A Linear Time Algorithm for the k Maximal Sums Problem

Gerth Stølting Brodal[1,] and Allan Grønlund Jørgensen[1,⋆]

BRICS[⋆⋆], MADALGO[⋆⋆⋆], Department of Computer Science, University of Aarhus, Denmark. E-mail: {gerth,jallan}@daimi.au.dk

Abstract. Finding the sub-vector with the largest sum in a sequence of n numbers is known as the maximum sum problem. Finding the k sub-vectors with the largest sums is a natural extension of this, and is known as the k maximal sums problem. In this paper we design an optimal $O(n+k)$ time algorithm for the k maximal sums problem. We use this algorithm to obtain algorithms solving the two-dimensional k maximal sums problem in $O(m^2 \cdot n + k)$ time, where the input is an $m \times n$ matrix with $m \leq n$. We generalize this algorithm to solve the d-dimensional problem in $O(n^{2d-1} + k)$ time. The space usage of all the algorithms can be reduced to $O(n^{d-1} + k)$. This leads to the first algorithm for the k maximal sums problem in one dimension using $O(n + k)$ time and $O(k)$ space.

1 Introduction

To solve the maximum sum problem one must locate the maximal sum sub-vector of an array A of n numbers. The maximal sub-vector of A is the sub-vector $A[i, \ldots, j]$ maximizing $\sum_{s=i}^{j} A[s]$. The problem originates from Ulf Grenander who defined the problem in the setting of pattern recognition [1]. Solutions to the problem also have applications in areas such as Data Mining [2] and Bioinformatics [3].

The problem, and an optimal linear time algorithm credited to Jay Kadane, are described by Bentley [1] and Gries [4]. The algorithm they describe is a scanning algorithm which remembers the best solution, $\max_{1\leq i\leq j\leq t} \sum_{s=i}^{j} A[s]$, and the best suffix solution, $\max_{1\leq i\leq t} \sum_{s=i}^{t} A[s]$, in the part of the input array, $A[1, \ldots, t]$, scanned so far. Both values are updated in $O(1)$ time in each step yielding a linear time algorithm using $O(1)$ space.

The problem can be extended to any number of dimensions. In two dimensions the input is an $m \times n$ matrix of numbers and the task is to find the connected sub-matrix with the largest aggregate. The two-dimensional version

⋆ Supported in part by an Ole Roemer Scholarship from the Danish National Science Research Council.

⋆⋆ Basic Research in Computer Science, research school.

⋆⋆⋆ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

Table 1. Previous and new results for the k maximal sums problem

Paper	Time complexity
Bae & Takaoka [8]	$O(n \cdot k)$
Bengtson & Chen [9]	$O(\min\{k + n \log^2 n, n\sqrt{k}\})$
Bae & Takaoka [10]	$O(n \log k + k^2)$
Bae & Takaoka [11]	$O((n + k) \log k)$
Lie & Lin [12]	$O(n \log n + k)$ expected
Cheng *et al.* [13]	$O(n + k \log k)$
Liu & Chao [14][1]	$O(n + k)$
This paper	$O(n + k)$

was the original problem, introduced as a method for maximum likelihood estimations of patterns in digitized images [1].

With $m \leq n$, the problem can be solved by a reduction to $\binom{m}{2} + m$ one-dimensional problems resulting in an $O(m^2 \cdot n)$ time algorithm. The same reduction technique can be applied iteratively to solve the problem in any dimension. But unlike the one dimensional case these algorithms are not optimal. In [5] and [6] asymptotically faster algorithms for the two-dimensional problem are described. In [6] Takaoka designed an $O(m^2 n \sqrt{\log \log m / \log m})$ time algorithm by a reduction to (min,+) matrix multiplication [7].

A simple extension of the maximum sum problem is to compute the k largest sub-vectors for $1 \leq k \leq \binom{n}{2} + n$. The sub-vectors are allowed to overlap, and the output is k triples of the form (i, j, sum) where $sum = \sum_{s=i}^{j} A[s]$. This extension was introduced in [8]. The solution for $k = 1$ described above does not seem to be extendable in any simple manner to obtain a linear algorithm for any k. Therefore, different solutions to this extended problem has emerged over the past few years. These results are summarized in Table 1.

A lower bound for the k maximal sums problem is $\Omega(n+k)$, since an adversary can force any algorithm to look at each of the n input elements and the output size is $\Omega(k)$.

1.1 Results

In this paper we close the gap between upper and lower bounds for the k maximal sums problem. We design an algorithm computing the k sub-vectors with the largest sums in an array of size n in $O(n + k)$ time. We also describe algorithms solving the problem extended to any dimension. We begin by solving the two-dimensional problem where we obtain an $O(m^2 \cdot n + k)$ time algorithm for an $m \times n$ input matrix with $m \leq n$. This improves the previous best result [13], which was

[1] The k maximal sums problem can also be solved in $O(n + k)$ time by a reduction to Eppstein's solution for the k shortest paths problem [15] which also makes essential use of Fredericksons heap selection algorithm. This reduction was observed independently by Hsiao-Fei Liu and Kun-Mao Chao [14].

an $O(m^2 \cdot n + k \log k)$ time algorithm. This solution is then generalized to solve the d dimensional problem in $O(n^{2d-1} + k)$ time, assuming for simplicity that all sides of the d-dimensional input matrix are equally long. Furthermore we describe how to minimize the additional space usage of our algorithms. The additional space usage of the one dimensional algorithm is reduced from $O(n+k)$ to $O(k)$. The input array is considered to be read only. The additional space usage for the algorithm solving the two-dimensional problem is reduced from $O(m^2 \cdot n + k)$ to $O(n+k)$ and for the general algorithm solving the d dimensional problem the space is reduced from $O(n^{2(d-1)} + k)$ to $O(n^{d-1} + k)$.

The main contribution of this paper is the first algorithm solving the k maximal sums problem using $O(n + k)$ time and $O(k)$ space. The result is achieved by generating a binary heap that implicitly contains the $\binom{n}{2} + n$ sums in $O(n)$ time. The k largest sums from the heap are then selected in $O(n+k)$ time using the heap selection algorithm of Frederickson [16]. The heap is build using partial persistence [17]. The space is reduced by only processing k elements at time. The resulting algorithm can be viewed as a natural extension of Kadane's linear time algorithm for solving the maximum sum problem introduced earlier.

1.2 Outline of Paper

The remainder of the paper is structured as follows. In Section 2 the overall structure of our solution is explained. Descriptions and details regarding the algorithms and data structures used to achieve the result are presented in Sections 3, 4 and 5. In Section 6 we combine the different algorithms and data structures completing our algorithm. This is followed by Section 7 where we show how to use our algorithm to solve the problem in d dimensions. Finally in Section 8 we explain how to reduce the additional space usage of the algorithms without penalizing the asymptotic time bounds.

2 Basic Idea and Algorithm

In this paper the term heap denotes a max-heap ordered binary tree. The basic idea of our algorithm is to build a heap storing the sums of all $\binom{n}{2}+n$ sub-vectors and then use Fredericksons binary heap selection algorithm to find the k largest elements in the heap.

In the following we describe how to construct a heap that implicitly stores all the $\binom{n}{2} + n$ sums in $O(n)$ time. The triples induced by the $\binom{n}{2} + n$ sums in the input array are grouped by their end index. The *suffix set* of triples corresponding to all sub-vectors ending at position j we denote Q^j_{suf}, and this is the set $\{(i, j, sum) \mid 1 \le i \le j \ \wedge sum = \sum_{s=i}^{j} A[s]\}$. The Q^j_{suf} sets can be incrementally defined as follows:

$$Q^j_{\text{suf}} = \{(j, j, A[j])\} \cup \{(i, j, s + A[j]) \mid (i, j-1, s) \in Q^{j-1}_{\text{suf}}\}. \tag{1}$$

As stated in equation (1) the suffix set Q^j_{suf} consists of all suffix sums in Q^{j-1}_{suf} with the element $A[j]$ added as well as the single element suffix sum $A[j]$.

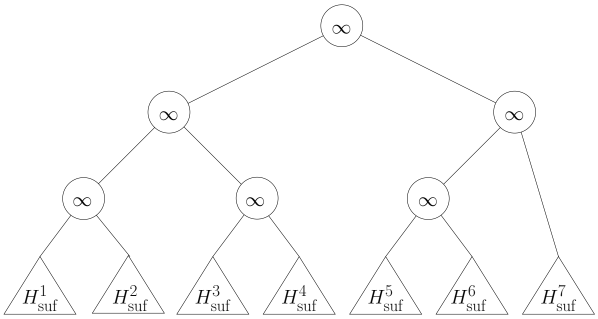

Fig. 1. Example of a complete heap H constructed on top of the H^j_{suf} heaps. The input size is 7.

Using this definition, the set of triples corresponding to all $\binom{n}{2} + n$ sums in the input array is the union of the n disjoint Q^j_{suf} sets. We represent the Q^j_{suf} sets as heaps and denote them H^j_{suf}. Assuming that for each suffix set Q^j_{suf}, a heap H^j_{suf} representing it has been build, we can construct a heap H containing all possible triples by constructing a complete binary heap on top of these heaps. The keys for the $n-1$ top elements is set to ∞ (see Figure 1). To find the k largest elements, we extract the $n-1+k$ largest elements in H using the binary heap selection algorithm of Frederickson [16] and discard the $n-1$ elements equal to ∞.

Since the suffix sets contain $\Theta(n^2)$ elements the time and space required is still $\Theta(n^2)$ if they are represented explicitly. We obtain a linear time construction of the heap by constructing an implicit representation of a heap that contains all the sums. We make essential use of a heap data structure to represent the Q^j_{suf} sets that supports insertions in amortized constant time.

Priority queues represented as heap ordered binary trees supporting insertions in constant time already exist. One such data structure is the self-adjusting binary heaps of Tarjan and Sleator described in [18] called Skew Heaps. The Skew heap is a data structure reminiscent of Leftist heaps [19, 20]. Even though the Skew heap would suffice for our algorithm it is able to do much more than we require. Therefore, we design a simpler heap which we will name *Iheap.* The essential properties of the Iheap are that it is represented as a heap ordered binary tree and that insertions are supported in amortized constant time.

We build H^{j+1}_{suf} from H^j_{suf} in $O(1)$ time amortized without destroying H^j_{suf} by using the partial persistence technique of [17] on the Iheap. This basically means that the H^j_{suf} heaps become different versions of the same Iheap. To make our Iheap partially persistent we use the node copying technique [17]. The cost of applying this technique is linear in the number of changes in an update. Since only the insertion procedure is used on the Iheap, the extra cost of using partial persistence is the time for copying amortized $O(1)$ nodes per insert operation. The overhead of traversing a previous version of the data structure is $O(1)$ per data/pointer access.

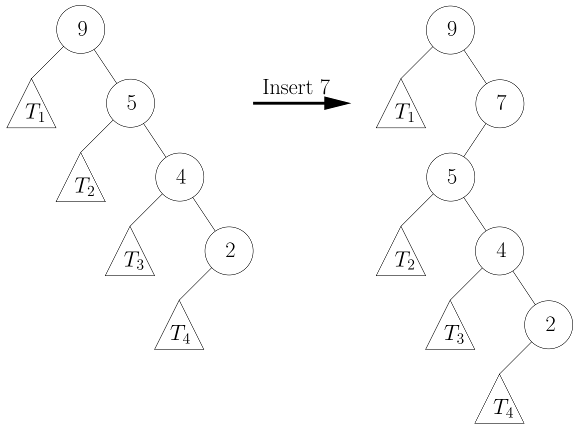

Fig. 2. An example of an insertion in the Iheap. The element 7 is compared to 2,4 and 5 in that order, and these elements are then removed from the rightmost path.

3 Binary Heaps

The main data structure of our algorithm is a heap supporting constant time insertions in the amortized sense. The heap is not required to support operations like deletions of the minimum or an arbitrary element. All we do is insert elements and traverse the structure top down during heap selection. We design a simple binary heap data structure Iheap by reusing the idea behind the Skew heap and perform all insertions along the rightmost path of the tree starting from the rightmost leaf.

A new element is inserted into the Iheap by placing it in the first position on the rightmost path where it satisfies the heap order. This is performed by traversing the rightmost path bottom up until a larger element is found or the root is passed. The element is then inserted as a right child of the larger element found (or as the new root). The element it is replacing as a right child (or as root) becomes the left child of the inserted element. An insertion in an Iheap is illustrated in Figure 2. If $O(\ell)$ time is used to perform an insertion operation because ℓ elements are traversed, the rightmost path of the heap becomes $\ell - 1$ elements shorter. Using a potential function on the length of the rightmost path of the tree we get amortized constant time insertions for the Iheap. Each element is passed on the rightmost path only once, since it is then placed on the left-hand side of element passing it, and never returns to the rightmost path.

Lemma 1. *The Iheap supports insertion in amortized constant time.*

4 Partial Persistence and H^j_{suf} Construction

As mentioned in Section 2 the H^j_{suf} heaps are build based on equation (1) using the partial persistence technique of [17] on an Iheap.

Data structures are usually *ephemeral*, meaning that an update to the data structure destroys the old version, leaving only the new version available for use. An update changes a pointer or a field value in a node. Persistent data structures allow access to any version old or new. Partially persistent data structures allow updates to the newest version, whereas fully persistent data structures allow updates to any version. With the partial persistence technique known as node copying, linked ephemeral data structures, with the restriction that for any node the number of other nodes pointing to it is $O(1)$, can be made partially persistent [17]. The Iheap is a binary tree and therefore trivially satisfies the above condition. The amortized cost of using the node copying technique is bounded by the cost of copying and storing $O(1)$ nodes from the ephemeral structure per update.

The basic idea of applying node copying to the Iheap is the following (see [17] for further details). Each persistent node contains one version of each information field in an original node, but it is able to contain several versions of each pointer (link to other node) differentiated by time stamps (version numbers). However, there are only a constant number of versions of any pointer, why each partially persistent Iheap node only uses constant space. Accessing relatives of a node in a given version is performed by finding the pointer associated with the correct time stamp. This is performed in constant time making the access time in the partially persistent Iheap asymptotically equal to the access time in an ephemeral Iheap.

According to equation (1), the set Q_{suf}^{j+1} can be constructed from Q_{suf}^{j} by adding $A[i+1]$ to all elements in Q_{suf}^{j} and then inserting an element representing $A[i+1]$. To avoid adding $A[i+1]$ to each element in Q_{suf}^{j}, we represent each Q_{suf}^{j} set as a pair $\langle \delta_j, H_{\mathrm{suf}}^{j} \rangle$, where H_{suf}^{j} is a version of a partial persistent Iheap containing all sums of Q_{suf}^{j} and δ_j is an element that must be added to all elements. With this representation a constant can be added to all elements in a heap implicitly by setting the corresponding δ. Similar to the way the Q_{suf}^{j} sets were defined by equation (1) we get the following incremental construction of the pair $\langle \delta_{j+1}, H_{\mathrm{suf}}^{j+1} \rangle$:

$$\langle \delta_0, H_{\mathrm{suf}}^{0} \rangle = \langle 0, \emptyset \rangle \ , \tag{2}$$

$$\langle \delta_{j+1}, H_{\mathrm{suf}}^{j+1} \rangle = \langle \delta_j + A[i+1], H_{\mathrm{suf}}^{j} \cup \{-\delta_j\} \rangle \ . \tag{3}$$

Let $\langle \delta_j, H_{\mathrm{suf}}^{j} \rangle$ be the latest pair built. To construct $\langle \delta_{j+1}, H_{\mathrm{suf}}^{j+1} \rangle$ from this pair, an element with $-\delta_j$ as key is inserted into H_{suf}^{j}. We insert this value, since δ_j has not been added to any element in H_{suf}^{j} explicitly, and because the sum $A[i+1]$ that the new element are to represent must be added to all elements in H_{suf}^{j} to obtain H_{suf}^{j+1}. Since we apply partial persistence on the heap, H_{suf}^{j} is still intact after the insertion, and a new version of the Iheap with the inserted element included has been constructed. H_{suf}^{j+1} is this new version and δ_{j+1} is set to $\delta_j + A[i+1]$. Therefore, the newly inserted element represents the sum $-\delta_j + \delta_j + A[i+1] = A[i+1]$. This ends the construction of the new pair $\langle \delta_{j+1}, H_{\mathrm{suf}}^{j+1} \rangle$. Since all sums from H_{suf}^{j} gets $A[i+1]$ added because of the increase of δ_{j+1} compared to δ_j and the new element represents $A[i+1]$ we conclude that $\langle \delta_{j+1}, H_{\text{suf}}^{j+1} \rangle$ represents the set Q_{suf}^{j+1}. The time needed for constructing H_{suf}^{j+1} is the time for inserting an element into a partial persistent Iheap. Since the size of an Iheap node is $O(1)$, by Lemma 1 and the node copying technique, this is amortized constant time

Lemma 2. *The time for constructing the n pairs $\langle \delta_j, H_{suf}^j \rangle$ is $O(n)$.*

5 Fredericksons Heap Selection Algorithm

The last algorithm used by our algorithm is the heap selection algorithm of Frederickson, which extracts the k largest[2] elements in a heap in $O(k)$ time. Input to this algorithm is an infinite heap ordered binary tree. The infinite part is used to remove special cases concerning the leafs of the tree, and is implemented by implicitly appending nodes with keys of $-\infty$ to the leafs of a finite tree. The algorithm starts at the root, and otherwise only explores a node if the parent already has been explored.

The main part of the algorithm is a method for locating an element, e, with $k \leq rank(e) \leq ck$ for some constant c[3]. After this element is found the input heap is traversed and all elements larger than e are extracted. Standard selection [21] is then used to obtain the k largest elements from the $O(k)$ extracted elements. To find e, elements in the heap are organized into appropriately sized groups named clans. Clans are represented by their smallest element, and these are managed in classic binary heaps [22].

By fixing the size of clans to $\log k$ one can obtain an $O(k \log \log k)$ time algorithm as follows. Construct the first clan by locating the $\lfloor \log k \rfloor$ largest elements and initialize a *clan-heap* with the representative of this clan. The children of the elements in this clan are associated with it and denoted its *offspring*.

A new clan is constructed from a set of $\log k$ nodes in $O(\log k \log \log k)$ time using a heap. However, not all the elements in an offspring set are necessarily put into such a new clan. The leftover elements are then associated to the newly created clan and are denoted the *poor relatives* of the clan.

Now repeatedly delete the maximum clan from the clan heap, construct two new clans from the offspring and poor relatives, and insert their representatives into the clan-heap. After $\lceil k/\lfloor \log k \rfloor \rceil$ iterations an element of rank at least k is found, since the representative of the last clan deleted, is the smallest of $\lceil k/\lfloor \log k \rfloor \rceil$ representatives. Since $2\lceil k/\lfloor \log k \rfloor \rceil + 1$ clans are created at most, each using time $O(\log k \log \log k)$, the total time becomes $O(k \log \log k)$.

By applying this idea recursively and then bootstrapping it Frederickson obtains a linear time algorithm.

Theorem 1 ([16]). *The k largest elements in a heap can be found in $O(k)$ time.*

[2] Actually Frederickson [16] considers min-heaps.
[3] The largest element has rank one.

6 Combining the Ideas

The heap constructed by our algorithm is actually a graph because the H^j_{suf} heaps are different versions of the same partially persistent Iheap. Also, the roots of the H^j_{suf} heaps include additive constants δ_j to be added to all of their descendants. However, if we focus on any one version, it will form an Iheap. This Iheap we can construct explicitly in a top down traversal starting from the root of this version, by incrementally expanding it as the partial persistent nodes are encountered during the traversal. Since the size of a partially persistent Iheap node is $O(1)$, the explicit representation of an Iheap node in a given version can be constructed in constant time.

However, the entire partially persistent Iheap does not need to be expanded into explicit heaps, only the parts actually visited by the selection algorithm. Therefore, we adjust the heap selection algorithm to build the visited parts of the heap explicitly during the traversal. This means that before any node in a H^j_{suf} heap is visited by the selection algorithm, it is build explicitly, and the newly built node is visited instead. We remark that the two children of an explicitly constructed Iheap node, can be nodes from the partially persistent Iheap.

The additive constants associated with the roots of the H^j_{suf} are also moved to the expanding heaps, and they are propagated downwards whenever they are encountered. An additive constant is pushed downwards from node v by adding the value to the sum stored in v, removing it from v, and instead inserting it into the children of v. Since nodes are visited top down by Fredericksons selection algorithm, it is possible to propagate the additive constants downwards in this manner while building the visited parts of the partially persistent Iheap. Therefore, when a node is visited by Fredericksons algorithm the key it contains is equal to the actual sum it represents.

Lemma 3. *Explicitly constructing t connected nodes in any fixed version of a partially persistent Iheap while propagating additive values downwards can be done in $O(t)$ time.*

Theorem 2. *The algorithm described in Section 2 is an $O(n+k)$ time algorithm for the k maximal sums problem.*

Proof. Constructing the pairs $\langle \delta_j, H^j_{\text{suf}} \rangle$ for $i = 1, \ldots, n$ takes $O(n)$ time by Lemma 2. Building a complete heap on top of these n pairs, see Figure 1, takes $O(n)$ time. By Lemma 3 and Theorem 1 the result follows.

7 Extension to Higher Dimensions

In this section we use the optimal algorithm for the one-dimensional k maximum sums problem to design algorithms solving the problem in d dimensions for any natural number d. We start by designing an algorithm for the k maximal sums problem in two dimensions, which is then extended to an algorithm solving the problem in d dimensions for any d.

Theorem 3. *There exists an algorithm for the two-dimensional k maximal sums problem, where the input is an $m \times n$ matrix, using $O(m^2 \cdot n + k)$ time and space with $m \leq n$.*

Proof. Without loss of generality assume that m is the number of rows and n the number of columns. This algorithm uses the reduction to the one-dimensional case mentioned in Section 1 by constructing $\binom{m}{2} + m$ one-dimensional problems. For all i, j with $1 \leq i \leq j \leq m$ we take the sub-matrix consisting of the rows from i to j and sum each column into a single entrance of an array. The array containing the rows from i to j can be constructed in $O(n)$ time from the array containing the rows from i to $j-1$. Therefore, we for each $i = 1, \ldots, m$ construct the arrays containing rows from i to j for $j = i, \ldots, m$ in this order.

For each one-dimensional instance we construct the n heaps H^j_{suf}. These heaps are then merged into one big heap by adding nodes with ∞ keys, by the same construction used in the one-dimensional algorithm, and use the heap selection algorithm to extract the result. This gives $(\binom{m}{2} + m) \cdot (n-1) + \binom{m}{2} + m - 1$ extra values equal to ∞.

It takes $O(n)$ time to build the H^j_{suf} heaps for each of the $\binom{m}{2} + m$ one-dimensional instances and $O(m^2 \cdot n + k)$ time to do the final selection. $\square$

The above algorithm is naturally extended to an algorithm for the d-dimensional k maximum sums problem, for any constant d. The input is a d-dimensional vector A of size $n_1 \times n_2 \times \cdots \times n_d$.

Theorem 4. *There exists an algorithm solving the d-dimensional k maximal sums problem using $O(n_1 \cdot \prod_{i=2}^{d} {n_i}^2)$ time and space.*

Proof. The dimension reduction works for any dimension d, *i.e.* we can reduce an d-dimensional instance to $\binom{n_d}{2} + n_d$ instances of dimension $d-1$. We iteratively use this dimension reduction, reducing the problem to one-dimensional instances. Let $A^{i,j}$ be the $d-1$-dimensional matrix, with size $n_1 \times n_2 \times \cdots \times n_{d-1}$ and $A^{i,j}[i_1] \cdots [i_{d-1}] = \sum_{s=i}^{j} A[i_1] \cdots [i_{d-1}][s]$.

We obtain the following incremental construction of a $d-1$-dimensional instance in the dimension reduction, $A^{i,j} = A^{i,j-1} + A^{j,j}$. Therefore, we can build each of the $\binom{n_d}{2} + n_d$ instances of dimension $d-1$ by adding $\prod_{i=1}^{d-1} n_i$ values to the previous instance. The time for constructing all these instances is bounded by:

$$
\begin{aligned}
T(1) &= 1 \\
T(d) &= \left(\binom{n_d}{2} + n_d \right) \cdot \left(T(d-1) + \prod_{i=1}^{d-1} n_i \right) ,
\end{aligned}
$$

which solves to $O(n_1 \cdot \prod_{i=2}^{d} {n_i}^2)$ for $n_i \geq 2$ and $i = 1, \ldots, d$. This adds up to $\prod_{i=2}^{d} (\binom{n_i}{2} + n_i) = O(\prod_{i=2}^{d} {n_i}^2)$ one-dimensional instances in total. For each one-dimensional instance the n_1 heaps, H^j_{suf}, are constructed. All heaps are assembled into one complete heap using $n_1 \cdot \prod_{i=2}^{d} \left(\binom{n_i}{2} + n_i \right) - 1$ infinity keys (∞) and heap selection is used to find the k largest sums. $\square$

8 Space Reduction

In this section we explain how to reduce the space usage of our linear time algorithm from $O(n + k)$ to $O(k)$. This bound is optimal in the sense that at least k values must be stored as output.

Theorem 5. *There exists an algorithm solving the k maximal sums problem using $O(n + k)$ time and $O(k)$ space.*

Proof. The original algorithm uses $O(n + k)$ space. Therefore, we only need to consider the case where $k \leq n$. Instead of building all n heaps at once, only k heaps are built at a time. We start by building the k first heaps, $H^1_{\text{suf}}, \ldots, H^k_{\text{suf}}$, and find the k largest sums from these heaps using heap selection as in the original algorithm. These elements are then inserted into an *applicant set*. Then all the heaps except the last one are deleted. This is because the last heap is needed to build the next k heaps. Remember the incremental construction of H^{j+1}_{suf} from H^j_{suf} defined in equation (3) based on a partial persistent Iheap.

We then build the next k heaps and find the k largest elements as before. These elements are merged with the applicant set and the k smallest are deleted using selection [21]. This is repeated until all H^j_{suf} heaps have been processed. The space usage of the last heap grows by $O(k)$ in each iteration, ruining the space bound if it is reused. To remedy this, we after each iteration find the k largest elements in the last heap and build a new Iheap with these elements using repeated insertion. The old heap is then discarded. Only the k largest elements in the last heap can be of interest for the suffix sums not yet constructed, thus the algorithm remains correct.

At any time during this algorithm we store an applicant set with k elements and k heaps which in total contains $O(k)$ elements. The time bound remains the same since there are $O(\frac{n}{k})$ iterations each performed in $O(k)$ time. □

In the case where $k = 1$, it is worth noticing the resemblance between the algorithm just described and the optimal algorithm of Jay Kadane described in the introduction. At all times we remember the best sub-vector seen so far. This is the single element residing in the applicant set. In each iteration we scan one entrance more of the input array and find the best suffix of the currently scanned part of the input array. Because of the rebuilding only two suffixes are constructed in each iteration and only the best suffix is kept for the next iteration. We then update the best sub-vector seen so far by updating the applicant set. In these terms with $k = 1$ our algorithm and the algorithm of Kadane are the same and for $k > 1$ our algorithm can be seen as a natural extension of it.

The original algorithm solving the two-dimensional version of the problem requires $O(m^2 \cdot n + k)$ space. Using the same ideas as above, we design an algorithm for the two-dimensional k maximal sums problem using $O(m^2 \cdot n + k)$ time and $O(n + k)$ space.

Theorem 6. *There exists an algorithm for the k maximal sums problem in two dimensions using $O(m^2 \cdot n + k)$ time where $m \leq n$ and $O(n + k)$ additional space.*

Proof. Using $O(n)$ space for a single array we iterate through all $\binom{m}{2} + m$ one-dimensional instances in the standard reduction creating each new instance from the last one in $O(n)$ time. We only store in memory $\lceil \frac{k}{n} \rceil$ instances at a time.

We start by finding the k largest sub-vectors from the first $\lceil \frac{k}{n} \rceil$ instances by concatenating them into a single one-dimensional instance separated by $-\infty$ values and use our one-dimensional algorithm. No returned sum will contain values from different instances because that would imply that the sum also included a $-\infty$ value. The k largest sums are saved in the applicant set. We then repeatedly find the k largest from the next $\lceil \frac{k}{n} \rceil$ instances in the same way and update the applicant set in $O(k)$ time using selection. When all instances have been processed the applicant set is returned.

If $k \leq n$ we only consider one instance in each iteration. The k largest sums from this instance is found and the applicant set is updated. This can all be done in $O(n + k) = O(n)$ time using the linear algorithm for the one-dimensional problem and standard selection. There are $\binom{m}{2} + m$ iterations resulting in an $O(m^2 \cdot n) = O(m^2 \cdot n + k)$ time algorithm.

If $k > n$ each iteration considers $\lceil \frac{k}{n} \rceil \geq 2$ instances. These instances are concatenated using $\lceil \frac{k}{n} \rceil - 1$ extra space for the ∞ values. The k largest sums from these instances are found from the concatenated instance using the linear one-dimensional algorithm in $O((\lceil \frac{k}{n} \rceil \cdot n) + k) = O(k)$ time. The number of iterations is $(\binom{m}{2} + m) / \lceil \frac{k}{n} \rceil \leq (\binom{m}{2} + m) \cdot \frac{n}{k}$, leading to an $O(m^2 \cdot n + k)$ time algorithm.

For both cases the additional space usage is at most $O(n + k)$ at any point during the iteration since only the applicant set, $\lceil \frac{k}{n} \rceil$ instances, and $\lceil \frac{k}{n} \rceil - 1$ dummy values are stored in memory at any one time. $\square$

The above algorithm is extended naturally to solve the problem for d dimensional inputs of size $n_1 \times n_2 \times \cdots \times n_d$.

Theorem 7. *There exists an algorithm solving the d-dimensional k maximal sums problem using* $O(n_1 \cdot \prod_{i=2}^{d} {n_i}^2)$ *time and* $O(\prod_{i=1}^{d-1} n_i + k)$ *additional space.*

Proof. As in Theorem 4, we apply the dimension reduction repeatedly, using $d-1$ vectors of dimension $1, 2, \ldots, d-1$ respectively, to iteratively construct each of the $\prod_{i=2}^{d}(\binom{n_i}{2} + n_i) = O(\prod_{i=2}^{d} {n_i}^2)$ one-dimensional instances. Every time a $d-1$-dimensional instance is created we recursively solve it. Again only $\lceil \frac{k}{n_1} \rceil$ one-dimensional instances and the applicant set is kept in memory at any one time and the algorithm proceeds as in the two-dimensional case. The space required for the arrays is $\sum_{i=1}^{d-1} \prod_{j=1}^{i} n_j = O(\prod_{i=1}^{d-1} n_i)$ with $n_i \geq 2$ for all i. $\square$

References

1. Bentley, J.: Programming pearls: algorithm design techniques. Commun. ACM **27**(9) (1984) 865–873
2. Fukuda, T., Morimoto, Y., Morishita, S., Tokuyama, T.: Data mining with optimized two-dimensional association rules. ACM Trans. Database Syst. **26**(2) (2001) 179–213

3. Allison, L.: Longest biased interval and longest non-negative sum interval. Bioinformatics **19**(10) (2003) 1294–1295
4. Gries, D.: A note on a standard strategy for developing loop invariants and loops. Sci. Comput. Program. **2**(3) (1982) 207–214
5. Tamaki, H., Tokuyama, T.: Algorithms for the maximum subarray problem based on matrix multiplication. In: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1998) 446–452
6. Takaoka, T.: Efficient algorithms for the maximum subarray problem by distance matrix multiplication. Electr. Notes Theor. Comput. Sci. **61** (2002)
7. Takaoka, T.: A new upper bound on the complexity of the all pairs shortest path problem. Inf. Process. Lett. **43**(4) (1992) 195–199
8. Bae, S.E., Takaoka, T.: Algorithms for the problem of k maximum sums and a vlsi algorithm for the k maximum subarrays problem. In: 7th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2004), 10-12 May 2004, Hong Kong, SAR, China, IEEE Computer Society (2004) 247–253
9. Bengtsson, F., Chen, J.: Efficient algorithms for k maximum sums. In Fleischer, R., Trippen, G., eds.: Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings. Volume 3341 of Lecture Notes in Computer Science., Springer (2004) 137–148
10. Bae, S.E., Takaoka, T.: Improved algorithms for the k-maximum subarray problem for small k . In Wang, L., ed.: Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings. Volume 3595 of Lecture Notes in Computer Science., Springer (2005) 621–631
11. Bae, S.E., Takaoka, T.: Improved algorithms for the k-maximum subarray problem. Comput. J. **49**(3) (2006) 358–374
12. Lin, T.C., Lee, D.T.: Randomized algorithm for the sum selection problem. In Deng, X., Du, D.Z., eds.: Algorithms and Computation, 16th International Symposium, ISAAC 2005, Sanya, Hainan, China, December 19-21, 2005, Proceedings. Volume 3827 of Lecture Notes in Computer Science., Springer (2005) 515–523
13. Cheng, C.H., Chen, K.Y., Tien, W.C., Chao, K.M.: Improved algorithms for the k maximum-sums problems. Theoretical Computer Science **362**(1-3) (2006) 162–170
14. Chao, K.M., Liu, H.F.: Personal communication (2007)
15. Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. **28**(2) (1999) 652–673
16. Frederickson, G.N.: An optimal algorithm for selection in a min-heap. Inf. Comput. **104**(2) (1993) 197–214
17. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. Journal of Computer and System Sciences **38**(1) (1989) 86–124
18. Sleator, D.D., Tarjan, R.E.: Self adjusting heaps. SIAM J. Comput. **15**(1) (1986) 52–69
19. Crane, C.A.: Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Dept. of Computer Science, Stanford University (1972)
20. Knuth, D.E.: The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)
21. Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. J. Comput. Syst. Sci. **7**(4) (1973) 448–461
22. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM **7**(6) (1964) 347–348