© The Author 2006. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
doi:10.1093/comjnl/bxl007

Improved Algorithms for the *K*-Maximum Subarray Problem

SUNG EUN BAE* AND TADAO TAKAOKA

Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand
**Corresponding author: seb43@student.canterbury.ac.nz*

The maximum subarray problem is to find the contiguous array elements having the largest possible sum. We extend this problem to find K maximum subarrays. For general K maximum subarrays where overlapping is allowed, Bengtsson and Chen presented $O(min\{K + n\log^2 n, n\sqrt{K}\})$ time algorithm for one-dimensional case, which finds unsorted subarrays. Our algorithm finds K maximum subarrays in sorted order with improved complexity of $O((n + K)\log K)$. For the two-dimensional case, we introduce two techniques that establish $O(n^3)$ and subcubic time.

Keywords: Maximum subarray, persistent 2-3 tree, selection in matrices with sorted columns, distance matrix multiplication

Received 12 August 2005; revised 22 December 2005

1. INTRODUCTION

The maximum subarray problem was first described by Bentley in his literature Programming Pearls [1, 2] as an example to discuss the efficiency of computer programs. This problem determines an array portion that sums to the maximum value with respect to all possible array portions within the input array. When the input array is two-dimensional, we find a rectangular subarray with the largest possible sum. If all elements of an array are non-negative, this problem is trivial, as the entire array represents the solution. Similarly, if all elements are non-positive, the solution is empty with value 0. So we consider a dataset containing both positive and negative values.

In practice, a bitmap image has all non-negative pixel values. When the average is subtracted from each pixel, we can apply the maximum subarray algorithm to find the brightest area within the image.

For the one-dimensional case, we have an optimal linear time sequential solution. A simple extension of this solution can solve the two-dimensional problem in $O(m^2n)$ time for an $m \times n$ array ($m \leq n$), which is cubic when $m = n$ [1, 2]. In this paper, if only n appears in complexities for the two-dimensional case, we assume $m = n$.

The subcubic time solution based on Takaoka's subcubic distance matrix multiplication (DMM) algorithm [3] is given by Tamaki and Tokuyama [4], which is further simplified by Takaoka [5]. In the context of parallel computations, time and cost optimal PRAM and mesh algorithms for the one-dimensional case are described in [6]. For the two-dimensional case, EREW PRAM solutions achieving $O(\log n)$ time with $O(n^3/\log n)$ processors are given in [7, 8] and comparable result on interconnection networks is given in [9]. The EREW PRAM version of the subcubic algorithm in [4, 5] is given in [10], which also features a VLSI algorithm based on the technique introduced in Bentley's paper. This VLSI algorithm for the maximum subarray problem achieves $T = m + n - 2$ steps, which is $O(n)$ time using $O(n^2)$ sized hardware circuit.

Finding K maximum sums is a natural extension. This problem is discussed in [11] and [12]. The former provides $O(Kn)$ and $O(Km^2n)$ time solutions for the one- and two-dimensional cases in the course of the development of a systolic array algorithm of $O(Kn)$ time using $O(n^2)$ size hardware for the two-dimensional case. The latter brings the worst case time down to $O(min\{K + n\log^2 n,\ n\sqrt{K}\})$ for a one-dimensional array. There is, however, a subtle difference in the problem definition. While K maximum sums produced by [11] are sorted, sortedness is not considered in the complexity given in [12]. We use the problem definition used in [11] in this paper, such that the final solution will be in order.

An improvement achieving $O(n\log K + K^2)$ time for small K is presented in the preliminary version of this paper [13]. This solution is better than the previous one when $K \leq \sqrt{n}\log n$ and is $O(n\log K)$ time when $K \leq \sqrt{n\log n}$.

This paper reviews the preliminary work [13] and first presents $O(n\log K)$ time solution for $K \leq n$. The extra K^2 term from $O(n\log K + K^2)$ is removed using a partially persistent data structure [14] and an efficient selection

algorithm in matrices with sorted columns [15]. Since K may be theoretically as large as $n(n+1)/2$, we extend this solution to show that the same framework can be used for any K up to $K = n(n+1)/2$ with a complexity of $O((n+K)\log K)$.

If we use the above algorithm directly for the two-dimensional (n, n)-array, we have $O(n^2(n+K)\log K)$ time complexity. We describe two techniques that improve this time complexity to $O(n^3)$ for $K \leq n^{1.5}/\sqrt{\log n}$ and even sub-cubic for smaller K. The first is based on the sampling technique and provides a subroutine to the second solution. With advanced algorithms for DMM [3, 16, 17], the second reduces the complexity to subcubic.

A related topic is a similar problem with K disjoint subarrays, which may be more practical in some applications. Within this category, we can define several problems, and only the one-dimensional case received some attention, especially in bio-informatics. Further discussion on a possible extension will be made in the section on concluding remarks.

2. REVIEW OF THE MAXIMUM SUBARRAY PROBLEM

We give a two-dimensional array $a[1..m, 1..n]$ as an input data set. The maximum subarray problem is to find a rectangular portion $a[r_1..r_2, c_1..c_2]$ such that the sum of contained elements should be greater than or equal to the sum of any other rectangular portions of the dataset. We suppose the upper-left corner has coordinates (1, 1).

EXAMPLE 1. Let a be given by

$$a = \begin{bmatrix} -1 & 2 & -3 & 5 & -4 & -8 & 3 & -3 \\ 2 & -4 & -6 & -8 & 2 & -5 & 4 & 1 \\ 3 & -2 & 9 & -9 & \lceil -1 & 10 & -5\rceil & 2 \\ 1 & -3 & 5 & -7 & \lfloor 8 & -2 & 2\rfloor & -6 \end{bmatrix}$$

The maximum subarray is the array portion $a[3..4, 5..6]$ surrounded by inner brackets, whose sum is 15.

Bentley introduced Kadane's algorithm that finds the maximum sum within a one-dimensional array, whose time is linear [1], and extended it to two-dimensions. In this section, we review another $O(n)$ algorithm that provides a framework for K-maximum subarray problem. Its simple extended version can find K maximum sums in $O(Kn)$ time, which is given in [11].

2.1. Finding the maximum sum in $O(n)$ time

The following algorithm has its central algorithmic concept in the prefix sum. The prefix sums $sum[1..n]$ of a one-dimensional array $a[1..n]$ are computed by

```
sum[0] ← 0
for i → 1 to n do
    sum[i] ← sum[i − 1] + a[i]
end for
```

ALGORITHM 1. Maximum sum in a one-dimensional array.

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

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 one-dimensional array, we have to find indices x, y that maximize $\sum_{i=x}^{y} a[i]$.

The notations *min* and *max* in italic font are used for variables and MIN and MAX are used for minimum and maximum operations. We will use, however, min for minimum operation inside O-notation following the convention. Array variable names are also used to express the set or list given by the array elements.

Let min_i be the minimum prefix sum for an array portion $a[1..i-1]$. Then the following is obvious.

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

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

Based on Lemma 1, we can design the linear time algorithm that finds the maximum sum in a one-dimensional array (Algorithm 1). Comments are given by //.

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. At the end, M is the maximum sum.

2.2. Selecting k largest elements

We discuss a simple technique to select k largest elements from n elements, which will be used extensively throughout this paper. Suppose the array L 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 L as the solution and exit. Otherwise, the selection is *valid*. We proceed to select *kthMax* by the linear selection algorithm [18]. We compare each element of

ALGORITHM 2. *Select* (k, L): Select k largest elements of L.

```
1: L1, L2, L3 ← ∅
2: if k > n then
3:    return L and exit
4: end if
5: kthMax ← kth Max of L[1..n]
6: Partition L into (L1, L2, L3), where
   L1 = {x | x ∈ L, x > kthMax},
   L2 = {x | x ∈ L, x = kthMax},
   L3 = {x | x ∈ L, x < kthMax}
7: if |L1| < k then
8:    append first k − |L1| elements of L2 to L1
9: end if
10: return L1
```

ALGORITHM 3. K maximum sums in a one-dimensional array for $1 \leq K \leq n$.

```
1: for k ← 1 to K do
2:    min[k] ← ∞, M[k] ← −∞
3: end for
4: sum[0] ← 0, min[1] ← 0, M[1] ← 0
5: for i ← 1 to n do
6:    sum[i] ← sum [i − 1] + a[i]
7:    for k ← 1 to K do
8:       cand[k] ← sum[i] − min[k]
9:    end for
10:   //Select(K, L) by Algorithm 2
11:   M ← Select(K, merge(M, cand))
12:   insert sum[i] into min
13: end for
```

L against *kthMax* and partition them into $L1$, $L2$, $L3$, where $L1$ contains elements greater than *kthMax*, $L2$ contains elements equal to *kthMax*. All elements smaller than *kthMax* are kept in $L3$. This is done in $O(n)$ time. If $|L1| = k$, we take $L1$ as the solution. If there are multiple elements of the same value as the k-th maximum, $|L1| < k$. We take first $k - |L1|$ elements from $L2$ and append them to $L1$. $L1$ now contains k largest values as required. The total time is bounded by $O(n)$.

LEMMA 2. *Selection of k largest values from a set of n elements takes* $O(n)$ *time*.

2.3. Finding K maximum sums in $O(Kn)$ time

Based on Algorithm 1, 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 2 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 $K \leq n$ and modify it for the general case.

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

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

Let max_i be the list of K maximum sums for $a[1..i]$. This list is maintained in M in Algorithm 3, sorted in non-increasing order. When the algorithm ends, M contains the final solution max_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. max_{i+1} *is the list of K maximum elements of* $merge(max_i, cand_{i+1})$.

In Algorithm 3, the list *min* at the beginning of the i-th iteration stands for 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.

Note that we initialize the list of tentative solutions by $M = \{0, -\infty \cdots, -\infty\}$.

The line 11 in the algorithm preserves the loop-invariant from step i to step $i + 1$ as stated in Lemma 3. 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. Inserting a prefix sum into the list of minimum prefix sums 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 7–11 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 1.

ALGORITHM 4. K maximum sums in a one-dimensional array for $1 \leq K \leq n(n+1)/2$.

```
1: C ← ∅
2: for k ← 1 to MIN {K, n} do
3:     min[k] ← ∞
4: end for
5:     sum[0] ← 0, min[1] ← 0, M[1] ← 0
6: for i ← 1 to n do
7:     sum[i] ← sum[i − 1] + a[i]
8:     for k ← 1 to MIN {K, i} do
9:         append sum[i] − min[k] to C
10:    end for
11:    insert sum[i] into min
12: end for
13: M ← Select(K, C)
14: sort M
```

Note that Algorithm 3 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 3 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 3 to handle the general case better. Specifically, the following modification no more relies on Lemma 3. In Algorithm 4, we declare an empty set C and append each candidate to C. Finally, we select K largest candidates from C by Algorithm 2 and sort them.

The total time is $O \leq (n * \min(K, n) + K \log K)$, 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 3. In an extreme case when $K = n(n+1)/2$, it is $O(n^2 \log n)$ or $O(K \log K)$ time. The space complexity of this algorithm is $O \leq (n * \min(K, n))$ 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 3 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 improving Algorithm 3 in this paper. When $K \leq n$, we can apply a simple sampling technique to reduce the number of candidates. In Sections 3 and 4, we assume $K \leq n$ and give improved algorithms based on the sampling technique. In Section 5, we show how such a technique can be used for $n < K \leq n(n+1)/2$.

3. $O(n \log K + K^2)$ 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 3. 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 $K \leq n$. This solution is introduced in the preliminary paper [13] and provides a starting point for the further improved algorithm in Section 4.

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 $sample$.

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

We then select the K-th largest value $KthSample$ by a linear time selection algorithm [18]. 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 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 need 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 $KthSample$.

Such an idea is implemented as Algorithm 5. We describe the details of this algorithm.

3.1. 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

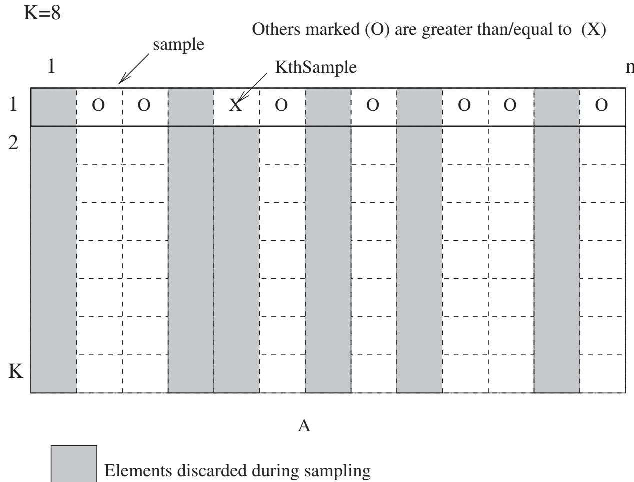

FIGURE 1. Selection of K *samples.*

$min_i[1..K]$ 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 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 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.

3.2. 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 1, but this does not improve the overall asymptotic complexity.

Now we analyse each part.

3.2.1. 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 latter part of this paper, Section 4.2.2, 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 $O(K^2)$.

ALGORITHM 5. Faster algorithm for K maximum sums in a one-dimensional array.

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

3.2.2. 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 $O(n \log K)$.

3.3. 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 ∞ 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 paper, Section 4.2.2. Both $min[1..K]$ and $M[1..K]$ are set in $O(K)$ time.

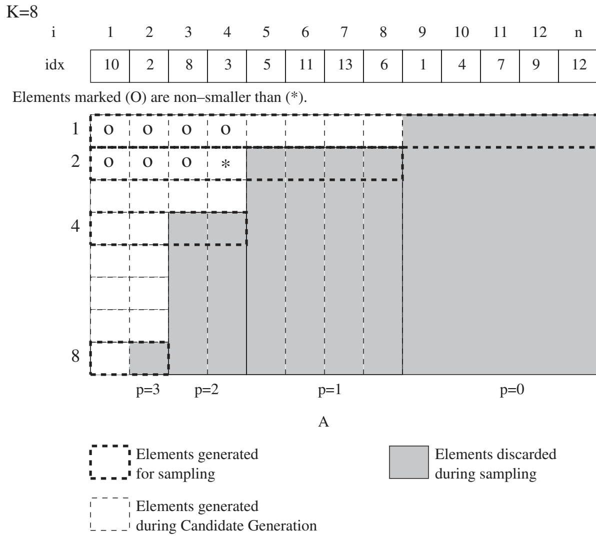

FIGURE 2. Sampling in row 1,2,4,8. . . .

The pre-process (sampling, selection and screening) is $O(n)$ time, when *KthSample* is selected by a linear time selection algorithm [18].

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 [12], 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 ($K \leq \sqrt{n\log n}$).

4. $O(n \log K)$ TIME ALGORITHM

The algorithm in Section 3 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 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 [15] 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, and 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 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 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 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 the termination of this process, the number of remaining elements is significantly smaller than in the previous solution shown in Figure 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 cannot 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 $A[1][x_1] \ldots A[1][x_K]$ 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 $A[2][x_1] \ldots A[2][x_K]$ 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 3 (lines 7–9) and Algorithm 5 (lines 21–23), $cand_i$ 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

ALGORITHM 6. Algorithm for K maximum sums in a one-dimensional array with generalized sampling technique.

```
 1: //Initialization
 2: for K ← 1 to K do min_0[k] ← ∞
 3: for i ← 1 to n do u[i] ← K
 4: sum[0] ← 0, min_1[1] ← 0
 5: //Pre-process
 6: for i ← 1 to n do
 7:    sum[i] ← sum[i − 1] + a[i]
 8:    insert sum[i] into min_i−1 // creates min_i
 9:    delete min_i[K + 1] // deletes from min_i to keep size K
10: end for
11: //Sampling/Reindexing
12: q ← n, q′ ← K, p ← 0, idx ← [1, 2, . . .n]
13: while 2^p ≤ K do
14:    //Compute A[2^p][1..q], contained in A
15:    for i ← 1 to q do
16:       A[i] ← sum[idx[i]] − min_idx[i][2^p]
17:    end for
18:    l ← q′-th max of A[2^p][1..q]
19:    Partition A into (A1, A2, A3), where
       A1 = {x|x ∈ A, x > l}, A2 = {x|x ∈ A, x = l},
       A3 = {x|x ∈ A, x < l}
20:    Copy prefix sum indices of elements in (A1, A2, A3) to idx[1..q]
21:    for i ← q′ + 1 to q do u[i] ← 2^p − 1
22:    p ← p + 1, q ← q′, q′ ← ⌈K/2^p⌉
23: end while
24: //Candidate Generation
25: C ← ∅
26: for i ← 1 to K do
27:    //u[i]: number of generation in col. i
28:    for k ← 1 to u[i] do
29:       append sum[idx[i]] − min_idx[i][k] to C
30:    end for
31: end for
32: //Final Selection of K maxima
33: M ← Select(K, C)
34: sort M
```

$sum[i]$ 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 [14] to overcome this problem.

In the following, we describe Algorithm 6 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 the ephemeral data structure for min, we use a partially persistent 2–3 tree for the 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 Section 4.2. Note that we use control variables i and k for row-wise and column-wise operations respectively in the following algorithm and its description.

4.1. Algorithm description

Algorithm 6 is composed of five major routines, namely, initialization, pre-process, sampling/reindexing, candidate generation and final selection of K maximum sums. We describe details of each routine.

4.1.1. 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.

4.1.2. Pre-process

Over n iterations, we compute the prefix sum and insert this prefix sum into min. We have n-versions of sorted sets $min_0[1..K] \cdots, min_n[1..K]$ maintained in a partially persistent 2-3 tree. We need to fix the number of leaf node to be K. Line 8 inserts a new prefix sum $sum[i]$ 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 10. Details of update operations to the persistent 2-3 tree are discussed in Section 4.2.

4.1.3. Sampling/reindexing

In our problem setting, we start with an empty array A whose dimension is $K \times n$. During the routine shown by lines 12–23, 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 reindexing. 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 2. Lines 15–17 show 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 18 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 2 such that all items greater than l are moved

to the left partition $A1$, equal to l to $A2$ and smaller to $A3$. Let $(A1,\ A2,\ A3) = \{sum[x_1] - min_{x_1}\ [1],\ \ sum[x_2] - min_{x_2}[1] \ldots\ sum[x_q] - min_{x_q}[1]\}$. 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 reindexing* 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 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. Note that $idx[1..K]$ at $p = 1$ correspond to the column indices marked (O) and (X) in Figure 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 labelled 'p = 0,1,2,3' in Figure 2 represent such aborted portions. Note that the first iteration of the 'while' loop at lines 13–23 is essentially equivalent to the sampling process used in Algorithm 5.

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 algorithm by Frederickson and Johnson.

Actual generation of non-aborted candidates is done in the next subroutine starting at line 25. We update the array u at line 21 to indicate how many candidates can be produced in each column of *A*. Due to the initialization, 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 reindexing is done, the columns of disqualified samples are not allowed to produce more than $(2^p - 1)$ candidates. When the 'while' loop at lines 13–23 terminates, we have $u[1..K] = \{8, 7, 3, 3, 1, 1, 1, 1\}$ for the example given in Figure 2. Further discussion is given in the next section.

Note that the update to q at line 22 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.

4.1.4. *Candidate generation*

We now produce all the elements that survived the sampling process. The final version of *idx* as shown in Figure 3 indicates

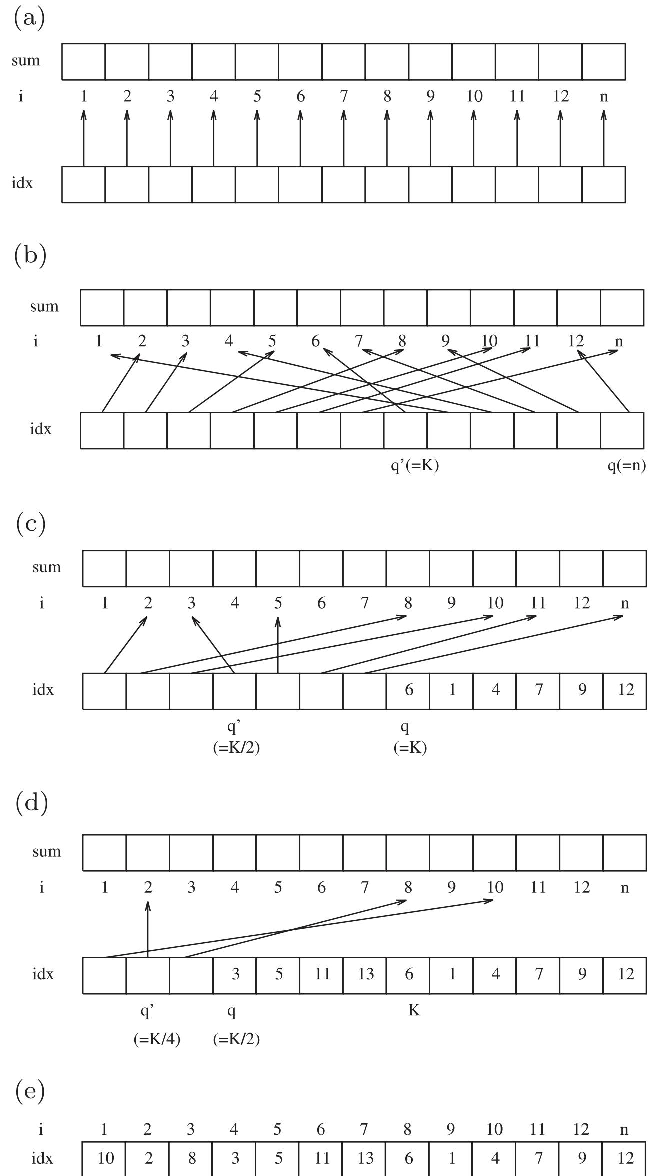

FIGURE 3. Rearrangement of index array *idx* when $n = 13$, $K = 8$. (a) Initialization. (b) p = 0. (c) p = 1. (d) p = 2. (e) Final values when p = 3.

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 4.1.3 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 4.2.2, 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 4.2.3 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 2.

We start with an empty set C, and append each generated element to a set C at lines 25–31. 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 $K + 2(K/2) + 4(K/4) + \cdots = O(K \log K)$.

While [15] 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 4.3.

4.1.5. *Final selection of the K maximum sums*

Finally, lines 33–34 describes the selection of K maximum elements in C. We sort such final K elements and obtain the sorted list of K maximum subarrays.

4.2. 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 [19], 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 [19, 20] 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[1].

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.* [14], there has been considerable development of persistent data structures [21, 22, 23, 24]. 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 [14]. As we only modify the newest version, a partially persistent structure will be sufficient.

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

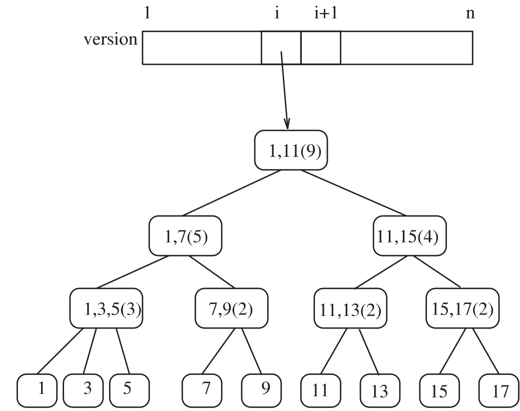

FIGURE 4. The i-th version of 2-3 tree. Leaf nodes represent sorted set in non-decreasing order. The number inside () shows the number of leaf nodes under this node. $K = 9$.

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 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 4.2.3.

4.2.1. *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 copy and update all the nodes in the path to the root in the same manner. Newly created nodes are shaded in grey in Figure 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

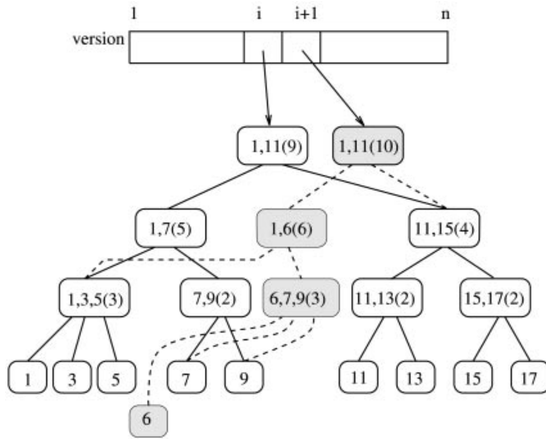

FIGURE 5. The (i+1)-th version is created when 6 is inserted.

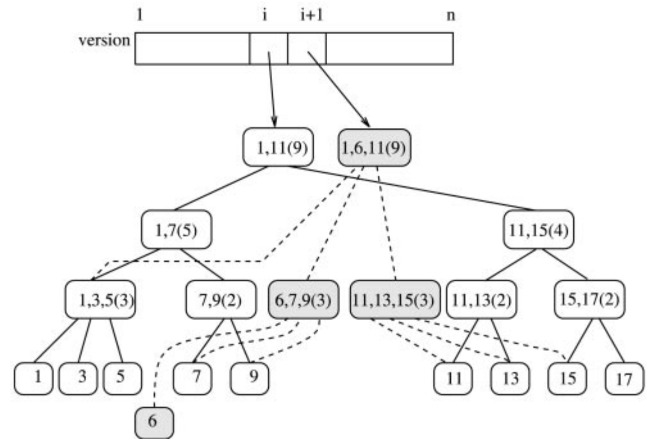

FIGURE 6. The largest item 17 is deleted from the (i+1)-th version to keep the size K.

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 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 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.

4.2.2. *Sequential access to leaf nodes*

We first examine the time for the following routine.

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

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.

4.2.3. *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, centre 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 centre subtree. Otherwise, it is in the right subtree. We perform at most three 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.

4.3 Analysis

First we analyse the time for the initialization. Inside the loop, K elements of min_1 are set sequentially. Due to Section 4.2.2, 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 4.2.1. The time for the pre-processing is thus $O(n \log K)$.

Let us examine the time complexity within the 'while' loop at lines 13–23. 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 lines 15–17. The first element of each n versions of min is retrieved spending $O(\log K)$ time each due to Section 4.2.3. The time spent by this line is thus $O(n \log K)$. Following routines (lines 18–20) are linear time operations on n samples. These $O(n)$ times are absorbed. The time spent by line 21 is $O(n - K)$, which is absorbed too.

When $p \geq 1$, at the p-th iteration, lines 15–17 generate $A[2^p][i]$ by $sum[idx[i]] - min_{idx[i]}[2^p]$ for $i = 1 \ldots q$, which are $q = \lceil K/2^{p-1} \rceil$ elements. When the loop is exited, the total number of samples generated is $K + K/2 + K/4 \ldots = O(K)$.

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 13–23, q is K, $\lceil K/2 \rceil$ ...etc. The time spent by lines 15–17 throughout 'while' loop ($p \geq 1$) is then $O((K + K/2 + K/4 \ldots) \log K) = O(K \log K))$.

Lines 18–20 perform linear operations on K, $\lceil K/2 \rceil$, $\lceil K/4 \rceil \ldots$ elements at each iteration. The total time is then $O(K + K/2 + K/4 + \ldots) = O(K)$. Similarly, the time by line 21 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 15–17 is the dominant one inside the 'while' loop.

The 'for' loop starting at line 26 involves the generation of non-discarded elements in the array A. There are $O(K \log K)$ elements remaining as discussed in Section 4.1.4. Note that lines 28–30 involve sequential reading from the sorted set maintained by 2-3 tree. It is done in linear time as discussed in Section 4.2.2. Then the total time for generating $O(K \log K)$ elements is bounded by $O(K \log K)$.

All the generated items are collected in C in no specific order. We proceed to line 33 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 34.

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

THEOREM 4.1. $\forall K \in [1..n]$, *the sorted list of K maximum subarrays is computed in $O(n \log K)$ time*.

We discuss the complexity for large K in the next section, Section 5.

Note that Frederickson and Johnson's algorithm [15] 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 whether 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 behaviour 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.

5. 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 5 nor Algorithm 6 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 5 is *invalid* as there are only n ($n < K$) elements. As previously

defined in Section 2.2, we use a term 'valid' to describe the opposite case, a meaningful application of the sampling technique.

For Algorithm 5 to support large K, we make a simple modification by combining it with Algorithm 4.

Since the sampling technique in Algorithm 5 is invalid for $K > n$, we skip the pre-process and perform lines 6–14 of Algorithm 4. Such a change gives $O((K + n)\log\min(K, n) + \min(K, n)^2)$ 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 5 does not improve Algorithm 4 for large K.

Now we consider Algorithm 6. It is obvious that the sampling in the first row is invalid as it was in Algorithm 5. 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 4.

For large K, notice that the size of each version of *min* will be at most n, for the same reason explained in Section 2.3. The size of the imaginary array A in Figure 2 is then (n, n). So we first let $K' = \mathrm{MIN}\{K, n\}$ in the beginning of Algorithm 6 and replace each appearance of K at line 2,3,9,13 and 26 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 $K/2^{p_0} \leq n$. 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 4.3.

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 4.1.4. 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 $K = O(n^2)$, $|C| = O(K)$.

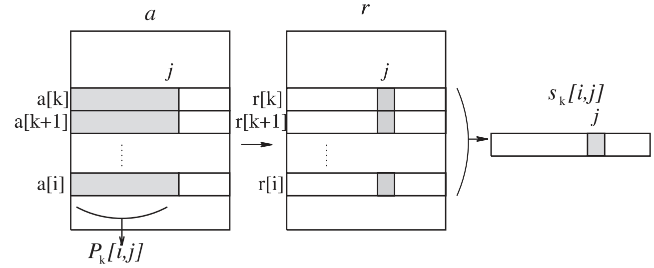

FIGURE 7. Perfix sum computation in two dimensions.

Each candidate needs $O(1)$ time for generation, making this subroutine $O(K\log n^2/K)$ 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 analysed the modified algorithm mostly for large K, this version also supports small K without affecting the complexity given in Theorem 4.1. We end this section with the following conclusion.

THEOREM 5.1. $\forall K \in [1..n(n+1)/2]$, *the sorted list of K maximum subarrays is computed in* $O((n + K)\log K)$ *time*.

6. SPEED-UP FOR TWO DIMENSIONS

If we use the algorithm in the previous section directly for an (n, n) array, we have an $O(n^2(n+K)\log K)$ time algorithm for $K \leq n(n+1)/2$. This is already more efficient than the $O(min\{n^2C, n^4\})$ by Bengtsson and Chen [12], where $C = min\{K + n\log^2 n, n\sqrt{K}\}$.

We further speed up the algorithm for the two-dimensional case. We introduce two different approaches based on the sampling technique and the divide-and-conquer method. The former achieves $O(n^3)$ time for $K \leq n^{1.5}/\sqrt{\log n}$. This solution is simpler and provides a subroutine to the latter, which achieves subcubic complexity.

6.1. Sampling in two dimensions

For an (n, n) array, $a[1..n, 1..n]$, we first build the row-wise prefix sum $r[1..n, 1..n]$ such that $r[i, j] = a[i, 1] + a[i, 2] + \cdots + a[i, j]$ for each row $i = 1..n$ as shown in Figure 7. This is done in $O(n^2)$ time. Let the portion $P_k[i, j]$ be the rectangular region of $a[k, 1]$ at the top-left corner and $a[i, j]$ at the bottom-right corner. We use a notation $s_k[i, j]$ to represent the sum of elements inside $P_k[i, j]$. The sum $s_k[i, j]$ is computed by $r[k, j] + \cdots + r[i, j]$. There are $n(n + 1)/2$ combinations of k and i for $1 \leq k \leq i \leq n$. Computing $s_k[i, 1..n]$ for all these combinations spends $O(n^3)$ time.

Each portion $P_k[i, n]$ can be regarded as a one-dimensional array where we perform the $O((n + K)\log K)$ solution

(Algorithm 6) using $s_k[i, 1..n]$ as its prefix sum. We get K maximum sums from each portion 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 2. The total time for two-dimensional array is then $O(n^2(n + K)\log K)$.

Using the sampling technique described in Section 3, 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.

For each portion $P_k[i, n]$, we compute the maximum sum by Algorithm 1. 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 the one-dimensional case on these selected portions only. Performing $O((n + K)\log K)$ time solution 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 2, 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 $O(n^3 + K(n + K)\log K + K^2 + K\log K)$. When K is small, this complexity is simply $O(n^3)$. For rest K, such that $n < K \leq n(n + 1)/2$, it is $O(n^3 + K^2 \log K)$ time. This is also cubic time if $K \leq n^{1.5}/\sqrt{\log n}$.

6.2. Divide-and-conquer

We examine the problem in divide-and-conquer methodology, which differs from the iterative approach that is adopted by the solutions previsouly discussed. Tamaki and Tokuyama present an $O(n^3\sqrt{\log\log n/\log n})$ time, which is subcubic, for the two dimensional maximum subarray problem based on divide-and-conquer approach [4] utilizing Takaoka's DMM algorithm [3]. The simplified solution with the same complexity is given in [5]. We modify this algorithm to compute the K-maximum subarray problem for two dimensions. The technique here is best illustrated by an analogy of extending a single-track railway into a double-track one when $K = 2$.

6.2.1. Distance matrix multiplication

The DMM is to compute the following distance product $C = AB$ for two (n, n)-matrices $A = [a_{ij}]$ and $B = [b_{ij}]$ whose elements are real numbers.

$$c_{ij} = \min_{1\leq k\leq n}\{a_{ik} + b_{kj}\}, \ (i, j = 1..n) \tag{1}$$

The operation in the right-hand side of Equation (1) is called DMM, of MIN-version, and A and B are called distance matrices in this context. If we use MAX instead, we call it the MAX-version.

Suppose we have an acyclic graph composed of three layers such that each layer has vertices $1 \ldots, n$, and the distances from the vertices in the first layer to those in the second are given by A and those from the second to the third are given by B. The intuitive meaning of DMM of MIN-version is that c_{ij} is the shortest path distance from vertex i in the first layer to vertex j in the third layer. Now we divide A, B, and C into (m, m)-submatrices for $N = n/m$ as follows:

$$\begin{pmatrix} A_{11} & \cdots & A_{1N} \\ \cdots & & \\ A_{N1} & \cdots & A_{NN} \end{pmatrix} \begin{pmatrix} B_{11} & \cdots & B_{1N} \\ \cdots & & \\ B_{N1} & \cdots & B_{NN} \end{pmatrix} = \begin{pmatrix} C_{11} & \cdots & C_{1N} \\ \cdots & & \\ C_{N1} & \cdots & C_{NN} \end{pmatrix}$$

Matrix C can be computed by

$$C_{ij} = \min_{1\leq k\leq N}\{A_{ik}B_{kj}\}(i, j = 1..N) \tag{2}$$

where the product of submatrices is defined similarly to Equation (1) and the MIN operation is defined on submatrices by taking the MIN operation component-wise. Since comparisons and additions of distances are performed in a pair, we omit counting the number of additions for measurement of the complexity. We have N^3 multiplications of distance matrices in Equation (2).

Let us assume that each multiplication of (m, m)-submatrices can be done in $T(m)$ time, assuming precomputed tables are available. The time for constructing the tables is reasonable when m is small. The time for MIN operations in Equation (2) is $O(n^3/m)$ in total. Thus the total time excluding table construction is given by $O(n^3/m + (n/m)^3T(m))$.

In the following, we show that $T(m) = O(m^{5/2})$. Thus the time becomes $O(n^3/m^{1/2})$.

Now we further divide the small (m, m)-submatrices into rectangular matrices in the following way. We rename the matrices A_{ik} and B_{kj} in (2) by A and B. Let $M = m/l$, where $1 \leq l \leq m$. Matrix A is divided into M (m, l)-submatrices $A_1, \ldots, A_M$ from left to right, and B is divided into $M(l, m)$-submatrices $B_1, \ldots, B_M$ from top to bottom. Note that A_k are vertically rectangular and B_k are horizontally rectangular. Then the product $C = AB$ can be given by

$$C = \min_{1\leq k\leq M}\{A_kB_k\} \tag{3}$$

We show later that A_kB_k can be computed in $O(l^2m)$ time, assuming a precomputed table is available. Then the right-hand side of Equation (3) can be computed in

$$O(Mm^2 + Ml^2m) = O(m^3/l + lm^2) \tag{4}$$

time, where the first term is for MIN operations component-wise and the second term is for computing the M distance products. Setting l to $m^{1/2}$, this time becomes $O(m^{5/2})$.

Rename again the matrices A_k and B_k in Equation (3) by A and B, and show how to compute AB, that is,

$$\underset{1\leq r\leq l}{\text{MIN}}\{a_{ir}+b_{rj}\}, \quad (i,j=1..m). \tag{5}$$

Assume that the lists of length m, $(a_{1r}-a_{1s},\ldots,a_{mr}-a_{ms})$, $(1\leq r<s\leq l)$, and $(b_{s1}-b_{r1},\ldots,b_{sm}-b_{rm})$, $(1\leq r<s\leq l)$ are already sorted for all r and s such that $1\leq r<s\leq l$. The time for sorting for all the lists is absorbed in the main complexity. Let E_{rs} and F_{rs} be the corresponding sorted lists. For each r and s, we merge lists E_{rs} and F_{rs} to form list G_{rs}. This takes $O(l^2m)$ time. Let H_{rs} be the list of ranks of $a_{ir}-a_{is}$ $(i=1..m)$ in G_{rs} and L_{rs} be the list of ranks of $b_{sj}-b_{rj}(j=1..m)$ in G_{rs}. Let $H_{rs}[i]$ and $L_{rs}[j]$ be the i-th and j-th components of H_{rs} and L_{rs} respectively. Then we have

$$G_{rs}[H_{rs}[i]]=a_{ir}-a_{is},\ G_{rs}[L_{rs}[j]]=b_{sj}-b_{rj}$$

The lists H_{rs} and L_{rs} for all r and s can be made in $O(l^2m)$ time, when the sorted lists are available.

We have the following obvious equivalence.

$$a_{ir}+b_{rj}\leq a_{is}+b_{sj}\Leftrightarrow a_{ir}-a_{is}\leq b_{sj}-b_{rj}\Leftrightarrow H_{rs}[i]\leq L_{rs}[j]$$

Fredman [26] observed that the information of ordering for all i, j, r, and s in the rightmost side of the above formula is sufficient to determine the product AB by a precomputed table. This information is essentially packed in the three dimensional space of $H_{rs}[i](i=1..m;\ r=1..l;\ s=r+1..l)$, and $L_{rs}[j](j=1..m;\ r=1..l;\ s=r+1..l)$. We call this the three-dimensional packing.

Takaoka [3] proposed that to compute each (i,j) element of AB, it is enough to know the above ordering for all r and s. We call this the two-dimensional packing. Note that the precomputed table must be obtained within the total time requirement. The two-dimensional packing will therefore allow a larger size of m, leading to a speed-up.

For simplicity, we omit i from $H_{rs}[i]$ and $L_{rs}[i]$ and define concatenated sequences H and L of length $l(l-1)/2$ by

$$H=H_{1,2}\ldots H_{1,l}H_{2,3}\ldots H_{2,l}\ldots H_{l,l-1}$$
$$L=L_{1,2}\ldots L_{1,l}L_{2,3}\ldots L_{2,l},\ \ldots L_{l,l-1}$$

For integer sequence $(x_1,\ldots,x_p)$, let $h(x_1,\ldots,x_p)=x_1\mu^{p-1}+\cdots+x_{p-1}\mu+x_p$. Let $h(H)$ and $h(L)$ be encoded integer values for H and L, where $p=l(l-1)/2$ and $\mu=2m$. The computation of h for H and L for all i takes $O(l^2m)$ time. By consulting a precomputed table *table* with the values of $h(H)$ and $h(L)$, we can determine the value of r that gives the minimum for Equation (5) in $O(1)$ time. For all i and j, it takes $O(m^2)$ time. Thus the time for one A_kB_k in Equation (3) is $O(l^2m)$, since $l^2=m$.

To compute *table*$[x][y]$, x and y are decoded into H and L. If $H_{s,r}>L_{s,r}$ for $s<r$ or $H_{r,s}<L_{r,s}$ for $r<s$, we can say r beats s in the sense that $a_{ir}+b_{rj}\leq a_{is}+b_{sj}$. We first fix r and check this condition for all such s. We repeat this for all r. If r is not beaten by any s, it becomes the table entry, that is, *table*$[x][y]=r$. Thus the table can be constructed in $O((l(l-1)/2)(2m)^{2l(l-1)/2})=O(c^{m\log m})$ time for some constant c. Let us set $m=\log n/(\log c\log\log n)$. Then we can compute the table in $O(n)$ time.

EXAMPLE 2. $m=5$, $2m=10$, $h(H)=456$, and $h(L)=329$. Since $H_{1,2}>L_{1,2}$ and $H_{2,3}<L_{2,3}$, the winner is 2, that is, *table*$[456, 329]=2$.

$$H=\begin{bmatrix} - & 4 & 5\\ - & - & 6\\ - & - & - \end{bmatrix}, L=\begin{bmatrix} - & 3 & 2\\ - & - & 9\\ - & - & - \end{bmatrix}$$

6.2.2. *Generalization of DMM*

To prepare for the K-maximum subarray problem, we extend Equation (1) in such a way that c_{ij} is the K-tuple of K minima of $\{a_{ik}+b_{kj}\,|\,k=1..n\}$. We call this definition K-distance matrix multiplication, or simply K-DMM. The intuitive meaning of K-DMM of MIN-version is that c_{ij} is the K shortest path distances from i to j in the same graph as described before.

Now we generalize the MIN and MAX operations on distance matrices. Let each element of a distance matrix be a K-tuple of real numbers such as $\mathbf{a}=(a_1\ \ldots,\ a_K)$. The MIN operation on the two K-tuples $\mathbf{a}$ and $\mathbf{b}$ is defined by MIN $\{\mathbf{a},\mathbf{b}\}=(c_1\ldots,c_K)$, where $(c_1\ldots,c_K)$ is the list of the K smallest elements of $\mathbf{a}\cup\mathbf{b}$. If there are equal values in $\mathbf{a}$ and/or $\mathbf{b}$, the union operation here is for multisets. Similarly we can define MAX $\{\mathbf{a},\mathbf{b}\}=\mathbf{a}\cup\mathbf{b}-(c_1\ \ldots,\ c_K)$.

The extended MIN and MAX operations can be performed by taking the smaller half and larger half from $\mathbf{a}\cup\mathbf{b}$, which can be done in $O(K)$ time by Algorithm 2. In the following we mainly describe the MIN-version. The MAX-version can be defined symmetrically.

If each element of distance matrices A_1 and A_2 is a K-tuple, the MIN operation on A_1 and A_2 is defined component-wise over corresponding K-tuples. To compute K-DMM, we use the extended MIN operation in Equation (2), where the elements of matrix $A_{ik}B_{kj}$ are K-tuples. The extended MIN operation is also done in Equation (3).

Let $K\leq O(\sqrt{\log n/\log\log n})$. We consider the problem of finding the minimum, second minimum, ..., K-th minimum in Equation (5) for $K\leq l$. We note that for this range of K, we can find the K minima in $O(K)$ time using an extended table. That is, the table entry for $h(H)$ and $h(L)$ gives the indices that give the values $\{a_{ir}+b_{rj}|r=1,\ldots,l\}$ in non-decreasing order, from which we can take the first K elements in $O(K)$ time.

EXAMPLE 3. Since $H_{1,2}>L_{1,2}$, $H_{2,3}<L_{2,3}$, and $H_{1,3}>L_{1,3}$, we have *table*$[456, 329]=231$.

We see the first term in the right-hand side of Equation (4) is multiplied by K by the extended MIN operation and so is the second term by the above approach, i.e. $O(Km^3/l+Klm^2)$.

The total time is $O(Kn^3/m + (n/m)^3T(m))$, where $T(m) = O(Km^{2.5})$. We conclude that K minima in Equation (1) can be computed in $O(Kn^3\sqrt{\log\log n/\log n})$ time, if $K \le l = \sqrt{\log n/(\log c \log\log n)}$.

To construct the table, we follow the method described in the last section. If we check the number of s's that r can beat for each r, we can determine the rank of r in the sorted list. Thus the time for table construction is still $O(n)$.

REMARK 1. In the above description, we can return K-tuples in sorted order. Thus we could have defined MIN{**a**, **b**} by merging, rather than selecting and filtering. We took the latter option as there might be a better algorithm to select the K smallest values from $\{a_{ir} + b_{rj} | r = 1..l\}$ in unsorted order.

6.2.3. *Subcubic time algorithm for the maximum subarray problem*

We review the divide-and-conquer approach given in [5]. Let a two-dimensional array $a[1..m, 1..n]$ of real numbers be given as input data. The maximum subarray problem here is to maximize the sum of the array portion $a[k..i, l..j]$, that is, to obtain such indices (k, l) and (i, j). We assume that $m \le n$ without loss of generality. We also assume that m and n are powers of 2. We will mention the general case of m and n later.

Bentley's algorithm finds the maximum subarray in $O(m^2n)$ time, which is cubic when $m = n$. The central algorithmic concept in this section is again that of prefix sum. We use DMMs of both MIN and MAX versions in this section.

We compute the prefix sums $s[i, j]$ for array portions of $a[1..i, 1..j]$ for all i and j with the boundary condition $s[i, 0] = s[0, j] = 0$. Note that $s[i, j]$ is a simplified form of the notation $s_1[i, j]$ given in Section 6.1. Obviously this can be done in $O(mn)$ time. The outer framework of the algorithm is given in Algorithm 7. Note that the prefix sums once computed are used throughout recursion.

In this algorithm, the column-centred 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

$$A_{centre} = \underset{\substack{0\le k\le i-1\\0\le l\le n/2-1\\1\le i\le m\\n/2+1\le j\le n}}{\mathrm{MAX}} \{s[i, j] - s[i, l] - s[k, j] + s[k, l]\}$$

In the above we first fix i and k, and maximize the above by changing l and j. Then the above problem is equivalent to maximizing the following.

For $i = 1..m$ and $k = 0..i - 1$,

$$A_{centre}[i, k] = \underset{\substack{0\le l\le n/2-1\\n/2+1\le j\le n}}{\mathrm{MAX}} \{-s[i, l] + s[k, l] + s[i, j] - s[k, j]\}$$

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

$$A_{centre}[i, k] = -\underset{0\le l\le n/2-1}{\mathrm{MIN}} \{s[i, l] + s^*[l, k]\} + \underset{n/2+1\le j\le n}{\mathrm{MAX}} \{s[i, j] + s^*[j, k]\} \quad (6)$$

ALGORITHM 7. 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.
   //Thus we assume $m \le n$
3: Let A_{left} be the solution for the left half.
4: Let A_{right} be the solution for the right half.
5: Let A_{centre} be the solution for the column-centred problem.
6: Let the solution be the maximum of those three.

The first part in the above is DMM of the MIN-version and the second part is of the MAX-version.

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

$$S_2S_2^* - S_1S_1^* \quad (7)$$

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 A_{centre} 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, n), where n is a power of 2. Then all parameters m and n appearing through recursion in Algorithm 7 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 A_{centre}'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 analyse the time for the work at level 0. We can multiply $(n, n/2)$ and $(n/2, n)$ matrices by 4 multiplications of size $(n/2, n/2)$, and there are two such multiplications in Equation (6). Let $M(n)$ be the time for multiplying two $(n/2, n/2)$ matrices. At level 0, we obtain an A_{centre} and two smaller A_{centre}'s, spending $12M(n)$ comparisons. Thus we have the following recurrence for the total time $T(n)$. The following lemma is obvious

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

LEMMA 4. *Let c be an arbitrary constant such that $c > 0$. Suppose $M(n)$ satisfies the condition $M(n) \ge (4 + c)\, M(n/2)$. Then the above $T(n)$* satisfies $T(n) \le 12(1 + 4/c)M(n)$.

Clearly the complexity of $O(n^3\sqrt{\log\log n/\log n})$ for $M(n)$ 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 $M(n)$.

ALGORITHM 8. *K*-Maximum subarray for two-dimensional array.

1: If the array becomes $n^{\alpha} \times n^{\alpha}$, return the solution by Algorithm A.
2..5: Same as Algorithm 7
6: Let the solution be the K-tuple of K maxima selected from $\{A_{left} \cup A_{right} \cup A_{centre}\}$

Suppose n is not given by a power of 2. By embedding the array a in an array of size (n', 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.

6.2.4. The K-maximum subarray problem

Now we describe the K-maximum subarray problem. When the recursion hits a (n^{α}, n^{α}) array for $0 \leq \alpha \leq 1$, we select K largest sums within this (n^{α}, n^{α}) array. The algorithm for the two-dimensional case in Section 6.1 solves this in $O(n^{3\alpha})$ time. Let us call this algorithm Algorithm A.

Suppose K is a power of 2. If not, we can choose the next power of 2 for K. Let $K \leq n^{\alpha}$. First we design Algorithm 8 by changing line 1 and line 6 in Algorithm 7.

Next we describe how to compute A_{centre} at each recursion. We first define the subtraction of two K-tuples, where K MAX $\{\mathbf{a}\}$ selects K largest elements in a set $\mathbf{a}$.

$$\mathbf{a} - \mathbf{b} = K\,\mathrm{MAX}\{a_i - b_j \mid a_i \in \mathbf{a}, \mathrm{b}_j \in \mathbf{b}, 1 \leq i, j \leq K\}$$

According to Frederickson and Johnson [15], selection of the K largest elements in Cartesian sum $X + Y$ is solved in $O(K)$ time, where $|X| = |Y| = K$. We can use this method for $\mathbf{a} - \mathbf{b}$ with $O(K)$ time. If we use $O(K^2)$ time algorithm by the exhaustive method, we can still achieve the final complexity as shown below.

The DMM of the MIN and MAX version in Equation (6) is replaced with the K-DMM of the MIN and MAX version. Note that each component of $S_2S_2^*$ and $S_1S_1^*$ in Equation (7) is now a K-tuple. The matrix subtraction is computed by $\mathbf{a} - \mathbf{b}$ operation component-wise.

Let us assume $K \leq O(\sqrt{\log n^{\alpha} / \log\log n^{\alpha}})$. Then we can use the K-DMM to compute the centre solution before hitting the bottom of recursion, and establish a recurrence equation similar to the one in Lemma 3, where the second term of $12M(n)$ is replaced by $12KM(n)$. As the complexity for Equation (7) in the recurrence is bounded by $O(n^2K^2) = O(n^2 \log n^{\alpha} / \log\log n^{\alpha})$ if we use the naive method, the complexity of this part is absorbed in the main complexity by increasing 12 slightly. The initial condition for T becomes $T(n^{\alpha}) = O(n^{3\alpha})$. As there are $n/n^{\alpha} \times n/n^{\alpha}$ subarrays at the bottom of recursion, the total time spent by Algorithm A is $O((n/n^{\alpha})^2 n^{3\alpha}) = O(n^{2+\alpha})$. If we use the $O(Kn^3\sqrt{\log\log n / \log n})$ time algorithm for K-DMM in Algorithm 8, the total time before hitting the bottom of recursion is $O(Kn^3\sqrt{\log\log n / \log n})$. Thus the total time is $O(Kn^3\sqrt{\log\log n / \log n} + n^{2+\alpha})$ for $K \leq O(\sqrt{\alpha \log n / \log\log n})$.

7. CONCLUDING REMARKS

In this paper, we studied K-maximum subarray problem for the one- and two-dimensional cases and presented improved algorithms.

For the one-dimensional case, we established $O((n + K)\log K)$ time algorithm. This solution produces K maximum subarrays in sorted order, while sortedness is not assumed in $O(min\{K + n\log^2 n, n\sqrt{K}\})$ time solution by Bengtsson and Chen [12]. Hence it requires extra $O(K \log K)$ time for sorting if necessary. Taking this account, our solution is more efficient than [12] for any K, $1 \leq \mathrm{K} \leq n(n + 1)/2$.

For the two-dimensional case, we showed that the worst-case time is cubic or subcubic if the value of K is relatively small. Specifically, $K \leq n^{1.5}/\sqrt{\log n}$ for cubic time, and $K \leq O(\sqrt{\alpha \log n / \log\log n})$, $(0 \leq \alpha \leq 1)$ for subcubic time.

If we find K maximum subarrays in a graphic image, those will heavily overlap. That is, we will find many array portions that only slightly differ in co-ordinates. If we are only interested in strictly disjoint portions, one way to solve this problem is the following greedy method. When we find the maximum sum using the two-dimensional version of Algorithm 1, we replace the value of each cell comprising the maximum sum with $-\infty$, and repeat this algorithm. By repeating this process, we can find the second maximum sum, the third etc. For a one-dimensional array, as each run takes $O(n)$ time, we can find the K-maximum subarray in $O(Kn)$ time. This is however solved in $O(n)$ time [27]. It remains to be seen if we can extend the $O(n)$ time algorithm to two dimensions with $O(n^3)$ time.

The sum of those maximum subarrays by this greedy method may not be the maximum of the total sum of K disjoint subarrays. This problem of minimizing the total sum of K disjoint subarrays has been solved in linear time for the one-dimensional case in [28]. To the authors' knowledge, the two-dimensional case has not been solved.

ACKNOWLEDGEMENTS

The authors express thanks to the referees whose constructive comments improved the technical quality of the paper to a great extent.

REFERENCES

[1] Bentley, J. (1984) Programming pearls: algorithm design techniques. *Commun. ACM*, **27**(9), 865–873.

[2] Bentley, J. (1984) Programming pearls: perspective on performance. *Commun. ACM*, **27**(11), 1087–1092.

[3] Takaoka, T. (1992) A new upper bound on the complexity of the all pairs shortest paths problem. *Inform Process. Lett.*, **43**(4), 195–199.

[4] Tamaki, H. and Tokuyama, T. (1998) Algorithms for the maximum subarray problem based on matrix multiplication.

In *Proc. SODA 1998*, San Francisco, CA, January 25–27, pp. 446–452. SIAM, Philadelphia, PA.

[5] Takaoka, T. (2002) Efficient algorithms for the maximum subarray problem by distance matrix multiplication. In *Proc. CATS 2002, Melbourne, Australia, January 28–February 1, ENTCS*, **61**, pp. 191–200. Elsevier, Amsterdam, The Netherlands.

[6] Miller, R. and Boxer, L. (2000) *Algorithms Sequential & Parallel—A Unified Approach*. Prentice Hall, Upper Saddle River, NJ.

[7] Perumalla, K. and Deo, N. (1995) Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Process. Lett.*, **5**(3), 367–373.

[8] Wen, Z. (1995) Fast parallel algorithms for the maximum sum problem. *Parallel Comput.*, **21**(3), 461–466.

[9] Qui, K. and Akl, S. G. (1999) Parallel maximum sum algorithms on interconnection networks. Department Computing and Information Science, Queen's University, Kingston, Ontario, Canada.

[10] Bae, S. E. and Takaoka, T. (2003) Parallel approaches to the maximum subarray problem. In *Proc. Japan–Korea Workshop on Algorithms and Computation 2003*, Sendai, Japan, July 3–4, pp. 94–104. Tohoku University, Sendai, Japan.

[11] Bae, S. E. and Takaoka, T. (2004) Algorithms for the problem of *K* maximum sums and a VLSI algorithm for the *K* maximum subarrays problem. In *Proc. ISPAN 2004*, Hong Kong, May 10–12, pp. 247–253. IEEE Computer Soceity Press, Los Alamitos, CA.

[12] Bengtsson, F. and Chen, J. (2004) Efficient algorithms for the k maximum sums. In *Proc. ISAAC 2004*, Hong Kong, December 20–22, *LNCS*, **3341**, pp. 137–148. Springer-Verlag, Berlin.

[13] Bae, S. E. and Takaoka, T. (2005) Improved algorithms for the *K*-maximum subarray problem for small *K*. In *Proc. COCOON 2005*, Kunming, Yunnan, China, August 16–19, pp. 621–631. Springer-Verlag, Berlin.

[14] Driscoll, J. R., Sarnak, N., Sleator, D. D. and Tarjan, R. E. (1986) Making data structures persistent. In *Proc. ACM STOC'86*, May 28–30, Berkeley, CA, pp. 109–121. ACM Press, New York, NY.

[15] Frederickson, G. N. and Johnson, D. B. (1982) The complexity of selection and ranking in X+Y and matrices with sorted columns. *J. Comput. Syst. Sci.*, **24**, 197–208.

[16] Takaoka, T. (2004) A faster algorithm for the all-pairs shortest path problem and its application. In *Proc. COCOON 2004*, Jeju Island, Korea, August 17–20, *LNCS*, **4106**, pp. 278–289. Springer-Verlag, Berlin.

[17] Zwick, U. (2004) A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. In *Proc. ISAAC 2004*, Hong Kong, December 20–22, *LNCS*, **3341**, pp. 921–932. Springer-Verlag, Berlin.

[18] Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L. and Tarjan, R. E. (1973) Time bounds for selection. *J. Comput. Syst. Sci.*, **7**(4), 448–461.

[19] Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

[20] Knuth, D. E. (1998) *The Art of Computer Programming*. Addison-Wesley, Reading, MA.

[21] Brodal, G. S. (1996) Partially persistent data structures of bounded degree with constant update time. *Nordic J. Comput.*, **3**(3), 238–255.

[22] Becker, B., Gschwind, S., Ohler, T., Seeger, B. and Widmayer, P. (1996) An asymptotically optimal multiversion B-tree. *VLDB J.*, **5**, 264–275.

[23] Sarnak, N. and Tarjan, R. E. (1986) Planar point location using persistent search trees. *Commun. ACM*, **29**(7), 669–679.

[24] Kaplan, H. and Tarjan, R. E. (1996) Purely functional representations of catenable sorted lists. In *Proc. ACM STOC'96*, Philadelphia, PA, May 22–24, pp. 202–211. ACM Press, New York, NY.

[25] Brown, M. R. and Tarjan, R. E. (1980) The design and analysis of a data structure for representing sorted lists. *SIAM J. Comput.*, **9**(3), 594–614.

[26] Fredman, M. (1976) New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, **5**, 83–89.

[27] Ruzzo, W. L. and Tompa, M. (1999) A linear time algorithm for finding all maximal scoring subsequences. In *Proc. ISMB'99*, Heidelberg, Germany, August 6–10, pp. 234–241. AAAI Press, Menlo Park, CA.

[28] Csürös, M. (2004) Algorithms for finding maxima-scoring segment sets. In *Proc. WABI 2004*, Bergen, Norway, September 14–17, *LNCS*, **3240**, pp. 62–73. Springer-Verlag, Berlin.