch over both subtrees. If a hole stretches over, we have $right.G = -\infty$ and $left.L = \infty$ due to Definition 4.5.5 as $lb > to - 1$ and $rb < from$, resulting in $M_{center} = -\infty$. Then it becomes case (1) or (2) above. So we consider that $left$ and $right$ have separate holes, which do not stretch over. Let the rightmost hole in $left$ be $(holeBegin_r, holeEnd_r)$ and the leftmost hole in $right$ be $(holeBegin_l, holeEnd_l)$. Due to Definition 4.5.5,

$$\begin{aligned} left.L &= \underset{holeEnd_r \le i \le left.to-1}{\text{MIN}} \{sum[i]\} \\ right.G &= \underset{right.from \le i \le holeBegin_l - 1}{\text{MAX}} \{sum[i]\} \end{aligned}$$

The resulting $M_{center}$ lies in $(holeEnd_r + 1, holeBegin_l - 1)$, which is disjoint from both holes. □

We design Algorithm 28 based on $P(x)$ above. This algorithm recursively visits h-nodes that belong to $(holeBegin, holeEnd)$ and updates them. As non h-nodes are irrelevant, the recursion terminates if *node* is disjoint from the hole. An h-node inside the hole are not explicitly deleted. Instead, we perform "pseudo-deletion" by setting *hole* attribute and $(L, M, G) \leftarrow (\infty, -\infty, -\infty)$. By terminating the recursion after this action, we consequently have the same effect as deleting such a node. Note that no special handling is needed for a leaf. A leaf is either disjoint from the hole or inside the hole. These two cases are handled by lines 1 and 2-3 respectively. If the currently visiting node has been "pseudo-deleted" already, we may terminate the recursion as per line 4.

If we update the tree in Figure 4.9 with a hole $(5, 8)$ by Algorithm 28, the second maximum subarray $54(1, 2)$ is obtained from the root.

**Algorithm 28** Update tournament $T$

```
procedure update(node, holeBegin, holeEnd) begin
//Execute by update(root(T), holeBegin, holeEnd)
//Attributes $L, M, G$ and hole are local to node.
//left and right are two children of node.
1: if the coverage of node is disjoint from the hole (non h-node) then return
2: if the coverage of node is completely inside the hole then
3:    hole ← true; $(L, M, G) \leftarrow (\infty, -\infty, -\infty)$; return
4: if hole = true and $(L, M, G) = (\infty, -\infty, -\infty)$ then return
5: update(left, holeBegin, holeEnd) //updates left subtree
6: update(right, holeBegin, holeEnd) //updates right subtree
7: if right.hole then $L \leftarrow right.L$ else $L \leftarrow \text{MIN}\{left.L, right.L\}$
8: if left.hole then $G \leftarrow left.G$ else $G \leftarrow \text{MAX}\{left.G, right.G\}$
9: if left.hole or right.hole then hole ← true
10: $M \leftarrow \text{MAX}\{M_{left}, M_{right}, M_{center}\}$, where $M_{center} \leftarrow right.G - left.L$
end
```

### 4.5.3 Analysis

We find the first maximum subarray by building $T$ in $O(n)$ time. To compute the next maximum subarray, we regard the previous solution as a hole and run Algorithm 28. The update involves traversing two paths from the root to *holeBegin* and *holeEnd*. Since the height of the tree is $O(\log n)$, the time for the next maximum subarray is bounded by $O(\log n)$. To obtain $K$ disjoint maximum subarrays in sorted order, the total time is therefore $O(n + K \log n)$. Note that $O(n + K \log n) = O(n + K \log K)$ for any integer $K$ due to Lemma 2.3.1. To obtain $K$ disjoint maximum subarrays in sorted order, this is asymptotically equivalent to Ruzzo and Tompa's algorithm [82], which requires extra time for sorting.

As we wish to output $K$ disjoint maximum subarrays in sorted order, a lower bound similar to Lemma 3.6.4 can be established.

**Lemma 4.5.8.** *The lower bound for the 1D K-DMSP is* $\Omega(n + K \log K)$

The proof for Lemma 3.6.4 is equally applicable. Since the lower bound matches the upper bound, we have,

**Theorem 4.5.9.** *Algorithm 27 augmented with Algorithm 28 is optimal*

## 4.6 New Algorithm for 2D

In this section, we extend the algorithm for one-dimension to two-dimensions.

### 4.6.1 Strip Separation

We apply the strip separation technique discussed in Chapter 2. We first compute the prefix sums $sum[1..m][1..n]$ and from them, we retrieve strip prefix $sum_{g,i}[1..n]$ for all pairs of $g$ and $i$, where $1 \leq g \leq i \leq m$. Each strip prefix array $sum_{g,i}[1..n]$ needs $O(n)$ time to build, and there are $m(m+1)/2$ combinations of $g$ and $i$, hence computing all strip prefixes takes $O(m^2n)$ time in total.

### 4.6.2 Finding the first maximum subarray

We apply Algorithm 27 to each strip prefix. The tournament produced from a strip prefix $sum_{g,i}$, is denoted by $T_{g,i}$. The $M$ attribute at the root of $T_{g,i}$, which we denote by $M_{g,i}$, is the maximum subarray that spans from the $g$-th row to the $i$-th row. Since each strip is treated as a one-dimensional problem, the locational information stored in $M_{g,i}$, $M_{g,i}.from$ and $M_{g,i}.to$, are the column addresses. Note that the rows where subarray starts and ends are subject to the strip and fixed to $g$ and $i$ respectively. Thus if $(M_{g,i}.from, M_{g,i}.to) = (h, j)$, the subarray corresponds to $M_{g,i}$ is $(g, h)|(i, j)$.

There are total of $O(m^2)$ tournaments and same number of $M_{g,i}$'s (for $1 \leq g \leq i \leq m$). Among them, the one with the largest sum can be found by a simple iteration, *2dMaxSelect()*.

Finding the maximum through the iteration is an $O(m^2)$ time process. As each tournament is built by Algorithm 27 spending $O(n)$ time, the overall time for finding the first maximum subarray is $O(m^2n)$.

### 4.6.3 Next maximum

Finding the next maximum subarray disjoint from the previous ones involves update operations in the tournament-level and *2dMaxSelect()*. Let us denote the $x$-th maximum subarray by $M(x)$.

**Algorithm 29** Find the maximum subarray of a 2D array

```
procedure 2dMaxSelect() begin
1: $M \leftarrow 0$
2: for $g \leftarrow 1$ to $m$ do
3:     for $i \leftarrow g$ to $m$ do
4:         if $M_{g,i} > M$ then $M \leftarrow M_{g,i}$; $r_1 \leftarrow g$; $r_2 \leftarrow i$
5:     end for
6: end for
7: return $M(r_1, M.from)|(r_2, M.to)$
end
```

Suppose *2dMaxSelect()* computed $M(1)$, the first maximum subarray, that is $M_{g_1,i_1}(g_1, h_1)|(i_1, j_1)$ coming from a tournament $T_{g_1,i_1}$. Here $M_{g_1,i_1}$ corresponds to the subarray $(g_1, M_{g_1,i_1}.from)|(i_1, M_{g_1,i_1}.to)$, thus $h_1 = M_{g_1,i_1}.from$ and $j_1 = M_{g_1,i_1}.to$ here.

Let us consider a tournament $T_{g,i}$. If its vertical coverage $(g, i)$ is disjoint from $(g_1, i_1)$, this tournament definitely produces $M_{g,i}$ that is disjoint from the first maximum. Otherwise, $M_{g,i}$ may possibly overlap $M(1)$.

There are $O(m^2)$ tournaments of such a possibility. By creating a hole $(h_1, j_1)$ in such tournaments and updating the tree as per Section 4.5.2, we ensure that $M_{g,i}$'s at the root of these tournaments are disjoint from $(g_1, h_1)|(i_1, j_1)$.

Now we run *2dMaxSelect()* to select $M(2)$. Subsequent disjoint maximum subarrays are found by repeating these steps. Each maximum subarray is computed by updating $O(m^2)$ tournaments followed by *2dMaxSelect()*, which take $O(m^2 \log n)$ time and $O(m^2)$ time respectively. The latter time is absorbed into the former. For $K$ maxima, it is $O(Km^2 \log n)$ time. Including the time for strip separation, the total time for $K$-disjoint maximum subarrays is $O(m^2 n + Km^2 \log n)$, which is a cubic time for $K \leq \frac{n}{\log n}$.

### 4.6.4 Improvement for $K > n$

It has been stated that $\bar{K}$ can be as large as $mn/2$ in two-dimensions. In such a case, the algorithm above is $O(m^3 n \log n)$ time.

As described in the previous section, on computation of each maximum

**Algorithm 30** Find $K$-disjoint maximum subarrays in 2D

```
1: for $k \leftarrow 1$ to $K$ do
2:     compute $M(k)$ that is $M_{g_k,i_k}(g_k,h_k)|(i_k,j_k)$ returned by 2dMaxSelect()
3:     for $g \leftarrow 1$ to $m$ do
4:         for $i \leftarrow g$ to $m$ do
5:             if not$(i < g_k$ or $g > i_k)$ then create a hole $(h_k, j_k)$ in $T_{g,i}$
6:         end for
7:     end for
8: end for
```

subarray $M^*(g^*,h^*)|(i^*,j^*)$, we update up to $O(m^2)$ tournaments $T_{g,i}$ whose vertical range $(g,i)$ overlaps $(g^*,i^*)$. Each update involves creation of a hole $(h^*,j^*)$ by Algorithm 28 taking $O(\log n)$ time.

Suppose a tournament has already created a hole $(h^*,j^*)$ in the previous computation and now we attempt to create a hole $(h^*,j^*)$ again. Clearly, this hole creation may be skipped. The algorithm above does not consider such a case and will spend another aforementioned $O(\log n)$ time, attempting to create the same hole again.

In general, when we wish to create a hole $(h^*,j^*)$ in a tournament, if there is a hole $(h_0^*,j_0^*)$ that contains $(h^*,j^*)$, we can skip the hole creation process. Let us refer to such a situation as the *skipping condition.*

Each tournament is regarded as a one-dimensional problem. Since $\bar{K}$ for one-dimension is $n/2$, each strip can not have more than $n/2 = O(n)$ holes. The total number of holes in all $O(m^2)$ tournaments is then bounded by $O(m^2 n)$.

If the skipping condition is checked before attempting to create the hole, we may avoid redundant computation. Then the total number of hole creating operations during the computation of $K$ disjoint maximum sums is bounded by $O(m^2 n)$, which means that the second term of the complexity can not exceed $O(m^2 n \log n)$ even if $K > n$. Assuming that the skipping condition is checked in less than $O(\log n)$ time, we propose the following improved complexity.

**Proposition 4.6.1.** *$K$ disjoint maximum subarrays for two-dimensions can be computed in $O(m^2 n + m^2 \cdot \min(K,n) \log n)$ time.*

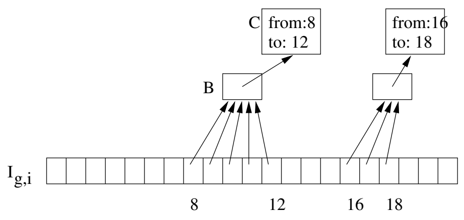


Figure 4.11: Disjoint set representation

We describe how we achieve this in the following.

### Union/Find problem

To achieve the suggested improvement for $K > n$, the skipping condition needs to be examined in less than $O(\log n)$ time.

We solve this by a classical *Union/Find problem* [34, 42, 51, 95, 13, 96]. Two major operations, *find* and *union*, are involved. Among the collection of disjoint sets, *find* operation tells whether certain two items are in the same set, and *union* operation merges two sets. We can make either *find* or *union* take $O(1)$ time, but not both simultaneously. We thus adopt a strategy supporting $O(1)$ time *find* operation to achieve fast skipping condition check in the context of our problem.

We first introduce an auxiliary array-based data structure $I$ for each strip to represent disjoint sets. Let $I_{g,i}$ be the array we prepare for a tournament $T_{g,i}$. Initially all the elements are null. The snapshot of $I_{g,i}$ is shown in Figure 4.11.

In this context, a set of contiguous array elements of $I_{g,i}$ that forms a disjoint set is referred to as a *block*. A block of $I_{g,i}[x..y]$ is made when we create a hole $(x, y)$ at $T_{g,i}$. However, a block can be concatenated with another adjacent block, to form a union of two blocks. For example, when two holes $(2, 5)$ and $(6, 8)$ are created at $T_{g,i}$, we have one concatenated block of $I_{g,i}[2..8]$. In this figure, two blocks of $(8, 12)$ and $(16, 18)$ are shown. Each element contained in a block references an intermediate $B$-node, then $B$-node references a $C$-node that stores two indices $from$ and *to*, the beginning and ending indices of the block. $B$ and $C$ stand for *block* and *concatenated block* respectively. Each $C$-node may have a number of $B$-nodes pointing

**Algorithm 31** Check if $(h^*, j^*)$ can skip hole creation

```
procedure canSkip($I_{g,i}, h^*, j^*$) begin
  1: return $I_{g,i}[h^*] \neq null$ and $I_{g,i}[j^*] \neq null$
  2:       and $(h^*, j^*) \subset (I_{g,i}[h^*].from, I_{g,i}[j^*].to)$
  3:       and $(I_{g,i}[h^*].from, I_{g,i}[h^*].to) = (I_{g,i}[j^*].from, I_{g,i}[j^*].to)$
end
```

to it as several blocks can eventually get connected. In such a case, we will perform *union* operation which will be described later. The size of $C$-node is determined by the number of $B$-nodes pointing to it.

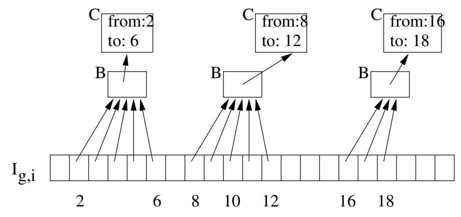
Suppose all the holes created in $T_{g,i}$ in the past are maintained in $I_{g,i}$. Before we attempt to make a hole $(h^*, j^*)$ in $T_{g,i}$, we visit $I_{g,i}$ and make a query by Algorithm 31. Note that we use a short notation such as $I_{g,i}[x].from$ to represent the $from$ attribute of the $C$ node pointed by the $B$ node pointed by $I_{g,i}$.

This algorithm returns *true* if all the elements within $(h^*, j^*)$ have been already marked as hole. For example, if $(h^*, j^*) = (9, 11)$, which is a subarray of $(8, 12)$ shown in Figure 4.11 is given, *true* is returned and we can safely skip the hole creation. When the query returns $false$, it implies that some elements within $(h^*, j^*)$ have not been marked yet and we still need to create a hole at $T_{g,i}$. It is obvious that such a query needs $O(1)$ time, satisfying the sub-$O(\log n)$ time requirement for the skipping condition check.

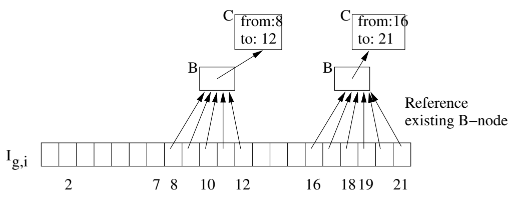
Nevertheless, to achieve the proposed complexity in Proposition 4.6.1, we need to show that the extra cost for maintenance of $I_{g,i}$ does not exceed the proposed complexity. In the following, we describe how $I_{g,i}$ is maintained.

When a hole $(h^*, j^*)$ is made at the tournament tree, we visit $I_{g,i}$ to mark that each element in this range at $sum_{g,i}$ constitutes a hole. If some elements in this range are already marked, we only process remaining unmarked portions. This way, each element of $I_{g,i}$ is processed at most once.

If $h^*$ is inside an existing block, we look up $I_{g,i}[h^*].to$ and update $h^*$ to the index of the first unmarked element. Likewise, if $j^*$ is inside another existing block, we update $j^*$ by $j^* \leftarrow I_{g,i}[j^*].from - 1$. Note that both $h^*$ and $j^*$ can not be inside the same existing block at this stage, as such a case would have returned *true* by the query above. After updating $h^*$ and $j^*$, the range



(a) Inserting $(2, 6)$ to Figure 4.11



(b) Inserting $(X, 21)$ (for $16 \leq X \leq 18$) to Figure 4.11

Figure 4.12: Marking where no union operation is needed

$(h^*, j^*)$ is now either totally disjoint from all existing blocks, or superarray of some existing blocks. We first describe a strategy for the former scenario.

Marking a range $(h^*, j^*)$ involves setting a pointer at $I_{g,i}[h^*..j^*]$ sequentially. As shown in Figure 4.12(a), we create one $B$-node and let the pointer from each array element reference it. We also create a $C$-node and make a pointer reference from the $B$-node. Finally indices $(h^*, j^*)$ are stored at $(from, to)$ of the $C$-node.

If there is an existing block adjacent to the beginning index $h^*$, we do not need to create an additional $B$-node. $I_{g,i}[h^*]$ and subsequent elements simply point to the $B$-node of the adjacent block. We need to update $I_{g,i}[h^*].to$ accordingly. Such an example is shown in Figure 4.12(b), where we insert $(X, 21)$ for $16 \leq X \leq 21$.

After marking, we see if a new block $I_{g,i}[h^*..j^*]$ happens to be connected to the existing block. If so, we perform *union* operation to form a concatenated block. This is essential to ensure the returned value of Algorithm 31 is correct.

Each union operation only involves the pointer redirection from $B$-nodes to a $C$-node. When two blocks are concatenated, let us suppose each block itself may be the result of previous union operations. Each block will be composed of one $C$-node pointed by multiple $B$-nodes, which are also pointed by array elements. We determine the larger $C$-node, the one pointed by greater number of $B$-nodes. The pointers of $B$-nodes referencing the smaller $C$-node are then redirected to the larger $C$-node. Certainly, $from$ and $to$ at the $C$-node need to be updated accordingly. Figure 4.13 illustrates such a case where insertion of a block is followed by a union operation. In the imple-

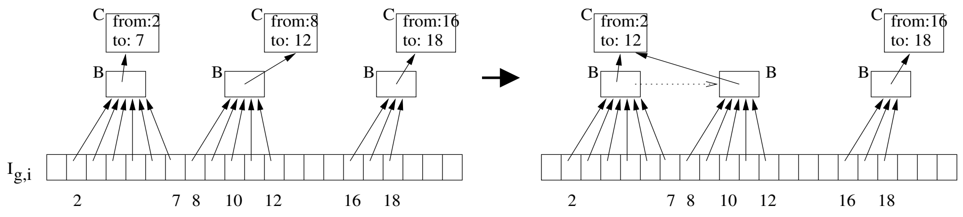


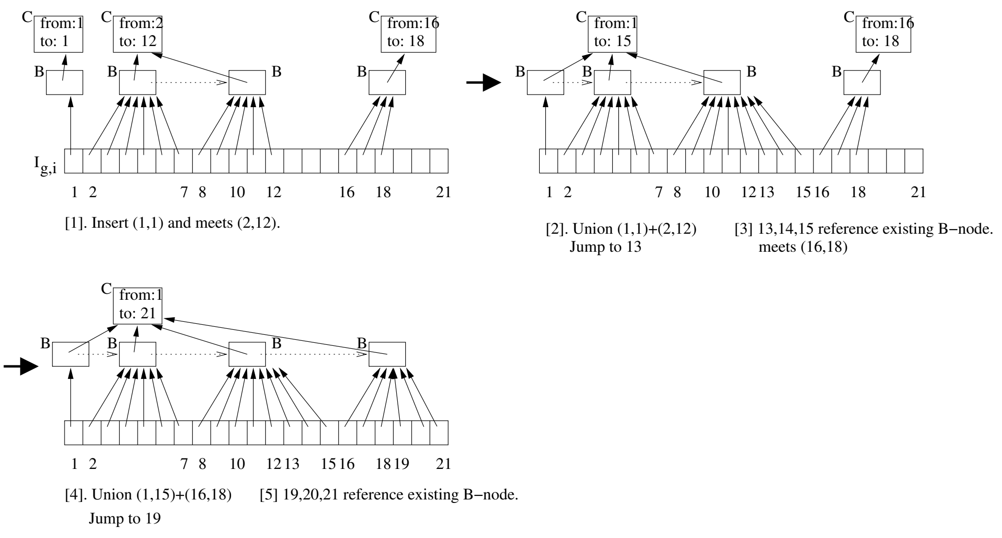
Figure 4.13: Marking followed by union operation:Inserting $(2, X)$ (for $7 \leq X \leq 12$) to Figure 4.11

mentation level, one may consider creating a linked-list of $B$-nodes during the union operation, as shown by the dotted line in the figure. As a group of $B$-nodes referencing the same $C$-node are all linked, we may simply follow these links and redirect the pointers to another $C$-node in the future union operations.

Even if a new entry $(h^*, j^*)$ is a superarray of some existing blocks as shown in Figure 4.14, the general framework described above can be applied without change. We can skip already marked array elements and process unmarked elements only. When a new $B$-node is required, it is created and the indices at $C$-node are updated to indicate the position of the concatenated block. Notice that we only need to create at most one additional $B$-node, but we may have to perform several union operations. In Figure 4.14, for example, it is only $I_{g,i}[1]$ that requires a new $B$-node, while other entries simply reference the existing $B$-nodes.

## Analysis

On insertion of a block, we create one or no $B$-node. As no more than $\min(K, n)$ blocks can be inserted to $I_{g,i}$, we have at most $\min(K, n)$ $B$-nodes. The number of union operations we need to process is dependent on the number of blocks. As each union operation puts two blocks into one concatenated block, the maximum number of union operations is $O(\min(K, n)) - 1$, since then everything is in one block. As the pointers of $B$-nodes referencing the smaller $C$-node are redirected to the larger $C$-node, the total time spent for $\min(K, n) - 1$ unions is $O(\min(K, n) \log \min(K, n))$. Note that this justifies separation of $B$- and $C$-nodes in the design of our data structure. Suppose



[1]. Insert (1,1) and meets (2,12).
[2]. Union (1,1)+(2,12) Jump to 13
[3] 13,14,15 reference existing B−node. meets (16,18)
[4]. Union (1,15)+(16,18) Jump to 19
[5] 19,20,21 reference existing B−node.

Figure 4.14: Inserting $(1, 21)$ to the result of Fig 4.13

we have $B^*$-node only, a combined version of $B$- and $C$-node. On each union operation, pointer redirection is made from each array element of $I_{g,i}$ to a $B^*$-node, resulting in $O(n \log n)$ time in total. It lacks efficiency when $K < n$.

In addition, marking each array element of $I_{g,i}$ is $O(1)$ time, thus $O(n)$ time in total. Then combined time for marking and union is $O(n + p \log p)$, where $p = \min(K, n)$. For all $O(m^2)$ strips, it is $O(m^2 n + m^2 \cdot p \log p)$ time, but this time can be absorbed into the suggested complexity in Proposition 4.6.1. Therefore the proposed complexity of $O(m^2 n + m^2 \cdot \min(K, n) \log n)$ is achieved.

## 4.7 Run-time Performance Consideration

In the current form of our algorithm, *2dMaxSelect()* in Algorithm 29 is given as a simple iteration. In fact, a binary heap (max-version) may be used instead without increasing the complexity, and considerable run-time speed up can be achieved with the following modification. We can say this is a two-level priority queue.

We build a heap each of whose nodes references the root of each tournament $T_{g,i}$ for some $g$ and $i$. Let the value of a node in the heap be determined by $M_{g,i}$, the $M$ attribute at the root of $T_{g,i}$. Then the maximum subarray for two-dimensions is retrieved from the root of the heap.

We prepare a list $HL$, which stands for a hole-list, associated with each tournament. When a maximum sum is found at the root of the heap, we add the range of a hole to $HL$ of all potentially overlapping tournaments in $O(1)$ time each, instead of creating a hole in each tournament.

While the root of the heap references a tournament that has a non-empty list, we take the first element of $HL$ and create a hole specified by it. The heap property may have been broken after this, so we adjust the heap. If the root of the heap has an empty list, we are assured that the subarray specified by the root is disjoint from all the previous solutions, and its $M_{g,i}$ is greater than that of other nodes in the heap. Thus $M_{g,i}$ is selected as the next maximum sum.

This effectively enables us to update some small number of tournaments whose $M_{g,i}$ are high-ranked in the heap.

Also, if there is a node in the heap referencing non-positive $M_{g,i}$, we can safely delete this node from the heap. Effectively, the size of heap may reduce over time. For example, if $\bar{K} = mn/2$, only $O(m)$ nodes referencing tournaments $T_{1,1}, T_{2,2}, ..T_{m,m}$ will be present in the heap.

Alternative routines to Algorithm 29 and Algorithm 30 are given in Algorithm 32.

Indeed, this modification does not improve the asymptotic complexity. Line 1 of the main routine takes $O(m^2)$ time. Inside the loop starting at line 2, the complexity of line 3, is still bounded by $O(m^2 \log n)$. In the worst case, the "while" loop starting at line 2 of *2dMaxSelectHEAP()* may run $O(n^2)$ times, and inside the loop, line 4 and line 5 or 6 are all $O(\log n)$ time process. The $O(m^2)$ time by lines 4–7 in the main routine is absorbed.

However considerable run-time speed up can be observed with random inputs. The quantitative analysis is still open at the stage.

## 4.8 Extending to Higher Dimensions

For a $d$-dimensional array of size $n \times \cdots \times n$, there are $O(n^{2d-2})$ 1D problems. We spend $O(n^{2d-1})$ time to build a tournament for each 1D problem. Upon the retrieval of each subsequent maximum sum in the $d$-dimensional array,

**Algorithm 32** Find $K$-disjoint maximum subarrays in 2D (heap version)

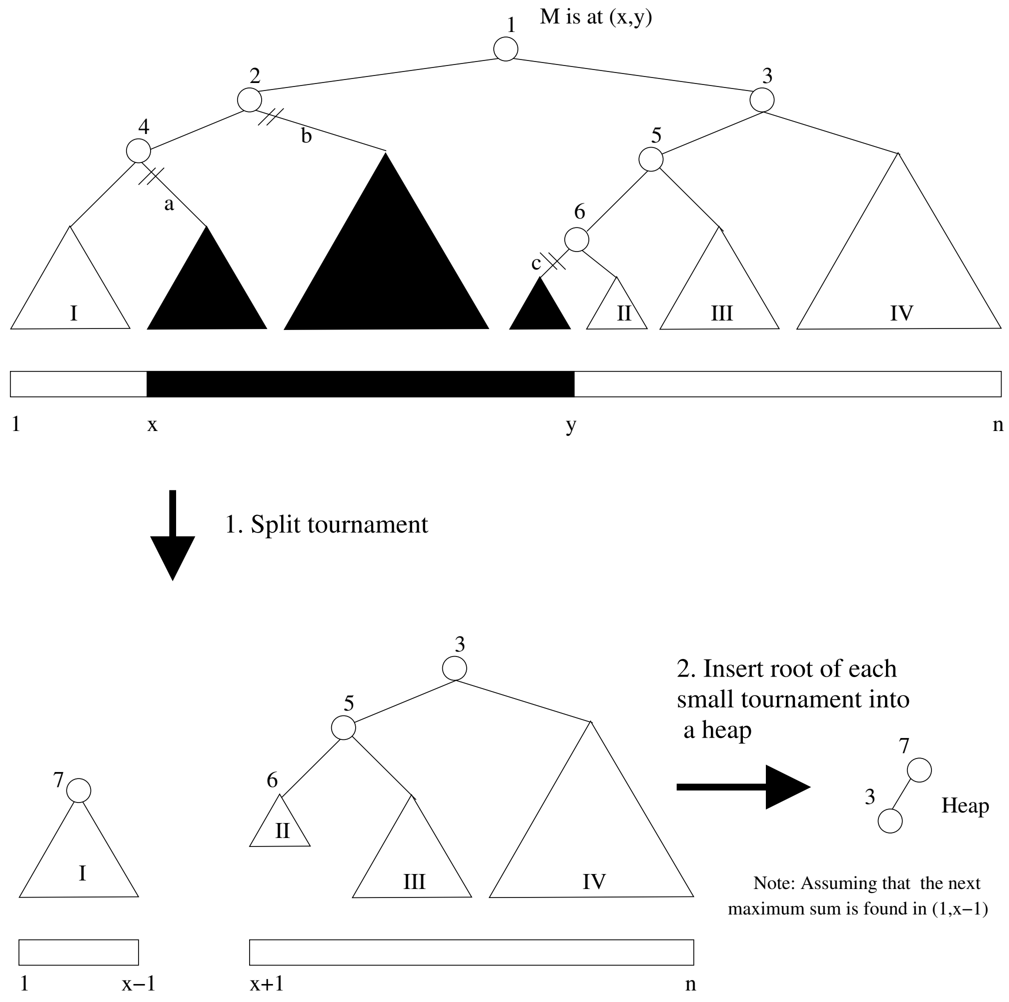
```
procedure 2dMaxSelectHEAP() begin
 1: take the root of the heap given as $M_{r_1,r_2}$
 2: while $HL_{r_1,r_2}$ is not empty do
 3:    take the first entry from $HL_{r_1,r_2}$ given as $(holeBegin, holeEnd)$
 4:    create a hole in $T_{r_1,r_2}$ // $M_{r_1,r_2}$ is updated.
 5:    if $M_{r_1,r_2} \leq 0$ then delete the root from the heap
 6:    else adjust the heap
 7:    take the root of the heap given as $M_{r_1,r_2}$
 8: end while
 9: return $M(r_1, M.from)|(r_2, M.to)$
end
begin // main
 1: build the heap where each node points to the root of each tournament
 2: for $k \leftarrow 1$ to $K$ do
 3:    compute $M(k)$ that is $M_{g_k,i_k}(g_k, h_k)|(i_k, j_k)$ returned by 2dMaxSelectHEAP()
 4:    for $g \leftarrow 1$ to $m$ do
 5:       for $i \leftarrow g$ to $m$ do
 6:          if not$(i < g_k$ or $g > i_k)$ then add a hole entry $(h_k, j_k)$ to $HL_{g,i}$
 7:       end for
 8:    end for
 9: end for
end
```

we need to update $O(n^{2d-2})$ tournaments. The total time for $K$ maximum sums is then $O(n^{2d-1} + Kn^{2d-2} \log n)$. Using the same technique described in Section 4.6.4, the second term can be bounded by $O(n^{2d-1} \log n)$, making the total time $O(n^{2d-1} + n^{2d-2} \cdot \min(K, n) \log n)$.

## 4.9 Alternative Algorithm for 1D

In relation to Algorithm 20 presented in the previous chapter, an alternative algorithm for 1D can be devised.

Suppose we have computed the first maximum subarray $(x, y)$ by Algorithm 27. To obtain the second maximum, we create a hole $(x, y)$ by explicitly deleting the relevant nodes and split the tournament into two smaller tour-



1
M is at (x,y)
2
3
4
b
5
a
6
c
I
II
III
IV
1
x
y
n
1. Split tournament
3
5
2. Insert root of each small tournament into a heap
7
6
7
3
Heap
II
I
III
IV
Note: Assuming that the next maximum sum is found in (1,x−1)
1
x−1
x+1
n

Figure 4.15: Alternative algorithm for 1D based on a combination of a heap and tournaments

naments, as shown in Figure 4.15. Two smaller tournaments cover the range $(1, x-1)$ and $(y+1, n)$ respectively. It is easy to see that the split takes $O(\log n)$ time. The split procedure may be similar to the technique used in Algorithm 22, but we do not copy nodes during the split as the original tournament does not need to be kept.

Each split results in at most two smaller tournaments. We take the root of each tournament and inserted them into an auxiliary heap. The root of the heap is the next maximum subarray. Note that this is disjoint from the previous one.

In general, to find the subsequent maximum, we delete the root of the heap, and visit the tournament that is associated with the root of the heap. In the tournament, we create a hole and split this tournament. Finally, the roots of new smaller tournaments are inserted into the heap. This routine is repeated until $K$ maximum subarrays are found.

The size of the heap is bounded by $\bar{K}$, and each maximum subarray is found in $O(\log n)$ time, making the total time including the tournament building $O(n + K \log n)$. This is intuitively easier than the algorithm described in this chapter (Algorithm 27 coupled with Algorithm 28), yet not particularly simpler to implement.

The major problem of the alternative algorithm arises from its difficulty in creating a *random* hole. Suppose that a tournament has been split into several tournaments over time, for example, five tournaments with coverage $(1, 3)$, $(6, 8)$, $(10, 12)$, $(15, 19)$ and $(22, 23)$ respectively. Five roots of these tournaments form a heap. Notice that it is difficult to create a hole $(2, 22)$. As shown in line 5 of Algorithm 30 or line 4 of Algorithm 32, a simple extension to 2D is heavily dependent on this random hole creation. The lack of such a capability makes the alternative algorithm difficult to extend to 2D.

## 4.10 Summary of Results

In this chapter, we established $O(n + K \log K)$ time for ranking $K$ disjoint maximum subarrays in 1D and extended this to 2D to achieve $O(m^2 n + Km^2 \log n)$ time, which is $O(m^2 n)$ time for small $K$. To author's knowledge, this is the first improvement to the trivial $O(Km^2 n)$ time solution. Since $\bar{K}$, the maximum possible $K$, can be as large as $mn/2$ depending on the data, reduction of the factor $K$ is significant. It will be an interesting question how to determine $\bar{K}$ in advance.

The improvement to the second term of complexity, $O(m^2 \cdot \min(K, n) \log n)$, was achieved by incorporating a simple solution for the union/find problem. It was shown that the complexity for the union/find problem, the extra cost for the computing the union operations, is tightly within the overall time. Thus, unless a different framework is used, a better solution for the union/find problem may be a prerequisite for any attempt to further improve the second term. The best-known result for the problem is slightly more than $O(n + \min(K, n))$ [95, 13, 96].

The algorithm for the 2D problem can be further extended. For a $d$-dimensional array of size $n \times \ldots \times n$, we need $O(n^{2d-1} + n^{2d-2} \cdot \min(K, n) \log n)$ time.

# Chapter 5

# Experimental Results

The time complexity of new algorithms for the $K$-OMSP and $K$-DMSP developed from this research were theoretically proved in earlier chapters. This chapter presents the results of experimental comparison of newly developed algorithms, and relates their practical performance to what is expected in theory. The experiments were performed using a 2.13 GHz Intel Core 2 Duo E6400 machine, with 2048MB of RAM. This CPU has two identical processor cores and 2048KB of L2 cache shared by two cores [54]. All algorithm implementations were written in the `C` programming language, and all are single-threaded. Hence the effect of the multi-core is negligible. The system is installed with Fedora Core 6 Linux operating system equipped with kernel version 2.6.18.

Each program processes identical input array of size $n$ that were randomly generated* with the same seed. They were compiled using the GNU compiler `gcc` (Version 4.1.3) with the `-O3` optimization flag. The maximum size of the input was determined not to exceed the available RAM, such that all algorithms would run without slow virtual memory paging. All algorithms were timed by GNU `time` (Version 1.7), and the amount of CPU time they used were obtained from the sum of system time and user time. The time is measured 20 times for each experiment and the average is shown in 2 decimal places.

* Using a built-in function in `C`, `rand() % 100 - 50`, such that the array elements will be roughly well distributed between $-50$ and 50

| | n=50,000 | | | n=500,000 | | |
|---|---|---|---|---|---|---|
| $K$ | Algo.15 | Algo.18 | Algo.20 | Algo.15 | Algo. 18 | Algo. 20 |
| 1 | 0.00 | 0.01 | 0.01 | 0.02 | 0.11 | 0.17 |
| 5 | 0.00 | 0.02 | 0.01 | 0.03 | 0.27 | 0.17 |
| 50 | 0.02 | 0.08 | 0.01 | 0.21 | 0.95 | 0.17 |
| 500 | 0.19 | 0.15 | 0.01 | 2.02 | 1.39 | 0.17 |
| 5,000 | 2.08 | 0.18 | 0.01 | 21.01 | 1.78 | 0.17 |
| 50,000 | 21.38 | 0.23 | 0.10 | 210.03 | 2.22 | 0.28 |
| 500,000 | - | - | - | 2108.91 | 2.85 | 1.71 |

Table 5.1: Total processing time (in seconds) spent by algorithms for the 1D $K$-OMSP for $n = 50,000$ and $500,000$

## 5.1 1D $K$-OMSP

Three algorithms, Algorithm 15, Algorithm 18 and Algorithm 20 were implemented and their running times were measured as shown in Table 5.1.

It is easy to observe that Algorithm 20 runs faster than the other two in most benchmarks, albeit it is initially even slower than Algorithm 15, the theoretically least efficient one. This is due to overhead involved in building tournaments and random access to the memory during the recursive routine. Maintenance of persistent 2-3 tree in Algorithm 18 and persistent tournament in Algorithm 20 requires frequent dynamic memory allocation. In contrast, Algorithm 15 is implemented using arrays of fixed size whose memory are allocated when the program starts. This explains Algorithm 15 running faster for small $K$.

While $K$ is small relative to $n$, the time by Algorithm 20 remains almost constant. However, a significant increase in time is observed when $K = 50000$ and $n = 50000$ or $n = 500000$.

The time measurements displayed are very close to the theoretical estimation. The pre-process time including the prefix sum computation and the time for building a tournament and a max-heap contribute to the first term of the complexity. This is a one-off process and its time is shown by the measurement for $K = 1$. The fact that the time remains near-constant suggests that the actual retrieval of maximum sums is very fast, and measured below

2 decimal places. For $K \geq 50000$, the effect of $K$ on the time is starting to be more pronounced. Excluding the initial setting time, the running times are 0.11 and 1.54 seconds for $K = 50000$ and 500000 at $n = 500000$, slightly more than 10 times increase, hinting the second term of the complexity, $O(K \log K)$, has become dominant.

Algorithm 18 is initially slower than Algorithm 15 for small $K$, but it eventually overtakes at $K = 500$ in both cases, $n = 50000$ and $n = 500000$. Note that the time increases approximately 10 times when $n$ increases from 50000 to 500000, which confirms the factor of $n$ in the complexity. Also its relatively slow increase in running time in relation to $K$ demonstrates $\log K$ factor.

Algorithm 15 is the least efficient algorithm theoretically, and the experimental result demonstrates the theoretical estimation. The factor of $K$ in the complexity is shown by the linear increase in running time. When $n$ increases from 50000 to 500000, for the same value of $K$, it is also observed that the running time increases ten times, which demonstrates the factor of $n$ in the theoretical complexity. This algorithm, while it is the least efficient, is simple to implement, and may still serve as a good option for some applications where very small value of $K$ is required.

Algorithm 18 requires $O(n \log K)$ space complexity due to $n$ versions of $min$ maintained in a persistent 2-3 tree. For $K = O(n)$, this is more complex space-wise than Algorithm 20 whose space complexity is $O(n+K \log n)$. Also notice that each node in a 2-3 tree takes up more memory than a node in a tournament does in Algorithm 20. In the implementations used in the experiment, each node in Algorithm 18 reserves 37 bytes of memory space[†], while a node in the tournament in Algorithm 20 consumes about half amount of memory. Due to this reason, Algorithm 18 exceeds the available memory and starts the virtual memory paging once the setting is more demanding than $n = 1600000$ and $K = 1600000$. On the other hand, Algorithm 20

[†] Based on a simple implementation such that a single structure can be used for any of 1..4-node, which allocates sufficient memory in case the node expands to a 4-node. We need four integer variables and four pointers to children nodes. In addition, we prepare another integer variable to store the number of leaf nodes under the node (similar to Figure 3.4), and a one-byte (such as `char`) variable to specify a node-type. For optimal memory usage, separate structures can be defined for each type of nodes

is more durable, and consistently stays below the physical memory until $n = 8000000$ and $K = 8000000$.

## 5.2 2D $K$-OMSP

The following four algorithms for the 2D $K$-OMSP are implemented and their running times are measured. For simplicity, we assume that $m = n$ here. Each implementation will be referred to by the keyword.

- **LOOP** (Quadruple-nested loops): $O(Kn^3)$ time. Extended from Algorithm 7 through a similar modification from Algorithm 3 to Algorithm 15.
- **SAMP** (Sampling in two dimensions): Refer to Section 3.7.1. $O(n^3 + K^2 \log n)$ time. Algorithm 20 is used to compute $K$ 1D problems.
- **HEAP** (Selection in two-level Heap): Refer to Section 3.7.2. $O(n^3 + K \log \min(K, n))$ time.
- **COMB** (Combination of SAMP and HEAP):

The fourth version, COMB, deserves some remark. If $K$ is relatively small, we do not need to build a tournament and a first-level heap for every strip. We first collect $K$ *good* strips by sampling process. As is in SAMP, we only construct a tournament and an associated first-level heap for these selected strips. Then the second-level heap has only $K$ nodes. For small $K$, it can be a considerable reduction from $n(n + 1)/2$, the number of nodes in the second-level heap will need to maintain otherwise. Note that it is not an asymptotic improvement on HEAP, since preprocessing of $O(n^2)$ strips still involves $O(n^3)$ time and retrieval of each subsequent maximum sum still requires $O(\log \min(K, n))$ time.

LOOP closely matches the theoretical estimation. The processing time increases in proportion to $K$. Also when $n$ is doubled, the processing time increases approximately eight times.

SAMP achieves noticeable efficiency through the sampling pre-process and application of an efficient algorithm for the 1D $K$-OMSP, Algorithm

| $m \times n$ | $64 \times 64$ | $128 \times 128$ | $256 \times 256$ | $512 \times 512$ | $1,024 \times 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.00 | 0.02 | 0.17 | 1.31 | 10.41 |
| 2 | 0.00 | 0.02 | 0.24 | 1.88 | 14.65 |
| 4 | 0.00 | 0.04 | 0.38 | 3.03 | 24.06 |
| 8 | 0.00 | 0.08 | 0.67 | 5.30 | 41.52 |
| 16 | 0.02 | 0.15 | 1.27 | 9.77 | 78.22 |
| 32 | 0.03 | 0.30 | 2.42 | 19.28 | 152.39 |
| 64 | 0.07 | 0.59 | 4.76 | 37.51 | 298.78 |
| 128 | 0.15 | 1.22 | 9.63 | 76.69 | 618.63 |
| 256 | 0.29 | 2.40 | 19.00 | 152.11 | 1212.70 |
| 512 | 0.60 | 4.76 | 37.88 | 303.80 | 2421.88 |
| $1,024$ | 1.49 | 11.72 | 93.47 | 743.88 | 5981.25 |

Table 5.2: Total processing time (in seconds) spent by LOOP

| $m \times n$ | $64 \times 64$ | $128 \times 128$ | $256 \times 256$ | $512 \times 512$ | $1,024 \times 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.00 | 0.00 | 0.05 | 0.35 | 2.71 |
| 2 | 0.00 | 0.00 | 0.05 | 0.36 | 2.70 |
| 4 | 0.00 | 0.00 | 0.05 | 0.36 | 2.72 |
| 8 | 0.00 | 0.00 | 0.05 | 0.36 | 2.71 |
| 16 | 0.00 | 0.00 | 0.05 | 0.36 | 2.70 |
| 32 | 0.00 | 0.00 | 0.05 | 0.37 | 2.71 |
| 64 | 0.00 | 0.01 | 0.06 | 0.38 | 2.72 |
| 128 | 0.01 | 0.02 | 0.07 | 0.41 | 2.75 |
| 256 | 0.04 | 0.08 | 0.15 | 0.50 | 2.91 |
| 512 | 0.20 | 0.26 | 0.40 | 0.85 | 3.34 |
| $1,024$ | 0.73 | 0.94 | 1.21 | 1.92 | 4.83 |

Table 5.3: Total processing time (in seconds) spent by SAMP

20. The processing time remains little influenced by the increase of $K$ while $K < 256$ for an array of every selected size. If we exclude the time spent for the initial setting, i.e. the time for $K = 1$, the change of running time with respect to $K$ becomes more apparent. For example, in case of an array of size $1024 \times 1024$, we subtract the initial setting time of 2.71 seconds from each running time and obtain the following table showing the times for computing the rest $(K - 1)$ maximum sums.

| $m \times n$ | $64 \times 64$ | $128 \times 128$ | $256 \times 256$ | $512 \times 512$ | $1,024 \times 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.04 | 0.38 | 2.89 | - | - |
| 2 | 0.04 | 0.37 | 2.93 | - | - |
| 4 | 0.03 | 0.37 | 2.92 | - | - |
| 8 | 0.03 | 0.38 | 2.92 | - | - |
| 16 | 0.04 | 0.37 | 2.92 | - | - |
| 32 | 0.03 | 0.37 | 2.93 | - | - |
| 64 | 0.04 | 0.37 | 2.92 | - | - |
| 128 | 0.04 | 0.37 | 2.93 | - | - |
| 256 | 0.04 | 0.37 | 2.92 | - | - |
| 512 | 0.04 | 0.37 | 2.92 | - | - |
| $1,024$ | 0.03 | 0.37 | 2.92 | - | - |

Table 5.4: Total processing time (in seconds) spent by HEAP

| $K$ | 2..32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| Time(sec) | 0.00 | 0.01 | 0.04 | 0.20 | 0.63 | 2.12 |

Times for $K \leq 32$ are almost constantly zero. After the sampling preprocess, only $K$ strips are left and we run Algorithm 20 $K$ times. Here, let us review the running time by Algorithm 20 for $n = 50000$ and $K = 50$ in Table 5.1. Assuming the validity of this measured data, $K$-times repetition of Algorithm 20 for $n = 1024$ and $K \leq 32$ is expected to run less than 0.01 second.

It is $K \geq 128$ where we can observe a noticeable increase in running time. Approximately, the time increase is proportional to the square of $K$ with slight deviation at $K = 256$ and $K = 1024$. What is shown in the second term of the complexity suggests the time will be proportional to $K^2$, and such a trend is roughly observed.

The size of input array is doubled vertically and horizontally, enlarging it 4 times from $64 \times 64$ to $128 \times 128$ etc. The impact of the size of input array on the running time is well displayed in the row with $K = 1$. Approximately 8 times increase in time is observed.

The performance of HEAP raises some interesting points. As estimated by the time complexity, the processing time almost remains constant while $K$ grows. Most of time is spent for the initial memory allocation, and retrieval of maximum sums is done very fast, even below the time measurement limit. Nevertheless, the large numbers of dynamic memory allocation makes it slower than SAMP in most cases, and even slower than LOOP when $K$ is relatively small. In total, memory spaces for $O(n^3 + K \log n)$ nodes are dynamically allocated. A related issue is that the algorithm fails to allocate required memory and starts virtual memory paging for an array of size $512 \times 512$. Hence the times for $512 \times 512$ and $1024 \times 1024$ are missing.

| $m \times n$ | $64 \times 64$ | $128 \times 128$ | $256 \times 256$ | $512 \times 512$ | $1,024 \times 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.00 | 0.00 | 0.05 | 0.40 | 2.94 |
| 2 | 0.00 | 0.00 | 0.05 | 0.40 | 2.96 |
| 4 | 0.00 | 0.00 | 0.05 | 0.40 | 2.99 |
| 8 | 0.00 | 0.00 | 0.05 | 0.40 | 2.98 |
| 16 | 0.00 | 0.00 | 0.05 | 0.40 | 2.96 |
| 32 | 0.00 | 0.00 | 0.05 | 0.40 | 2.99 |
| 64 | 0.00 | 0.01 | 0.06 | 0.41 | 2.99 |
| 128 | 0.00 | 0.01 | 0.06 | 0.41 | 3.01 |
| 256 | 0.00 | 0.01 | 0.07 | 0.44 | 3.01 |
| 512 | 0.01 | 0.02 | 0.10 | 0.49 | 3.13 |
| $1,024$ | 0.02 | 0.05 | 0.15 | 0.59 | 3.34 |

Table 5.5: Total processing time (in seconds) spent by COMB

COMB is implemented to overcome this undesirable characteristics of HEAP. For relatively small $K$, it performs as fast as SAMP. Yet, its running time is much less affected by $K$. This is because the incorporation of HEAP reduces the second term of the complexity to $O(K \log \min(K, n))$. Still, the increase of processing time depending on $K$ is slightly more pronounced than HEAP. This is mainly attributed to the diminishing gain of sampling. The sampling pre-process is most effective for small $K$ as many number of strips can be discarded. The gain of sampling diminishes as $K$ grows and sampling is no longer useful or even unnecessary once $K$ reaches $n(n + 1)/2$ since no strips are discarded and all the tournaments need to be built. One may skip the sampling in such a case to avoid extra overhead.

## 5.3 1D $K$-DMSP

Three algorithms for the 1D $K$-DMSP were implemented and their running times were measured.

- **RK** (Repeated application of Kadane's algorithm (Algorithm 1)) : $O(Kn)$ time.
- **RT** (Extended version of Ruzzo and Tompa's algorithm (Algorithm 25): $O(n + K \log K)$) time.
- **NEW** (New Algorithm based on Section 4.5): $O(n + K \log K)$) time.

For a fair comparison, RT extends the original Ruzzo and Tompa's algorithm by adding an extra selection and sorting process, such that it will output the $K$ maximum sums in sorted order. We adopt Algorithm 11 for this purpose [‡]. This makes RT asymptotically equivalent to NEW.

In this experiment, if $\bar{K}$, the number of positive maximum subarrays in the input array, is fewer than the desired $K$, the program terminates and outputs only $\bar{K}$ values. Note that the value of $\bar{K}$ is dependent on the randomly generated input array $a$, and may vary with a different pseudo random number generator and seed.

When $n = 100000$, the processing times of RT and NEW remain almost constant, and indeed, the time by RT is below measurement limit. It is interesting to note that these algorithms are $O(n)$ time for $K \leq n/\log n$, and $n/\log_2 n = 6020.6$ for $n = 100000$. In this experiment, only $\bar{K} = 5945$ disjoint maximum subarrays are available.

It appears that sorting, the extra overhead required by RT, needs negligibly short running time. The most time consuming process in NEW appears to be the one of tournament building due to frequent dynamic memory allocation. The first maximum sum is obtained as a direct result of tournament building. The rest $(K-1)$ maximum sums are obtained spending minimal time. The near-constant running time of RT and NEW makes the value of $K$ almost irrelevant, supporting the $O(n)$ time complexity. However, the

[‡] Instead of a tournament, a heap is used as it reduces the memory consumption by half

- $n = 100,000$

| $K$ | RK | RT | NEW |
|---|---|---|---|
| 1 | 0.00 | 0.00 | 0.02 |
| 10 | 0.00 | 0.00 | 0.02 |
| 100 | 0.02 | 0.00 | 0.02 |
| 1,000 | 0.27 | 0.00 | 0.02 |
| $\bar{K}$=5,945 | 1.42 | 0.00 | 0.02 |

- $n = 1,000,000$

| $K$ | RK | RT | NEW |
|---|---|---|---|
| 1 | 0.02 | 0.06 | 0.30 |
| 10 | 0.05 | 0.06 | 0.30 |
| 100 | 0.22 | 0.06 | 0.30 |
| 1,000 | 2.33 | 0.06 | 0.30 |
| 10,000 | 27.50 | 0.06 | 0.31 |
| $\bar{K}$=60,790 | 151.39 | 0.08 | 0.42 |

Table 5.6: Total processing time (in seconds) spent by algorithms for the 1D $K$-DMSP

frequent dynamic memory allocation incurred by NEW makes it running considerably slower than RT, which can be implemented exclusively using arrays only.

The running times of NEW for $K = 1$ and $n = 1000000$ illustrate the overhead caused by dynamic memory allocation more evidently. For $K = 1$, NEW is approximately 5 times slower than RT and 15 times slower than RK. Indeed, it is roughly 5 times slower than RT for all values of $K$ tested.

In this experiment, the maximum value for $K$, that is $\bar{K}$, is 60790. Both RT and NEW show slight time increase at $K = 60790$ but their running times are near-constant under this point. In theory, both RT and NEW would be $O(K \log K)$ time once $K$ exceeds $n/\log_2 n$, that is $K > 50171.7$, as the second term in the complexity becomes dominant. While such a behavior is not very clearly pronounced in RT, NEW experiences a noticeable increase in running time for $K = 60790$.

The slight increase in time for RT at this point is attributed to the increased number of values to sort, since it would run in $O(n)$ time regardless of the value of $K$ if sorting is not performed. On the other hand, the increased running time for NEW is caused by more number of the tournament updates. Certainly, the overhead involved in updating a tournament is heavier, as it requires to traverse two paths and modify many attributes of each node visited. On the other hand, RT maintains an array-implemented heap, which involves much lighter overhead. Excluding the initial setting time, the running time of NEW for computing the remaining $(K-1)$ maximum sums is 0.01 and 0.12 seconds for $K = 10000$ and $K = 60790$ respectively. This is slightly steeper than linear increase, but whether its time is affected by the dominant second term, $O(K \log K)$, is not clear. While it will need more measured data to confirm the trend, an attempt to generate a different random sequence that would lead to higher value for $\bar{K}$ was not very successful. In many random sequences, $\bar{K}$ tends to be relatively small with respect to $n$.

The theoretical estimation for RK is best demonstrated for $K \geq 100$, where running time increases linearly in proportion to $K$.

## 5.4 2D $K$-DMSP

The following three algorithms were implemented for the 2D $K$-DMSP:

- RK (Repeated application of Kadane's algorithm (Algorithm 9)): $O(Km^2n)$ time
- NEW1 (New algorithm based on Section 4.6): $O(m^2n + Km^2 \log n)$ time
- NEW2 (New algorithm based on Section 4.7 (Algorithm 32)): $O(m^2n + Km^2 \log n)$ time

For each algorithm, we prepare two tables, for $m = 128$ and $m = 256$ with values of $n$ ranging from 64 to 1024 and $K$ ranging from 1 to 1024.

RK clearly demonstrates its $O(Km^2n)$ time complexity. The running time increases in proportion to $K$ and $n$, also $m^2$, which is observed by four-fold time increase when $m$ is increased from 128 to 256.

NEW1 and NEW2 are slower than RK for small $K$ due to expensive setting time that arises from tournament building. The complexity of the

| $m = 128$ | $n = 64$ | $n = 128$ | $n = 256$ | $n = 512$ | $n = 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.00 | 0.00 | 0.01 | 0.02 | 0.04 |
| 2 | 0.00 | 0.01 | 0.02 | 0.04 | 0.09 |
| 4 | 0.01 | 0.02 | 0.04 | 0.10 | 0.19 |
| 8 | 0.02 | 0.04 | 0.09 | 0.18 | 0.38 |
| 16 | 0.03 | 0.09 | 0.17 | 0.37 | 0.74 |
| 32 | 0.08 | 0.16 | 0.35 | 0.72 | 1.46 |
| 64 | 0.14 | 0.30 | 0.63 | 1.36 | 2.82 |
| 128 | 0.28 | 0.59 | 1.23 | 2.61 | 5.47 |
| 256 | 0.52 | 1.11 | 2.34 | 4.86 | 10.28 |
| 512 | 1.03 | 2.15 | 4.44 | 9.05 | 19.29 |
| $1,024$ | 2.00 | 4.18 | 8.42 | 17.78 | 36.71 |

| $m = 256$ | $n = 64$ | $n = 128$ | $n = 256$ | $n = 512$ | $n = 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.01 | 0.02 | 0.04 | 0.08 | 0.17 |
| 2 | 0.01 | 0.03 | 0.08 | 0.16 | 0.33 |
| 4 | 0.04 | 0.08 | 0.16 | 0.33 | 0.65 |
| 8 | 0.07 | 0.16 | 0.32 | 0.66 | 1.30 |
| 16 | 0.15 | 0.32 | 0.66 | 1.30 | 2.56 |
| 32 | 0.28 | 0.61 | 1.26 | 2.56 | 5.10 |
| 64 | 0.56 | 1.17 | 2.39 | 5.07 | 9.97 |
| 128 | 1.08 | 2.22 | 4.56 | 9.67 | 19.66 |
| 256 | 2.12 | 4.29 | 9.11 | 18.51 | 37.56 |
| 512 | 4.14 | 8.38 | 17.45 | 35.55 | 73.25 |
| $1,024$ | 7.95 | 16.39 | 33.65 | 68.77 | 139.88 |

Table 5.7: Total processing time (in seconds) spent by RK

second term is relatively high, and results in the running time for retrieval of the rest $(K-1)$ maximum sums after the initial setting more pronounced than similar experiment with algorithms for the 2D $K$-OMSP.

If we subtract the initial setting time measured at $K = 1$ from the rest measured times in the same column, the pattern of increase in time with respect to $K$ becomes more visible. For example, the running times of NEW1 for $m = 128$ and $n = 1024$ in Table 5.8 excluding the initial setting are given as:

| $m = 128$ | $n = 64$ | $n = 128$ | $n = 256$ | $n = 512$ | $n = 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.08 | 0.17 | 0.34 | 0.73 | 1.43 |
| 2 | 0.09 | 0.18 | 0.35 | 0.75 | 1.44 |
| 4 | 0.10 | 0.19 | 0.36 | 0.76 | 1.45 |
| 8 | 0.12 | 0.21 | 0.39 | 0.77 | 1.47 |
| 16 | 0.13 | 0.25 | 0.43 | 0.85 | 1.54 |
| 32 | 0.16 | 0.29 | 0.52 | 0.95 | 1.67 |
| 64 | 0.18 | 0.32 | 0.63 | 1.11 | 1.95 |
| 128 | 0.22 | 0.40 | 0.78 | 1.33 | 2.37 |
| 256 | 0.26 | 0.46 | 0.91 | 1.60 | 3.13 |
| 512 | 0.34 | 0.56 | 1.08 | 2.04 | 4.02 |
| 1,024 | 0.50 | 0.74 | 1.28 | 2.54 | 5.30 |

| $m = 256$ | $n = 64$ | $n = 128$ | $n = 256$ | $n = 512$ | $n = 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.35 | 0.70 | 1.39 | 2.88 | 5.62 |
| 2 | 0.37 | 0.72 | 1.43 | 2.93 | 5.63 |
| 4 | 0.38 | 0.74 | 1.47 | 2.99 | 5.68 |
| 8 | 0.43 | 0.80 | 1.52 | 3.13 | 5.75 |
| 16 | 0.50 | 0.92 | 1.71 | 3.28 | 5.96 |
| 32 | 0.58 | 1.06 | 1.95 | 3.71 | 6.26 |
| 64 | 0.75 | 1.29 | 2.33 | 4.28 | 7.25 |
| 128 | 1.06 | 1.65 | 2.88 | 5.18 | 8.78 |
| 256 | 1.62 | 2.33 | 3.66 | 6.28 | 10.66 |
| 512 | 2.72 | 3.56 | 4.99 | 8.12 | 13.77 |
| 1,024 | 4.78 | 5.86 | 7.55 | 11.39 | 18.20 |

Table 5.8: Total processing time (in seconds) spent by NEW1

| $K$ | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|
| Time(sec) | 0.01 | 0.02 | 0.04 | 0.11 | 0.24 | 0.52 | 0.94 | 1.70 | 2.59 | 3.87 |

Since $m$ and $n$ are fixed, a linear increase in time with respect to $K$ is theoretically estimated. Indeed, the time increases almost linearly while $K \leq 256$, but a slight deviation is observed for large $K$'s.

There are two reasons for such a non-linearity for large $K$'s. Note that in NEW1, all $O(m^2)$ strips are examined by lines 3–7 of Algorithm 30, but actual number of strips that require a hole creation (line 5) is dependent on $g_k$ and $i_k$ of each maximum subarray $M(k)$.

| $m = 128$ | $n = 64$ | $n = 128$ | $n = 256$ | $n = 512$ | $n = 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.08 | 0.17 | 0.34 | 0.73 | 1.42 |
| 2 | 0.08 | 0.17 | 0.34 | 0.74 | 1.39 |
| 4 | 0.08 | 0.17 | 0.34 | 0.74 | 1.42 |
| 8 | 0.09 | 0.17 | 0.35 | 0.74 | 1.43 |
| 16 | 0.09 | 0.18 | 0.35 | 0.74 | 1.41 |
| 32 | 0.11 | 0.20 | 0.37 | 0.75 | 1.45 |
| 64 | 0.14 | 0.24 | 0.40 | 0.79 | 1.52 |
| 128 | 0.21 | 0.33 | 0.52 | 0.93 | 1.66 |
| 256 | 0.28 | 0.44 | 0.71 | 1.21 | 2.07 |
| 512 | 0.41 | 0.59 | 0.95 | 1.60 | 2.74 |
| 1,024 | 0.60 | 0.83 | 1.28 | 2.11 | 3.74 |

| $m = 256$ | $n = 64$ | $n = 128$ | $n = 256$ | $n = 512$ | $n = 1,024$ |
|---|---|---|---|---|---|
| $K = 1$ | 0.35 | 0.71 | 1.40 | 2.91 | 5.64 |
| 2 | 0.35 | 0.71 | 1.40 | 2.88 | 5.54 |
| 4 | 0.35 | 0.71 | 1.38 | 2.88 | 5.63 |
| 8 | 0.37 | 0.73 | 1.39 | 2.94 | 5.60 |
| 16 | 0.40 | 0.75 | 1.41 | 2.93 | 5.66 |
| 32 | 0.43 | 0.78 | 1.47 | 3.01 | 5.72 |
| 64 | 0.51 | 0.90 | 1.58 | 3.05 | 5.86 |
| 128 | 0.72 | 1.15 | 1.83 | 3.38 | 6.00 |
| 256 | 1.07 | 1.66 | 2.37 | 4.09 | 6.73 |
| 512 | 1.65 | 2.41 | 3.31 | 5.66 | 8.49 |
| 1,024 | 2.69 | 3.68 | 4.94 | 7.82 | 11.64 |

Table 5.9: Total processing time (in seconds) spent by NEW2

For optimization purposes, an additional condition was used to implement line 5, which reads,

```
5: if not($i < g_k$ or $g > i_k$) and $M_{g,i} > 0$ then  create a hole $(h_k, j_k)$ in $T_{g,i}$
```

This extra condition prevents updating $T_{g,i}$ if the current root of a tournament, $M_{g,i}$, is non-positive. As $T_{g,i}$ will no longer produce a positive maximum sum, we do not need to waste $O \log n)$ time for updating it. Note that this extra condition becomes increasingly effective for large $K$, as many tournaments may have a non-positive root after many updates have been performed.

Hence, each iteration of the outer-most loop (lines 1–8) would require somewhere between $O(m^2)$ and $O(m^2 \log n)$ time, not exact $O(m^2 \log n)$ time. This explains the non-linear increase in time for large $K$'s.

As expected in Section 4.7, NEW2 has a slight performance improvement to NEW1, especially for relatively large $K$. Notice that the running times are almost constant for small $K$. In such a case, the hole-lists associated with each first-level tournament are near-empty, making $2dMaxSelectHeap()$ in Algorithm 32 take negligible amount time.

Due to large memory requirement, the physical memory is totally consumed by two new algorithms when $m$, the number of rows in the array, is 512. Note that HEAP in the 2D $K$-OMSP had the same problem and failed to run for an array of size $512 \times 512$. The optimization technique based on the sampling pre-process used for the 2D $K$-OMSP can not be applied. In the case of the 2D $K$-DMSP, many subarrays that appear to be promising in the early stage will be eventually eliminated over time when an overlapping subarray with greater sum is selected and holes are created as a result. Hence we can not easily predict if a strip has absolutely no chance of producing a subarray that will be included in the final $K$ subarrays.

The improvement for $K > n$ described in Section 4.6.4 is not implemented as it inevitably adds extra memory requirement. Even with this idea incorporated, little change is expected for Table 5.8 and Table 5.9, as there are few cases for $K > n$.

### 5.4.1 Experiment with Image Data

If we have some prior knowledge of the data to process, further optimization is possible. To illustrate this point, let us test the algorithm with image data and discuss some implementation-level optimization techniques.

From the USC-SIPI image database[§], we chose 85 image files under *Aerials* and *Miscellaneous* categories. The image sizes vary between $256 \times 256$

[§] Signal & Image Processing Institute, University of Southern California. Retrieved from the institute website: `http://sipi.usc.edu/database`

| $m = 256$ | $n = 256$ | $n = 512$ |
|---|---|---|
| $K = 1$ | 1.31 (0.01) | 2.72 (0.05) |
| 2 | 1.39 (0.04) | 2.82 (0.07) |
| 4 | 1.45 (0.06) | 2.89 (0.09) |
| 8 | 1.52 (0.08) | 2.97 (0.11) |
| 16 | 1.58 (0.10) | 3.04 (0.13) |
| 32 | 1.66 (0.12) | 3.13 (0.15) |
| 64 | 1.79 (0.15) | 3.26 (0.17) |
| 128 | 1.97 (0.18) | 3.46 (0.21) |
| 256 | 2.28 (0.21) | 3.79 (0.25) |
| 512 | 2.83 (0.28) | 4.44 (0.32) |
| 1024 | 4.13 (0.30) | 5.47 (0.46) |

Table 5.10: Average processing time (in seconds) and standard deviation (inside parenthesis) for processing image files by NEW2

and $2250 \times 2250$, but they all have been resized to $256 \times 256$ and $256 \times 512$[¶]

When an image file is loaded, we obtain the luminance value ($Y = 0.30R + 0.59G + 0.11B$) of each pixel and subtract an anchor value from each pixel, to ensure that the array is not all non-negative. The processed 2D array is then fed to NEW2. In this experiment, the sum of the average pixel value and the standard deviation was used as the anchor value. This makes the input 2D array slightly negative-biased and tends to extract tighter subarrays.

The average processing time and the standard deviation for varying $K$ are given in Table 5.10.

Considering the standard deviation, this result is fairly consistent with Table 5.9 which was obtained from processing the random data. For larger $K$, the image data processing is observed to be slightly faster. This is possibly attributed to the second-level heap. Typically in the image data, pixels with similar luminance level are likely to be positioned close to each other. In the second-level heap, nodes associated with the tournament containing high luminance pixels are located near the root and other nodes related to the low luminance regions will be occupying the lower part of the heap. After we

[¶] In graphics, the dimension of the image is usually represented by *width*×*height*, so it is indeed $512 \times 256$ according to the norm.

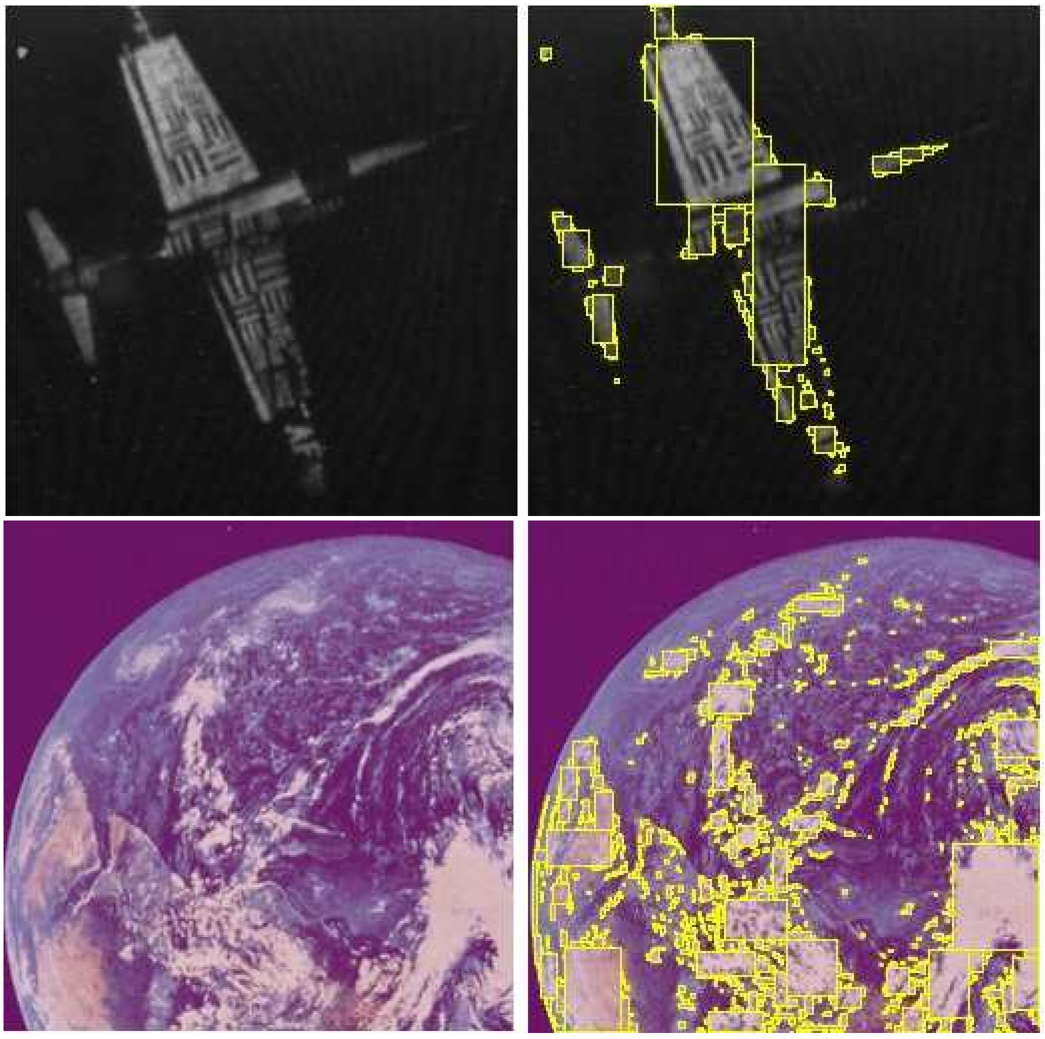


Figure 5.1: Example: Processing image data by NEW2

create a hole in the tournament referenced by the root node, the tournament is still likely to contain some large positive subarrays, resulting in a positive new value for the node. When the second-level heap is adjusted, the updated root node will not be pushed down too far. Such a behavior is less pronounced with the random data, and may explain the slightly faster processing speed when a large $K$ is tried with an image data.

### 5.4.2 Implementation-level Optimization

Considering the characteristics of the image data, some optimization techniques can be proposed.

#### Filtering and choice of a good anchor value

It was previously pointed out that the sampling technique, as it is, is not applicable to the 2D $K$-DMSP. However, we can still reduce some number of strips at the early stage. As a pre-process, we compute the first maximum sum only in each strip by Algorithm 7 or Algorithm 8. During the pre-process, only $O(n^2)$ memory space is used. If the first maximum sum in a strip is not positive, we avoid making a tournament out of this strip.

Obviously, this simple *filtering* is *weak* and may only achieve satisfactory result when the array contains predominantly many negative values. With this filtering incorporated, in some favorable conditions[‖], NEW1 and NEW2 were able to compute an array of size $256 \times 1024$ for $K = 1024$ at significantly improved 0.98 seconds. Under the same condition, NEW2 managed to process an array of size $1024 \times 1024$ in 5.84 seconds for $K = 1024$. Processing an array of such a size was previously impossible without running into the memory problem, but in this particular case, more than 90% strips were discarded before tournament is being built. When the default setting for the random number generation is used, no strip is discarded at all, and both NEW1 and NEW2 fail to process an array of size $512 \times 512$.

The filtering pre-process may be usefult for image data processing, as we can make the array filtering-friendly by subtracting a large anchor value.

With addition of the filtering pre-process while preserving the same setting as the experiment above, the processing time and standard deviation were obtained as shown in Table 5.11. Compared with Table 5.10, a significant reduction in processing time is observed. The standard deviation is higher as the effect of the filtering pre-process varies depending on each image data.

With larger anchor value, the filtering pre-process is expected to be more

[‖] The elements in the input array are generated by `rand()%100-70`, such that the array may consist of many negative values

| $m = 256$ | $n = 256$ | $n = 512$ |
|---|---|---|
| $K = 1$ | 0.52 (0.36) | 1.10 (0.77) |
| 2 | 0.53 (0.37) | 1.11 (0.78) |
| 4 | 0.53 (0.39) | 1.11 (0.79) |
| 8 | 0.56 (0.41) | 1.14 (0.82) |
| 16 | 0.57 (0.43) | 1.16 (0.84) |
| 32 | 0.62 (0.45) | 1.19 (0.86) |
| 64 | 0.66 (0.49) | 1.27 (0.90) |
| 128 | 0.75 (0.51) | 1.39 (0.90) |
| 256 | 0.88 (0.58) | 1.56 (0.99) |
| 512 | 1.15 (0.72) | 1.78 (1.15) |
| $1,024$ | 1.52 (1.01) | 2.27 (1.38) |

Table 5.11: With filtering: Average processing time (in seconds) and standard deviation (inside parenthesis) for processing image files by NEW2

effective, reducing the memory usage as well as running time. However, some loss of precision may occur, if far too a large anchor value is chosen. What defines the precision may be a complex topic itself, so we simply regard it as the satisfactory level of matching our perception.

If we have some domain knowledge of image data to process, we may be able to determine a good anchor value heuristically, ensuring a good balance of speed and precision. For example, an astronomy image typically contains predominantly many low-valued pixels. The average pixel value of the satellite image of size $m = 512$ and $n = 1024$ shown in Figure 5.2[**] is 14, which is very low. If we use 128 as the anchor value, after subtraction, the resulting 2D array is heavily biased to negative values. The filtering pre-process becomes very effective, managing to reject $129,001$ out of $131,328$ total strips. Only $2,327$ strips survived the filtering. This 98% rejection rate greatly reduced the computation speed as well as the memory usage, resulting in $K = 1,024$ maximum subarrays computed in 1.28 second. Each maximum subarray roughly corresponds to a major city in the world, matching our perception fairly well.

[**] Earth at Night: November 27, 2000. Astronomy Picture of the Day.
National Aeronautics and Space Administration (NASA).
Retrieved from `http://antwrp.gsfc.nasa.gov/apod/ap001127.html`.

### Amalgamation

In a typical image data, the pixel value distribution is reasonably smooth in a sense that neighboring pixels are likely to have close values. When we build a tournament for each strip, we may consider amalgamating all consecutive positive values and all consecutive negative values. For example, the array $a = \{3, 51, -41, -57, 52, 59, -11, 93, -55, -71, 21, 21\}$ is amalgamated into $\{54, -98, 111, -11, 93, -55, -71, 42\}$. In this case, when we build a tournament with the latter array, we have 33% reduction in the memory usage. Note that the original indices should be preserved after the amalgamation. When a tournament is build, the $from$ and $to$ attributes of the leaf node representing 54 should be 1 and 2 respectively.

This idea can be easily generalized to a strip with multiple rows. An interval where the strip prefix sum monotonously increases is amalgamated into one positive value. Conversely, an interval with monotonously decreasing strip prefix sum is converted to a single negative value. Considering the smooth value distribution observed in a typical image data, this amalgamation technique is expected to give a significant saving in the memory usage. When a hole is created in a strip, a special care needs to be taken to update the tournament correctly. This will involve some modification to Algorithm 28.

### Height constraints

Let us suppose that we know the approximate dimension of the maximum subarrays to extract. In particular, as discussed in Section 3.9, if the height of the maximum subarrays should be between $L_h$ and $U_h$, the total number of strips to process is only $O(mD_h)$ where $D_h = U_h - L_h$. If $D_h$ is considerably smaller than $m$, this leads to a sharp reduction in the number of tournaments to build, i.e., from $O(m^2)$ to $O(mD_h)$. This idea obviously requires more involved user input, as the user should know the approximate height of the expected maximum subarrays in advance. For this reason, this idea may be only suitable if the user has some prior knowledge of the image data that he/she wishes to process.

The experiment of the last two techniques are left as future work.

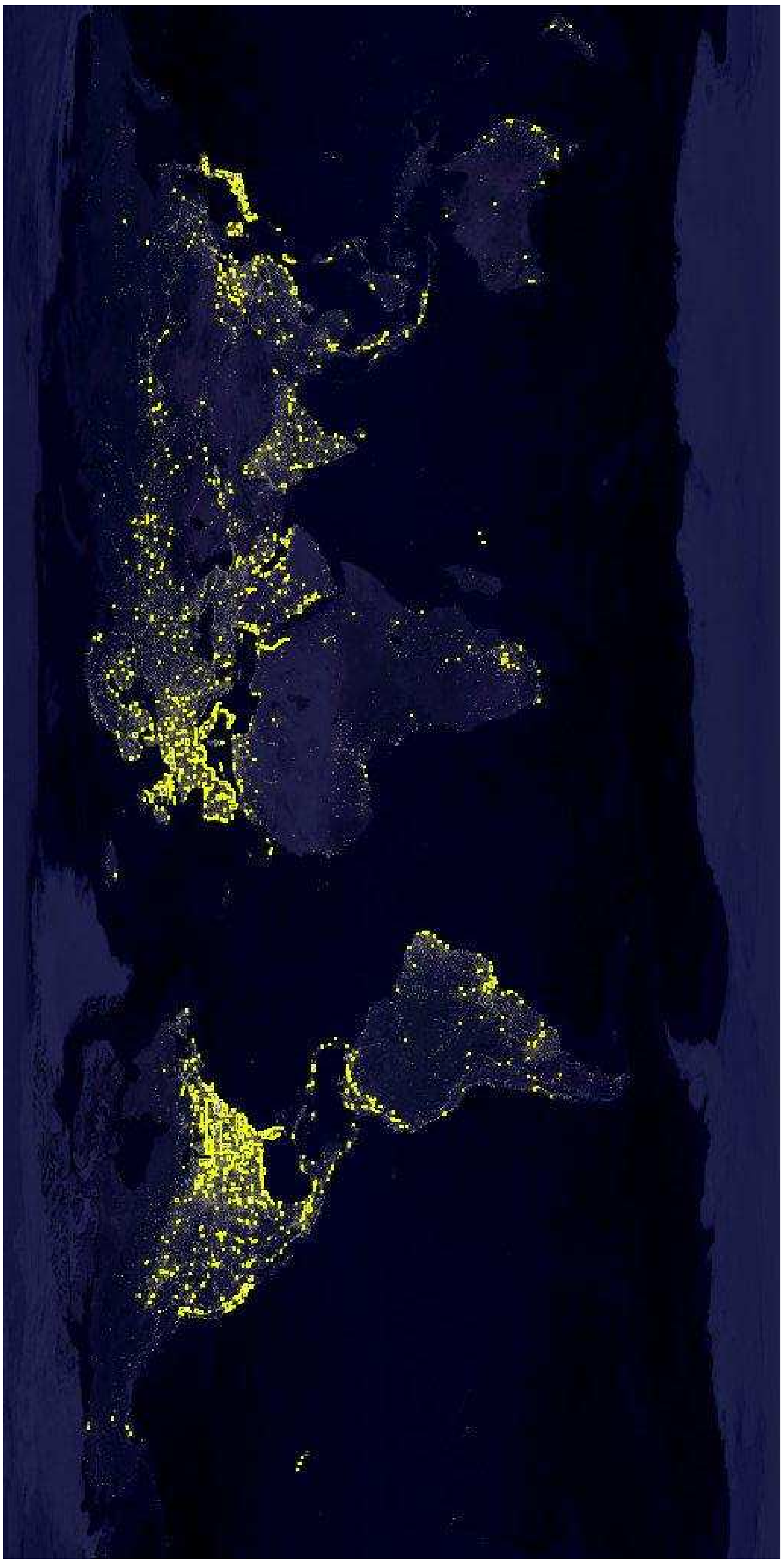


Figure 5.2: $K$=1,024 brightest spots by NEW2 with filtering pre-process.

# Chapter 6

# Concluding Remarks and Future Work

## 6.1 Concluding Remarks

In Part I, we identified two categories of $K$-maximum subarray problem, the $K$-overlapping maximum subarray problem ($K$-OMSP) and the $K$-disjoint maximum subarray problem ($K$-DMSP), and presented various techniques to speed up computing these problems.

We adopted the general framework based on the subtraction of minimum prefix sum from each prefix sum to produce candidates for the final solution. Given the framework, various methods have been investigated to compute the $K$-OMSP efficiently. The major challenges involved in the problem are reduction of the total number of candidates generated and maintenance of the list of minimum prefix sums.

We initially studied the iterative framework based on Algorithm 15 and improved the time complexity from $O(Kn)$ to $O(K^2 + n \log K)$ through a simple technique based on sampling. Each prefix sum $sum[i]$ produces the maximum candidate $cand_i[1]$ by subtracting the minimum prefix sum prior to $sum[i]$, that is $min_i = \text{MIN}\,\{sum[0], ..sum[i-1]\}$. Among these $n$ samples, we execute the linear time selection algorithm (Algorithm 13) and select the $K$-th largest. We can identify only $(K-1)$ prefix sums that requires further candidate generation, then the total number of candidates produced are bounded by $O(K^2)$. It was also discussed that a 2-3 tree is a suitable data structure to efficiently maintain the minimum prefix sums.

We further explored this simple idea and reduced the number of candidates to $O(K \log K)$ by incorporating the efficient method for selection in sorted columns. When we have a two-dimensional array of $Kn$ candidates, $A$, whose element $A[i][j]$ is $cand_j[i] = sum[j] - min_j[i]$. Each column is sorted in non-increasing order. The algorithm for selection in sorted columns proposed by Frederickson and Johnson [38] examines small number of elements in rows 1,2,4.. and rearranges the array such that $O(K \log K)$ maximum elements in $A$ would be relocated at the top-left corner of $A$. However, the selection technique is not directly applicable, since we do not have $A$ pre-built, and cannot afford to spend time to fully build it.

We observed that a slight modification to the original selection technique alleviates this requirement, and showed that the modified selection technique can be applied *on-the-fly* such that elements in rows 1,2,4.. are being produced and examined at the same time, i.e. being sampled. The sampling and rearrangement can be done in $O(K \log K)$ time, if an appropriate data structure for the minimum prefix sums is used. We proposed to use a persistent 2-3 tree to maintain the minimum prefix sums. The $i$-th version of the minimum prefix sums, $min_i$, is maintained by the $i$-th version of the 2-3 tree. This combination of the sampling/rearrangement technique and a persistent 2-3 tree resulted in $O((n+K)\log K)$ time.

Our final effort to improve this result included the design of a new framework for selection in Cartesian sums. We devised a simple algorithm using a two-level heap to retrieve the $K$ maximum values in $X + Y = \{x_i + y_j \mid x_i \in X,\ y_j \in Y\}$ in sorted order, in $O(n + K \log \min(K, n))$ time. This simple algorithm is easy to modify to compute the $K$-OMSP. We use a *persistent tournament* to maintain different versions of $min$, and compute the $K$ maximum subarray in $O(n + K \log \min(K, n))$ time.

For two-dimensions, we explored two methods to speed up. It is trivial to extend a 1D algorithm to 2D through the strip separation technique, which would result in $O(n^2)$ factor multiplied to the complexity of 1D.

A simple sampling technique can be applied, such that we select $K$ strips that would contain the final $K$ maximum sums. This way, we can disqualify $O(\frac{n(n+1)}{2} - K)$ strips at early stage, and can solve the problem in $O(n^3)$ time for $K \leq \sqrt{\frac{n^3}{\log n}}$.

Through a technique based on the selection in a two-level heap, we prepare a heap-tournament pair for each strip, and build a second-level heap with values of the root of each heap-tournament pair. The second-level heap has $O(n^2)$ elements, and its root holds the maximum sum in 2D. To get the next maximum sum in 2D, we update the second-level heap as well as the first-level heap-tournament pair. The total time is $O(n^3 + K \log \min(K, n))$, that is $O(n^3)$ for $K \leq \frac{n^3}{\log n}$. In general, we need $O(n^{2d-1} + K \log \min(K, n^d))$ time to compute the $K$-OMSP in a $d$-dimensional array of size $n \times \cdots \times n$.

When we have the length constraints $L$ and $U$, such that the length of $K$ maximum subarrays would be between $L$ and $U$, the proposed algorithms can be extended with minor modifications. Based on the algorithm that finds the minimum prefix sums satisfying the length constraints by Fan et al.[32], the first maximum subarray meeting the length constraints is computed in $O(n)$ time. For each position of the prefix sum, we can retrieve a tournament tree that maintains the associated minimum prefix sums meeting the length constraints easily, and compute $K$ maximum subarrays meeting the length constraints in $O(n + K \log \min(K, n - L))$ time.

A tournament is proved to be an efficient data structure for the $K$-OMSP, and we applied the same data structure to compute the $K$-DMSP. While Ruzzo and Tompa's algorithm [82] computes all disjoint maximum subarrays in linear time, the problem definition is slightly different. As their algorithm does not produce solutions in order, it would spend extra $O(K \log K)$ time if the sorted order is required. This algorithm is also optimized for 1D and seems difficult to extend to higher dimensions. We devised a new algorithm for 1D that computes the first maximum subarray recursively and builds a tournament during the computation. The subsequent maximum subarray can be retrieved in $O(\log n)$ time by updating the tournament, meaning that $K$ maximum subarrays can be computed in $O(n + K \log K)$ time. As the solutions are sorted by default, this new algorithm is considered asymptotically equivalent to Ruzzo and Tompa's.

This new algorithm for 1D is extended to 2D through the strip separation, resulting in $O(n^2)$ tournaments. When a maximum subarray $(r_1, c_1)|(r_2, c_2)$ is selected, we create a hole $(c_1, c_2)$ in a tournament $T_{g,i}$ whose row coverage $(g, i)$ overlaps $(r_1, r_2)$. In the worst case, we would need to create a hole in $O(n^2)$ tournaments before the next maximum subarray can be computed. This results in $O(n^3 + Kn^2 \log \min(K, n))$ time, which is $O(n^3)$ time for $K \leq \frac{n}{\log n}$. Considering that the only currently available alternative is $O(Kn^3)$ time process based on repeated application of Kadane's algorithm, this algorithm has enhanced practicality. Marginal improvement of $O(n^3 + \min(K,n) \cdot n^2 \log \min(K,n))$ time has been also presented through the incorporation of *union/find* method. In general, we need $O(n^{2d-1} + n^{2d-2} \cdot \min(K,n) \log n)$ time to compute the $K$-DMSP in a $d$-dimensional array of size $n \times \cdots \times n$.

The new algorithms developed from this thesis contribute to understanding the $K$-maximum subarray problem associated with the selection problem. It has been identified that the $K$-OMSP and $K$-DMSP are essentially the extended version of the selection problem, especially the tournament-based selection described in Chapter 2.

## 6.2 Future Works

We concluded that both 1D $K$-OMSP and $K$-DMSP can be computed in $O(n + K \log K)$ time for $K \leq n$. This complexity suggests that the tournament(or heap)-based selection of $K$ maximum elements (Algorithm 11) is computationally equivalent to both kinds of $K$-MSP's.

This observation poses an interesting open problem. Considering that a linear time selection of the $K$-th maximum is possible (Algorithm 13), is it also possible to compute the $K$-th maximum subarray, not $K$ maximum subarrays in sorted order, in $O(n)$ time? It will be also interesting to see if we can compute $K$ maximum subarrays in $O(n+K)$ time without the sorted order requirement*.

For the $K$-DMSP, the selection of the $K$-th maximum subarray can be done in $O(n)$ time through the combination of Ruzzo and Tompa's algorithm (Algorithm 25) and the linear time selection (Algorithm 13). We find all $\bar{K}$-disjoint maximum subarrays, then select the $K$-th ($K \leq \bar{K}$) maximum. Our algorithm for the $K$-DMSP can not enjoy any speed up as it must produce $K$ maximum subarrays in sorted order.

The biggest difference between the $K$-OMSP and the $K$-DMSP is, however, the maximum value of $K$, i.e., $\bar{K}$. In the $K$-OMSP, $\bar{K} = \frac{n(n+1)}{2}$, while in the $K$-DMSP, $\bar{K} = \frac{n}{2}$. Hence, it is unrealistic to expect an algorithm that

* See the end of this section for the latest results.

computes the $K$-th maximum subarray in the overlapping condition in $O(n)$ time when $K > n$. For this reason, an attempt to establish an $O(n)$ time will need to impose a condition on the range of $K$, such that $K \leq n$.

An interesting related problem will be a combined version of the $K$-OMSP and $K$-DMSP, that is the problem of $K$ *partially overlapping* maximum subarrays. For example, we wish to find the $k$-th maximum subarray, that is at most 50% overlapping the previously determined solutions. This problem seems not very difficult for $K = 2$. When the first maximum subarray $M(from, to)$ is found, we create a hole $(\frac{from+to-1}{2}, to)$ and find the next maximum subarray in $a[1..to]$, such that it is disjoint from the hole. This maximum subarray overlaps $(from, to)$ at most by 50%. Similarly, we create another hole $(from, \frac{from+to+1}{2})$ and find the next maximum subarray in $a[from + to + 1..n]$. This again overlaps $(from, to)$ at most by 50%. The greater of two is then the desired second maximum sum. A solution for general $K$ is open.

While the algorithms for 1D have the same complexity in both cases, the $K$-OMSP and $K$-DMSP, there is a gap between the presented algorithms for 2D. Namely, $O(n^3 + K \log \min(K, n))$ time versus $O(n^3 + Kn^2 \log \min(K, n))$ time. It is expected that the second term in the complexity of the 2D $K$-DMSP can match that of the 2D $K$-OMSP. If it is achieved, as $\bar{K} \leq n^2/2$, the total complexity will be simply $O(n^3)$ for any $K$.

Nevertheless, it appears that the $n^2$ factor in the second term is inherently difficult to eliminate unless a different approach is taken. The simple technique based on strip separation inevitably introduces $\frac{n(n+1)}{2}$ strips to handle. In the 2D $K$-OMSP, we could reduce the number of necessary strips by the sampling technique as it is guaranteed that the $K$ maximum subarrays are contained in the $K$ selected strips. It was observed that this strategy is, however, not applicable to the 2D $K$-DMSP. An attempt to improve the $n^2$ factor, therefore, might involve a new framework specifically developed for 2D problem, such as the approach based on the distance matrix multiplication (DMM).

## 6.3 Latest Results

After this thesis was submitted for the examination, some new results have been reported.

As expected in Section 6.2, $O(n+K)$ worst-case time algorithm has become finally available for the 1D $K$-OMSP when the sorted order requirement is waived. Brodal and Jørgensen [24] achieved $O(n+K)$ time by combining the partially persistent *Iheap*, a binary heap optimized for insertion operation, and Frederickson's algorithm for selection in a min-heap [39]. It is also reported that Hsiao-Fei Liu and Kun-Mao Chao independently observed the same result by applying Eppstein's algorithm for finding the $k$ shortest paths in a directed acyclic graph (DAG) [31].

For the same problem, Lin and Lee [73] presented an expected $O(n \log n + K)$ time algorithm, which needs only $O(n)$ space. For $K \leq n$, however, this is not as efficient as the $O(n + K \log \min(K, n))$ worst-case time solutions given in Section 3.6 and [26, 16][†].

[†] Latest form of these papers are also available as [27] and [17].

# Part II

# Parallel Algorithms

# Chapter 7

# Parallel Algorithm

## 7.1 Introduction

You cannot have a baby in one month by getting nine women pregnant.
–Fred Brooks, "The Mythical Man-Month"[*]

Gordon E. Moore, the co-founder of INTEL, predicted that the speed of a central processing unit (CPU) will be doubled every 24 months [76], and this prediction has been empirically observed for the last four decades in terms of the increase in frequency of operation. However, considering the clock speed alone, *Moore's law* is unlikely to hold in the foreseeable future, as improving the performance of individual processors has become increasingly difficult. Higher clock speed corresponds to exponential increases in temperature, which makes it very difficult to make a CPU that runs reliably at a fast clock speed. There are fundamental barriers too. In terms of size, transistors built on a CPU may reach the limits of miniaturization at atomic levels. Also the speed of electronic circuit is limited by the speed of light.

Recently, INTEL and ADVANCED MICRO DEVICES (AMD) have instead opted for another strategy to increase the computation speed by providing multiple computing cores packaged in a single CPU. A multi-core CPU enables parallel processing of multi-threaded application, thus results in increased computation speed. With dual-core CPUs becoming increasingly popular in the consumer market, such as INTEL's CORE DUO and AMD's ATHLON 64 X2 processors, processors with tens or hundreds cores are expected within the next decade [56].

[*] Sometimes quoted as "One woman can have a baby in nine months, but nine women can't have a baby in one month." or "The bearing of a child takes nine months, no matter how many women are assigned"

While the *parallel computing* with a multi-core CPU has become a recent phenomenon in the consumer market, the history of parallel computing can be traced back to early 1960s. Daniel Slotnick at the University of Illinois designed two early parallel computers, SOLOMON in early 1960s and ILLIAC IV in early 1970s. At Carnegie-Mellon University, C.MMP and CM* were constructed during the 1970s. In the early 1980s, Caltech built the COSMIC CUBE, the ancestor of multi-computers built by AMETEK, INTEL, and NCUBE. Brief history of early parallel computers is given in [80].

It should be noted that having multiple processors does not automatically mean faster computation. In fact, not all problems can be parallelized. This is because many algorithms are inherently sequential in nature. The quotation in the beginning of this chapter, often referred to as Brooks' law, describes this situation colloquially. Bearing of a baby is inherently sequential process, thus having multiple women cannot speed up. More formally, Amdahl's law [4] addresses this problem. The speedup possible through parallel processing is limited by the portion of the computation that must be performed sequentially.

Let $T_{seq}$ be the fastest known worst-case running time of a sequential algorithm for one problem. Obviously, the best upper bound on the parallel time achievable using $P$ processors is $T_{par} = O(T_{seq}/P)$. A parallel algorithm that achieves this running time is said to be *optimal.* The total cost of a parallel algorithm is thus defined as $C$, such that,

$$C = PT_{par}$$

Here, if $C = T_{seq}$, the parallel algorithm is optimal. In practice, linear speedup, the speed-up proportional to the number of processors is difficult to achieve, and indeed is the major challenge.

Often, a parallel algorithm can be constructed by redesigning a sequential algorithm to make effective use of parallel hardware. Previously known parallel algorithms for the maximum subarray problem (MSP) [103, 77, 79] are also derived from the sequential solutions. While these previous results are made for both one- and two-dimensional problems, the parallel computation for one-dimensional problem is less interesting since the sequential algorithm

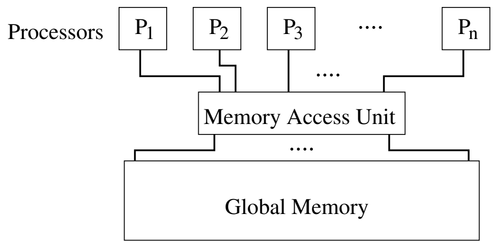


Figure 7.1: The structure of PRAM machine. It consists of a set of processors connected to a global memory through a memory access unit. All memory accesses are assumed to take $O(1)$ time.

can already computes in $O(n)$ time.

In the remaining part of this thesis, we briefly describe these parallel algorithms for the MSP and propose a new parallel algorithm for the two-dimensions. Since a parallel algorithm inherently is designed for a specific architecture, we describe several architectures adopted by the previous algorithms.

## 7.2 Models of Parallel Computation

### 7.2.1 Parallel Random Access Machine(PRAM)

The random access machine (RAM) is an abstract machine, a traditional sequential model of computation. This model is also called the von Neumann model. The RAM contains a single processor, which operates under the control of a sequential algorithm where one instruction at a time is issued. The parallel RAM (PRAM) is a parallel extension of the RAM. The PRAM was developed in order to provide a platform upon which people could design theoretical algorithms that would behave as predicted by the asymptotic analysis on real parallel computers. The advantage of the PRAM is that it ignores communication issues and allows the user to focus on the potential parallelism available in the design of an efficient solution to the given problem. The PRAM maintains $n$ processors, $P_1$,$P_2$...$P_n$, each of which is identical to a RAM processor and a global memory shared by all processors. Each processor is assumed to access to every memory location in $O(1)$ time.

The processors in the PRAM are not directly connected to each other, meaning that if two processors wish to communicate during the computation, the communication must be made through the global memory. This condition, however, may cause the classical *race condition*, where two processors are trying to read/write to the same memory location. We discuss the three PRAM models that are designed to resolve the memory access conflict issue.

- CREW: Concurrent Read, Exclusive Write. Multiple processors are allowed to read from the same memory location during a clock cycle, but only one processor is allowed to write to a given memory location. If multiple processors attempt to write to the same memory location during the same clock cycle, it is considered to be an error.

- EREW: Exclusive Read, Exclusive Write. This is the most restrictive PRAM model as it forbids both concurrent reads and concurrent writes. This restriction presents a challenge to design efficient algorithms for the EREW PRAM model.

- CRCW: Concurrent Read, Concurrent Write. To allow multiple processors to write to the same memory location during the same clock cycle, various schemes have been proposed, such as Priority CW, Common CW, Arbitrary CW, Combining CW.

In fact, ERCW model is also possible, but it is hardly interesting. Indeed, it is hard to imagine a machine that is sophisticated enough to support the concurrent write, but not the easier concurrent read.

Despite the popularity as bases for parallel algorithm design, no actual PRAM machines have been built. Possibly, a PRAM machine with relatively few processors may be built, but such a machine cannot scale to large numbers of processors while preserving uniformly fast access time to memory. This is mainly due to current technological limitations in connecting processors and memory. In the end, it is not feasible to allow $n$ processors to access any $n$-memory locations simultaneously.

Nevertheless, the PRAM is a powerful model for studying the logical structure of parallel computation under conditions that permits theoretically optimal communication. If a PRAM algorithm is cost optimal, it may be a suitable basis for the design of a parallel program targeted to a real parallel computer [80].

### 7.2.2 Interconnection Networks

As the global memory in the PRAM model is technically difficult, it is natural to consider distributed-memory machines as an alternative to construct a real parallel computer. An *interconnection network* is a parallel computer made of processor-memory pairs (which we shall refer to as "processors") connected to each other in a well-defined pattern. Unlike the PRAM model where a global memory is used, parallel computers based on interconnect network need to employ some kind of routing to enable passing of messages between nodes that are not directly connected.

Clearly, the topology providing more connections mean faster communication between processors. For example, if processors are organized in a complete graph, where every processor is connected to all others, it guarantees one-step communication between any pair of processors. However, such a system is very expensive to build and scale.

There are number of ways to organize processors, such as a linear array, ring, mesh, tree, pyramid, mesh-of-trees and hypercube. As the description of all these processor organizations is beyond the scope of this thesis, we describe the mesh and the hypercube models with respect to the parallel algorithm for the MSP.

#### Mesh

A *mesh* is a 2-dimensional, checkerboard-type organization of processors, where each processor has four neighbors.

Communication is allowed only between neighboring processors and there is no global communication, even control flow is strictly local.

The mesh is therefore very restrictive in comparison to other topologies. However, this structure is considered as a good option to develop a chip-based parallel algorithm, because its regular grid structure is easy to be built on a one-level chip. In case of a complete graph, for example, this is infeasible

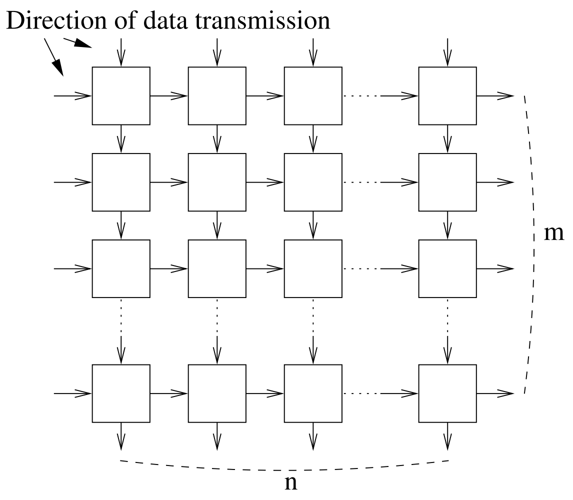


Figure 7.2: Mesh network

if there are more than 4 processors, since more than 4 nodes in a complete graph means it is not planar.

The mesh is often implemented as a *systolic array* [68]. The word "systole" refers to the rhythmical contraction of the heart by which the blood is forced onward and the circulation kept up[†]. Systolic array imitates the pumping of blood. A processor starts computation when data from its neighbors arrives, then outputs results to other neighbors. The regular communication pattern with the systolic array means it requires no routing.

Examples of mesh algorithms usually involve two-dimensional array inputs, such as the matrix multiplication [36, 78, 97, 98] and the all-pairs shortest-paths (APSP) problem [92] achieving the time bound of $\Omega(n)$.

We save further description for the detailed discussion in Chapter 8.

## Hypercube

A *hypercube* of size $n$ consists of $n$ processors, where $n$ is a power of 2. Let the index of a processor $P_{b(i)}$, $b(i)$ be a binary representation of an integer $i$ for $0 \leq i < n$. For example, when $n = 4$, processors are indexed $P_{00}$, $P_{01}$,

[†] systole. (n.d.). Merriam-Webster's Medical Dictionary. Retrieved October 25, 2006, from Dictionary.com website: `http://dictionary.reference.com/browse/systole`

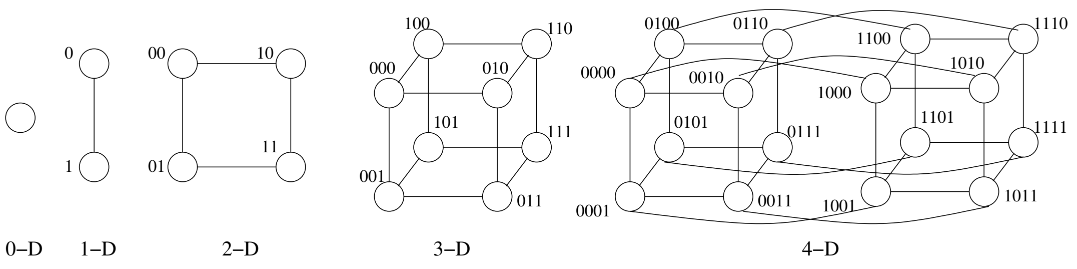


Figure 7.3: Hypercube-connected architectures of zero, one, two , three and four dimensions.

$P_{10}$ and $P_{11}$. Note that the length of the binary index is $\log_2 n$.

Processors A and B are connected if the binary indices differ in exactly one position. So suppose than $n = 8$, then the processor $P_{011}$ is connected to $P_{111}$, $P_{001}$ and $P_{010}$.

Constructing a hypercube is easily done in recursive manner. A hypercube of size $n$ can be constructed from two hypercubes of size $n/2$, which we refer to as $H_1$ and $H_2$.

We place $H_1$ and $H_2$ side by side and add a leading zero to every binary index of $H_1$ and a leading 1 to every binary index of $H_2$. Finally, we connect the processors of $H_1$ and $H_2$ if two processors differ only in their leading bit of the index.

Note that a processor in a hypercube of size $n$ is connected to exactly $\log_2 n$ other processors. Thus all processors are considered identical in terms of the number of attached communication links.

## 7.3 Example: Prefix sum

As a working example of the parallel algorithms designed for a specific architecture, we describe the prefix sum computation. This choice is based on two reasons. Firstly, the prefix sum provides the basis for the MSP computation. Secondly, the prefix sum computation has been well studied for various architectures and almost all parallel computers contain hardware implementation for this problem. Note that the prefix sum computation can be done in $O(n)$ time sequentially, i.e. $T_{seq} = O(n)$. We will also examine the cost of the

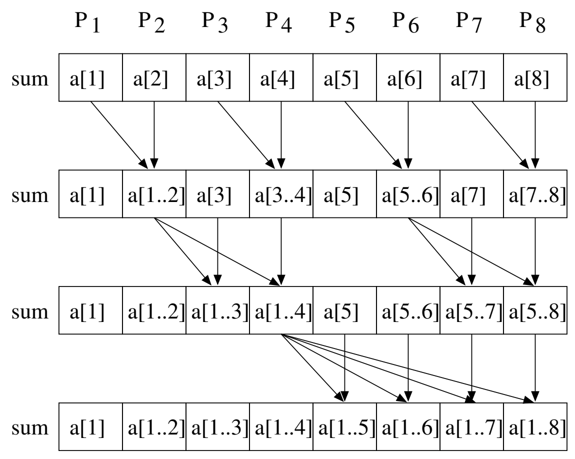


Figure 7.4: Computing prefix sums on a CREW PRAM, for $n = 8$. We denote $\sum_{i=p}^{q} a[i]$ by $a[p..q]$ in the figure.

parallel algorithm is optimal.

### 7.3.1 PRAM

The PRAM prefix sum computation is covered in various texts [52, 80, 72, 57], and we present the description of CREW PRAM algorithm given in [52].

Without the loss of generality, we assume that $n$ is power of 2. We first present an $n$-processor and $O(\log n)$ time algorithm (Algorithm 33).

**Algorithm 33** CREW PRAM algorithm to find prefix sums of an $n$ element list using $n$ processors.

```
procedure prefixsum($p$,$q$) begin
// computes $\sum_{i=p}^{p} a[i]$,$\sum_{i=p}^{p+1} a[i]$,...$\sum_{i=p}^{q} a[i]$. Set $sum[i] \leftarrow a[i]$ initially
1: if $p = q$ then return $sum[p]$
2: call prefixsum($p$,$\frac{p+q-1}{2}$) and prefixsum($\frac{p+q+1}{2}$,$q$) in parallel
3: for all $P_i$, where $\frac{p+q+1}{2} \leq i \leq q$ in parallel do
4:     $sum[i] \leftarrow sum[i] + sum[\frac{p+q-1}{2}]$
5: end for
end
```

This algorithm runs recursively. It continuously halves the input until only one element remains (line 1), then a processor $P_i$ adds the prefix sum of the first half, $sum[\frac{p+q-1}{2}]$, which holds the sum $a[p] + .. + a[\frac{p+q-1}{2}]$, to $sum[i]$ whose current value is $a[\frac{p+q+1}{2}] + .. + a[i]$. Figure 7.4 illustrates this recursive procedure.

As shown in the figure, one memory location may be concurrently read by multiple processors, but not written concurrently. So this is a CREW PRAM algorithm. Let $T(n)$ be the run time of the above algorithm. Line 2 takes $T(\frac{n}{2})$ time as two recursive calls with halved input are executed in parallel. The rest takes $O(1)$ time. The following recurrence relation for $T(n)$ holds.

$$T(n) = T\left(\frac{n}{2}\right) + O(1), \quad T(1) = 1$$

This solves to $T(n) = O(\log n)$. Then $C = PT_{par} = n \log n = O(n \log n)$. Here, we have $C > T_{seq}$, meaning this parallel algorithm is not cost-optimal. Note that $P_1$ is redundant as it merely sits idle throughout the computation. Indeed, only $n/2$ processors are involved in the computation at each step, and it is possible to run this algorithm with $n/2$ processors. Still, there is no improvement to the total cost.

Now we consider a cost-optimal algorithm that needs only $\frac{n}{\log n}$ processors , while keeping the run time the same.

On termination of Algorithm 34, we regard the final prefix $sum[1..n]$ as the concatenation of the result computed by each processor, such that,

$$sum[1..n] = sum_1[1.. \log n] \cup sum_2[1.. \log n] \cup .. \cup sum_{\frac{n}{\log n}}[1.. \log n]$$

Each step of the algorithm is bounded by $O(\log n)$ time, therefore $O(\log n)$ total time using $\frac{n}{\log n}$ CREW PRAM processors.

A similar algorithm with the same cost running on EREW PRAM is possible [57].

This result can be extended to two-dimensions. For an array of size $n \times n$, we can easily obtain an algorithm employing $O(n^2/\log n)$ processors running in $O(\log n)$ time. In phase one, we assign $n/\log n$ processors to each row such that each group of $n/\log n$ processors compute the row-wise prefix sums, $r[i][j] = a[i][1] + .. + a[i][j]$ on each row $1..n$ in parallel by Algorithm 34. In phase two, we process $r[1..n][1..n]$ vertically. Notice that

**Algorithm 34** Optimal $O(\log n)$ time prefix computation using $\frac{n}{\log n}$ CREW PRAM processors

1. We first divide the input into $\frac{n}{\log n}$ groups each containing $\log n$ elements. Each processor is allocated to each group, and computes the prefix sums of its $\log n$ assigned elements sequentially. Note that all processors performs this sequential computation simultaneously. This takes $O(\log n)$ time. Let the result be contained in $sum_i[1..\log n]$ for the $i$-th group.

2. Each processor collects the last prefix sum of each group as a delegate, forming an auxiliary list containing $\frac{n}{\log n}$ elements. A total of $\frac{n}{\log n}$ processors run Algorithm 33 on this auxiliary list, taking $O(\log \frac{n}{\log n})$=$O(\log n)$ time. Let the result be contained in $sum_{del}[1..\frac{n}{\log n}]$.

3. Processor $P_i$ $(2 \leq i \leq \frac{n}{\log n})$ collects $sum_{del}[i-1]$, and performs $sum_i[1..\log n]$ + $sum_{del}[i-1]$ sequentially. Each processor spends $O(\log n)$ time, where all processors work in parallel. Let the result be stored back in $sum_i[1..\log n]$.

$sum[i][j] = r[1][j] + .. + r[i][j]$, the prefix sum of $r[1..i][j]$. Again, we assign $n/\log n$ processors to each column and each group of $n/\log n$ processors perform similar parallel computation. Both phase one and two are $O(\log n)$ time, and we have $C = O(n^2)$, an optimal cost.

### 7.3.2 Mesh

The two-dimensional structure of a mesh makes it readily suited to the two-dimensional prefix sum computation. Certainly, a one-dimensional input can be processed on a mesh [52], but it would be less interesting when our motivation is to speed up the computation of the two-dimensional MSP through parallel processing. Hence we focus on the two-dimensional prefix sum in the following.

A processor in the mesh is often referred to as a *cell*. A cell located at the $i$-the row and the $j$-th column is denoted by $cell(i, j)$. Each cell contains a set of registers that is private to itself. For the prefix sum computation,

**Algorithm 35** Mesh prefix sum algorithm using $m \times n$ processors

```
1: for all $cell(i, j)$ where $1 \le i \le m$ and $1 \le j \le n$ in parallel do
2:     if $cell(i, j)$ is active then
3:         $r(i, j) \leftarrow r_{in}(i, j-1) + a(i, j)$
4:         $s(i, j) \leftarrow s_{in}(i-1, j) + r(i, j)$
5:     end if
6: end for
```

we have register $a$, $r$ and $s$, and these registers at *cell*$(i, j)$ are denoted by $a(i, j)$, $r(i, j)$ and $s(i, j)$ respectively. We assume that $a[i][j]$ resides on the $a$ register of *cell*$(i, j)$, i.e. $a(i, j) = a[i][j]$. We store the row-wise (horizontal) prefix sum in $r(i, j)$ and the prefix sum in $s(i, j)$.

We design the mesh network to behave as a systolic array, such that the outcome of a cell is collected by its neighboring cells. Scheduling when each cell starts computation is one of the most significant design issue to ensure the correct result.

We initiate the *control signal* at the left boundary of the network, that is gradually propagated towards right. Each cell becomes *active* when it receives the signal. An active cell can communicate with its neighbors and perform internal computation. The data flow from one cell to its neighbors takes one communication *step*, and the control signal also takes one step to transmit from one cell to its right neighbor. In general, at step=$\alpha$, we consider all cells in the first MIN $(\alpha, n)$ columns are active.

An active cell sends a *packet* containing set of register values. There are two types of packet in Algorithm 35. A packet delivered to right contains the register value $r$, and the other delivered downwards contains the register value $s$.

We assume that $r$ and $s$ registers in the network are initialized to 0. We also assume that $s(0, 1..n) = 0$ and $r(1..m, 0) = 0$, to deal with the index 0 at lines 3 and 4. We denote the name of a register contained in the incoming packet with a subscript "*in*", for example, $r_{in}(i-1, j)$ means the $r$ register value from *cell*$(i-1, j)$.

Here, each prefix sum $sum[i][j]$ will be retrieved from $s(i, j)$, and the sum of all elements, $sum[m][n]$ will be computed at step=$m + n - 1$.

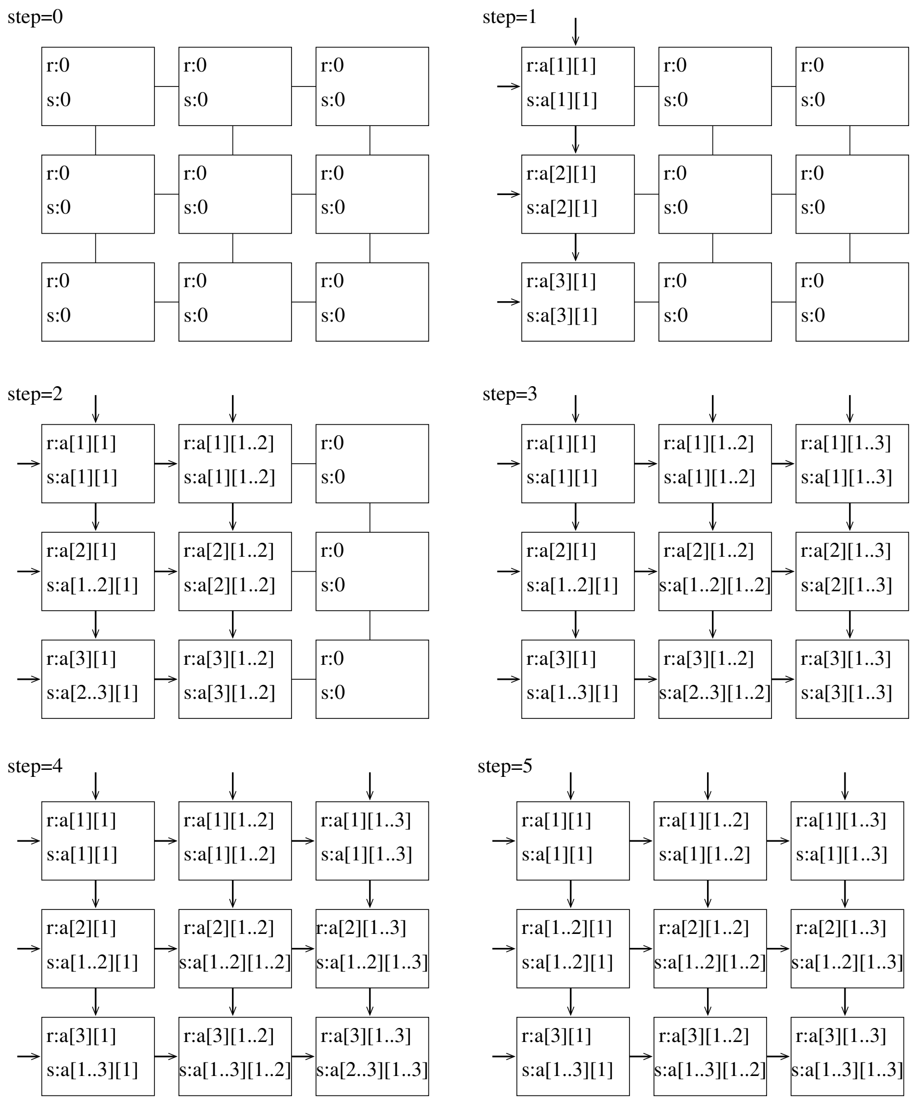


Figure 7.5: Computing prefix sums on a $3 \times 3$-processor mesh. $a(i, j)$ is not shown. We denote $\sum_{i=r_1}^{r_2} \sum_{j=c_1}^{c_2} a[i][j]$ by $a[r_1..r_2][c_1..c_2]$ in the figure.

In a interconnection network, the *network diameter* is the maximum distance between any pair of processors. The mesh of size $m \times n$ has *diameter* $m+n-2$. For $s(m, n)$ to be the correct prefix sum, $s(1, 1)$, the sum of $a[1][1]$ alone must travel to $cell(m, n)$. Since $s(1, 1)$ is computed at step 1, the total number of steps is therefore $(m + n - 2) + 1$.

If the input array is of size $n \times n$, this algorithm is $O(n)$ time with $O(n^2)$

processors, and the total cost is $O(n^3)$. This is clearly non-optimal, as for two-dimensions, we know that $T_{seq} = O(n^2)$.

The major limitation for the mesh, in this case, is the network diameter. In order to reduce the network diameter, the only option is to have smaller sized mesh network, and assign more work to each cell. The extra work allocated to a cell is computed sequentially. We call such a configuration a *coarse-grained* mesh. On the contrary, when a single input element is assigned to each cell, it is *fine-grained.* The extreme cases, a coarse-grained mesh can be of size 1, or a sequential machine, which runs the algorithm in $O(n^2)$ time. A move from Algorithm 33 to obtain Algorithm 34 with CREW PRAM is based on a similar motivation.

While it may be possible to achieve a cost-optimal algorithm by finding a balance between the network diameter and the amount of work each processor must perform, we omit the further discussion. Later we will see that this non-optimal mesh algorithm still suits as a basis for the MSP in Chapter 8.

Once the prefix sum is computed by the mesh, we could use the same architecture to compute the MSP. However, we design a parallel algorithm that computes the MSP spending the minimum number of parallel steps.

### 7.3.3 Hypercube

To compute the prefix sum $sum[1...n]$, we prepare a $d(= \log n)$-dimensional hypercube, which has $n$ processors. Each processor is labeled with $b(i)$, the binary representation of $i$ for $0 \leq i \leq n-1$.

Each processor $P_{b(i)}$ maintains two values, $s(i)$ and $t(i)$. In the beginning, $s(i)$ and $t(i)$ are both set to $a[i+1]$. In the end, we want each local value $s(i)$ to be $sum[i+1]$. Each processor exchanges data with its all directly connected neighboring processors, one at a time. The *partner* processor is determined by line 5 of Algorithm. The bit-wise operator "XOR" returns 1 if "1 XOR 0" or "0 XOR 1". For example, if $n = 8$ and $d = 3$, the 5-th processor $P_{b(5)} = P_{101}$ communicates with $P_{100}$ first, because 101 XOR 001$(= 2^0)$ is 100, followed by $P_{111}$ and $P_{001}$ in order. Inside the sequential "for" loop between lines 4 and 10, hence $x = 4, 7$ and 1 respectively at each iteration.

At the end of a communication step, $t(x)$, the incoming value from a

**Algorithm 36** Hypercube prefix sum algorithm using $n$ processors

```
1: for all $P_{b(i)}$ where $0 \leq i \leq n-1$ in parallel do
2:    $s(i) \leftarrow a[i+1]$
3:    $t(i) \leftarrow s(i)$
4:    for $j \leftarrow 0$ to $d-1$ do
5:       $partner \leftarrow P_{b(x)}$ where $b(x) \leftarrow b(i)$ XOR $2^j$
6:       send $t(i)$ to $partner$
7:       receive $t(x)$ from $partner$
8:       $t(i) \leftarrow t(i) + t(x)$
9:       if $x < i$ then $s(i) \leftarrow s(i) + t(x)$
10:   end for
11: end for
end
```

neighboring processor is added to the result $s(i)$ only if the message comes from a processor with a smaller label than that of the recipient processor, i.e. $x < i$. The contents of the outgoing message kept in $t(i)$ are updated with every incoming message. For instance, after the first communication step, $P_{000}$, $P_{010}$, and $P_{100}$ do not add the data received from $P_{001}$, $P_{011}$ and $P_{101}$ to their $s(i)$. However, their $t(i)$ are updated. The detailed description of the algorithm may be found in [67, 81, 28].

As each processor communicates with its all $\log n$ directly-connected processors sequentially, the overall time $T_{par}$ is $O(\log n)$ given $n$ processors. Again, $C = n \log n$ and this algorithm is not optimal. Certainly, optimal algorithm can be designed using a coarse-grained hypercube, where we have only $p$ ($p < n$) processors and each processor maintains $n/p$ elements, a similar transition from Algorithm 33 to Algorithm 34. Qui and Akl presented that $O(n/p + \log p)$ times is possible [79], meaning that when $p = \log n$, $T_{par} = O(n/\log n)$ with $O(\log n)$ processors. Here $C = O(n)$ and therefore it is optimal. Qui and Akl applied this optimal parallel prefix sum algorithm as a basis for the MSP computation on the hypercube.

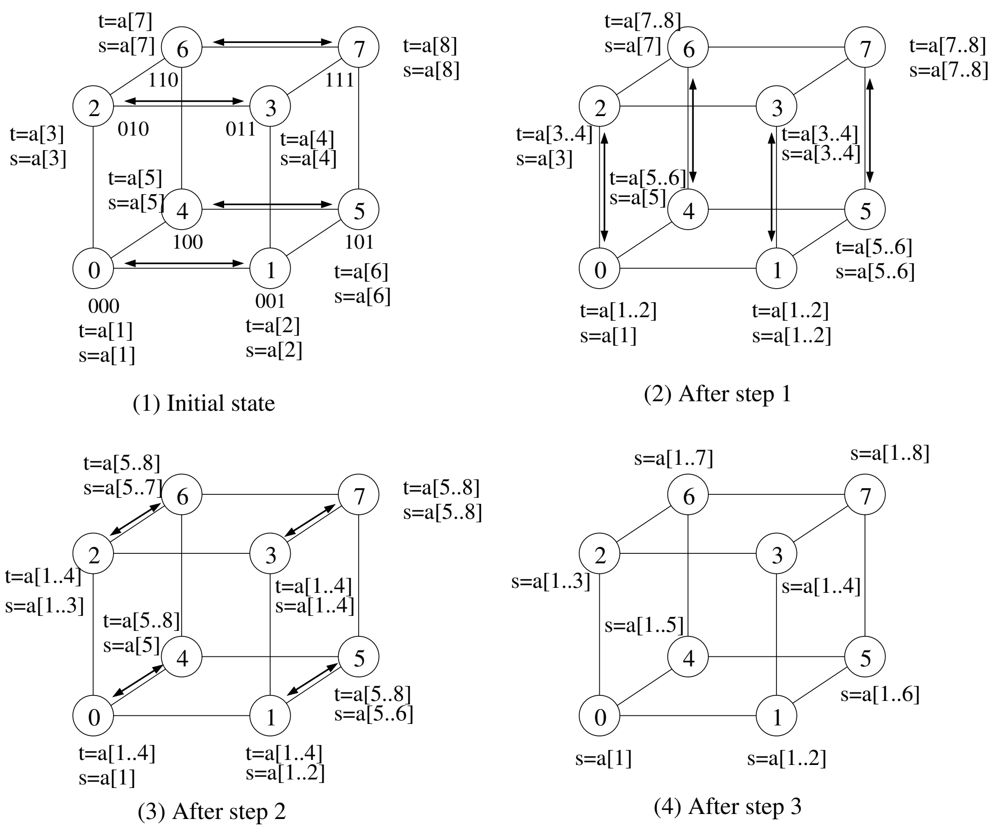


Figure 7.6: Computing prefix sums on an eight-processor hypercube. We denote $\sum_{i=p}^{q} a[i]$ by $a[p..q]$ in the figure.

# Chapter 8

# Parallel Algorithm for Maximum Subarray Problem

## 8.1 Parallel Maximum Subarray Algorithm

In the previous chapter, we have reviewed $O(\log n)$ time PRAM and hypercube parallel algorithms for the prefix sums using $O(n/\log n)$ processors. Considering the close relation between the prefix sum computation and the MSP, a parallel algorithm for the MSP is well expected.

Perumalla and Deo [77]*, Wen [103] and Qiu and Akl [79] gave optimal parallel algorithms. They all run in $O(\log n)$ time with $O(n/\log n)$ processors for one-dimension, and $O(\log n)$ time with $O(n^3/\log n)$ processors for two-dimensions. For easy comparison, we present Table 8.1.

Table 8.1: Parallel Algorithms for the MSP

| | Model | 1D | | | 2D | | |
|---|---|---|---|---|---|---|---|
| | | $T_{par}$ | $P$ | Read | $T_{par}$ | $P$ | Read |
| P. & D.[77] | PRAM | $O(\log n)$ | $\frac{n}{\log n}$ | ER | $O(\log n)$ | $\frac{n^3}{\log n}$ | CR |
| Wen [103] | PRAM | $O(\log n)$ | $\frac{n}{\log n}$ | ER | $O(\log n)$ | $\frac{n^3}{\log n}$ | ER |
| Q. & A.[79] | Hypercube | $O(\log n)$ | $\frac{n}{\log n}$ | - | $O(\log n)$ | $\frac{n^3}{\log n}$ | - |

For CREW PRAM, EREW PRAM, and Hypercube, we know the following fact.

* It should be noted that the authors of [77] adopted the parallel prefix sum computation by Ladner and Fisher [71] which takes $O(\log n)$ time, but with $O(n)$ processors for Phase 1 of Algorithm 37. It is believed that the reference to [71] is incorrectly given, and the authors meant an optimal EREW PRAM algorithm [57].

**Theorem 8.1.1.** *Prefix (or suffix) sums of $n$ elements can be computed in $O(\log n)$ time using $O(n/\log n)$ processors.*

Note that Theorem 8.1.1 also holds true with any associative operator $\oplus$, meaning that prefix minimum, for example, can be computed similarly, when MIN operator is used instead of $+$.

Let us examine Algorithm 37, an EREW PRAM-type algorithm by Perumalla and Deo [77]. This can be viewed as a parallel version of Algorithm 3.

**Algorithm 37** EREW PRAM Algorithm for 1D MSP

```
//Phase 1
for all $P_i$, where $i \leftarrow 1$ to $n/\log n$ in parallel do
    compute prefix sum $sum[1..n]$
end for
//Phase 2
for all $P_i$, where $i \leftarrow 1$ to $n/\log n$ in parallel do
    compute $min[1..n]$ //$min[i] = \text{MIN}\{sum[0], ..sum[i-1]\}$
end for
//Phase 3
for all $P_i$, where $i \leftarrow 1$ to $n/\log n$ in parallel do
    compute $cand[1..n]$ //$cand[i] = sum[i] - min[i]$
end for
//Phase 4
for all $P_i$, where $i \leftarrow 1$ to $n/\log n$ in parallel do
    compute $M[1..n]$ //$M[i] = \text{MAX}\{cand[1], ..cand[i]\}$
end for
output $M[n]$
```

Theorem 8.1.1 is applied to Phases 1,2 and 4, where the associative operators $+$, MIN and MAX are used respectively, hence they take $O(\log n)$ time each. Each processor in Phase 3 computes $sum[x] - min[x]$ sequentially $\log n$ times, hence it is $O(\log n)$ time too.

The hypercube algorithm by Qiu and Akl [79] follows essentially the same framework, with some detailed techniques inherent to the hypercube model. Qiu and Akl also presented similar results for other interconnection networks, including pancakes and stars.

Wen's EREW PRAM algorithm [103] employs Smith's recursive linear time algorithm (Algorithm 5) [84] and does not separate four different phases. Its full details are omitted.

When the one-dimensional MSP is solved in $O(\log n)$ time with $O(n/\log n)$ processors, it is fairly straightforward to extend to two-dimensions. In a two-dimensional array of size $n \times n$, there are $O(n^2)$ one-dimensional problems, i.e. strips. We prepare $O(n^3/\log n)$ processors, where each group of $O(n/\log n)$ processors are assigned to solve each strip in $O(\log n)$ time. The total cost is then $O(n^3)$.

## 8.2 Introduction: Mesh Algorithms for the 2D MSP

All the previous parallel algorithms for the MSP are extended from the parallel prefix sum algorithms for a specified architecture.

For an input array of size $n \times n$, the mesh algorithm for the 2D prefix sums (Algorithm 35) is non-optimal with $T_{par} = O(n)$ and $P = O(n^2)$. While this parallel algorithm alone may not be considered efficient, its simplicity makes it a good platform upon which we build a mesh algorithm for the MSP.

Considering that the sequential algorithms (Algorithm 9, Algorithm 8) are $O(n^3)$ time, designing a mesh algorithm for the 2D MSP achieving $O(n)$ time with $O(n^2)$ processors is a worthwhile endeavor, which we set as the objective of this chapter.

Such a mesh algorithm, at best, is arguably slower than other parallel algorithms that achieve $O(\log n)$ time. Still, $O(n)$ time computation with much less number of processors poses a reasonable compromise between the time and the hardware complexity. In addition, it is fairly straightforward to build a VLSI circuit that embeds a mesh algorithm, implying that the parallel algorithm of this type can be materialized at the fraction of the hardware cost of other parallel algorithms.

Another justification for making an algorithm for the 2D MSP can be made. It is known that the MSP, the distance matrix multiplication (DMM) and the all-pairs shortest-paths (APSP) problems are of the same sequential time complexity [89, 87, 94]. mesh algorithms for the matrix multiplication [36, 78, 97, 98] and the APSP [92] all achieve $\Omega(n)$ time. We are naturally compelled to solve this problem in a similar manner.

We assume the same mesh model given in Figure 7.2.2. Each processing unit, a *cell* is connected to four neighboring cells and exchanges data with them.

This is an asynchronous array processing which uses the flow of data to initiate the operation inside the cell, not the global control or global synchronization [69]. As only directly connected cells are allowed to communicate, no consideration for routing is required.

The data flow from one cell to its neighboring cells takes one communication *step*. The overall time spent by this network will be given in terms of the total number of steps.

We start with choosing a sequential algorithm upon which the mesh algorithm will be based. Kung [37] suggested a list of desirable properties of a "good" sequential algorithm for mesh implementation. He stated a desirable algorithm should

- be built upon a simple cell design
- not require many different types of cells
- have an iterative routine that is simple and regular

Iterative structure of Algorithm 8 and Algorithm 9 fit this criteria. The sub-cubic algorithms [89, 94] are not favored due to *divide-and-conquer*, that is, recursive nature.

In this chapter, we design a mesh algorithm based on Algorithm 8, which can be trivially derived from the mesh prefix sum algorithm (Algorithm 35). We also present a mesh-Kadane algorithm, a mesh implementation of Algorithm 9 in the later part of this section.

## 8.3 Mesh Implementation of Algorithm 8

Algorithm 37 suggested that the MSP is solved by three phases of prefix computation (Phases 1,2 and 4). By replacing the operator "+" with MIN and MAX for Phase 2 and Phase 4 respectively, it is easy to see that three

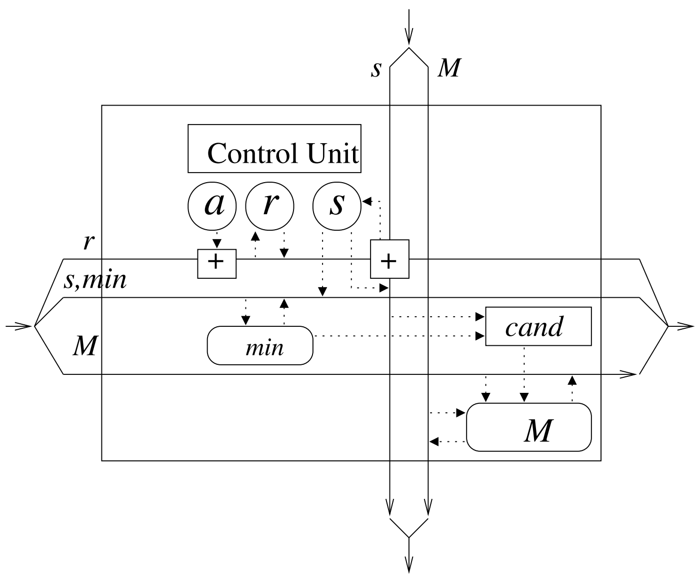


Figure 8.1: Design of a Cell

runs of Algorithm 35 can solve the MSP spending $3(m + n - 1)$ steps of communication.

In this section, however, we show that $m + n - 1$ steps are sufficient. This is only a constant factor improvement, not decreasing the asymptotic time complexity. It should be noted, however, that when a mesh algorithm is concerned, reduction by a constant factor is still regarded as a significant improvement.

In comparison to Algorithm 35, each $cell(i, j)$ has two more registers $min$ and $M$. We may prepare extra set of registers $(r_1, c_1)|(r_2, c_2)$ to hold the location of the subarray of sum $M$. For simplicity, let us neglect them in the following description. Note that the register names follow the usage of the variable names in Algorithm 8

Following the notation settled for Algorithm 35, we denote a register value of an incoming packet with a subscript "$in$", and each register of $cell(i, j)$ is denoted with a suffix $(i, j)$, for example, $a(i, j)$ etc. The register value at step $\alpha$ is marked with a subscript "$\alpha$" such as $r(i, j)_\alpha$.

The solid lines in Figure 8.1 deliver the value of a register, and the dotted lines show how these registers are updated.

**Algorithm 38** Mesh version of Algorithm 8: Initialize and update registers of $cell(i, j)$

```
Initialize:: $cell(i, j)$ do begin
1: $a(i, j) \leftarrow a[i][j]$ //value assigned to $cell(i, j)$, that is $a[i][j]$
2: $r(i, j) \leftarrow 0$//row-wise sum $a[i][1] + .. + a[i][j]$
3: $s(i, j) \leftarrow 0$//prefix sum
4: $min(i, j) \leftarrow 0$//minimum prefix sum
5: $M(i, j) \leftarrow 0$//Maximum sum
end
Update:: $cell(i, j)$ do begin
6: if $cell(i, j)$ is active then
7:     $r(i, j) \leftarrow r_{in}(i, j-1) + a(i, j)$
8:     $min(i, j) \leftarrow \text{MIN}\{s_{in}(i, j-1), min_{in}(i, j-1)\}$
9:     $s(i, j) \leftarrow s_{in}(i-1, j) + r(i, j)$
10:    $cand(i, j) \leftarrow s(i, j) - min(i, j)$
11:    $M(i, j) \leftarrow \text{MAX}\{M_{in}(i-1, j), M_{in}(i, j-1), M(i, j), cand(i, j)\}$
12: end if
end
```

### 8.3.1 Initialization

We assume there is a zeroth row and column in the input array, such that the input data $a[0][1..n] = 0$ and $a[1..m][0] = 0$. We also assume that each cell is aware of its coordinates $(i, j)$.

Actually, cells not on the boundary of the network have no direct communication line to the input array, hence line 1 is not executable. For the sake of simple description, for now, we assume that the input array is already residing in the network. We address the data loading issue in detail later in Section 9.2.

When initiated, $a(i, j)$ loads $a[i][j]$, and all others are initialized to 0.

### 8.3.2 Update

The update routine is an extended version of Algorithm 35. Instead of having separate three phases of parallel prefix computation exemplified in Algorithm 37, we integrate prefix sums, prefix minimum, and prefix maximum computations into one phase processing. Lines 8, 10 and 11 are the additional operations for such an integration. Since separated runs of similar prefix computations would require three times more communications, this integrated one-phase processing reduces the total number of communication steps, effectively maximizing the system throughput.

The control signal is initiated at the left boundary and propagated towards right each step, triggering the vertical data flow. Each cell in a column receives the control signal simultaneously from the left column. *Active* cells are those which received the control signal. Once activated, they remain active throughout the process.

The horizontal data flow is responsible for the prefix sum in each row, while the vertical data flow takes care of the strip prefix sums. Specifically, a packet sent to right contains register values $r$, $s$, $min$ and $M$, and the other sent down delivers $s$ and $M$. Note that $M$ is sent in both directions.

### 8.3.3 Boundary

In this mesh algorithm, each cell receives data from up and left neighbors, meaning that the cells on the left and top boundary will try to receive data from outside the network. However, a special treatment to the cells on the boundary should be avoided to keep every cell homogeneous. Maintaining homogeneity not only results in simpler hardware implementation, but also ensures good *modularity* and *regularity*, two essential properties to make the network indefinitely extensible. We resolve this issue by the imaginary 0-th row and 0-th column and appropriate register initialization. While the cells on the boundary gets data flows from these imaginary cells, the incoming register values should not affect the correct computation.

### 8.3.4 Analysis

To prove the correctness of Algorithm 38, we identify the invariant of each register at a specific time.

***Register*** $r(i,j)$ ***at step*** $\alpha$ Register $r(i,j)$ maintains the row-wise prefix sum. This register remains the initial value 0 until step $j$, when it receives

$r(i,j-1)_{j-1}$ from its left neighbor. At step $j$ and onwards, the value of $r(i,j)$ remains unchanged. It is easy to see that,

$$r(i,j)_\alpha = \sum_{q=1}^{j} a(i,q)$$

***Register*** $s(i,j)$ ***at step*** $\alpha$ Register $s(i,j)$ is designed to maintain the sum of row-wise prefix sums. At step $j$, $s(i,j) = r(i,j)$. Each subsequent step, $s(i-1,j)$ from the upper neighbor is delivered and added to $r(i,j)$.

$$s(i,j)_\alpha = \sum_{p=i+j-\alpha}^{i} r(p,j) = \sum_{p=i+j-\alpha}^{i} \sum_{q=1}^{j} a(p,q)$$

If $p = i+j-\alpha \leq 0$, we reset $p$ and start from 1. At step $j$, the value of $s(i,j)$ is the strip prefix sum $sum_{i,i}[j]$. Each step, the top-boundary of strip extends one row, such that $sum_{i-1,i}[j]$ at step $j+1$, $sum_{i-2,i}[j]$ at step $j+2$ etc.

We can formulate the value of $s(i,j)$ as follows.

**Lemma 8.3.1.** $s(i,j)_\alpha = sum_{g,i}[j]$ ,where $g = \text{MAX}\{1, i+j-\alpha\}$

At step $(i+j-1)$, $s(i,j) = sum_{1,i}[j]$, the prefix sum. This proves the correctness of Algorithm 35.

***Register*** $min(i,j)$ ***at step*** $\alpha$ Due to line 8, $min(i,j)_\alpha$ is computed by the following operation, which we simplify.

$$\begin{aligned}
min(i,j)_\alpha &= \text{MIN}\{s(i,j-1)_{\alpha-1}, min(i,j-1)_{\alpha-1}\} \\
&= \text{MIN}\{s(i,j-1)_{\alpha-1}, \text{MIN}\{s(i,j-2)_{\alpha-2}, min(i,j-2)_{\alpha-2}\}\} \\
&= \text{MIN}\{s(i,j-1)_{\alpha-1}, s(i,j-2)_{\alpha-2}, min(i,j-2)_{\alpha-2}\} = ... \\
&= \text{MIN}\{s(i,j-1)_{\alpha-1}, s(i,j-2)_{\alpha-2}, .., s(i,1)_{\alpha-j+1}, s(i,0)_{\alpha-j}\} \\
&= \underset{0\leq q\leq j-1}{\text{MIN}}\{s(i,q)_{\alpha-j+q}\}
\end{aligned}$$

Combining with Lemma 8.3.1, we establish an invariant of $min(i,j)_\alpha$.

**Lemma 8.3.2.** $min(i,j)_\alpha = \text{MIN}_{0 \le q \le j-1} \{sum_{g,i}[q]\}$ ,where $g = \text{MAX}\{1, i+j-\alpha\}$

Lemma 8.3.2 means that the minimum of prefix sums in $sum_{g,i}[0..j-1]$ is stored in $min(i,j)_\alpha$.

***Register*** $cand(i,j)$ ***at step*** $\alpha$ Notice that both $s(i,j)_\alpha$ and $min(i,j)_\alpha$ are specific to the same strip that spans from row $g$ to $i$, and the subtraction of $s(i,j)_\alpha - min(i,j)_\alpha$ produces $cand(i,j)$, the candidate maximum subarray in that strip.

**Lemma 8.3.3.** $cand(i,j)_\alpha = sum_{g,i}[j] - \text{MIN}\{sum_{g,i}[0..j-1]\}$ ,where $g = \text{MAX}\{1, i+j-\alpha\}$

***Register*** $M(i,j)$ ***at step*** $\alpha$ Due to line 11, $M(i,j)_\alpha$ is computed by selecting the maximum of the four values.

$$M(i,j)_\alpha = \text{MAX}\ \{M(i-1,j)_{\alpha-1}, M(i,j-1)_{\alpha-1}, M(i,j)_{\alpha-1}, cand(i,j)_\alpha\}$$

To simplify this, we first remove $M(i,j)_{\alpha-1}$ using the recurrence relation and get,

$$M(i,j)_\alpha = \text{MAX}\{M(i-1,j)_{0..\alpha-1}, M(i,j-1)_{0..\alpha-1}, cand(i,j)_{1..\alpha}\}$$

We further simplify the recurrence relation. Notice that we can leave only $cand(i,j)_{j..\alpha}$ as $cell(i,j)$ is not active until step $j$, nullifying $cand(i,j)_{1..j-1}$.

$$M(i,j)_\alpha = \underset{1 \le p \le i, 1 \le q \le j}{\text{MAX}} \{0, cand(p,q)_{q..\beta}\}\ \text{, where } \beta = \alpha + (p+q) - (i+j)$$

Combining this with Lemma 8.3.3, we have the following lemma.

**Lemma 8.3.4.** *$M(i,j)_\alpha$ is the maximum sum in $a[g..i][1..j]$* ,where $g = \text{MAX}\{1, i+j-\alpha\}$

The maximum sum in $a[1..i][1..j]$ is found in $M(i,j)$ when $g=1$, which is at step=$i+j-1$. The maximum sum in the whole array is then retrieved from $M(m,n)$ at step $m+n-1$.

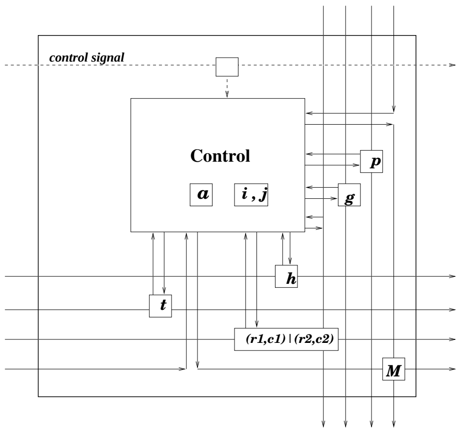


Figure 8.2: Design of a Cell

**Theorem 8.3.5.** *Algorithm 38 computes the maximum sum of* $a[1..m][1..n]$ *in* $m + n - 1$ *steps, and returns* $M(m, n)_{m+n-1}$ *as the solution.*

## 8.4 Mesh Implementation of Algorithm 9: Mesh-Kadane

Another version of mesh algorithm based on Algorithm 9, namely *mesh-Kadane* algorithm can be similarly designed.

### 8.4.1 Cell and its registers

Each $cell(i, j)$ has six registers $a, p, t, M, g$ and $h$. These register names are the same as the variables in Algorithm 9. Their specific roles are described in Algorithm 39.

The visual configuration of a cell may be similar to Figure 8.2. The solid lines in this figure show how the values of each register are conveyed and dotted line shows how a control signal is delivered. The control unit performs the operation, and controls the flow of data. Each small square on the line holds the current value of a register and send it to the control unit and the neighboring cells.

Again, the register of an incoming packet is marked with a subscript "*in*" such as $p_{in}(i, j)$. The register value at step $\alpha$ is marked with a subscript "$\alpha$".

Let the *scope* stand for the region that corresponds to a specific register.

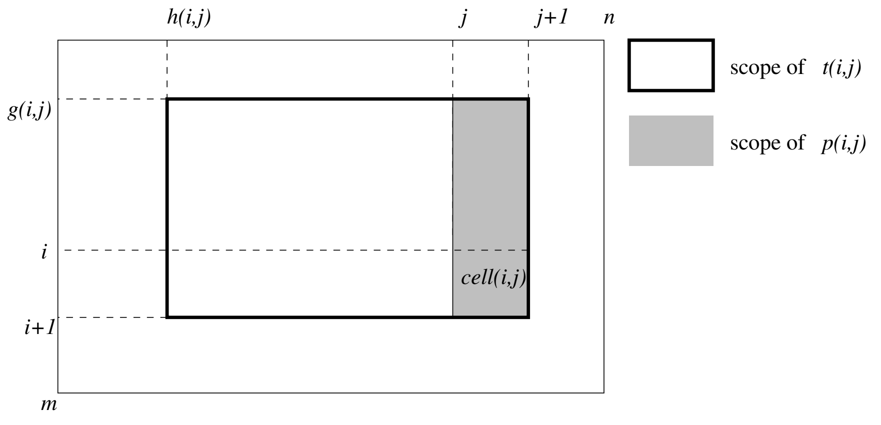


Figure 8.3: The scopes of $p(i, j)$ and $t(i, j)$ and scope registers $g(i, j)$ and $h(i, j)$

For example, $(r_1, c_1)|(r_2, c_2)$ is the scope of $M(i, j)$. For simplicity, we only focus on the maximum sum, leaving the location of the maximum subarray $(r_1, c_1)|(r_2, c_2)$ out of our discussion in the following description.

The registers $g(i, j)$ and $h(i, j)$ are *scope register*s. The scope of $p(i, j)$ is $(g(i, j), j)|(i, j)$ and that of $t(i, j)$ is $(g(i, j), h(i, j))|(i, j)$. In Figure 8.3, the scope of $t(i, j)$ is the rectangle surrounded by thicker lines, whereas the scope of $p(i, j)$ is the shaded rectangle.

The basic idea of our parallel algorithm is to execute Kadane's algorithm on horizontal strips in parallel, while propagating candidate maximum sum values down and right. Each cell has two operation routines, *Initialize* and *Update*. We show that asynchronous operation of cells correctly computes the maximum subarray. We describe each cell operation routine in Algorithm 39, which is executed for all $i$ and $j$ in parallel.

### 8.4.2 Initialization

Before initialization, we assume that the input array has the auxiliary 0-th row and 0-th column, such that $a[0][1..n] = 0$ and $a[1..m][0] = 0$, and the network also has corresponding 0-th row and column. *Initialize* routine in Algorithm 39 is designed to work universally.

When *cell*$(i, j)$ is initialized, for $0 \leq i \leq m$ and $0 \leq j \leq n$, the value of

**Algorithm 39** Initialize and update registers of $cell(i,j)$

```
Initialize:: $cell(i,j)$ do begin
1: $a(i,j) \leftarrow a[i][j]$ //value assigned to $cell(i,j)$, that is $a[i][j]$
2: $p(i,j) \leftarrow 0$ //column-wise sum $a[g][j] + ..a[i][j]$
3: $t(i,j) \leftarrow 0$ //horizontal accumulation of $p$'s
4: $g(i,j) \leftarrow i{+}1$ //$g$-th row is the top of the scope of $p(i,j)$ and $t(i,j)$
5: $h(i,j) \leftarrow j{+}1$ //$h$-th column is the left boundary of the scope of $t(i,j)$
6: $M(i,j) \leftarrow 0$ //the largest sum found so far
end
Update:: $cell(i,j)$ do begin
7: if $cell(i,j)$ is active then
8:     /* Vertical Data flow */
9:     $p(i,j) \leftarrow p_{in}(i\text{-}1,j) + a(i,j)$
10:    $g(i,j) \leftarrow g_{in}(i\text{-}1,j)$
11:    /* Horizontal Data flow */
12:    $t(i,j) \leftarrow t_{in}(i,j\text{-}1) + p(i,j)$
13:    if $t(i,j) > 0$ then $h(i,j) \leftarrow h_{in}(i,j\text{-}1)$
14:    else $t(i,j) \leftarrow 0$, $h(i,j) \leftarrow j{+}1$
15:    /* Update $M(i,j)$ */
16:    $M(i,j) \leftarrow \text{MAX}\{M_{in}(i\text{-}1,j), M_{in}(i,j\text{-}1), M(i,j), t(i,j)\}$
17: end if
end
```

$a[i][j]$ is copied to $a(i,j)$ (line 1). The register $p(i,j)$ and $t(i,j)$ are set to 0. The scope registers $g$ and $h$ indicate where the scope begins, thus need to be at least 1. They are initialized by line 4 and 5. The scopes of $p(i,j)$ and $t(i,j)$ are initially empty. Finally, the current maximum sum, $M(i,j)$ is set to 0.

### 8.4.3 Update

The flow of data takes place in two directions- from up and left. Update is performed when a cell gets data flow from its neighboring cells. In addition, we discuss how we deal with the cells on the top and the left boundaries which have no neighboring cells to receive register values from. Finally, we show the area obtained by this data flow policy is a legitimate subarray, a rectangle.

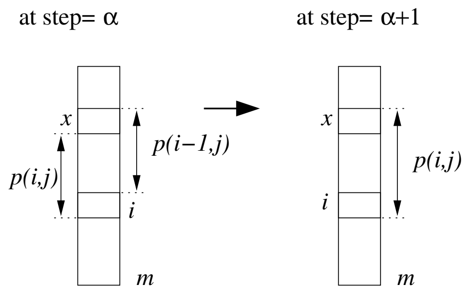


Figure 8.4: Update of register $p(i,j)$

The control signal is transmitted in the same way as the first mesh algorithm (Algorithm 38).

### Vertical data flow

The vertical data flow manages computation of the column-wise partial sum $p(i,j)$ and delivery of the current maximum sum $M(i,j)$ and the scope register $g(i,j)$. This data flow is triggered by the control signal.

If $cell(i,j)$ is active, it causes $p(i,j) \leftarrow p_{in}(i-1,j) + a(i,j)$, i.e., receives the column-wise partial sum of the upper neighbor and adds $a(i,j)$ to it. (line 9).

Suppose the scope of $p(i,j)_\alpha$, which is $p(i,j)$ at step $\alpha$, is $(x+1,j)|(i,j)$. It implies that $g(i,j)_\alpha = x+1$ as shown in Figure 8.4. Then the scope of $p(i-1,j)_\alpha$ is $(x,j)|(i-1,j)$, implying $g(i-1,j)_\alpha = x$. By adding $p_{in}(i-1,j)$ and $a(i,j)$, the scope of $p(i,j)$ effectively extends one cell upwards causing $g(i,j)_{\alpha+1} = x$ . Note that the scope of $p(i,j)_\alpha$ has the equal length for every $i$ in the same column.

The vertical data flow also delivers the value $M_{in}(i-1,j)$ for the selection of the maximum sum $M(i,j)$ given in line 16.

### Horizontal data flow

The horizontal data flow is the core of this algorithm which performs the original Kadane's solution. The data flow in this direction delivers $t(i,j)$, $h(i,j)$ as well as $M(i,j)$.

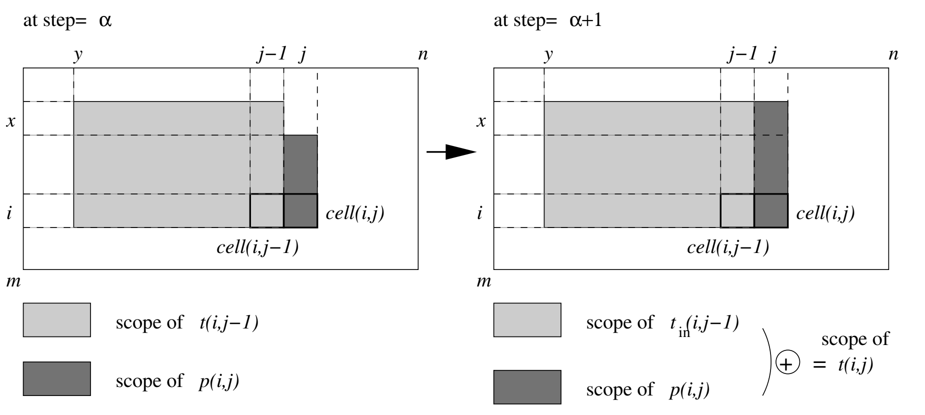


Figure 8.5: Update of register $t(i, j)$

Figure 8.5 illustrates how $t(i, j)$ is updated in general case, where $cell(i, j)$ gets the data flow from both directions, left and up.

Let us assume that at step $\alpha$, the register $t$ of the left neighbor, $t(i, j-1)_\alpha$, has the scope $(x, y)|(i, j-1)$ shaded in light gray. At the same time, the scope of $p(i, j)_\alpha$ is $(x+1, j)|(i, j)$ shaded in dark gray. Relevant scope registers are $g(i, j-1)_\alpha = x$, $h(i, j-1)_\alpha = y$ and $g(i, j)_\alpha = x+1$ respectively.

At step $\alpha+1$, the value of $t(i, j-1)_\alpha$ travels to $cell(i, j)$. This incoming value is denoted by $t_{in}(i, j-1)$. Meanwhile, the scope of $p(i, j)_\alpha$ extends one cell upwards and $p(i, j)_{\alpha+1}$ is obtained as shown in Figure 8.4. Then the vertical scope of $t_{in}(i, j-1)$ is precisely aligned with the scope of $p(i, j)_{\alpha+1}$, forming a rectangular merged area $(x, y)|(i, j)$. When $t(i, j)_{\alpha+1}$ is computed by line 12, this area becomes the scope of $t(i, j)_{\alpha+1}$.

Following the idea of Algorithm 9, if $t(i, j)_{\alpha+1}$ is non-positive, we no longer need to keep this value. In this case, we reset $t(i, j)_{\alpha+1}$ to 0. Depending on the value of $t(i, j)_{\alpha+1}$, the scope register $h(i, j)_{\alpha+1}$ is updated accordingly (lines 13, 14).

During the horizontal data flow, the value of maximum sum computed at the left neighbor is received. This value $M_{in}(i, j-1)$ along with $M_{in}(i-1, j)$ obtained from the vertical data flow are compared with the locally computed $t(i, j)$ and the current $M(i, j)$ to select the largest (line 16). In Section 8.4.4, we show the correct maximum sum is obtained this way.

### Timing

As shown in Algorithm 39 as well as Figure 8.5, $p(i,j)$ is updated prior to $t(i,j)$. The $i$-th row of the array lags one step behind the $(i-1)$-th row, and the $j$-th column lags one step behind the $(j-1)$-th column. The data flow in both directions takes exactly one step. This ensures that $t_{in}(i,j-1)$ and $p(i,j)$ are available when $t(i,j)$ is updated.

### Final Solution

After $m+n-1$ steps, all the data flow inside the network is completed and the final solution for the whole array $a[1..m][1..n]$ is held by the register $M(m,n)$. Details of time analysis is given in the next section.

## 8.4.4 Correctness of Algorithm 39

We define the invariants of each register of $cell(i,j)$ when the current time $\alpha$ exceeds the column number $j$ $(\alpha \geq j)$. In other cases, each register retains its initial value. With the invariants, we prove that Algorithm 39 correctly computes the maximum subarray.

***Register*** $g(i,j)$ ***at step*** $\alpha$ The initial value of $g(i,j)$, $g(i,j)_0$, is $i+1$ (line 4) and receives the value of $g(i-1,j)_{\alpha-1}$ from up to obtain $g(i,j)_\alpha$. The reception of $g(i-1,j)_{\alpha-1}$ starts at step $\alpha = j$ when $cell(i,j)$ becomes active. This effectively decreases the value of $g(i,j)$ by 1 each step. Note that $g(i,j)_\alpha$ can not be less than $g(0,j)$, whose value is initialized and fixed at 1. It is $\alpha = i+j-1$ when $g(i,j)_\alpha$ becomes 1. Once this point is reached, we consider this cell has completed the column-wise computation.

The value of $g(i,j)$ at step $\alpha$ is defined by,

$$g(i,j)_\alpha \quad = \quad i+1-(\alpha-j+1) = i+j-\alpha \tag{8.1}$$

***Register*** $p(i,j)$ ***at step*** $\alpha$ $p(i,j)$ at step $\alpha$ is the column-wise partial sum of $a[g(i,j)_\alpha..i][j]$, which is expressed as:

$$p(i,j)_\alpha \;=\; \sum_{q=g(i,j)_\alpha}^{i} a(q,j) \tag{8.2}$$

***Register*** $h(i,j)$ ***and*** $t(i,j)$ ***at step*** $\alpha$ As suggested by lines 13 and 14 of Algorithm 39, the register $h(i,j)$ and $t(i,j)$ are inter-related. Depending on the value of $t(i,j)$, the way $h(i,j)$ is updated is affected. On the other hand, $h(i,j)$ defines the scope of $t(i,j)$ such that $t(i,j)$ may represent the sum of subarray $(g(i,j),h(i,j))|(i,j)$.

The line 12 of the algorithm dictates how $t(i,j)$ is updated.

$$t(i,j)_\alpha = p(i,j)_\alpha + t(i,j-1)_{\alpha-1}$$

This is naturally equivalent to

$$\begin{aligned} t(i,j)_\alpha =& p(i,j)_\alpha + p(i,j-1)_{\alpha-1} + ... + p(i,h(i,j)_\alpha)_{\alpha-(j-h(i,j)_\alpha)} \\ =& \sum_{r=h(i,j)_\alpha}^{j} p(i,r)_{\alpha-j+r} \end{aligned} \tag{8.3}$$

By definition, the invariant of $t(i,j)$ at step $\alpha$ is given as follows. For simplicity, let us assume $t(i,j)_\alpha$ is 0 when $h(i,j)_\alpha > j$. Note that line 13 sets $h(i,j) \leftarrow j+1$ if $t(i,j)$ is non-positive. So the resulting $t(i,j)$ is always at least 0.

$$t(i,j)_\alpha = \sum_{q=g(i,j)_\alpha}^{i} \; \sum_{r=h(i,j)_\alpha}^{j} a(q,r) \tag{8.4}$$

Combining (8.1),(8.2) and (8.3), we can show that (8.3) and (8.4) are equivalent, which proves that the algorithm computes $t(i,j)$ correctly.

As $t(i,j)_\alpha$, the sum of subarray $(g(i,j)_\alpha, h(i,j)_\alpha)|(i,j)$, is the correct representation of $t$ in Algorithm 9, it inherits the same characteristics of $t$ given in Corollary 2.2.1, therefore,

**Corollary 8.4.1.** $t(i,j)_\alpha$ *is the maximum sum that ends at* $a[i][j]$ *with the top boundary* $g(i,j)_\alpha$*.*

When the top boundary $g(i,j)_\alpha$ is fixed, no subarray $(g(i,j)_\alpha, x)(i,j)$ for $x \neq h(i,j)_\alpha$ has greater sum than $t(i,j)_\alpha$. In order words, $h(i,j)_\alpha$ decides the scope that maximizes $t(i,j)_\alpha$.

***Register*** $M(i,j)$ ***at step*** $\alpha$ In Algorithm 39, the value of $M(i,j)$ at step $\alpha$ is obtained by,

$$M(i,j)_\alpha = \text{MAX}\ \{M(i-1,j)_{\alpha-1}, M(i,j-1)_{\alpha-1}, M(i,j)_{\alpha-1}, t(i,j)_\alpha\} \tag{8.5}$$

We prove that $M(i,j)$ obtained from the above computation is the correct maximum sum.

For simplicity, let us introduce a term *coverage*. Suppose we have examined the array portion from $a[e][f]$ to $a[i][j]$, i.e., $(e,f)|(i,j)$, and found the maximum sum $M$ whose scope is $(r_{1,}, c_1)|(r_2, c_2)$. Certainly, we refer to $M$ as the maximum sum within $(e,f)|(i,j)$. Here, let $(e,f)|(i,j)$ be the *coverage* of the maximum sum $M$.

In the context of mesh solution, after the coverage $(e,f)|(i,j)$ has been examined, we want the maximum sum $M$ to be available at $cell(i,j)$. More specifically, we want $M(i,j)$ to be the true representation of $M$ in Algorithm 9 such that it is the maximum sum within the coverage $(e,f)|(i,j)$. For example, after the mesh network completes the whole process, we will have found the value of $M(m,n)$, the maximum sum within the coverage $(1,1)|(m,n)$, the whole array.

**Lemma 8.4.2.** *The coverage of* $M(i,j)_\alpha$ *is* $(g(i,j)_\alpha, 1)|(i,j)$.

*Proof.* Suppose the coverage of $M(i,j)$ is $(e,f)|(i,j)$. It takes one step for the register values of $cell(i-1,j)$ and $cell(i,j-1)$ to arrive at $cell(i,j)$. These incoming register values were made one step earlier such as $M(i-1,j)_{\alpha-1}$ and $M(i,j-1)_{\alpha-1}$. Inductively, it is observed that the register values of $cell(i-i_1, j-j_1)$ arriving at $cell(i,j)$ at step $\alpha$ are those made at step $\alpha-(i_1+j_1)$, such as $M(i-i_1, j-j_1)_{\alpha-(i_1+j_1)}$ etc.

For $(e,f)|(i,j)$ to be the coverage of $M(i,j)_\alpha$, $cell(e,f)$ should be located $(\alpha-1)$ steps away from $cell(i,j)$ such that the register values of $cell(e,f)$, such as $M(e,f)_1$, may arrive at $cell(i,j)$ taking $(\alpha-1)$ steps.

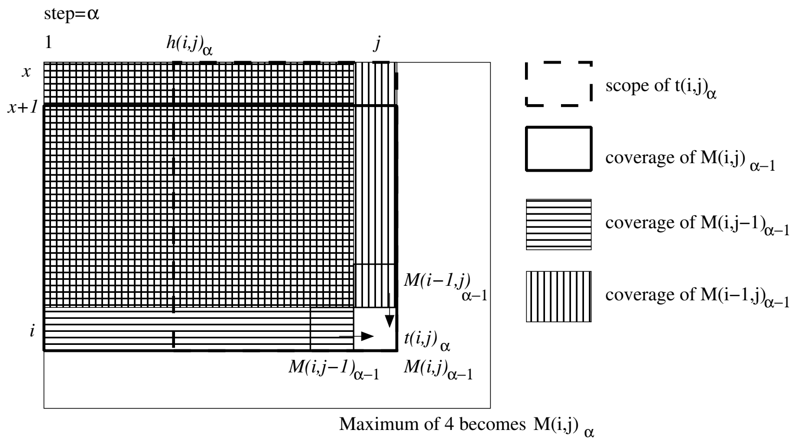


Figure 8.6: Computation of $M(i,j)_\alpha$

At step $\alpha$, the top boundary of the scope of $t(i,j)_\alpha$ and $p(i,j)_\alpha$ is $g(i,j)_\alpha$ meaning that $cell(i,j)$ has received data originated from up as far as $g(i,j)_\alpha$. Since $cell(g(i,j)_\alpha, j)$ is the farthest cell in the same column whose register values could reach $cell(i,j)$, the top boundary of the coverage of $M(i,j)_\alpha$ is $e = g(i,j)_\alpha$.

Due to $e = g(i,j)_\alpha$, the vertical distance between $cell(e,f)$ and $cell(i,j)$ is $i - g(i,j)_\alpha$. Then the horizontal distance, $(j-f)$, should be $\alpha - 1 - (i - g(i,j)_\alpha)$. From (8.1), $f = 1$. □

Now we prove $M(i,j)_\alpha$ computed by (8.5) is the correct maximum sum.

Due to Lemma 8.4.2, the coverage of $M(i-1,j)_{\alpha-1}$ is $(g(i-1,j)_{\alpha-1}, 1)|(i-1,j)$ and the coverage of $M(i,j-1)_{\alpha-1}$ is $(g(i,j-1)_{\alpha-1}, 1)|(i,j-1)$. From (8.1), we know that $g(i,j)_\alpha = g(i-1,j)_{\alpha-1} = g(i,j-1)_{\alpha-1}$. Let $g(i,j)_\alpha = x$. The scope of $t(i,j)_\alpha$ shares the same top boundary $x$ and the coverage of $M(i,j)_{\alpha-1}$ is $(x+1,1)|(i,j)$. These four areas are shown in Figure 8.6. Let us assume $M(i,j)_{\alpha-1}$, $M(i,j-1)_{\alpha-1}$ and $M(i-1,j)_{\alpha-1}$ are the maximum sum found within their own coverage. Due to Corollary 8.4.1, $t(i,j)_\alpha$ is the maximum sum ending at $a[i][j]$ with the top boundary $x$. Inductively, the largest of four must be the maximum sum within the coverage $(x,1)|(i,j)$, which proves the correctness of (8.5), and Algorithm 39.

### 8.4.5 Analysis: Total Communication Cost

Let $T$ be the total number of data flow steps required for the computation for whole array. When the whole process in the mesh network is complete, i.e., at step $T$, the network has examined all candidate subarrays as Algorithm 9 does sequentially, and the register value $M(m,n)_T$ is the maximum sum within the coverage $(1,1)|(m,n)$, the whole array $a$.

The top boundary of the coverage is $g(m,n)_T$. To find $T$ satisfying $g(m,n)_T = 1$, we apply (8.1) such that $g(m,n)_T = m + n - T = 1$. Solving this equation, $T = m + n - 1$.

We conclude that this algorithm solves the maximum subarray problem in $O(n)$ time with a mesh network of $O(n^2)$ size. The solution is found at $M(m,n)$. In the snapshot shown in Section 8.6, at step=7, $M(4,4)$ is the maximum subarray, that is $17(3,1)|(4,3)$.

## 8.5 Summary of Results

We have discussed previous parallel algorithms for the MSP and presented two new mesh algorithms for the 2D MSP. They both achieve $O(n)$ time with a mesh network of size $O(n^2)$. These mesh algorithms are designed to minimize the communication cost by integrating several parallel prefix computations, and achieve $2n - 1$ steps in total for an input array of size $n \times n$.

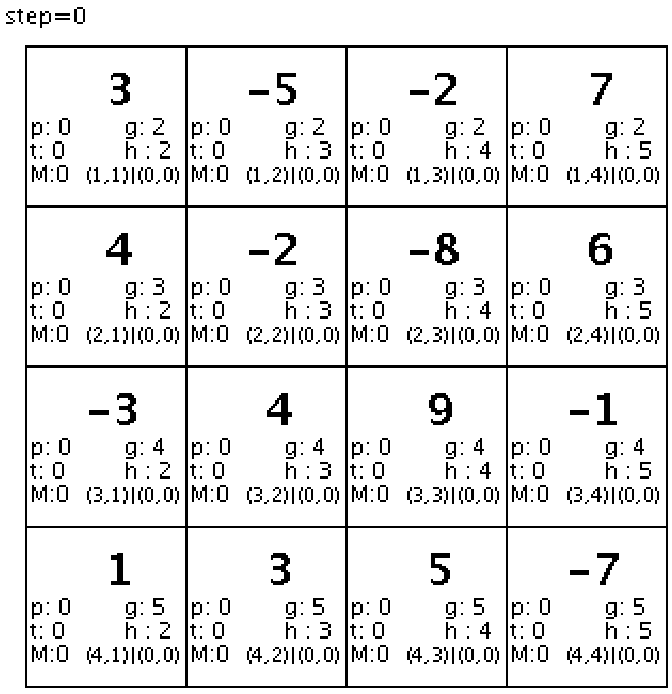
In terms of the time and hardware complexity, these two mesh algorithms are equivalent, while Algorithm 38 is conceptually easier to follow. Algorithm 39 is, however, of some historical value in a sense that it is the first parallel algorithm based on a classical work by Jay Kadane. Indeed, Algorithm 39 was designed first and its preliminary version was published in [6] prior to [7] which presented an earlier form of Algorithm 38.

These mesh algorithms are not as fast as $O(\log n)$ time parallel algorithms previously developed, but require less number of processing units and provide better practicality. Their regular interconnected structure and modular design are desirable characteristics for a hardware implementation whereas previous solutions are in the realm of pure theoretics or at best, expensive to realize. Weddell and Langford, from the department of Electrical and Computer Engineering at the University of Canterbury, implemented Algorithm 39 on a VIRTEX-II, FPGA , and experimentally demonstrated the validity of the design [101].

So far, we intentionally avoided addressing the data loading issue and assumed that the input array is loaded onto the network before the computation. The data loading issue as well as further enhancements to these mesh algorithms will be given in the next chapter.

## 8.6 Example: Trace of Algorithm 39

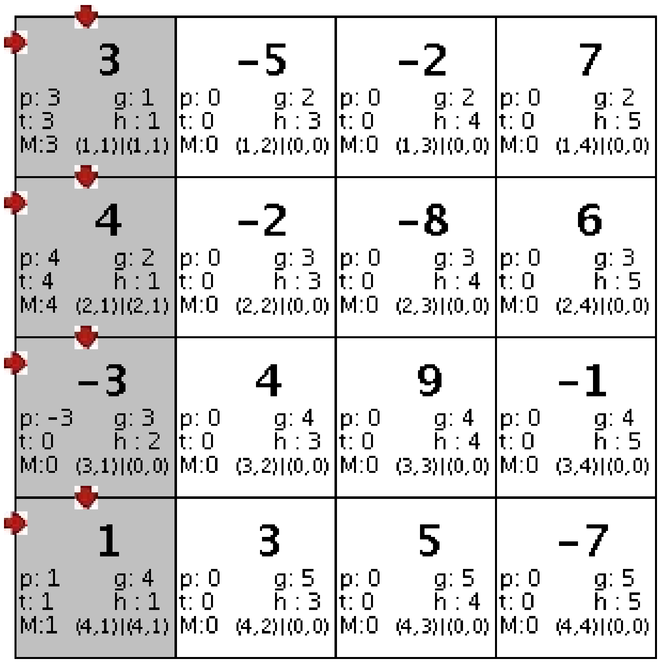
Snapshots taken from a Java simulator running Algorithm 39 for a $4 \times 4$ input array. The input array is pre-loaded onto the network.The arrow indicates the direction of the data flow. The value in larger font is $a$ register. Cells activated are shaded. The set of coordinates next to the value of $s$ register represents the location. The second set being (0,0) indicates no subarray of positive sum has been found.



step=0

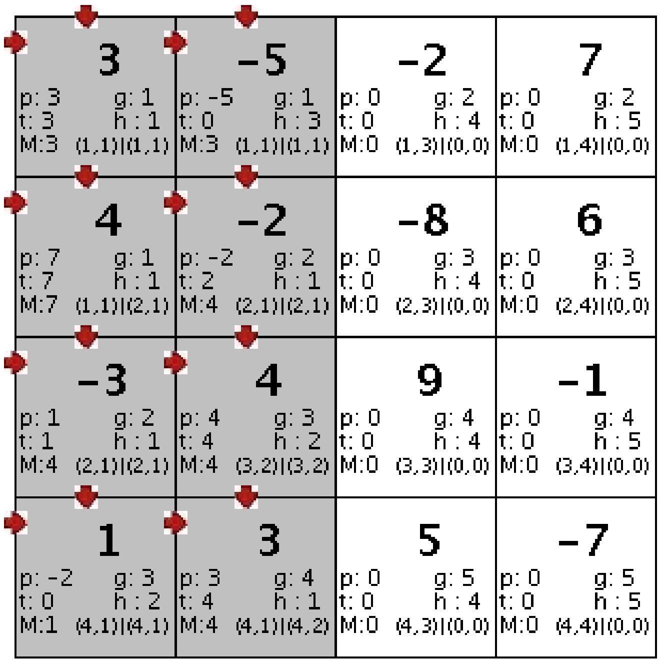
| | | | |
|---|---|---|---|
| **3** p: 0 g: 2 t: 0 h : 2 M:0 (1,1)\|(0,0) | **-5** p: 0 g: 2 t: 0 h : 3 M:0 (1,2)\|(0,0) | **-2** p: 0 g: 2 t: 0 h : 4 M:0 (1,3)\|(0,0) | **7** p: 0 g: 2 t: 0 h : 5 M:0 (1,4)\|(0,0) |
| **4** p: 0 g: 3 t: 0 h : 2 M:0 (2,1)\|(0,0) | **-2** p: 0 g: 3 t: 0 h : 3 M:0 (2,2)\|(0,0) | **-8** p: 0 g: 3 t: 0 h : 4 M:0 (2,3)\|(0,0) | **6** p: 0 g: 3 t: 0 h : 5 M:0 (2,4)\|(0,0) |
| **-3** p: 0 g: 4 t: 0 h : 2 M:0 (3,1)\|(0,0) | **4** p: 0 g: 4 t: 0 h : 3 M:0 (3,2)\|(0,0) | **9** p: 0 g: 4 t: 0 h : 4 M:0 (3,3)\|(0,0) | **-1** p: 0 g: 4 t: 0 h : 5 M:0 (3,4)\|(0,0) |
| **1** p: 0 g: 5 t: 0 h : 2 M:0 (4,1)\|(0,0) | **3** p: 0 g: 5 t: 0 h : 3 M:0 (4,2)\|(0,0) | **5** p: 0 g: 5 t: 0 h : 4 M:0 (4,3)\|(0,0) | **-7** p: 0 g: 5 t: 0 h : 5 M:0 (4,4)\|(0,0) |

step=1

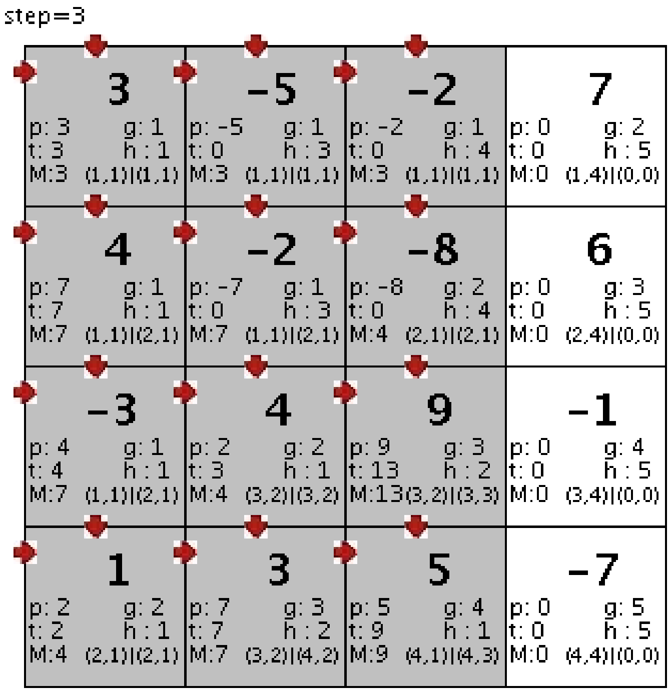


| | | | |
|---|---|---|---|
| **3**<br>p: 3 g: 1<br>t: 3 h : 1<br>M:3 (1,1)\|(1,1) | **-5**<br>p: 0 g: 2<br>t: 0 h : 3<br>M:0 (1,2)\|(0,0) | **-2**<br>p: 0 g: 2<br>t: 0 h : 4<br>M:0 (1,3)\|(0,0) | **7**<br>p: 0 g: 2<br>t: 0 h : 5<br>M:0 (1,4)\|(0,0) |
| **4**<br>p: 4 g: 2<br>t: 4 h : 1<br>M:4 (2,1)\|(2,1) | **-2**<br>p: 0 g: 3<br>t: 0 h : 3<br>M:0 (2,2)\|(0,0) | **-8**<br>p: 0 g: 3<br>t: 0 h : 4<br>M:0 (2,3)\|(0,0) | **6**<br>p: 0 g: 3<br>t: 0 h : 5<br>M:0 (2,4)\|(0,0) |
| **-3**<br>p: -3 g: 3<br>t: 0 h : 2<br>M:0 (3,1)\|(0,0) | **4**<br>p: 0 g: 4<br>t: 0 h : 3<br>M:0 (3,2)\|(0,0) | **9**<br>p: 0 g: 4<br>t: 0 h : 4<br>M:0 (3,3)\|(0,0) | **-1**<br>p: 0 g: 4<br>t: 0 h : 5<br>M:0 (3,4)\|(0,0) |
| **1**<br>p: 1 g: 4<br>t: 1 h : 1<br>M:1 (4,1)\|(4,1) | **3**<br>p: 0 g: 5<br>t: 0 h : 3<br>M:0 (4,2)\|(0,0) | **5**<br>p: 0 g: 5<br>t: 0 h : 4<br>M:0 (4,3)\|(0,0) | **-7**<br>p: 0 g: 5<br>t: 0 h : 5<br>M:0 (4,4)\|(0,0) |

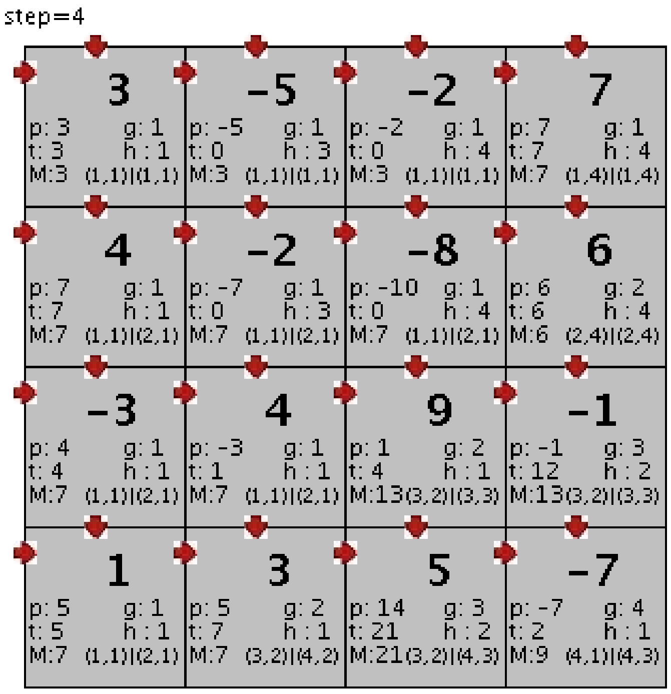
step=2



| | | | |
|---|---|---|---|
| **3**<br>p: 3 g: 1<br>t: 3 h : 1<br>M:3 (1,1)\|(1,1) | **-5**<br>p: -5 g: 1<br>t: 0 h : 3<br>M:3 (1,1)\|(1,1) | **-2**<br>p: 0 g: 2<br>t: 0 h : 4<br>M:0 (1,3)\|(0,0) | **7**<br>p: 0 g: 2<br>t: 0 h : 5<br>M:0 (1,4)\|(0,0) |
| **4**<br>p: 7 g: 1<br>t: 7 h : 1<br>M:7 (1,1)\|(2,1) | **-2**<br>p: -2 g: 2<br>t: 2 h : 1<br>M:4 (2,1)\|(2,1) | **-8**<br>p: 0 g: 3<br>t: 0 h : 4<br>M:0 (2,3)\|(0,0) | **6**<br>p: 0 g: 3<br>t: 0 h : 5<br>M:0 (2,4)\|(0,0) |
| **-3**<br>p: 1 g: 2<br>t: 1 h : 1<br>M:4 (2,1)\|(2,1) | **4**<br>p: 4 g: 3<br>t: 4 h : 2<br>M:4 (3,2)\|(3,2) | **9**<br>p: 0 g: 4<br>t: 0 h : 4<br>M:0 (3,3)\|(0,0) | **-1**<br>p: 0 g: 4<br>t: 0 h : 5<br>M:0 (3,4)\|(0,0) |
| **1**<br>p: -2 g: 3<br>t: 0 h : 2<br>M:1 (4,1)\|(4,1) | **3**<br>p: 3 g: 4<br>t: 4 h : 1<br>M:4 (4,1)\|(4,2) | **5**<br>p: 0 g: 5<br>t: 0 h : 4<br>M:0 (4,3)\|(0,0) | **-7**<br>p: 0 g: 5<br>t: 0 h : 5<br>M:0 (4,4)\|(0,0) |

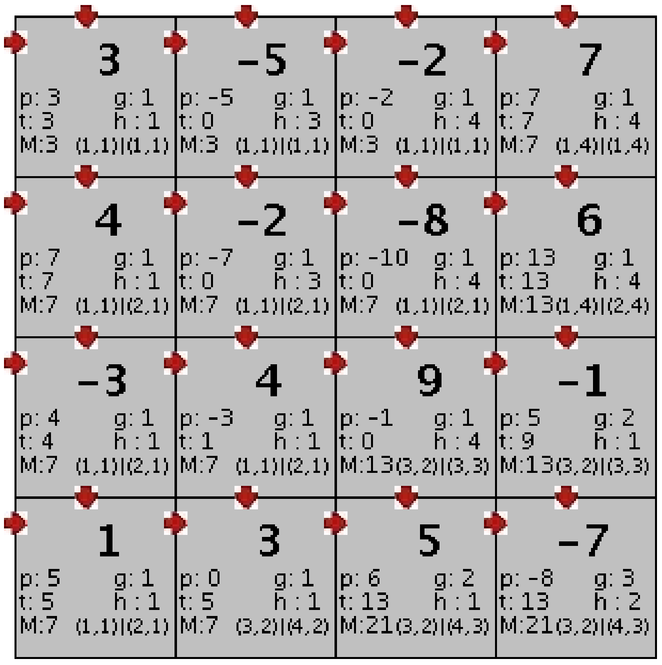


```
step=3
      3                  -5                 -2                  7
p: 3      g: 1      p: -5     g: 1      p: -2     g: 1      p: 0      g: 2
t: 3      h: 1      t: 0      h: 3      t: 0      h: 4      t: 0      h: 5
M:3 (1,1)|(1,1)     M:3 (1,1)|(1,1)     M:3 (1,1)|(1,1)     M:0 (1,4)|(0,0)

      4                  -2                 -8                  6
p: 7      g: 1      p: -7     g: 1      p: -8     g: 2      p: 0      g: 3
t: 7      h: 1      t: 0      h: 3      t: 0      h: 4      t: 0      h: 5
M:7 (1,1)|(2,1)     M:7 (1,1)|(2,1)     M:4 (2,1)|(2,1)     M:0 (2,4)|(0,0)

     -3                   4                  9                 -1
p: 4      g: 1      p: 2      g: 2      p: 9      g: 3      p: 0      g: 4
t: 4      h: 1      t: 3      h: 1      t: 13     h: 2      t: 0      h: 5
M:7 (1,1)|(2,1)     M:4 (3,2)|(3,2)     M:13(3,2)|(3,3)     M:0 (3,4)|(0,0)

      1                   3                  5                 -7
p: 2      g: 2      p: 7      g: 3      p: 5      g: 4      p: 0      g: 5
t: 2      h: 1      t: 7      h: 2      t: 9      h: 1      t: 0      h: 5
M:4 (2,1)|(2,1)     M:7 (3,2)|(4,2)     M:9 (4,1)|(4,3)     M:0 (4,4)|(0,0)
```



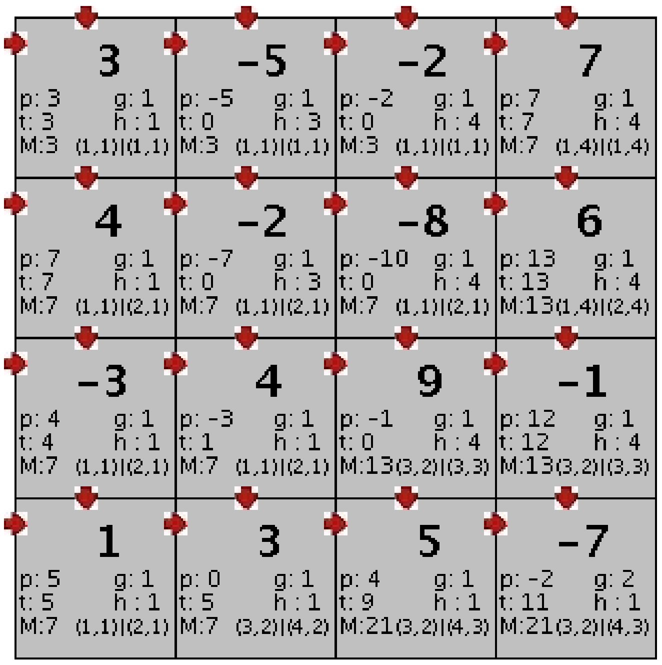
```
step=4
      3                  -5                 -2                  7
p: 3      g: 1      p: -5     g: 1      p: -2     g: 1      p: 7      g: 1
t: 3      h: 1      t: 0      h: 3      t: 0      h: 4      t: 7      h: 4
M:3 (1,1)|(1,1)     M:3 (1,1)|(1,1)     M:3 (1,1)|(1,1)     M:7 (1,4)|(1,4)

      4                  -2                 -8                  6
p: 7      g: 1      p: -7     g: 1      p: -10    g: 1      p: 6      g: 2
t: 7      h: 1      t: 0      h: 3      t: 0      h: 4      t: 6      h: 4
M:7 (1,1)|(2,1)     M:7 (1,1)|(2,1)     M:7 (1,1)|(2,1)     M:6 (2,4)|(2,4)

     -3                   4                  9                 -1
p: 4      g: 1      p: -3     g: 1      p: 1      g: 2      p: -1     g: 3
t: 4      h: 1      t: 1      h: 1      t: 4      h: 1      t: 12     h: 2
M:7 (1,1)|(2,1)     M:7 (1,1)|(2,1)     M:13(3,2)|(3,3)     M:13(3,2)|(3,3)

      1                   3                  5                 -7
p: 5      g: 1      p: 5      g: 2      p: 14     g: 3      p: -7     g: 4
t: 5      h: 1      t: 7      h: 1      t: 21     h: 2      t: 2      h: 1
M:7 (1,1)|(2,1)     M:7 (3,2)|(4,2)     M:21(3,2)|(4,3)     M:9 (4,1)|(4,3)
```

step=5



| | | | |
|---|---|---|---|
| 3<br>p: 3 g: 1<br>t: 3 h : 1<br>M:3 (1,1)\|(1,1) | -5<br>p: -5 g: 1<br>t: 0 h : 3<br>M:3 (1,1)\|(1,1) | -2<br>p: -2 g: 1<br>t: 0 h : 4<br>M:3 (1,1)\|(1,1) | 7<br>p: 7 g: 1<br>t: 7 h : 4<br>M:7 (1,4)\|(1,4) |
| 4<br>p: 7 g: 1<br>t: 7 h : 1<br>M:7 (1,1)\|(2,1) | -2<br>p: -7 g: 1<br>t: 0 h : 3<br>M:7 (1,1)\|(2,1) | -8<br>p: -10 g: 1<br>t: 0 h : 4<br>M:7 (1,1)\|(2,1) | 6<br>p: 13 g: 1<br>t: 13 h : 4<br>M:13(1,4)\|(2,4) |
| -3<br>p: 4 g: 1<br>t: 4 h : 1<br>M:7 (1,1)\|(2,1) | 4<br>p: -3 g: 1<br>t: 1 h : 1<br>M:7 (1,1)\|(2,1) | 9<br>p: -1 g: 1<br>t: 0 h : 4<br>M:13(3,2)\|(3,3) | -1<br>p: 5 g: 2<br>t: 9 h : 1<br>M:13(3,2)\|(3,3) |
| 1<br>p: 5 g: 1<br>t: 5 h : 1<br>M:7 (1,1)\|(2,1) | 3<br>p: 0 g: 1<br>t: 5 h : 1<br>M:7 (3,2)\|(4,2) | 5<br>p: 6 g: 2<br>t: 13 h : 1<br>M:21(3,2)\|(4,3) | -7<br>p: -8 g: 3<br>t: 13 h : 2<br>M:21(3,2)\|(4,3) |

step=6



| | | | |
|---|---|---|---|
| 3<br>p: 3 g: 1<br>t: 3 h : 1<br>M:3 (1,1)\|(1,1) | -5<br>p: -5 g: 1<br>t: 0 h : 3<br>M:3 (1,1)\|(1,1) | -2<br>p: -2 g: 1<br>t: 0 h : 4<br>M:3 (1,1)\|(1,1) | 7<br>p: 7 g: 1<br>t: 7 h : 4<br>M:7 (1,4)\|(1,4) |
| 4<br>p: 7 g: 1<br>t: 7 h : 1<br>M:7 (1,1)\|(2,1) | -2<br>p: -7 g: 1<br>t: 0 h : 3<br>M:7 (1,1)\|(2,1) | -8<br>p: -10 g: 1<br>t: 0 h : 4<br>M:7 (1,1)\|(2,1) | 6<br>p: 13 g: 1<br>t: 13 h : 4<br>M:13(1,4)\|(2,4) |
| -3<br>p: 4 g: 1<br>t: 4 h : 1<br>M:7 (1,1)\|(2,1) | 4<br>p: -3 g: 1<br>t: 1 h : 1<br>M:7 (1,1)\|(2,1) | 9<br>p: -1 g: 1<br>t: 0 h : 4<br>M:13(3,2)\|(3,3) | -1<br>p: 12 g: 1<br>t: 12 h : 4<br>M:13(3,2)\|(3,3) |
| 1<br>p: 5 g: 1<br>t: 5 h : 1<br>M:7 (1,1)\|(2,1) | 3<br>p: 0 g: 1<br>t: 5 h : 1<br>M:7 (3,2)\|(4,2) | 5<br>p: 4 g: 1<br>t: 9 h : 1<br>M:21(3,2)\|(4,3) | -7<br>p: -2 g: 2<br>t: 11 h : 1<br>M:21(3,2)\|(4,3) |

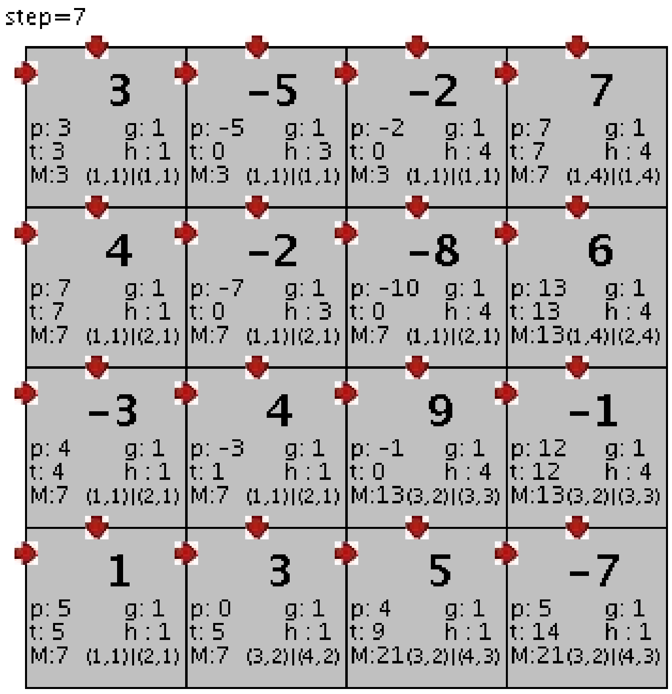


Figure 8.7: Example: Trace of Algorithm 39

At step 7, $M(4,4)$ and its associated coordinates represent the maximum subarray $21(3,2)|(4,3)$.

# Chapter 9

# Enhancements to the Mesh MSP Algorithms

## 9.1 Introduction

In this chapter, we describe possible extensions and enhancements to the mesh MSP algorithms. The correctness of the mesh MSP algorithms were already proved. However, there are two details that need to be addressed to utilize these algorithms in practice. The first is the data loading issue, which we discuss in Section 9.2. Secondly, in Section 9.3, we show how to run these algorithms on a coarse-grained mesh. This is particularly important to be able to process the input array larger than the available mesh network. We then present a constant factor improved version of Algorithm 38 by employing bi-directional horizontal data flow in Section 9.4. The remaining part of this chapter will give a description of modified mesh MSP algorithms to compute $K$ maximum subarrays.

## 9.2 Data Loading

So far it has been assumed that the each cell in the network is initialized and an input array element $a[i][j]$ $(1 \leq i \leq m,\ 1 \leq j \leq n)$ is loaded onto the corresponding cell of the network, $cell[i][j]$, before the actual computation at no cost. Consequently, the aforementioned time analysis did not count data loading cost. In practice, the time for data loading should be considered as another factor that affects the overall performance.

A problem involved in data loading is due to the structure of the mesh network. Since the network consists of cells which have only four connections with neighboring cells, direct loading from the input array $a[i][j]$ into $cell[i][j]$ is prohibited. Only cells on the boundary may accept input data from external sources.

We suggest a separate *parallel data loader* to resolve this issue. We assume a parallel data loader similar to linear image sensors which are commonly found in scanning applications such as paper copiers, fax machines, and film scanners [75]. This model has $m$ cells itself, and scans the input array a column at a time and injects these values simultaneously through the connections from the right boundary of the network in one step, so that the network can gradually build up a full array.

The first column of the input array is loaded onto the $n$-th column of the network. When the next column of the input array is loaded, the current data in the $n$-th column of the network shift left to evacuate the space for the new one. This way, all input array elements can be loaded in $n$ steps. If we perform Algorithm 38 or Algorithm 39 after the data loading is complete, it needs $T = m + 2n - 1$ steps for the whole process.

Now we describe how we eliminate these extra $n$ steps.

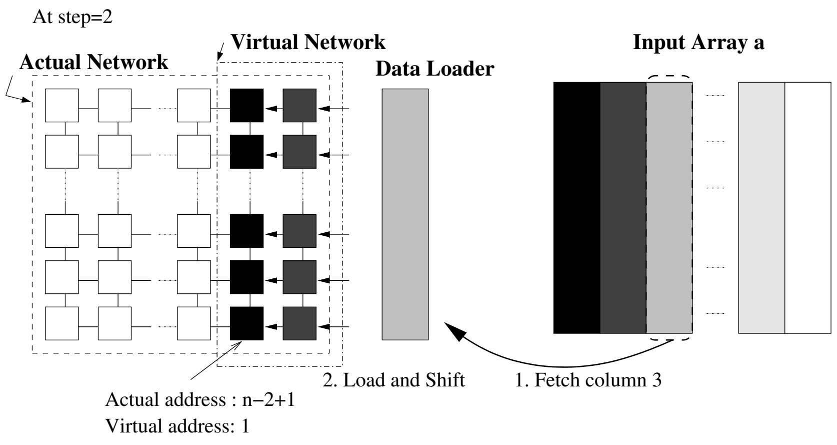
In Algorithms 38 and 39, the column $j$ receives the control signal at step $j$, and only active cells perform data flows. Non-active cells sit idle until they become active. We observe the following.

**Lemma 9.2.1.** *The $j$-th column of the input array $a[1..m][j]$ is not necessary to compute Algorithm 38 and 39 until step $j$.*

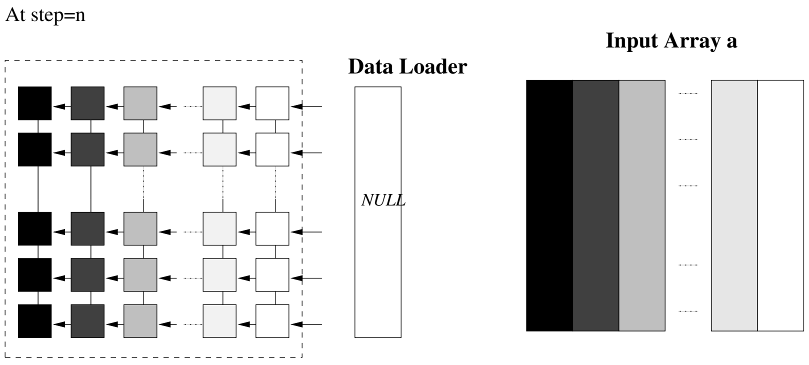
With the parallel data loader, the first $j$ columns of the input array are loaded onto the network by step $j$. All required data for executing the algorithm for this step are already available, so we do not need to wait for the data loading to complete. We instead, execute the algorithm *on-the-fly* and optimize overall throughput of the network.

The parallel data loader fetches the $\alpha$-th column of the input array at step $\alpha - 1$, such that this column can enter the network at the next step. Then at step $\alpha$ ($1 \leq \alpha \leq n$), as shown in Figure 9.1(a), the columns $1..\alpha$ of the input array are accommodated by the columns $(n-\alpha+1)..n$ of the network. This part of the network may be viewed as a *virtual network.* Note that, for $j \leq \alpha$, the $j$-th column of the input array is found in the $j$-th column of the virtual network.

The $(n-\alpha+1)$-th column of the “actual” network is then regarded as the



(a) At $step = 2(\leq n)$: Phase in



(b) At $step = \alpha = n$: Total Eclipse

Figure 9.1: Parallel Data Loading

first column of the virtual network. We let this virtual network emulate step $\alpha$ of the mesh algorithm . In general, the $j$-th column ($j \leq \alpha$) of the virtual network is actually located in the $(n - \alpha + j)$-th column of the network at step $\alpha$.

By step $n$, all $n$ columns of the input array have been loaded and the network is totally eclipsed by the virtual one as shown in Figure 9.1(b). Note that from this point on, the parallel data loader simply prepares null data and feeds them to the network at the subsequent step. When a column of null data is loaded onto the network, the data occupying the first column of the network are phased out and the data in the rest of columns shift left.

One column of the input array stays in the network for $n$ steps before it is eventually phased out. It is step $2n$ when the $n$-th column will leave the network, but we never reach this point as $2n > m + n - 1 (= T)$ due to the assumption $m \leq n$.

Phasing out has no ill-effect on the correctness of the algorithm. The $j$-th column of the input array enters the network at step $j$ and shifts to the left each step. It eventually phases out at step $n+j$, meaning that it stays within the network for $n$ steps. All horizontal data flow needed for this column is done within this time frame. Similarly, all vertical data flow for this column takes $m - 1$ steps, which can be done within these $n$ steps since $m \leq n$ is assumed.

It has been discussed that we can achieve $T = m + n - 1$ steps with data loading inclusive.

We now design a modified network that can accommodate this idea. Unlike the previous design, $cell(i, j)$ will be no longer hard-wired to one specific array element, $a[i][j]$. It will only store $a[i][j']$, one of $a[i][1..n]$ at one step and pass it to its neighbor at the next step. Thus each cell is provided with an extra register $j'$ to learn which element it is currently processing. Certainly, the horizontal data flow will be in the opposite direction, towards left, and it needs to deliver $a$ register value as well as this new $j'$ register value.

The downwards data flow in the original design needs to change the direction. To illustrate the situation, let us consider a classical example in physics. Imagine a train that is moving along at a constant speed from right to left. A ball is suspended by a wire attached to the ceiling. When we cut the wire, it falls straight down to the floor, since the initial horizontal velocity of the ball relative to the train is zero.In theory, the ball moves at the same horizontal speed as the train as it falls, but so do the ceiling and the floor. The downwards data flow from the $i$-th row to the $(i + 1)$-th row can be regarded as the falling ball from the ceiling to the floor. In both rows, we have data moving to the left at constant speed, one column at a step. These two rows should see this downwards data flow as the straight-down movement. To do so, we provide the connections between two rows in the *down-left* direction as shown in Figure 9.2.

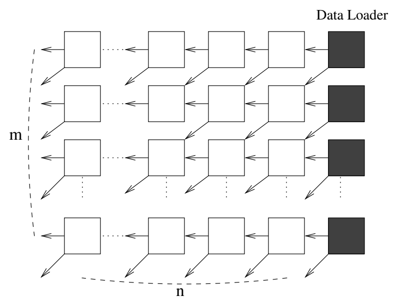


Figure 9.2: Revised mesh network with run-time data loading support

### Modifying Algorithm 39

In the following description, let us discuss how Algorithm 39 can be accommodated by the revised mesh network.

When a cell receives data flow from its right and upper-right neighbors, it receives the register values of $a$, $j'$ and $M$ from the right, and $p$, $g$ and $M$ from the upper-right. Note that $t$ register value no longer needs to be delivered in either direction. We give a detailed description of the the parallel data loader and revised update policy for each cell in Algorithm 40 and Algorithm 41 respectively. Both are executed in parallel for all $i$ or for all $i$ and $j$ respectively.

**Algorithm 40** Loading input array into the mesh

```
Load(col):: loader[i] do begin
//col is the column to be loaded. If $\alpha < n$, $col = \alpha + 1$. Otherwise $col = 0$
and we load a set of null data
 1: $j' \leftarrow col$
 2: $a \leftarrow a[i][j']$
 3: $p, t, M \leftarrow 0$
 4: $g \leftarrow i+1$
 5: $h \leftarrow j'+1$
end
```

One may notice a similarity between Algorithm 40 and *Initialize* routine in Algorithm 39. In fact, the data loader is designed to initialize registers on behalf of cells in the network. We explain this shortly. The parallel

**Algorithm 41** Revised routine for $cell(i, j)$ for run-time data loading

```
Update:: cell(i, j) do begin
1:  /* Data flow from right */
2:  $j'(i,j) \leftarrow j'_{in}(i,j{+}1)$
3:  $a(i,j) \leftarrow a_{in}(i,j{+}1)$
4:  /* Data flow from upper-right */
5:  $p(i,j) \leftarrow p_{in}(i\text{-}1,j{+}1) + a(i,j)$
6:  $g(i,j) \leftarrow g_{in}(i\text{-}1,j{+}1)$
7:  /* Check if null data received */
8:  if $j'(i,j) = 0$ //received null data then
9:      /* Skip internal computation. Results in $t(i,j) = M(i,j) = 0$ and
        $h(i,j) = 1$ */
10:     $h(i,j) \leftarrow h_{in}(i,j{+}1)$, $t(i,j) \leftarrow t_{in}(i,j{+}1)$, $M(i,j) \leftarrow M_{in}(i,j{+}1)$
11: else
12:     /* Internal Computation */
13:     $t(i,j) \leftarrow t(i,j) + p(i,j)$
14:     if $t(i,j) \leq 0$ then $t(i,j) \leftarrow 0$, $h(i,j) \leftarrow j'(i,j){+}1$
15:     /* Update $M(i,j)$ */
16:     $M(i,j) \leftarrow \text{MAX}\{M_{in}(i\text{-}1,j{+}1), M(i,j), M_{in}(i,j{+}1), t(i,j)\}$
17: end if
end
```

loader consists of $m$ components and each handles the array element in the corresponding row. Each component, $loader[i]$ $(0 \leq i \leq m)$ has an alias, $cell(i, n{+}1)$, and has the same set of registers and compatible data flow interface as ordinary cells in the network. The $a$ register of $loader[i]$, for example, is seen as $a(i, n{+}1)$ by $cell(i, n)$ performing Algorithm 41.

Since the data flow from a loading component to the connected cell in the network also takes 1 step, each component of the data loader, $loader[i]$, fetches the first column of the array at step 0 and initializes its own registers. These register values are loaded on to the $n$-th column of the network at step 1. Note here that $t(i, n)_1$ is the sum of $t(i, n)_0$ and $p(i, n)_1$. While $t(i, n)_0$ was not explicitly defined, for now, we assume its value is 0. We later show that this assumption is valid.

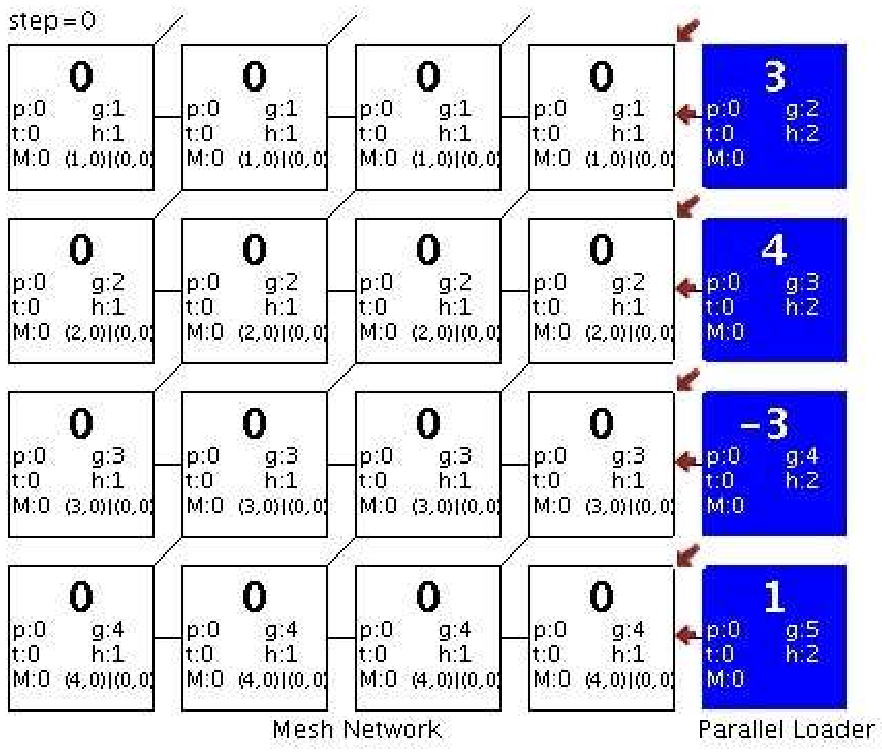
In general, at step=$\alpha$, the fetched array element is $a[i][\alpha + 1]$ if $\alpha < n$. When there is no remaining data to process, the loader prepares a set of null data to signal the end of computation. This null data travels the network just like ordinary set of register values. When a cell receives this null data, as shown in line 8 of Algorithm 41, it regards this as a terminating signal and skips the internal computation. Instead, it passively copies the received set of register values, which effectively initializes the registers. Then this cell relays the null data to the left neighbor, effectively causing gradual initialization of the entire network. This explains the lack of an explicit initialization routine in Algorithm 41.

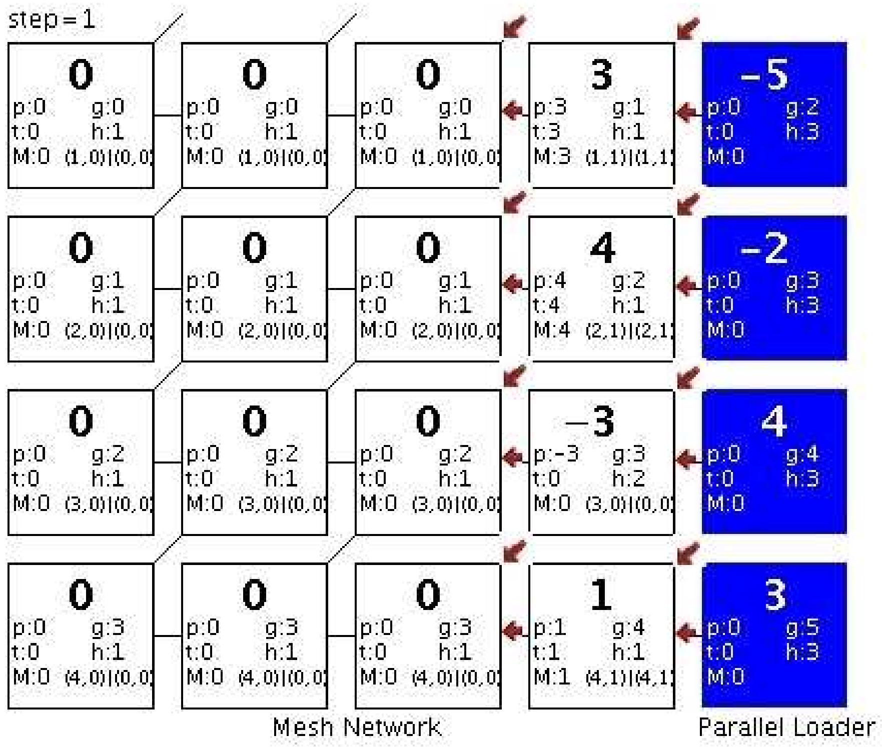
When a new input array needs to be processed right after the computation for one input is complete, the data loader prepares the null data only once, and starts processing the new input in the next step. With this one interleaving step, each column of the network will have been initialized *just-in-time*, one step before the arrival of new data, meaning that no more than one step delay is required before processing a new input. Once the network is ready to accept a new input, it is therefore $t(i,n)_0 = 0$. Effectively, such a *pipelining* operation maximizes the throughput of the network.

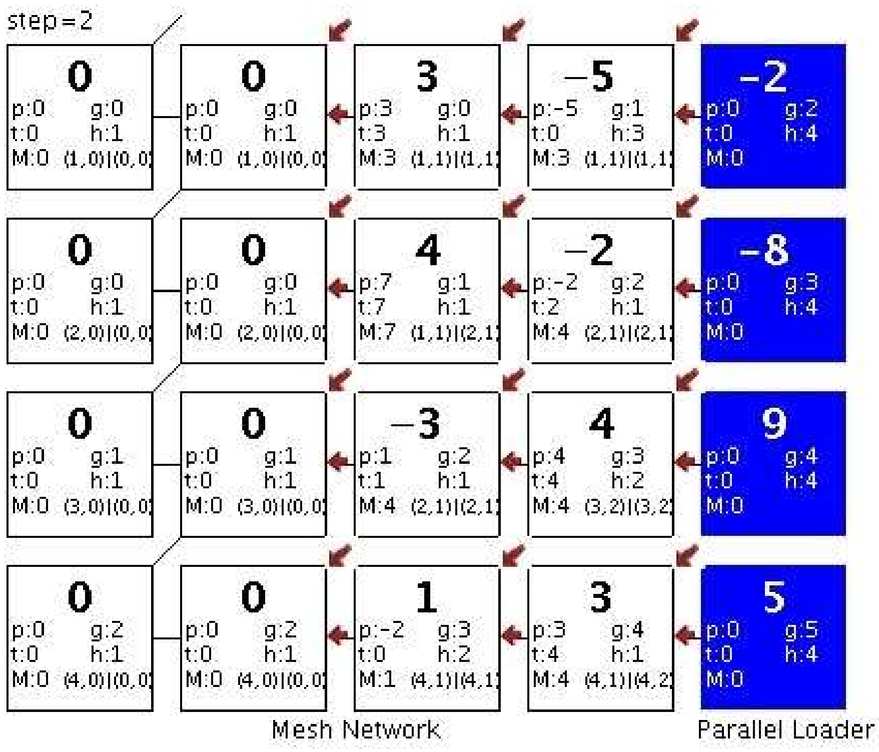
Note that there is a dummy 0-th row in both network and the loader. This is to provide every cell in the network with homogeneous interconnection settings. Cells in this row, $cell(0, 1..n)$, do not perform any update operation, but supply register values to the neighbor cells below. Specifically, $p(0, 1..n) = M(0, 1..n) = 0$ and $g(0, 1..n) = 1$.

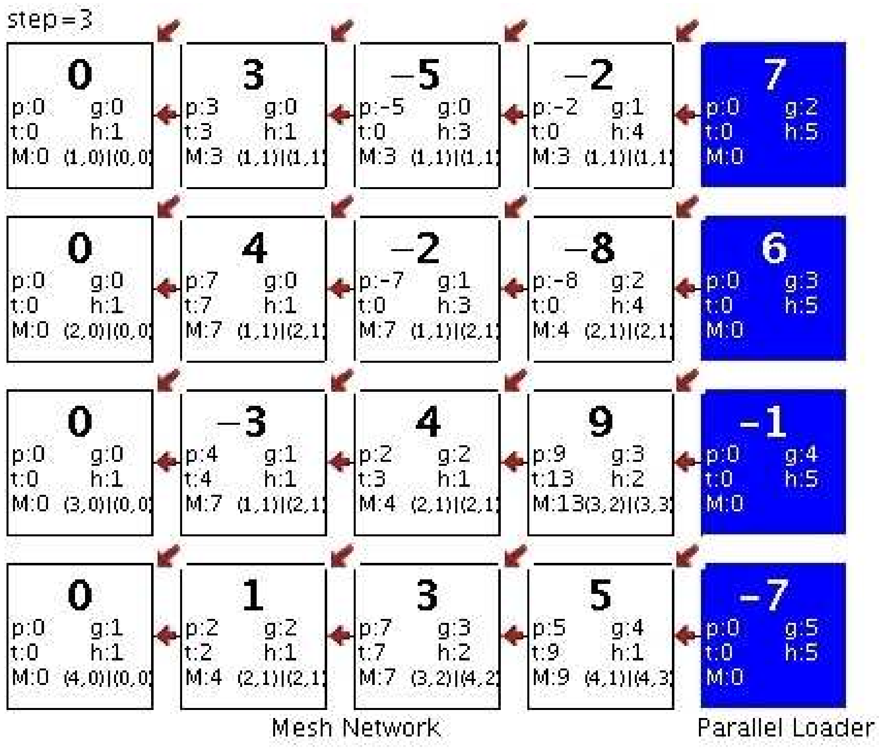
With this modification, the $n$-th column of the input array enters the mesh network at step $n$ and is located at the $n - m + 1$-th column of the network at step $m + n - 1$. The maximum sum and its location can then be retrieved from $cell(m, m + n - 1)$.

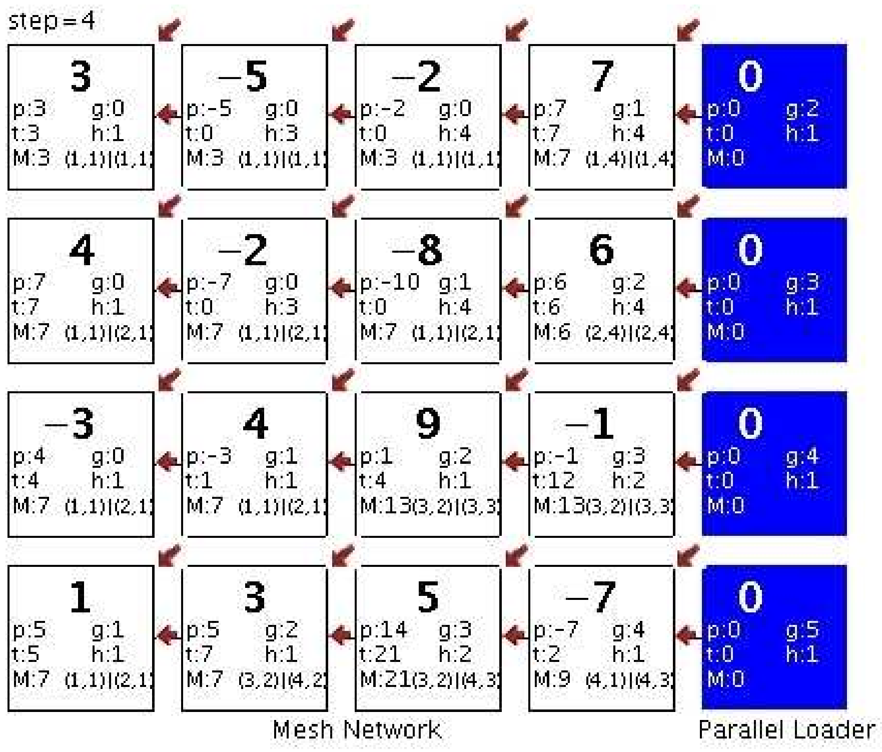
Figure 9.3 show the trace of running Algorithm 41 in conjunction with Algorithm 40. The same input in Figure 8.7 is used. $M(4, 1)$ and its associated coordinates at step 7 represent the maximum subarray $21(3, 2)|(4, 3)$. We assume that the network has processed a stream of null data previously, and every cell is initialized. In practice, only the $n(= 4)$-th column needs to be initialized, which can be done by loading a set of null data one step before the process. For the same reason, the network is ready to accept a new input at step 5.

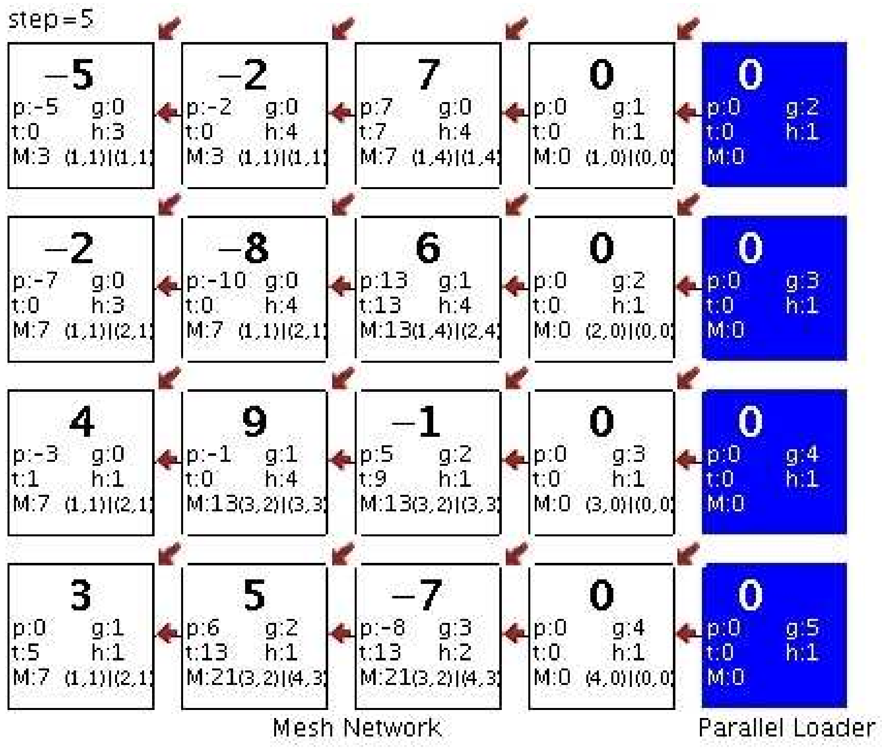

step=0
3
4
-3
1
Mesh Network
Parallel Loader


step=1
3
4
-3
1
-5
-2
4
3
Mesh Network
Parallel Loader


step=2
Mesh Network
Parallel Loader


step=3
Mesh Network
Parallel Loader


step=4
3 p:3 g:0 t:3 h:1 M:3 (1,1)|(1,1)
−5 p:-5 g:0 t:0 h:3 M:3 (1,1)|(1,1)
−2 p:-2 g:0 t:0 h:4 M:3 (1,1)|(1,1)
7 p:7 g:1 t:7 h:4 M:7 (1,4)|(1,4)
0 p:0 g:2 t:0 h:1 M:0
4 p:7 g:0 t:7 h:1 M:7 (1,1)|(2,1)
−2 p:-7 g:0 t:0 h:3 M:7 (1,1)|(2,1)
−8 p:-10 g:1 t:0 h:4 M:7 (1,1)|(2,1)
6 p:6 g:2 t:6 h:4 M:6 (2,4)|(2,4)
0 p:0 g:3 t:0 h:1 M:0
−3 p:4 g:0 t:4 h:1 M:7 (1,1)|(2,1)
4 p:-3 g:1 t:1 h:1 M:7 (1,1)|(2,1)
9 p:1 g:2 t:4 h:1 M:13(3,2)|(3,3)
−1 p:-1 g:3 t:12 h:2 M:13(3,2)|(3,3)
0 p:0 g:4 t:0 h:1 M:0
1 p:5 g:1 t:5 h:1 M:7 (1,1)|(2,1)
3 p:5 g:2 t:7 h:1 M:7 (3,2)|(4,2)
5 p:14 g:3 t:21 h:2 M:21(3,2)|(4,3)
−7 p:-7 g:4 t:2 h:1 M:9 (4,1)|(4,3)
0 p:0 g:5 t:0 h:1 M:0
Mesh Network
Parallel Loader


step=5
−5 p:-5 g:0 t:0 h:3 M:3 (1,1)|(1,1)
−2 p:-2 g:0 t:0 h:4 M:3 (1,1)|(1,1)
7 p:7 g:0 t:7 h:4 M:7 (1,4)|(1,4)
0 p:0 g:1 t:0 h:1 M:0 (1,0)|(0,0)
0 p:0 g:2 t:0 h:1 M:0
−2 p:-7 g:0 t:0 h:3 M:7 (1,1)|(2,1)
−8 p:-10 g:0 t:0 h:4 M:7 (1,1)|(2,1)
6 p:13 g:1 t:13 h:4 M:13(1,4)|(2,4)
0 p:0 g:2 t:0 h:1 M:0 (2,0)|(0,0)
0 p:0 g:3 t:0 h:1 M:0
4 p:-3 g:0 t:1 h:1 M:7 (1,1)|(2,1)
9 p:-1 g:1 t:0 h:4 M:13(3,2)|(3,3)
−1 p:5 g:2 t:9 h:1 M:13(3,2)|(3,3)
0 p:0 g:3 t:0 h:1 M:0 (3,0)|(0,0)
0 p:0 g:4 t:0 h:1 M:0
3 p:0 g:1 t:5 h:1 M:7 (1,1)|(2,1)
5 p:6 g:2 t:13 h:1 M:21(3,2)|(4,3)
−7 p:-8 g:3 t:13 h:2 M:21(3,2)|(4,3)
0 p:0 g:4 t:0 h:1 M:0 (4,0)|(0,0)
0 p:0 g:5 t:0 h:1 M:0
Mesh Network
Parallel Loader

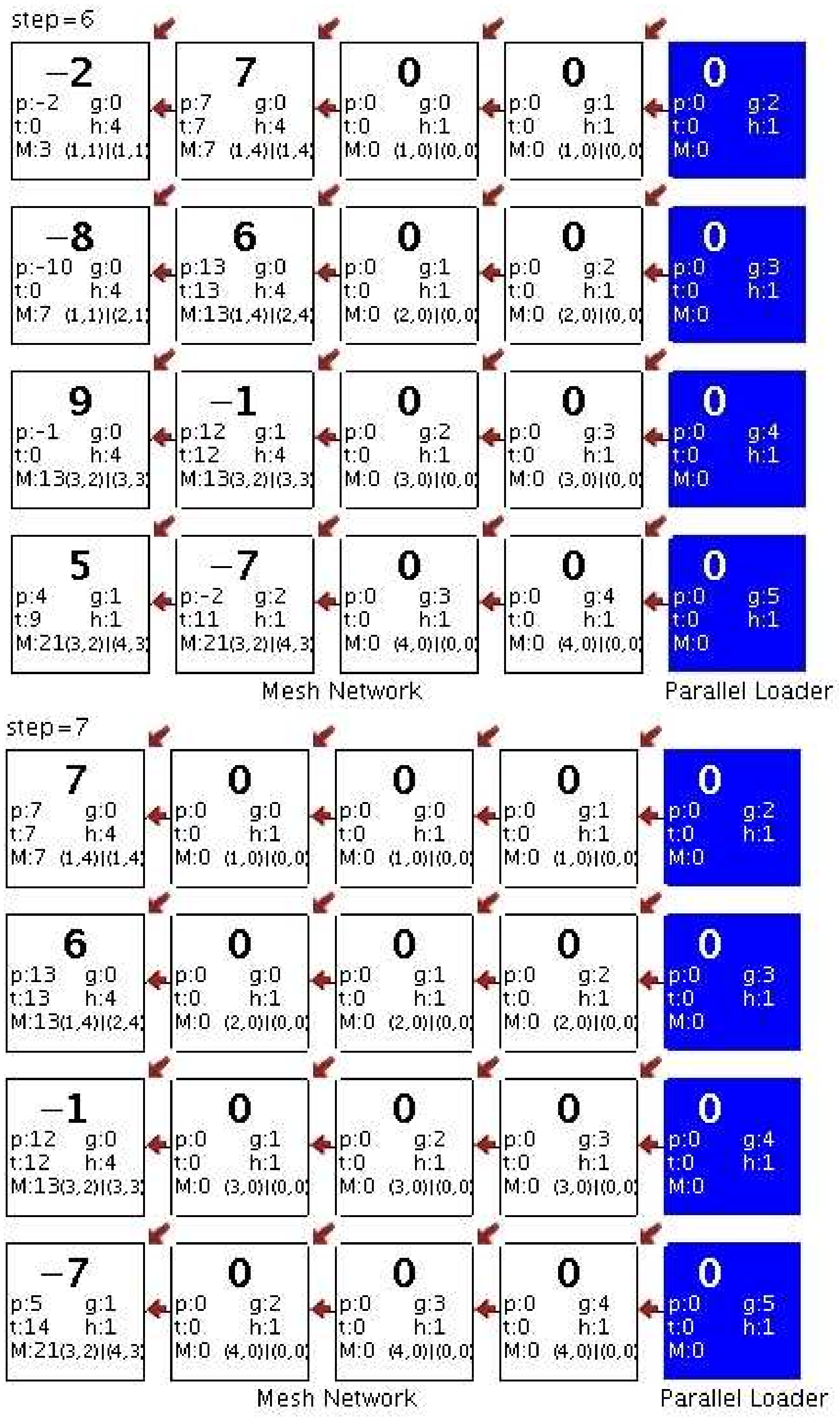


Figure 9.3: Example: Trace of Algorithm 41

We conclude this revised algorithm running on the modified network correctly computes the maximum subarray with no extra time for data loading. A cell in this revised network strictly communicates only with its neighbors following the structural definition of the mesh model, accomplishing completely asynchronous operation throughout. Similar modification to Algorithm 38 can be made, whose details are omitted.

## 9.3 Coarse-grained Mesh: Handling Large Array

It has been shown that the maximum subarray in a two-dimensional array of size $m \times n$ is computed by a mesh network consisting of $m \times n$ cells, i.e. a *fine-grained* mesh. This size is far below the requirement of $O(n^3/\log n)$ size by previous parallel solutions [103, 77, 79], and each processing unit used in this solution is simple enough so that modern technology can include millions of them in a single VLSI chip.

For any given array, however, there often arise situations where the size of the network available is not as large as the size of the problem. When the input array is larger than the network capacity, appropriate "segmenting" is necessary to fit the input array into the network [69], and we run the mesh algorithm on the *coarse-grained* mesh.

We divide the array of size $(m, n)$ into arrays of size $(r_1, r_2)$ as shown in Figure 9.4. There are $(m/r_1) \times (n/r_2)$ such segments. Each segment may itself be a smaller mesh network of size $(r_1, r_2)$ that processes the mesh algorithm (Algorithm 39) or a single processing unit that emulates it sequentially. The former option may be useful when the hardware implementation of this solution is available in quantity. Such a hardware version (e.g. a VLSI chip) will have a fixed size, so many of them can be interconnected to form a larger mesh network to fit the problem size. This is due to the homogeneous interconnection pattern of the mesh network offering good modularity and regularity. The second option may be practiced in a distributed network where each unit is programmable (e.g. a PC) to handle variable size of the segment.

In this arrangement on each parallel step, some of the data flows are taken place inside the segment while others are made between segments. As these

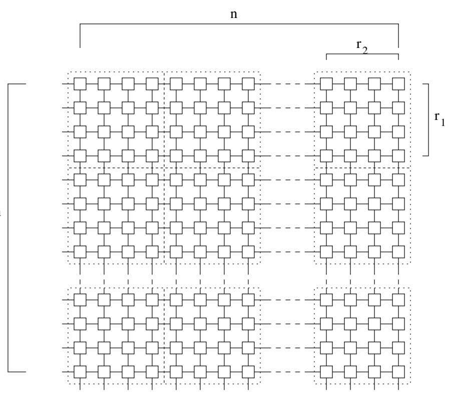


Figure 9.4: Segmenting into $r_1 \times r_2$ arrays

two data flows may have different speed, let us separate the time spent inside the segment ($T_1$) and the time spent between segments ($T_2$) respectively for one step. The time for one step is dependent on the slower speed of the two, i.e., $O(max(T_1, T_2))$ or simply $O(T_1 + T_2)$. For simplicity, let $m = n$ and $r_1 = r_2 = r$ for further analysis.

When a segment itself is a mesh network of size $(r, r)$, the interconnected segments behaves as if it was a mesh network of size $(n, n)$. We simply follow the algorithm and the total time after $2n - 1$ steps is $O\left((T_1 + T_2)n\right) = O(n)$ since $(T_1 + T_2)$ is a constant coefficient.

When a segment is emulated by a single processing unit, each of $p = n^2/r^2$ $(1 \leq p \leq n^2)$ processors spends $T_1 = O(r^2)$ time to emulate one parallel step. The processing unit assigned to the segment controls the data flow to the neighboring segments, making $T_2 = O\left(r\tau\right)$ time where $\tau$ is the time to deliver a group of register values of one cell $(r(i,j), t(i,j)$, etc.) to the neighboring segment.

Total time after $2n - 1$ steps is $O\left((T_1 + T_2)n\right) = O\left((r^2 + r\tau)n\right)$. When $\tau < r$, $T_2$ is absorbed into $T_1$ making the total time $O(r^2 n) = O(n^3/p)$. This is $O(r^2)$ times slower than the full parallel execution whose time is $O(n)$, but still gains $O(p)$ times of speedup compared with $O(n^3)$ time by a single processor performing a sequential algorithm. We conclude that our mesh algorithms are *scalable* in a sense that they can handle an input array larger than the capacity of the hardware [69].

## 9.4 Improvement to Algorithm 38

The mesh architecture provides limited inter-processor communication, favoring an iterative sequential algorithm to be used as a basis. A recursive algorithm, for example, Algorithm 5 would be difficult to parallelize with a mesh architecture.

However, the essence of this recursive algorithm still carries implication.

$$M = \mathrm{MAX}\,\{M_{left}, M_{right}, M_{center}\}$$

Suppose we divide the mesh into two halves, and let the left half compute $M_{left}$, the maximum subarray in the left half of the input array, and the right half compute $M_{right}$ simultaneously. Obviously, $M_{left}$ and $M_{right}$ are both computed in $m + n/2 - 1$ steps. Can we also compute $M_{center}$ in the mean time? If so, without spending extra time, can these three values meet at a specific processor, such that the maximum of three can be selected?

Let us investigate the latter issue first. When $M_{left}$ and $M_{right}$ are computed in a normal way, they will be located at the bottom-right corner of left half, and right half respectively. These solutions are $n/2$ positions apart, and there is no direct communication line. These two values are local to the cell they are located, hence they cannot be compared unless extra steps are spent to put them together.

To resolve this, we consider the right half of the mesh is the mirror version of the left, such that the horizontal data flow is made from right to left. This way, after $m+n/2-1$ steps, $M_{right}$ will be available at the bottom-left corner of the right half of the mesh, only one position away from $M_{left}$, where one step communication is possible. Suppose we have extra column between two halves of the mesh, *center*. Let the $i$-th processor in this column be $center(i)$.

Without loss of generality, we assume that $n$ is an even number. Otherwise, we can add an extra column with all $-\infty$ to either side of the input array.

As shown in Figure 9.5, $M_{left}$ and $M_{right}$ are available at $cell(m, n/2)$ and $cell(m, n/2+1)$ respectively at step $m+n/2-1$, and $center(m)$ receives them in the next step.

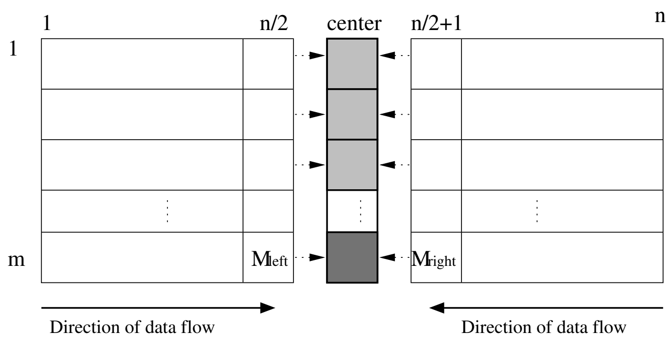


Figure 9.5: Bidirectional horizontal data flow. $M_{left}$ and $M_{right}$ are computed in parallel and meet at $center(m)$ at step$=m + n/2$.

Provided the bidirectional horizontal data flow, we describe how $M_{center}$ is computed. The prefix sum-based algorithm, Algorithm 38, can be modified with minimal modification.

$M_{center}$ in two dimensions is defined as

$$M_{center} = \underset{\substack{1 \le g \le i \le m \\ 1 \le h \le n/2 \le j \le n}}{\text{MAX}} \left\{ \sum_{p=g}^{i} \sum_{q=h}^{j} a[p][q] \right\} \tag{9.1}$$

For fixed rows $g$ and $i$, let $MC_{g,i}$ be defined as,

$$MC_{g,i} = \underset{1 \le h \le n/2 \le j \le n}{\text{MAX}} \left\{ \sum_{p=g}^{i} \sum_{q=h}^{j} a[p][q] \right\}$$

(9.1) above is then,

$$M_{center} = \underset{1 \le g \le i \le m}{\text{MAX}} \{MC_{g,i}\} \tag{9.2}$$

Now we describe how to compute $MC_{g,i}$. We obtain strip prefix sums $sum_{g,i}[1..n/2]$ from a strip $a_{g,i}$, that is $a[g..i][1..n]$. Let us focus on the left half first. Let the left part of $MC_{g,i}$ be denoted by $MC^{left}_{g,i}$, and likewise $MC^{right}_{g,i}$ for the right part. Obviously, $MC^{left}_{g,i}$ is meant to be,

$$MC^{left}_{g,i} = sum_{g,i}[n/2] - \text{MIN}\{sum_{g,i}[0..n/2-1]\}$$

Note that $sum_{g,i}[0] = 0$.

Now, let us examine Algorithm 38 processing the left half. Due to Lemma 8.3.3, we know that $cand(i, n/2)_\alpha = MC^{left}_{g,i}$, where $\alpha = n/2 + i - g$.

The right half of the mesh has the horizontal data flow running from right to left, and each register $s(i, j)$ effectively maintains suffix sums $ssum_{g,i}[n/2+1..n]$. Here, we assume that $ssum_{g,i}[n + 1] = 0$.

Similarly,

$$MC^{right}_{g,i} = ssum_{g,i}[n/2 + 1] - \text{MIN}\,\{ssum_{g,i}[n/2 + 2..n + 1]\}$$

Again, $cand(i, n/2 + 1)_\alpha = MC^{right}_{g,i}$, where $\alpha = n/2 + i - g$.

Now both $MC^{left}_{g,i}(=cand(i, n/2))$ and $MC^{right}_{g,i}(=cand(i, n/2 + 1))$ are available at step $n/2 + i - g$.

In the next step, at step $n/2+i-g+1$, $center(i)$ collects and adds them to get $MC_{g,i}$.

The following relation holds.

$$\begin{aligned} \text{At step } \alpha,\ MC_{g,i} &= cand(i, n/2)_{\alpha-1} + cand(i, n/2 + 1)_{\alpha-1} \qquad (9.3) \\ &,\quad \text{where } \alpha \geq n/2 + 1,\ g = \text{MAX}\,\{1, n/2 + i + 1 - \alpha\} \end{aligned}$$

Due to Lemma 8.3.3, it is step $n/2 + 1$ when $center(i)$ becomes active and computes $MC_{i,i}$. Each step, the strip currently being processed extends the top boundary by one row. For example, $center(i)$ computes $MC_{i-1,i}$ at step $n/2 + 2$, and $MC_{i-2,i}$ one step later etc. It is step $n/2 + i$ when $MC_{1,i}$ is computed at $center(i)$.

We prepare a register $M$ in $center(i)$, which is denoted by $M(i)$, and initialized to 0. We maintain $M(i)$ by the update operation given as (9.4).

$$M(i) \leftarrow \text{MAX}\,\{M_{in}(i - 1), M(i), cand_{in}(i, n/2) + cand_{in}(i, n/2 + 1)\} \quad (9.4)$$

Now we examine the invariant of $M(i)$ at step $\alpha$. From (9.4), we have,

$$M(i)_\alpha = \text{MAX}\,\{M(i-1)_{\alpha-1}, M(i)_{\alpha-1}, cand(i, n/2)_{\alpha-1} + cand(i, n/2+1)_{\alpha-1}\}$$

At step $n/2+1$, $M(i)$ is computed first time, meaning that $M(i-1)_{n/2} =$

$M(i)_{n/2} = 0$. Due to (9.3), we have $M(i)_{n/2+1} = MC_{i,i}$, and $MC(i)_{n/2+2} = \text{MAX}\,\{MC_{i-1,i-1},\, MC_{i,i},\, MC_{i-1,i}\}$ etc.

Inductively, this is reduced to

$$M(i)_\alpha = \underset{g \le p \le q \le i}{\text{MAX}} \{MC_{p,q}\}, \text{ where } g = \text{MAX}\,\{1, n/2 + i + 1 - \alpha\} \tag{9.5}$$

This means that,

$$M(m)_{n/2+m} = \underset{1 \le p \le q \le m}{\text{MAX}} \{MC_{p,q}\}$$

, and due to (9.2), it further means that,

$$M(m)_{n/2+m} = M_{center}$$

Running Algorithm 38 on each half of the input in parallel, having the horizontal data flow in the opposite direction in the right half, we have $M_{left}$ and $M_{right}$ available as $M(m, n/2)$ and $M(m, n/2 + 1)$ at step $m + n/2 - 1$. As shown in Figure 9.5, these two maximum values enter *center*$(m)$ at the next step, that is step $m + n/2$. It is precisely when $M_{center}$ is finalized. Now the maximum of three can be selected accordingly, to finalize the overall maximum sum.

We have shown that the following theorem holds.

**Theorem 9.4.1.** *The modified mesh algorithm for the 2D MSP takes $m+n/2$ communication steps.*

We design an update routine for *center*$(i)$ as given in Algorithm 42 whose update routine is slightly modified from (9.4) to incorporate the selection of three maximum sums, $M_{left}$, $M_{right}$ and $M_{center}$.

Note that the bidirectional technique is partly due the divide-and-conquer approach. Each half problem is solved in parallel and the center problem is solved by the "conquer" step. The recursion is only one level deep. It would be a natural question if we could further this idea and achieve more reduction in the number of steps. Due to the inherent limitation in the mesh architecture, we may not be able to reduce the number of steps further, unless extra interconnection is offered. For example, we divide the mesh into

**Algorithm 42** Update $center(i)$ to compute $M_{center}$

```
Update:: center(i) do begin
1: if $center(i)$ is active //current step $\geq n/2+1$ then
2:     $M(i) \leftarrow \mathrm{MAX}\{M_{in}(i\text{-}1), M_{in}(i,n/2), M_{in}(i,n/2{+}1), M(i), cand_{in}(i,n/2)$
       $+cand_{in}(i,n/2{+}1)\}$
3: end if
end
```

two halves twice, emulating two-level deep recursion, such that a quarter of the network is in charge of a quarter of the input array. From the left-most quarter, let them call $LL$, $LR$, $RL$ and $RR$. Suppose $LL$ and $LR$ execute the bidirectional horizontal data flow as described above, $M_{left}$, the maximum subarray in the left half of the whole array, will be located between $LL$ and $LR$ after $m + n/4$ steps. Similarly $M_{right}$ will be located between $RL$ and $RR$. These two solutions are $n/2$ positions apart and comparing them in one step is not possible. Another difficulty arises in computing $M_{center}$. If we are allowed to provide extra communication lines linking centers of each half, this is possibly tractable. Even if it is possible, it may be arguable whether extra complexity in the network can justify the reduction of communication steps.

## 9.5 $K$ maximum subarrays

### 9.5.1 2D $K$-OMSP

If we apply the same idea used in extending Algorithm 3 to Algorithm 15, the mesh algorithm (Algorithm 38) is easily extended to compute the $K$-OMSP. Specifically, we prepare lists of registers $min[1..K]$, and $M[1..K]$, and produce $cand[1..K]$ by $s - min[1..K]$. Note that $m$ is in non-decreasing order, hence $cand$ is in non-increasing order. We maintain $M$ in non-increasing order too. Selection of $K$ largest values among $M_{in}(i-1, j)$, $M_{in}(i, j-1)$, $M(i, j)$ and $cand$ is easily done in $O(K)$ time by merging them while maintaining the order.

As a cell now transmits $O(K)$ register values, each inter-processor communication, would take $O(K)$ times more. If we have $O(K)$ communication lines between cells, this can be done in $O(1)$ time. Still, the time for each step is $O(K)$ since each cell spends $O(K)$ time internally.

While the total number steps is still $m+n-1$, the actual computing time is hence $K$ times more. For $n \times n$ array, it is $O(Kn)$ time.

The bidirectional data flow can be applied similarly to reduce the number of steps. To obtain $M_{center}$, we compute Cartesian sums with $M^{left}_{center}$ and $M^{right}_{center}$, such that the $K$ largest elements in $\{x_i + y_j | x_i \in M^{left}_{center}, y_j \in M^{right}_{center}\}$ are selected. Frederickson and Johnson [38] showed that selection in $X+Y$ can be done in $O(m+p\log(K/p))$ time, where $|X| = n \leq m = |Y|$ and $p =$. In this context, $m = n = K$, hence $O(K)$ time. Note that, $M_{center}$ will not be in sorted order. When $M_{center}$ is not sorted, each *center*$(i)$ $(i = 1..m)$ can no longer decide $M[1..K]$ by simple merging method, so they will need to perform the linear time selection algorithm [20]. It follows that the overall $K$ maximum subarrays unloaded from *center*$(m)$ after $m + n/2$ steps, will not be in sorted order. When the order is required, we can perform sequential sorting in extra $O(K \log K)$ time, or we can prepare additional sorting network for parallel sorting. With the first option, the total time is $O(Kn + K \log K)$, where the second term is absorbed since $K = O(n^4)$ in extreme. The second option with the sorting network may be considered more than necessary.

Note that the mesh algorithm with $T = O(Kn)$ and $P = O(n^2)$ for the 2D $K$-OMSP is not optimal. In Chapter 3, we showed that this can be solved in $O(n^3 + K \log \min(K, n))$ time sequentially. The optimal sequential algorithm, however, is intractable to run on the mesh, due to its framework based on recursion. It is expected that PRAM or hypercube will be a suitable architecture, which we leave as a future work.

### 9.5.2 2D $K$-DMSP

It was discussed that the trivial solution for the $K$-disjoint maximum subarrays is based on repeated application of Kadane's algorithm. We find the first maximum subarray $M_1(r_1, c_1)|(r_2, c_2)$, then replace every element within this subarray with $-\infty$ and run Kadane's algorithm again to find the second maximum subarray, etc. This simple approach is easily applicable with the mesh algorithm, either Algorithm 39 or Algorithm 38, such that each run of $T = m + n - 1$ steps will find the each subsequent maximum subarray. The algorithm repeats the following three steps $K$ times.

1. Compute $M_k(r_1, c_1)|(r_2, c_2)_k$, the $k$-th maximum subarray
2. Update $a$ by replacing all elements in $(r_1, c_1)|(r_2, c_2)_k$ with $-\infty$.
3. Load the updated input array $a$ into the mesh network to compute the next maximum subarray.

The algorithm based on the three steps above requires to update the input array before each run. Hence the major problem is how to update the input array and load it into the mesh network.

We incorporate the data loader discussed in Section 9.2 to execute the step 2 and 3. This requires that each loading component to know the coordinates of the previous maximum subarray, $(r_1, c_1)|(r_2, c_2)_k$.

In the conventional mesh, this *broadcasting* problem takes $O(m)$ time, meaning that each subsequent maximum subarray requires extra interleaving $O(m)$ steps. For now, we assume a hybrid model of mesh and CREW PRAM, such that this information is stored in the global memory and each loading component can concurrently read it. We may ignore the cost for broadcasting.

Now, we modify Algorithm 40 based on this assumption. Each loading component updates the input element (if required), and loads the element onto the mesh network *on-the-fly*, as shown in Algorithm 43. Again, $\alpha$ represents the current step.

This modified routine for $loader[1..m]$ is combined with the mesh algorithm for the single maximum subarray, such as Algorithm 41 or a modified version of Algorithm 38 that is configured for run-time data loading.

At each *run* of the mesh algorithm for the $k$-th maximum subarray, we spend $n$ steps to load the updated input array, and extra $m - 1$ steps to feed the null data until the $k$-th maximum is ready at $cell(m, n - m + 1)$. The coordinates $(r_1, c_1)|(r_2, c_2)_k$ is concurrently read by $loader[1..m]$, and the data loader restart loading the input array, updating the input when required.

**Algorithm 43** Update and load input array

```
**Load**($col$):: $loader[i]$ do begin
//$col = (\alpha+1)\text{mod}(m+n-1)$. If $col > n$, set $col = 0$ and we load a set of null data
1: $j' \leftarrow col$
2:  /* $r_1$, $r_2$, $c_1$ and $c_2$ are from $(r_1,c_1)|(r_2,c_2)_k$. */
3: if $i \in [r_1,r_2]$ and $j' \in [c_1,c_2]$ then  $a[i][j'] \leftarrow -\infty$, $a \leftarrow a[i][j']$
4: else $a \leftarrow a[i][j']$
5: $p, t, M \leftarrow 0$
6: $k \leftarrow i+1$
7: $l \leftarrow j'+1$
end
```

Each disjoint maximum subarray is found in $m+n-1$ steps. For $K$-disjoint maximum subarrays, the total number of steps required is therefore $K(m+n-1)$.

If we conform to the conventional mesh, the broadcasting will take $m/2$ steps. We establish a single line connection between $cell(m, n-m+1)$ and $loader(\frac{m}{2}, 1)$. In $m/2$ steps, the packet containing the coordinates can arrive at every loading component. It is then $K(\frac{3m}{2}+n-1)$ steps in total. If Algorithm 38 is exclusively used, and it is combined with the bidirectional computation as well as the run-time data loading scheme, we may reduce $n/2$ steps for each run. This may compensate the extra $m/2$ steps incurred due to the broadcasting. We omit the details of combining these techniques.

If we adopt less strict connection rules, we may allow $m$ connections from $cell(m, n-m+1)$ to all $loader[1..m]$'s. Then each loading component can receive the packet in one step. As we have one extra step before start computing the next maximum subarray, the total number of steps is then $K(m+n-1)+K-1$=$K(m+n)-1$. This result can be also improved with the bidirectional computation.

We conclude that whichever option for the broadcasting is applied, the total time for computing $K$-disjoint maximum subarrays is $O(Kn)$ if $m=n$. The time is comparable to that of the mesh algorithm for the 2D $K$-OMSP.

## 9.6 Summary of Results

In this chapter, we presented possible enhancements to the mesh algorithms for the 2D MSP. Run-time data loading can be implemented to minimize possible delays that involves input array loading. A coarse-grained configuration is possible to run the algorithm even if the size of input array is beyond the capacity of the mesh network.

We showed that Algorithm 38 can be further modified to achieve a constant factor improvement of $n/2$ steps.

For an array of size $n \times n$, we can easily modify Algorithm 38 to compute the 2D $K$-OMSP and achieved $O(Kn)$ time, and a simple repetition of either of two mesh algorithms can compute the 2D $K$-DMSP in $O(Kn)$ time. The latter problem requires slightly more number of communication steps due to the cost for broadcasting if the mesh topology is strictly complied.

Neither of the mesh algorithms for the 2D $K$-OMSP nor the 2D $K$-DMSP is, however, optimal. Note that we have $O(n^3 + K \log K)$ time and $O(n^3 + \min(K, n)n^2 \log n)$ time sequential algorithms. These algorithms are recursive in nature and not suitable for mesh implementation. We instead opted for less efficient sequential algorithms, whose times are both $T_{seq} = O(Kn^3)$. At least, the total cost of our mesh algorithms match this time complexity.

# Chapter 10

# Concluding Remarks and Future Work

## 10.1 Concluding Remarks

Sequential algorithms for the MSP are cubic time or slightly less than cubic time. We discussed further speedup option through parallel processing. Among various parallel architectures, we chose a *mesh* as an appropriate platform to compute the MSP.

The main focus in designing mesh algorithms for the MSP are laid upon their practicality. While the mesh algorithms for the 2D MSP are not as fast as $O(\log n)$ time previous parallel algorithms, their biggest advantage over previous solutions are its hardware implementability.

It is observed that the MSP is solved by four separate parallel phases, which are computation of prefix sums, prefix minima, candidates and prefix maxima. Previous parallel algorithms for the MSP are based on this observation and utilize the parallel prefix computation algorithm for a target architecture.

A simple 2D prefix computation algorithm can be easily devised such that it would run in $O(n)$ time ($2n-1$ steps) with a mesh network of size $O(n^2)$. From the previous parallel algorithms for the MSP, we know that three runs of the parallel prefix computations with different operator each time, such as $+$, $\text{MIN}$, $\text{MAX}$ would suffice computing the maximum subarray.

Even though no mesh algorithm for the 2D MSP has been previously reported, an $O(n)$ time ($3(2n-1)$ steps) mesh algorithm is well anticipated. For the 2D MSP, $O(n^3)$ time is practically the best known upper bound, hence the mesh algorithm based on this simple strategy can be considered cost-optimal. It is also interesting to observe that the use of a non-optimal, yet a simple mesh prefix computation algorithm results in an optimal algorithm for the 2D MSP.

The mesh algorithm based on such separate prefix computations has a room for further optimization. Due to tight communication restriction in a mesh, a constant factor improvement is highly considered. Our first mesh algorithm (Algorithm 38) is an extended version of the mesh prefix sum algorithm (Algorithm 35). Instead of running three prefix computations (+, MIN and MAX ) separately, they are executed internally in one step, achieving total of $2n - 1$ steps. The constant factor of 3 above is hence eliminated. The second mesh algorithm is based on Kadane's algorithm, and also completes the computation in $2n - 1$ steps. Algorithm 38 is further improved by $n/2$ steps with introduction of bi-directional data flow. This improvement may be viewed as a parallel execution of the iterative sequential algorithm with a limited divide-and-conquer flavor.

Some practical considerations have been taken. Particularly, efficient data loading into the mesh might be an important issue in practice. Considerable modification has been made to Algorithm 39 to enable run-time data loading, which can be similarly applied to Algorithm 38. In a situation where the size of mesh is smaller than the size of input, the mesh algorithms can be configured to run on a coarse-grained mesh.

The mesh algorithms for the 2D MSP can be extended to compute $K$ maximum subarrays. Algorithm 38 in particular can be easily modified to compute the 2D $K$-OMSP in $O(Kn)$ time. A simple repetition of either of two mesh algorithms can compute the 2D $K$-DMSP in $O(Kn)$ time too. The latter problem requires slightly more number of communication steps due to the cost for broadcasting.

The mesh architecture is inherently not as powerful as other parallel computational models, in terms of limited inter-processor communication capability. While this limitation often results in difficulty in increasing computational speed, its grid-like layout is desirable for a single layer chip implementation, and provides considerably superior practical implications. Various techniques to design the mesh algorithm and to optimize the performance discussed in the thesis are not limited to computing the MSP. Among various problems, those which involve a 2D array and have known sequential iterative algorithms are particularly suitable. Due to its network diameter,

$O(n)$ running time is the theoretical lower bound in a mesh computation. Considering the total cost, one that requires at least $O(n^3)$ time to compute sequentially may be a good problem to consider a mesh implementation.

## 10.2 Future Work

The presented mesh algorithm for the 2D $K$-DMSP is based on a simple repeated run of either Algorithm 39 or Algorithm 38 followed by the update to the input array. To facilitate loading the updated array, the data loader employed in the run-time loading scheme is introduced. To update the input array, such that the elements contained in the latest maximum subarray are updated to $-\infty$, each loading component should be aware of the location of the latest maximum subarray. This inherently involves broadcasting from $cell(m, n - m + 1)$ to every single loading component. If the conventional mesh topology is closely followed, zero-time consuming broadcasting is infeasible, making the mesh algorithm for the 2D $K$-DMSP running slower than its counterpart for the 2D $K$-OMSP. If there are $K$ connection lines between neighboring cells, the 2D $K$-OMSP takes $m + n - 1$ steps, while each cell spends $O(K)$ time for internal computation. A matching result for the 2D $K$-DMSP is an open problem. It is, however, expected to involve the design of a completely new sequential algorithm as a basis. Algorithm 30 is not suitable for mesh implementation due to its extensive use of global memory. For example, it maintains $O(m^2)$ tournaments, and the storage of these tournaments are obstructive to localize.

The FPGA implementation of Algorithm 39 has been undertaken by Steve Weddell and Bevan Langford from the department of Electrical and Computer Engineering at the University of Canterbury. Currently, the FPGA can handle the input array of size up to $14 \times 14$. Weddell plans to undertake further improvements, including implementation on a larger FPGA, incorporation of a pipelined data loading, and implementation of a serial interface such as USB or RS232.

# References

[1] Aho, A., Hopcroft, J., and Ullman, J. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, Mass., 1974.

[2] Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. Basic local alignment search tool. *Journal of Molecular Biology 215* (1990), 403–410.

[3] Altschul, S. F. Evaluating the statistical significance of multiple distinct local alignments. In *Theoretical and Computational Methods in Genomic Research*, S. Suhai, Ed. Plenum Press, 1997, pp. 1–14.

[4] Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings* (1967), vol. 30, pp. 483–485.

[5] Asano, T., Chen, D. Z., Katoh, N., and Tokuyama, T. Polynomial-time solutions to image segmentation. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1996), Society for Industrial and Applied Mathematics, pp. 104–113.

[6] Bae, S. E., and Takaoka, T. Parallel approaches to the maximum subarray problem. In *Proc. of the seventh Japan-Korea Workshop on Algorithms and Computation* (2003), pp. 94–104.

[7] Bae, S. E., and Takaoka, T. Algorithms for the problem of k maximum sums and a vlsi algorithm for the k maximum subarrays problem. In *Proc. of the International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN 2004)* (2004), pp. 247–253.

[8] Bae, S. E., and Takaoka, T. Improved algorithm for the k-maximum subarray problem for small k. In *Proc. of International Computing and Combinatorics Conference (COCOON 2005) , LNCS 3595* (2005), pp. 621–631.

[9] Bae, S. E., and Takaoka, T. Algorithm for $k$ disjoint maximum subarrays. In *Proc. of the International Conference on Computational Science (ICCS 2006), Part I* (2006), pp. 595–602.

[10] Bae, S. E., and Takaoka, T. Improved algorithms for the $k$-maximum subarray problem. *Computer Journal 49*, 3 (2006), 358–374.

[11] Bae, S. E., and Takaoka, T. Ranking cartesian sums and k-maximum subarrays. Tech. Rep. TR-COSC 03/06, Department of Computer Science and Software Engineering, University of Canterbury, 2006.

[12] Bae, S. E., and Takaoka, T. Algorithms for $k$-disjoint maximum subarrays. *International Journal of Foundations of Computer Science 18*, 2 (2007), 319–339.

[13] Banachowski, L. A complement to tarjan's result about the lower bound on the complexity of the set union problem. *Information Processing Letters 11* (1980), 59–65.

[14] Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. An asymptotically optimal multiversion b-tree. *The VLDB Journal 5*, 4 (1996), 264–275.

[15] Bengtsson, F., and Chen, J. Efficient algorithms for the k maximum sums. *ISAAC 2004, LNCS, Springer 3341* (2004), 137–148.

[16] Bengtsson, F., and Chen, J. A note on ranking k maximum sums. Tech. Rep. 2005:08, Luleå University of Technology, 2005.

[17] Bengtsson, F., and Chen, J. Ranking k maximum sums. *Theoretical Computer Science 377* (2007), 229–237.

[18] Bentley, J. Programming pearls: algorithm design techniques. *Communications of the ACM 27*, 9 (1984), 865–873.

[19] Bentley, J. Programming pearls: perspective on performance. *Communications of the ACM 27*, 11 (1984), 1087–1092.

[20] Blum, M., Floyd, R., Pratt, V., Rivest, R., and Tarjan, R. Time bounds for selection. *Journal of Computer and System Sciences 7*, 4 (1973), 448–461.

[21] Boguski, M., Hardison, R., Schwartz, S., and Miller, W. Analysis of conserved domains and sequence motifs in cellular regulatory proteins and locus control regions using new software tools for multiple alignment and visualization. *New Biology 4* (1992), 247–260.

[22] Brendel, V., Bucher, P., Nourbakhsh, I., Blaisdell, B. E., and Karlin, S. Methods and algorithms for statistical analysis of protein sequences. In *Proc. of the National Academy of Science USA* (1992), vol. 89, pp. 2002–2006.

[23] Brodal, G. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing 3*, 3 (1996), 238–255.

[24] Brodal, G. S., and Jørgensen, A. G. A linear time algorithm for the $k$ maximal sums problem. In *Proc. of Mathematical Foundations of Computer Science (MFCS 2007), LNCS 3708* (2007), pp. 442–453.

[25] Brown, M. R., and Tarjan, R. E. The design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing 9*, 3 (1980), 594–614.

[26] Cheng, C., Chen, K., Tien, W., and Chao, K. Improved algorithm for the k maximum sums problem. In *Proc. of ISAAC 2005, LNCS 3827* (2005), pp. 799–808.

[27] Cheng, C.-H., Chen, K.-Y., Tien, W.-C., and Chao, K.-M. Improved algorithms for the maximum-sums problems. *Theoretical Computer Science. 362* (2006), 162–170.

[28] Cosnard, M., and Trystram, D. *Parallel Algorithms and Architectures.* International Thomson Computer Press, London, 1995.

[29] Davuluri, R., Grosse, I., and Zhang, M. Computational identification of promoters and first exons in the human genome. *Nature Genetics 29* (2001), 412–417.

[30] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1986), ACM Press, pp. 109–121.

[31] Eppstein, D. Finding the $k$ shortest paths. *SIAM Journal on Computing 28*, 2 (1998), 652–673.

[32] Fan, T. H., Lee, S., Lu, H. I., Tsou, T. S., Wang, T. C., and Yao, A. An optimal algorithm for maximum-sum segment and its application in bioinformatics. In *Proc. of CIAA 2003, LNCS 2759* (2003), Springer-Verlag, pp. 251–257.

[33] Fariselli, P., Finelli, M., Marchignoli, D., Martelli, P. L., Rossi, I., and Casadio, R. Maxsubseq: and algorithm for segment-length optimization. the case study of the transmembrane spanning segments. *Bioinformatics 19*, 4 (2003), 500–505.

[34] Fischer, M. Efficiency of equivalence algorithms. In *Complexity of computer computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 153–168.

[35] Floyd, R. W., and Rivest, R. L. Expected time bounds for selection. *Communications of the ACM 18*, 3 (1975), 165–172.

[36] Flynn, M. J., and Kosaraju, S. R. Processes and their interactions. *Kybernetics* (1976), 159–163.

[37] Foster, M. J., and Kung, H. T. Design of special-purpose vlsi chips: Example and opinions. In *Proceedings of the 7th annual symposium on Computer Architecture* (1980), ACM Press, pp. 300–307.

[38] Frederickson, G., and Johnson, D. The complexity of selection and ranking in X+Y and matrices with sorted rows and columns. *Journal of Computer and System Sciences 24* (1982), 197–208.

[39] Frederickson, G. N. An optimal algorithm for selection in a min-heap. *Information and Computation 104*, 2 (1993), 197–214.

[40] Fukuda, K., and Takaoka, T. Analysis of air pollution ($pm_{10}$) and respiratory morbidity rate using $k$-maximum sub-array (2-d) algorithm. In *Proc. of the Twenty-second Annual ACM Symposium on Applied Computing (SAC 2007)* (2007), pp. 153–157.

[41] Fukuda, T., Morimoto, Y., Morishita, S., and Tokuyama, T. Data mining with optimized two-dimensional association rules. *ACM Trans. Database Syst. 26*, 2 (2001), 179–213.

[42] Galler, B. A., and Fisher, M. J. An improved equivalence algorithm. *Communications of the ACM 7*, 5 (1964), 301–303.

[43] Goldwasser, M. H., Kao, M.-Y., and Lu, H.-I. Linear-time algorithms for computing maximum-density sequence segments with bioinformatics applications. *Journal of Computer and System Sciences 70* (2005), 128–144.

[44] Graham, R. L., Yao, A. C., and Yao, F. F. Information bounds are weak in the shortest distance problem. *Journal of the ACM 27*, 3 (1980), 428–444.

[45] Grenander, U. *Pattern Analysis.* Springer Verlag, 1978.

[46] Gries, D. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming 2* (1982), 207–214.

[47] Hannenhalli, S., and Levy, S. Promoter prediction in the human genome. *Bioinformatics 17* (2001), S90–S96.

[48] Hardison, R., Slighton, J., Gumucio, D., Goodman, M., Stojanovic, N., and Miller, W. Locus control regions of mammalian beta-globin gene clusters: combining phylogenetic analyses and experimental results to gain functional insights. *Gene 205* (1997), 73–94.

[49] Hoare, C. A. R. Algorithm 63 (partiton) and algorithm 65 (find). *Communications of the ACM 4*, 7 (1961), 321–322.

[50] Hoare, C. A. R. Quicksort. *Computer Journal 5*, 1 (1962), 10–15.

[51] Hopcroft, J., and Ullman, J. Set merging algorithms. *SIAM Journal on Computing 2* (1973), 294–303.

[52] Horowitz, E., Sahni, S., and Rajasekaran, S. *Computer Algorithms.* Computer Science Press, 1997.

[53] Huang, X. An algorithm for identifying regions of a dna sequence that satisfy a content requirement. *Computer Applications in the Biosciences 10* (1994), 219–225.

[54] Intel Co. Product brief: Intel Core 2 Duo desktop processor. website:www.intel.com/products/processor/core2duo/prod_brief.pdf, 2006.

[55] International Telecommunication Union. *Recommendation ITU-R BT. 470-6,7, Conventional Analog Television Systems*, 1998.

[56] ITWeek.co.uk. Intel predicts 100-core processors. website:www.itweek.co.uk/itweek/news/2151478/intel-predicts-100-core, March 2006.

[57] Já Já, J. *Parallel Algorithms: Design and Analysis.* Addison-Wesley, 1992.

[58] Jacoboni, I., Martelli, P., Fariselli, P., De Pinto, V., and Casadio, R. Prediction of the transmembrane regions of beta barrel membrane proteins with a neural network based predictor. *Protein Science 10* (2001), 779–787.

[59] Johnson, D. B., and Mizoguchi, T. Selecting the kth element in X + Y and $x_1 + x_2 + ... + x_m$. *SIAM Journal on Computing 7*, 2 (1978), 147–153.

[60] Kaplan, H., and Tarjan, R. E. Purely functional representations of catenable sorted lists. In *STOC '96: Proc. of the twenty-eighth annual ACM symposium on Theory of computing* (New York, NY, USA, 1996), ACM Press, pp. 202–211.

[61] Karlin, S., and Brendel, V. Chance and significance in protein and dna sequence analysis. *Science 257* (1992), 39–49.

[62] Karlin, S., Bucher, P., Brendel, V., and Altschul, S. F. Statistical-methods and insights for protein and dna-sequences. *Annual Review of Biophysics and Biophysical Chemistry 20* (1991), 175–203.

[63] Knuth, D. *The art of computer programming.* Addison-Wesley, Reading, Mass., 1998.

[64] Koza, J. R., Bennett III, F. H., Andre, D., and Keane, M. A. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann, 1999.

[65] Krogh, A., Larsson, B., von Heijne, G., and Sonnhammer, E. L. L. Predicting transmembrane protein topology with a hidden markov model: Application to complete genomes. *Journal of Molecular Biology 305* (2001), 567–580.

[66] Kuan Yu, C., and Kun Mao, C. On the range maximum-sum segment query problem. In *Proc. of ISAAC 2004,LNCS 3341* (2004), Springer-Verlag, pp. 294–305.

[67] Kumar, V., Grama, A., Gupta, A., and Karypis, G. *Introduction to parallel computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Redwood, CA, 1994.

[68] Kung, H. T., and Leiserson, C. Systolic arrays (for vlsi). In *Sparse Matrix* (1978), SIAM, pp. 256–282.

[69] Kung, S. Y. *VLSI Array Processors.* Prentice Hall, 1988.

[70] Kyte, J., and Doolittle, R. F. A simple method for displaying the hydropathic character of a protein. *Journal of Molecular Biology 157*, 1 (1982), 105–132.

[71] Ladner, R. E., and Fisher, M. J. Parallel prefix computation. *Journal of the ACM*, 4 (1980), 831–838.

[72] Leighton, T. *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes.* Morgan-Kaufmann, 1992.

[73] Lin, T.-C., and Lee, D. Randomized algorithm for the sum selection problem. *Theoretical Computer Science 377* (2007), 151–156.

[74] Lin, Y. L., Jiang, T., and Chao, K. M. Efficient algorithms for locating the length-constrained heaviest segments with applications to biomolecular sequence analysis. *Journal of Computer and System Science 65* (2002), 570–586.

[75] Microelectronics Technology Division. Solid state image sensors terminology. Tech. Rep. DS 00-001, Eastman Kodak Company, Rochester, NY, 1994.

[76] Moore, G. Cramming more components onto integrated circuits. *Electronics* (1965).

[77] Perumalla, K., and Deo, N. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters 5*, 3 (1995), 367–373.

[78] Preparata, F. P., and Vuillemin, J. E. Area-time optimal vlsi networks for multiplying matrices. *Information Processing Letters 11*, 2 (1980), 77–80.

[79] Qiu, K., and Akl, S. Parallel maximum sum algorithms on interconnection networks. Tech. Rep. 99–431, Queen's University, Department of Computer and Information Science, 1999.

[80] Quinn, M. J. *Parallel Computing- Theory and Practice.* McGraw-Hill, 1994.

[81] Ranka, S., and Sahni, S. *Hypercube algorithms: with applications to image processing and pattern recognition.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[82] Ruzzo, W. L., and Tompa, M. A linear time algorithm for finding all maximal scoring subsequences. In *7th Intl. Conf. Intelligent Systems in Molecular Biology, AAAI Press* (1999), pp. 234–241.

[83] Sarnak, N., and Tarjan, R. E. Planar point location using persistent search trees. *Communications of the ACM 29*, 7 (1986), 669–679.

[84] Smith, D. R. Applications of a stategy for designing divide-and-conquer algorithms. *Science of Computer Programming 8* (1987), 213–229.

[85] Srikant, R., and Agrawal, R. Mining quantitative association rules in large relational tables. In *Proc. of ACM SIGMOD International Conference on Management of Data* (1996), pp. 1–12.

[86] Stojanovic, N., Florea, L., Riemer, C., Gumucio, D., Slightom, J., Goodman, M., Miller, W., and Hardison, R. Comparison of five method for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acid Research 27* (1999), 3899–3910.

[87] Takaoka, T. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters 43*, 4 (1992), 195–199.

[88] Takaoka, T. Mid-year examination 2001, cosc 229 algorithm. University of Canterbury, 2001.

[89] Takaoka, T. Efficient algorithms for the maximum subarray problem by distance matrix multiplication. In *Electronic Notes in Theoretical Computer Science* (2002), J. Harland, Ed., vol. 61, Elsevier.

[90] Takaoka, T. A faster algorithm for the all-pairs shortest path problem and its application. *COCOON 2004, LNCS, Springer 3106* (2004), 278–289.

[91] Takaoka, T. An $O(n^3 \log\log / \log n)$ time algorithm for the all-pairs shortest path problem. *Information Processing Letters 96*, 5 (2005), 155–161.

[92] Takaoka, T., and Umehara, K. An efficient vlsi algorithms for the all pairs shortest path problem. *Journal of Parallel and Distributed Computing 16*, 3 (1992), 265–270.

[93] Takaoka, T., Voges, K., and Pope, N. Algorithms for data mining. In *Business Applications and Computational Intelligence*, K. Voges and N. Pope, Eds. Idea Group Publishing, 2005, pp. 291–315.

[94] Tamaki, H., and Tokuyama, T. Algorithms for the maximum subarray problem based on matrix multiplication. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms* (1998), SIAM, pp. 446–452.

[95] Tarjan, R. E. Efficiency of a good but not linear set union algorithm. *Journal of the ACM 22*, 2 (1975), 215–225.

[96] Tarjan, R. E., and van Leeuwen, J. Worst-case analysis of set union algorithms. *Journal of the ACM 31*, 2 (1984), 245–281.

[97] Ullman, J. D. *Computational Aspect of VLSI.* Computer Science Press, Rockville, Maryland, 1984.

[98] Van Scoy, F. L. The parallel recognition of classes of graphs. *IEEE Transactions on Computers 29*, 7 (July 1980), 563–570.

[99] Viola, P., and Jones, M. Robust real-time face detection. *Internation Journal of Computer Vision 57*, 2 (2004), 137–154.

[100] Walder, R., Garrett, M., McClain, A., Beck, G., Brennan, T., Kramer, N., Kanis, A., Mark, A., Rapp, J., and Sheffield, V. Short tandem repeat polymorphic markers for the rat genome from marker-selected libraries associated with complex mammalian phenotypes. *Mammallian Genome 9* (1998), 1013–1021.

[101] Weddell, S., and Langford, B. Hardware implementation of the maximum subarray algorithm for centroid estimation. In *Proc. of Twenty-first Image and Vision Computing Conference New Zealand (IVCNZ 2006)* (2006), pp. 511–515.

[102] Weiss, M. *Data Structures and Algorithm Analysis in C*, 2 ed. Addison-Wesley, Menlo Park, CA, 1997.

[103] Wen, Z. Fast parallel algoritihms for the maximum sum problem. *Parallel Computing 21*, 3 (1995), 461–466.

[104] Yoda, K., Fukuda, T., Morimoto, Y., Morishita, S., and Tokuyama, T. Computing optimized rectilinear regions for association rules. In *Proc. of the 3rd International Conference on Knowledge Discovery and Data Mining* (1997), AAAI Press, pp. 96–103.

# Appendix A

# Proteins and Amino acids

## A.1 Protein Sequences

A protein is composed of large numbers of amino acids. The list of 20 amino acids and their hydropathy index by Kyte and Doolittle [70] are given in Table A.1.

| | **1-Letter Code** | **3-Letter Code** | **Name** | **Hydropathy Index** |
|---|---|---|---|---|
| 1 | A | Ala | Alanine | 1.8 |
| 2 | R | Arg | Arginine | -4.5 |
| 3 | N | Asn | Asparagine | -3.5 |
| 4 | D | Asp | Aspartic acid | -3.5 |
| 5 | C | Cys | Cysteine | 2.5 |
| 6 | Q | Gln | Glutamine | -3.5 |
| 7 | E | Glu | Glutamic acid | -3.5 |
| 8 | G | Gly | Glycine | -0.4 |
| 9 | H | His | Histidine | -3.2 |
| 10 | I | Ile | Isoleucine | 4.5 |
| 11 | L | Leu | Leucine | 3.8 |
| 12 | K | Lys | Lysine | -3.9 |
| 13 | M | Met | Methionine | 1.9 |
| 14 | F | Phe | Phenylalaline | 2.8 |
| 15 | P | Pro | Proline | -1.6 |
| 16 | S | Ser | Serine | -0.8 |
| 17 | T | Thr | Threonine | -0.7 |
| 18 | W | Trp | Tryptophan | -0.9 |
| 19 | Y | Tyr | Tyrosine | -1.3 |
| 20 | V | Val | Valine | 4.2 |

Table A.1: The 20 Amino Acids and Their Official Codes

**Example A.1.1.** In 1955, Frederick Sanger, a two time Nobel laureate in Chemistry, determined the first complete amino-acid sequence of insulin. The sequence of human insulin is the following chain of 110 residues*. The sequence is given in FASTA format:

```
>P01308|INS_HUMAN Insulin [Contains: Insulin B chain; Insulin
A chain] - Homo sapiens (Human).
MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAEDLQV
GQVELGGGPGAGSLQPLALEGSLQKRGIVEQCCTSICSLYQLENYCN
```

* Retrieved 20 December, 2006 from Expert Protein Analysis System (ExPASy) Proteomics Server. website: `http://ca.expasy.org/uniprot/P01308`

# Appendix B

# Data Mining and Numerical Attributes

## B.1 Example: Supermarket Database

The following tables are reproduction of those given in [93].

| customer | items | expenditure |
|---|---|---|
| 1 | ham, cheese, cereal, milk | \$42 |
| 2 | bread, cheese, milk | \$22 |
| 3 | ham, bread, cheese, milk | \$37 |
| 4 | bread, milk | \$12 |
| 5 | bread, cereal, milk | \$24 |
| 6 | ham, bread, cheese, cereal | \$44 |

Table B.1: Customers' Purchase Data

| customer | name | gender | age | annual income | address |
|---|---|---|---|---|---|
| 1 | Anderson | female | 33 | \$20,000 | suburb A |
| 2 | Bell | female | 45 | \$35,000 | suburb A |
| 3 | Chen | male | 28 | \$25,000 | suburb B |
| 4 | Dickson | male | 50 | \$60,000 | suburb B |
| 5 | Elias | male | 61 | \$65,000 | suburb A |
| 6 | Foster | female | 39 | \$45,000 | suburb B |

Table B.2: Customers' Personal Data

# Appendix C

# Publications

Early forms of the research contained in this thesis were published in six articles. The references to these articles are duplicated below:

[6] Bae, S. E., and Takaoka, T. Parallel approaches to the maximum subarray problem. In *Proc. of the seventh Japan-Korea Workshop on Algorithms and Computation* (2003), pp. 94–104.

[7] Bae, S. E., and Takaoka, T. Algorithms for the problem of k maximum sums and a VLSI algorithm for the k maximum subarrays problem. In *Proc. of the International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN 2004)* (2004), pp. 247–253.

[8] Bae, S. E., and Takaoka, T. Improved algorithm for the k-maximum subarray problem for small k. In *Proc. of International Computing and Combinatorics Conference (COCOON 2005) , LNCS 3595* (2005), pp. 621–631.

[9] Bae, S. E., and Takaoka, T. Algorithm for $k$ disjoint maximum subarrays. In *Proc. of the International Conference on Computational Science (ICCS 2006)* (2006).

[10] Bae, S. E., and Takaoka, T. Improved algorithm for the $k$-maximum subarray problem. *Computer Journal 49*, 3 (2006), 358–374.

[12] Bae, S. E., and Takaoka, T. Algorithms for $k$-disjoint maximum subarrays. *International Journal of Foundations of Computer Science 18*, 2 (2007), 319–339.

In accordance with Section 8(c) of the University of Canterbury PhD Regulations and Guidelines 2003, the following statement identifies my own contribution to these articles.

These articles constitute my own research which was carried out in co-operation with Prof. Tadao Takaoka. The summary of my own contribution to each article is given as below.

- [6]: Design of Mesh algorithm for the 2D MSP
- [7]: Design of Mesh algorithm for the 2D $K$-OMSP
- [8]: Establishing $O(K^2 + n \log K)$ time for the 1D $K$-OMSP through sampling before candidate generation
- [10]: Extending [8] by combining it with selection in sorted columns and a persistent 2-3 tree. Sampling in two dimensions.
- [9]: Hole creation in a tournament for the next disjoint maximum subarray and extension to the 2D $K$-DMSP
- [12]: Application of Union/Find algorithm to the 2D $K$-DMSP and the second-level heap

Prof. Tadao Takaoka provided the general topic of research and contributed useful advice which added to the clarity of these articles, especially regarding the general framework based on the prefix sums and the recursive computation of the maximum subarray.

Signed:____________________ Date:________________