k-Maximum Subarrays for Small k: Divide-and-Conquer made simpler

Ovidiu Daescu and Hemant Malik
University of Texas at Dallas, Richardson TX 75080, USA
{daescu, malik}@utdallas.edu

Abstract. Given an array \( A \) of \( n \) real numbers, the maximum subarray problem is to find a contiguous subarray which has the largest sum. The \( k \)-maximum subarrays problem is to find \( k \) such subarrays with the largest sums. For the 1-maximum subarray the well known divide-and-conquer algorithm, presented in most textbooks, although suboptimal, is easy to implement and can be made optimal with a simple change that speeds up the combine phase. On the other hand, the only known divide-and-conquer algorithm for \( k > 1 \), that is efficient for small values of \( k \), is difficult to implement, due to the intricacies of the combine phase. In this paper we give a divide-and-conquer solution for the \( k \)-maximum subarray problem that simplifies the combine phase considerably while preserving the overall running time.

In the process of designing the combine phase of the algorithm we provide a simple, sublinear, \( O(\sqrt{k} \log^3 k) \) time algorithm, for finding the \( k \) largest sums of \( X + Y \), where \( X \) and \( Y \) are sorted arrays of size \( n \) and \( k \leq n^2 \). The \( k \) largest sums are implicitly represented, and can be enumerated with an additional \( O(k) \) time.

Unlike previous solutions, that are fairly complicated and sometimes difficult to implement, ours rely on simple operations such as merging sorted arrays, binary search, and selecting the \( k \)-th smallest number in an array. We have implemented our algorithms and report excellent performance as compared to previous results.

Keywords: K-Maximum Subarrays · Divide and Conquer · \( X + Y \) · Sublinear

1 Introduction

The well known problem of finding the maximum sum (contiguous) subarray of a given array of real numbers has been used in various applications and received a lot of attention over time. Some of the applications are in data mining \[10\][8], pattern recognition \[7\], and image processing and communication \[9\].

Given an array \( A \) of \( n \) real numbers and an integer \( k \), such that \( 1 \leq k \leq n(n + 1)/2 \), the \( k \)-maximum subarrays problem is to find \( k \) contiguous subarrays with the largest sums (not necessarily in sorted order of the sums). If \( k=1 \), Kadane’s algorithm \[1\] solves the maximum subarray problem in \( O(n) \) time.
using an iterative method. On the other hand, the well known divide and conquer algorithm [2], found in virtually all algorithms textbooks, has a suboptimal $O(n \log n)$ running time. An $O(n)$ time divide-and-conquer algorithm is briefly presented in [6].

For $k > 1$, Bengtsson and Chen [3] presented an algorithm that takes time $O(\min\{k+n \log^2 n, n\sqrt{k}\})$, where the second term, $O(n\sqrt{k})$, comes from a divide-and-conquer solution. That divide-and-conquer algorithm is difficult to implement, due to the intricacies of the combine phase.

In this paper we propose a competitive, $O(n\sqrt{k})$ time divide and conquer solution to find the $k$-maximum subarrays, which is optimal for $k = O(1)$ and $k = O(n^2)$. Our algorithm is simpler than the one in [3] due to a more direct way of performing the combine phase. Specifically, the combine phase we propose is itself a simple recursive procedure. To this end, we also address the following subproblem: Given two sorted arrays of real numbers, $X$ and $Y$, each of size $n$, let $S$ be the set $S = \{(x,y) \mid x \in X \text{ and } y \in Y\}$, with the value of each pair in the set defined as $\text{Val}(x,y) = x + y$. Find the $k$ pairs from $S$ with largest values. This problem is closely related to the famous pairwise sum $(X + Y)$ problem [11,12], that asks to sort all pairwise sums. Our main contribution is a sublinear, $O(\sqrt{k}\log^3 k)$ time algorithm, for finding the $k$ largest sums of $X + Y$. The $k$ largest sums are implicitly represented, and can be enumerated with an additional $O(k)$ time. A key feature of our solution is its simplicity, compared to previous algorithms [11,12], that find and report the $k$ largest sums in $O(k + \sqrt{k})$ time. Our algorithm is very simple, using only operations such as merging sorted arrays, binary search, and selecting the $k$-th largest number of an array.

We have implemented our main algorithms in JAVA and performed extensive experiments on macOS High Sierra with 3.1 GHz intel i5 processor and 8 GB of RAM, reporting excellent performance. For example, on random arrays of size $10^6$, with $k = 10^6$, we can find the $k$ maximum subarrays in about 52 seconds.

The rest of the paper is organized as follows. In Section 2 we discuss previous results. In Section 3 we describe in more details the linear time divide and conquer solution in [6], for the maximum subarray problem ($k = 1$). We then show how to extend this approach to find $k$-maximum subarrays in Section 4, and continue on to give our main, $O(n\sqrt{k})$ time divide-and-conquer algorithm, in Section 5. Section 5 also presents our $O(\sqrt{k}\log^3 k)$ time solution for finding the $k$ largest sums of $X + Y$. We report on implementation, experimental results and comparison with previous results in Section 6.

2 Previous Work

Bengtsson, and Chen [3] provided a complex, $O(\min\{k + n \log^2 n, n\sqrt{k}\})$ time algorithm to solve the $k$-maximum subarray problem. Their main algorithm, for general $k$, has five phases. First, the problem is reduced to finding the top $k$ maximum values over all the "good" elements in some matrix of size $n \times n$. In the second phase, repeated constraint searches are performed, which decrease the number of candidate elements to $O(\min\{kn, n^2\})$. In third phase, a range
A reduction procedure is performed to reduce the number of candidates further to $\theta(k)$. In the fourth phase, a worst-case linear-time selection algorithm is used on the remaining candidates, resulting in an element $x$, that is the $k$-th largest sum. The final phase involves finding the "good" elements with values not less than $x$. The $O(n\sqrt{k})$ part of the running time comes from a divide-and-conquer solution, and is useful for small values of $k$. The combine phase of the divide and conquer algorithm uses the $O(\sqrt{k})$ time algorithm from [12] to find the $k^{th}$ largest element in a sorted matrix, which is fairly difficult to understand and tedious to implement. A trivial lower bound for this problem is $O(n + k)$.

In the same year (2006), Bae and Takaoka [4] provided an $O((n + k) \log k)$ solution that reports the $k$ maximum subarrays in sorted order. Still in 2006, Cheng, Chen, Tien, and Chao [5] provided an algorithm with running time $O(n + k \log(\min\{n, k\}))$. The authors adapted an iterative strategy where the table of possible subarray sums is build partially after every iteration and the algorithm terminates in $O(\log n)$ iterations, which yields a time complexity of $O(n + k \log(\min\{n, k\}))$.

Finally, in 2007, Bengtsson and Chen [6] provided a solution that takes time $O(n + k \log n)$ in the worst case to compute and rank all $k$ subsequences. They also proved that their algorithm is optimal for $k = O(n)$, by providing a matching lower bound. Their approach is different from the previous ones. In particular, although only briefly described, their solution provides an $O(n)$ time algorithm for the maximum subarray ($k=1$) problem. They give a tree based algorithm that uses a full binary tree, augmented with information about prefix sums, suffix sums, sums, and ranking among subsequences with respect to their sums. There are two phases of this algorithm. In the first phase, initial information (prefix sum, suffix sum, sum, largest elements) is computed and stored in the tree. The tree is constructed in a bottom-up fashion. The algorithm is based on the well known observation that the maximum sum can be obtained from the left branch or the right branch, or from a subsequence spanning over the left and right branches (subarrays). The second phase is the query phase which uses a binary heap to compute the $k$-maximum subarrays. A special property of this algorithm is that if $l$ largest sums are already computed then the $(l + 1)$-th largest sum can be found in $O(\log n)$ time.

Frederickson and Johnson [12] provided an efficient algorithm to find the $k^{th}$ maximum element of a matrix with sorted rows and columns. When the sorted matrix has $k$ rows and $k$ columns, their algorithm finds the $k^{th}$ largest element in $O(\sqrt{k})$ time. It can then be used to find and report the $k$ largest elements in the matrix in an additional $O(k)$ time. This corresponds to finding and reporting the $k$ largest values of $X + Y$. The algorithm is not simple, and is tedious to implement.
In this section we give a detailed description of a simple, linear time divide-
and-conquer algorithm to find the maximum subarray \((k = 1)\), by placing
the algorithm in [6] in a standard divide-and-conquer framework.

Given an array \(A\) of \(n\) real numbers, the maximum subarray problem is to
find a contiguous subarray whose sum of elements is maximum over all possible
subarrays, including \(A\) itself. The divide and conquer algorithm divides \(A\)
to two subarrays of equal size, makes two recursive calls, and then proceeds with
the combine step while keeping track of the maximum subarray sum found in
the process.

In the combine phase, at an internal node, we have two subarrays, \(A_1\) (from
left child) and \(A_2\) (from right child). We define the following variables which are
used to find the maximum subarray (see also Figure 1):

| Variable          | Description                                      |
|-------------------|--------------------------------------------------|
| max_left          | maximum subarray starting from left most index    |
| max_right         | maximum subarray starting from right most index   |
| sum               | sum of all elements in array                      |
| max_cross         | maximum crossing subarray                         |
| max_sub           | maximum subarray                                  |

The idea is to make the combine phase run in \(O(1)\) time instead of the \(O(n)\)
time, as described in [2]. For that, the values (and corresponding array indexes)
of \(max\_left\), \(max\_right\), and \(sum\) must also be passed up from the recursive
calls. The \(sum\) value at a given node can be found by adding up the sums from
the children. The value \(max\_left\) is either the \(max\_left\) from the left child or the
\(sum\) value from the left plus the \(max\_left\) value from the right child. Similarly,
the value \(max\_right\) is either the \(max\_right\) from the right child or the \(sum\)
value from the right plus the \(max\_right\) value from the left child. The following
divide and conquer algorithm, \texttt{Maximum\_Subarray}, takes in the input an array
\(A\) of size \(n\) and two integers, \(low\) and \(high\), which correspond to the start index
and end index of subarray \(A[low...high]\), and finds and returns the maximum
subarray of \(A[low, high]\).
Algorithm 1. Maximum_Subarray (A, low, high)

1. if (low == high)
2. 
3. max_left = A[low];
4. max_right = A[low];
5. sum = A[low];
6. max_sub = A[low];
7. return (max_left, max_right, sum, max_sub)
8. mid = ⌊(low+high)/2⌋
9. (max_left1, max_right1, sum1, max_sub1) = Maximum_Subarray(A, low, mid)
10. (max_left2, max_right2, sum2, max_sub2) = Maximum_Subarray(A, mid+1, high)
11. max_left = max(max_left1, sum1 + max_left2);
12. max_right = max(max_right2, sum2 + max_right1)
13. sum = sum1 + sum2
14. max_cross = max(max_right1 + max_left2
15. max_sub = max(max_cross, max_sub1, max_sub2)
16. return (max_left, max_right, sum, max_sub)

In above algorithm, steps 1-7 take O(1) time. Steps 8-9 correspond to the recursive calls. Steps 10-15 take O(1) time. Therefore, the time taken by Algorithm 1 is: T(n) = 2T(n/2) + O(1) = O(n).

In the next section, we extend this algorithm to find the k-maximum subarrays.

4 k-maximum subarrays by divide and conquer

Given an array A of n numbers and an integer k, where 1 ≤ k ≤ n(n+1)/2, the k-maximum subarrays problem is to find the k contiguous subarrays with the largest sums. In this section we address the k-maximum subarrays problem and provide a warm-up divide and conquer solution.

We extend the approach in Section 3, used to find the maximum subarray. We recursively divide the array into two subarrays with equal elements until we reach a base case (of size √k), and then perform the combine step. The main difference, and implied difficulty, is that recursive calls return information about k largest subarrays from corresponding subproblems, including k largest sums from the left and from the right, and we are finding k largest subarray values in the combine step. A detailed description of the generic algorithm is given in [3]. Our main goal is to simplify the combine phase.

In the remaining of the paper, notations like max_left, max_right, max_cross, and max_sub refer to arrays of size k, holding the corresponding k largest sum values. Except for max_cross, these arrays are sorted in non-increasing order.

Consider the left and right subarrays, Ar and Ar, of some internal node v in the recursion tree. The k largest sums at v are among the k largest sums from Ar, the k largest sums from Ar, and the k largest sums of contiguous subarrays that cross between Ar and Ar (we call these last sums crossing sums). The difficult
part is to efficiently compute the $k$ crossing sums and the various $k$ largest sums that need to be passed up to the parent node.

In this section we provide a solution for the combine step that is simple yet efficient, easy to implement, and sets up the stage for our better solution in the following section.

The function $\text{MERGE}$ used in the following algorithms is similar to the one in the $\text{merge-sort}$ sorting algorithm, except that we stop after finding the largest $k$ values, and takes $O(k)$ time. By a slight abuse, we allow the $\text{MERGE}$ function to work with a constant number of arrays in the input, rather than just two arrays. If there are more than two arrays passed to the $\text{MERGE}$ function we perform pairwise merge to find the $k$ largest numbers. Similarly, function $\text{SELECT}$, whenever mentioned, is the standard linear time selection function [13], that finds the $k$-th (and thus $k$) largest number(s) of a given set of $(O(k)$ in our case) numbers.

The function $\text{MAX\_SUM}(a, A)$ used below takes in the input an array $A$ of size $k$ and an integer $a$ and adds $a$ to each entry of $A$. It is used to add the value of the sum of elements of the left (or right) child of $v$ to the $k$ largest sum prefix (suffix) values of the right (resp., left) child of $v$.

The following procedure takes as input two arrays, $A$ and $B$, each of size $k$, sorted in non-increasing order, and outputs the $k$-maximum sums of the pairwise addition of $A$ and $B$. For our purpose, $A$ would contain the $k$ largest sums of $A_l$ for subarrays starting at the rightmost entry of $A_l$, while $B$ would contain the $k$ largest sums of $A_r$ for subarrays starting at the leftmost entry of $A_r$. We use a priority queue $Q$ implemented as a binary heap to store pairwise sums, as they are generated. An AVL tree $T$ is also used, to avoid placing duplicate pairs $(i,j)$ in $Q$.

**Algorithm 2. MAX\_SUM\_CROSS (A, B)**

1. $k = \text{sizeof}(A)$
2. $Q \leftarrow \text{null}$ //Max Priority Queue
3. $M[k] \leftarrow \text{null}$; //Output Array
4. $T \leftarrow \text{null}$; //AVL Tree
5. $m \leftarrow 0$;
6. add $(0, 0)$ to $Q$ with priority $A[0] + B[0]$
7. store $(0, 0)$ in $T$
8. while $k > 0$
9.   $(i, j) = \text{pop} Q$
10.  $M[m] = A[i] + B[j]$
11.  $m = m + 1$; $k = k - 1$
12.  if $(i < k$ and $(i + 1, j) \not\in T)$
13.     store $(i+1, j)$ in $T$
14.  add $(i + 1, j)$ to $Q$ with priority $(A[i + 1] + B[j])$
15.  if $(j < k$ and $(i, j + 1) \not\in T)$
16.     store $(i, j + 1)$ in $T$
17.  add $(i, j + 1)$ to $Q$ with priority $(A[i] + B[j + 1])$
18. return $M$
Time Complexity of Algorithm MAX\_SUM\_CROSS: Lines 12, 15 take \(O(\log k)\) time for searching \(T\), lines 13, 16 take \(O(\log k)\) time to store indices in \(T\), lines 9, 14, 17 take \(O(\log k)\) to add or remove an element in the priority queue, and the while loop in line 8 runs \(k\) times. Therefore, the time complexity for algorithm MAX\_SUM\_CROSS is \(O(k \log k)\).

The Max-k algorithm below computes the \(k\)-maximum sums (subarrays) of the given array \(A\). The values \(low\) and \(high\) correspond to the start index and end index of the subarray \([low...high]\).

Algorithm 3. Max-k (A, low, high)

1. if \((low + \sqrt{k} \geq high)\) then find \(max\_left, max\_right, sum, max\_sub\) by brute force and return \((max\_left, max\_right, sum, max\_sub)\)
2. mid = \(\lfloor \frac{low + high}{2} \rfloor\)
3. \((max\_left1, max\_right1, sum1, max\_sub1) = Max-k (A, low, mid)\)
4. \((max\_left2, max\_right2, sum2, max\_sub2) = Max-k (A, mid + 1, high)\)
5. \(max\_left = \text{MERGE}(max\_left1, MAX\_SUM (sum1, max\_left2))\)
6. \(max\_right = \text{MERGE}(max\_right2, MAX\_SUM (sum2, max\_right1))\)
7. \(sum = sum1 + sum2\)
8. \(max\_cross = MAX\_SUM\_CROSS(max\_right1, max\_left2)\)
9. \(max\_sub = \text{MERGE}(max\_cross, max\_sub1, max\_sub2)\)
10. return \((max\_left, max\_right, sum, max\_sub)\)

Running time of Max-k: It can be easily seen that the running time of algorithm Max-k is described by the recurrence:

\[
T(n, k) = 2T(n/2, k) + O(k \log k), \text{ with } T(\sqrt{k}, k) = O(k).
\]

Using substitution method, we have \(T(n, k) = 2^i T(n/2^i, k) + O(\sum_{j=0}^{i-1} 2^j k \log k)\). Letting \((n/2^i)^2 = k\) results in \(2^i = n/\sqrt{k}\).

\[
T(n, k) = (n/\sqrt{k}) T(\sqrt{k}, k) + O(n \sqrt{k} \log k).
\]

Since \(T(\sqrt{k}, k) = O(k)\) the overall time complexity is \(O(n \sqrt{k} \log k)\). The algorithm is an \(O(k \log k)\) factor slower than the one in \([3]\), while being very simple to describe and implement.

In the next section we provide a simple, \(O(k)\) time prune-and-search algorithm for the MAX\_SUM\_CROSS procedure, which improves the overall time complexity of algorithm Max-k to \(O(n \sqrt{k})\).

5 An improved algorithm for \(k\)-maximum subarrays

In this section, we improve the results in the previous section by providing an \(O(k)\) time divide-and-conquer solution for the combine phase. To this end, we first find the \(k\)-th largest element \(x\) of the pairwise sum \(A + B\) [1112], and then scan \(A\) and \(B\) for elements in \(A + B\) greater than or equal to \(x\). If this output would be sorted then it will again lead to an \(O(k \log k)\) running time. However, as explained later in this section, there is no need to sort these elements, that correspond to the values of the crossing sums.

Frederickson and Johnson [12] provided an algorithm that can find the \(k\)-th maximum element of a matrix consisting of \(k\) rows and \(k\) columns, each sorted
in nonincreasing order, in \(O(\sqrt{k})\) time. Given a sorted matrix \(M\), the algorithm extracts a set \(S\) of submatrices of different shapes which guarantee to contain all elements greater than or equal to the \(k^{th}\) largest element of \(M\). Note that they also contain elements which are less than the \(k^{th}\) largest element. After extracting the set of submatrices, it forms a new matrix with the help of dummy matrices (matrices where all entries are \(-\infty\)). The new matrix is also sorted. The submatrices are referred to as cells, and for each cell \(C\), \(\min(C)\) and \(\max(C)\) represent the smallest and largest elements in this cell. Initially, there is a single cell which is the matrix formed from dummy matrices and the set \(S\). After each iteration, a cell is divided into four subcells. From all the subcells formed the algorithm computes some values that allow to discard a few cells guaranteed not to contain the \(k^{th}\) largest element. The algorithm is not simple to follow, and tedious to implement. Also the construction of sorted matrix from set of sorted submatrices and dummy matrices is not easy to understand and implement.

Intuitively, for our problem, the rows and columns of the matrix are generated by the sums in \(A + B\), where \(A\) and \(B\) are sorted arrays of size \(k\) each. In row \(i\), \(A[i]\) is summed over the entries in array \(B\). Similarly, in column \(j\), \(B[j]\) is summed over the entries in array \(A\). The matrix does not have to be explicitly stored as the matrix entries can be generated as needed from the values in \(A\) and \(B\). Thus, using the algorithm in [12] one can compute the \(k^{th}\) maximum element \(x\) of \(A + B\) in \(O(\sqrt{k})\) time. Retrieving the elements of \(A + B\) that are greater than (or equal) to \(x\) takes an additional \(O(k)\) time. This makes the algorithm \textsc{Max-Sum-Cross} in previous section run in \(O(k)\) time. Since the \(k\) largest crossing sum values are no longer sorted, we replace the \textsc{Merge} call in line 9 of algorithm \textsc{Max-k} with a \textsc{Select} call. The algorithm, as presented above, has been described in [3].

The only place where the \(k\) largest crossing sum values are used at an internal node \(u\) of the recursion tree is in the calculation of the \(k\) largest sum values at \(u\), given the \(k\) largest sum values from the left and the right children of \(u\). Let \(v\) be the parent node of \(u\). Node \(u\) needs to pass up to \(v\) the \(k\) largest sum values of subarrays that start at the leftmost entry, and the \(k\) largest sum values of subarrays that start at the rightmost entry (\textit{max-left} and \textit{max-right} arrays at \(u\)) and these subarrays are either distinct from the crossing subarrays at \(u\) or computed independently of those subarrays by function \textsc{Max-Sum}. See Figure 1.

The following lemma is implicitly used in [3].

**Lemma 1.** The \(k\) largest crossing sum values do not need to be sorted for algorithm \textsc{Max-k} to correctly report the \(k\) largest sum values of \(A\).

Then, the running time of the \textsc{Max-k} algorithm is now described by the recurrence \(T(n, k) = 2T(n/2, k) + O(k)\), with \(T(\sqrt{k}, k) = O(k)\) As described in [3], using the substitution method, we have \(T(n, k) = 2^iT(n/2^i, k) + O(\sum_{j=0}^{i-1} k2^j)\) Letting \((n/2^i)^2 = k\) results in \(2^i = n/\sqrt{k}\).

\[ T(n, k) = (n/\sqrt{k})T(\sqrt{k}, k) + O(n\sqrt{k}). \] Thus, \(T(n) = O(n\sqrt{k})\).
As mentioned earlier, the algorithm for finding the $k$-th largest entry in $A + B$, as presented in [12], is complex and tedious to implement. In what follows, we provide a simple algorithm to find the $k$-th largest element in $A + B$, which takes $O(\sqrt{k} \log^3 k)$ time and is easy to implement. Moreover, unlike the algorithm in [12], our algorithm is a simple prune-and-search procedure. Also, unlike in [12], our algorithm implicitly finds the $k$ largest elements of $A + B$ in the process. The total time needed to report all $k$ largest elements in $A + B$ is then $O(k + \sqrt{k} \log^3 k)$ which is still $O(k)$, and thus the final time complexity of algorithm Max-k remains $O(n \sqrt{k})$.

Let $A$ and $B$ be arrays of size $n$ and let $k$ be an integer such that $k \leq n^2$. We now show how to find the $k$th-largest element of $A + B$ and an implicit representation of the $k$ largest elements of $A + B$ in sublinear, $O(\sqrt{k} \log^3 k)$ time.

Consider a matrix $M$ with $n$ rows and $n$ columns, such that the element in matrix entry $M[i, j]$ is the sum $A[i] + B[j]$. We have $\sqrt{k} \leq n$. It is easy to observe that all rows and columns in matrix $M$ are sorted and the $k$ largest values will not lie in the submatrix $M[\sqrt{k}, \sqrt{k}; \min\{n, k\}, \min\{n, k\}]$, which is shaded in Figure 2. Without loss of generality, assume that $k \leq n$. We call $M$ a sorted matrix. Note that for the $k$-maximum subarray problem, matrix $M$ is of size $k \times k$.

Matrix $M$ is only considered for better understanding of the algorithms presented in this section, but there is no need to store it explicitly. Instead, its entries are computed only as needed. The only information required is the start and end index of each row $\in [1, \sqrt{k}]$ and each column $\in [1, \sqrt{k}]$, that define the "active" entries of $M$ at a given step. Whenever a matrix is passed in a function, we are passing arrays, storing the start and end indexes of these rows and columns.

A staircase matrix $M_S$ is a subset of adjacent rows and columns of $M$, where each row and each column are described by a start and an end index. For a row (column) $i + 1$, its start index is no larger than the start index of row (column) $i$ and its end index is no larger than the end index of row (column) $i$.

Let max_cross be an array of size $k$. Initially, all entries of max_cross are set to minus infinity. Let $p \leq \sqrt{k}$ and $m, r \leq k$ be positive integers.
The notation $M[i][0 : j]$ denotes the entries in row $i$ of the matrix $M$, columns 0 to $j$. The notation $M[0 : i][j]$ denotes the entries in column $j$ of matrix $M$, rows 0 to $i$.

Given matrix $M$ and a pair $(m, r)$ such that $A[m] + B[r] \geq M[\sqrt{k}, \sqrt{k}]$, the following algorithm will find and return:

(i) a staircase matrix of $M$ where all elements are greater than or equal to $M[m, r]$

(ii) the total number $T$ of elements in the staircase matrix.

**Algorithm 4. STAIRCASE(M, m, r, p)**

1. $x = M[m, r]$
2. for each row $i \in [1, p]$ of $M$, use binary search to find the maximum index $\alpha_i$ such that $M[i][0 : \alpha_i] \geq x$.
3. for each column $j \in [1, p]$ of $M$, use binary search to find the maximum index $\beta_j$ such that $M[0 : \beta_j][j] \geq x$.
4. Let $M_S$ be the (implicitly defined, staircase) submatrix of $M$ formed by elements larger or equal than $x$ found in step 2 and step 3.
5. Let $T$ be the total number of elements in $M_S$
6. return $M_S, T$

In algorithm **STAIRCASE**, binary search on each row or column requires $O(\log k)$ time. There are $p$ rows and $p$ columns, and $p \leq \sqrt{k}$. The staircase matrix $M_S$ is defined implicitly, by start-end pairs for rows and columns. Therefore the total time of algorithm **STAIRCASE** is $O(p \log k)$.

Notice that we need to pay attention to not double count the entries in $M[0 : p][0 : p]$, which can be easily done in $O(p)$ time.

**Fig. 3.** (a) Two staircase matrices $M_1$ and $M_2$. Matrix $M_1$ has total number of elements less than or equal to $k$ while matrix $M_2$ has more than $k$ elements. (b) Illustrating $M_2 \setminus M_1$

The idea behind the following algorithm is to find two consecutive diagonal index $i$ and $i+1$ of matrix $M$ such that the staircase matrix computed by **STAIRCASE**(M, $i$, $i$, $i$) contain only elements which are greater than or equal to the $k^{th}$ largest element of matrix $M$, while elements which are not part of the
staircase matrix \(\text{STAIRCASE}(M, i + 1, i + 1, i + 1)\) are guaranteed to be less than the \(k\)th largest element of \(M\). It is easy to notice that the \(k\)th largest element lies in the subtraction of the matrices \(\text{STAIRCASE}(M, i + 1, i + 1, i + 1)\) and \(\text{STAIRCASE}(M, i, i, i)\).

The following algorithm takes as input a sorted matrix \(M\) and computes a staircase submatrix of \(M\) containing the \(k\) largest entries in \(M\). As we will see, it does that in \(O(p \log^3 k)\) time, using an implicit representation of submatrices of \(M\). The \(k\) largest entries can then be reported in an additional \(O(k)\) time.

**Algorithm 5. MAX\_SUM\_CROSS-1(M)**

1. Use binary search on 1, 2, ..., \(\sqrt{k}\) to find index \(i\) such that the total number of elements returned by \(\text{STAIRCASE}(M, i, i, i)\) is at most \(k\) and the total number of elements returned by \(\text{STAIRCASE}(M, i + 1, i + 1, i + 1)\) is greater than \(k\) (This binary search on the diagonal of matrix \(M\) is illustrated in Figure 3-a).
   - (a) Let \(M_1, T_1 = \text{STAIRCASE}(M, i, i, i)\).
   - (b) Let \(M_2, T_2 = \text{STAIRCASE}(M, i + 1, i + 1, i + 1)\)
2. if \(T_1 = k\)
3. return \(M_1\)
4. totalElementsLeft = \(k - T_1\) //elements to be found in \(M_2 \setminus M_1\)
5. \(M_S = M_2 \setminus M_1\)
6. \(M_{\text{final}} = \text{FIND\_INDEX}(M_S, \text{totalElementsLeft}, i+1)\)
7. return \(M_{\text{final}}\)

In algorithm **MAX\_SUM\_CROSS-1**, \(M_2 \setminus M_1\) corresponds to the staircase matrix formed by deleting elements of matrix \(M_1\) from \(M_2\). Step 1 requires \(O(p \log^2 k)\) and finds the tuples \((M_1, T_1)\) and \((M_2, T_2)\). In step 5, we store \(O(p)\) indexed pairs into \(M\), which takes \(O(p)\) time. Let \(\Gamma\) be the running time for algorithm **FIND\_INDEX** (step 6). The total time taken by algorithm **MAX\_SUM\_CROSS-1** is then \(O(\max\{p \log^2 k, \Gamma\})\).

Let \(M_S\) be the staircase matrix which corresponds to \(M_2 \setminus M_1\). \(M_S\) is implicitly defined and stored. For row \(i\) of \(M_S\), let the lowest index be \(i_l\), and the highest index be \(i_h\). The median value of each row can be found in constant time, at entry \((i_l + i_h)/2\). A similar notation is used to denote the lowest and the highest entries of a column of \(M_S\).

Algorithm **FIND\_INDEX** below takes as input \(M_S\), an integer which stores the rank of the element we need to find in \(M_S\), and an index \(p\) useful for computing staircase matrices, and returns the \(k\)th largest elements of matrix \(M\) in an implicit representation.

**Algorithm 6. FIND\_INDEX(**\(M_S, \text{totalElementsLeft}, p\)**)

1. Find the median value in each row 1 to \(p\) and in each column 1 to \(p\) of \(M_S\) and place them into an array \(X\).
2. Sort \(X\) in non-increasing order. For element \(x_i\) at \(i\)-th position in array \(X\), let \(\alpha_i\) be the total number of elements of \(M_S\) greater than or equal to \(x_i\), and let \(\beta_i\) be the total number of elements of \(M_S\) strictly greater than \(x_i\).
3. Use binary search on $X$ together with the STAIRCASE function to find the maximum index $i$ and the minimum index $j$ in array $X$ such that $totalElementsLeft - \alpha_i \geq 0$ and $totalElementsLeft - \alpha_j < 0$. Find corresponding $\beta_i$ and $\beta_j$. Notice that $j = i + 1$. When searching, the last argument passed to the STAIRCASE function is $p$, while $m$ and $r$ correspond to the row and column in $M_S$ for the current search value in $X$.

4. if $\exists i, j$ in step 3,
   (a) if $totalElementsLeft - \alpha_i = 0$
       i. Return staircase matrix of $M$ with smallest element as $x_i$
   (b) else if $totalElementsLeft - \beta_j = 0$
       i. Return staircase matrix of $M$ with all elements greater than $x_j$
   (c) else if $totalElementsLeft - \beta_j < 0$
       i. Let $M'$ be the staircase matrix obtained by removing all elements greater or equal to $x_i$ and less than or equal to $x_j$ from $M$
       ii. $totalElementsLeft = totalElementsLeft - \alpha_i$
       iii. FIND_INDEX($M'$, $totalElementsLeft$, $p$)
   (d) else if $totalElementsLeft - \beta_j > 0$
       i. $totalElementsLeft = totalElementsLeft - \beta_j$
       ii. Return staircase matrix of $M$ with all element greater than $x_j$ and $totalElementsLeft$ number of element equal to $x_j$

5. else if $\exists j$ and $\not{\exists} i$
   (a) if $totalElementsLeft - \beta_j = 0$
       i. Return staircase matrix of $M$ with all element greater than $x_j$
   (b) else if $totalElementsLeft - \beta_j < 0$
       i. Let $M'$ be the staircase matrix obtained by removing all elements less than or equal to $x_j$ from $M_S$
       ii. FIND_INDEX($M'$, $totalElementsLeft$, $p$)
   (c) else if $totalElementsLeft - \beta_j > 0$
       i. $totalElementsLeft = totalElementsLeft - \beta_j$
       ii. Return staircase matrix of $M$ with all element greater than $x_j$ and $totalElementsLeft$ number of element equal to $x_j$

6. else if $\exists i$ and $\not{\exists} j$
   (a) if $totalElementsLeft - \alpha_i = 0$
       i. Return staircase matrix of $M$ with smallest element as $x_i$
   (b) else
       i. Let $M'$ be the staircase matrix obtained by removing all elements greater than or equal to $x_i$ from $M_S$
       ii. $totalElementsLeft = totalElementsLeft - \alpha_i$
       iii. FIND_INDEX($M'$, $totalElementsLeft$, $p$)

We now analyze the running time of algorithm FIND_INDEX. Step 1 requires $O(p)$ time, Step 2 requires $O(p \log k)$ time, and Step 3 requires $O(p \log^2 k)$ time. Computing the staircase matrix and the total elements in steps 4, 5, and 6 requires $O(p \log k)$ time. In steps 4, 5, and 6, half of elements are removed before function FIND_INDEX is called recursively. With $p = O(\sqrt{k})$, the time complexity is then described by $T(k, p) = T(k/2, p) + p \log^2 k$, with $T(1, p) = O(1)$.
Using substitution method, we have \( T(k, p) = T(k/2^i, p) + O(\sum_{j=0}^{i-1} p \log^2(k/2^j)) \)

Letting \( k/2^i = 1 \) results in \( i = \log k \) and
\[
T(k, p) = T(1, p) + O(\sum_{j=1}^{\log k - 1} p \log^2(k/2^j))
\]
which solves for \( T(k, p) = O(p \log^3 k) \). With \( T = O(\log^3 k) \), algorithm \textsc{MAX\_SUM\_CROSS-1} thus takes \( O(p \log^3 k) \) time where \( p = O(\sqrt{k}) \).

Let's summarize our result below.

**Theorem 1.** Algorithm \textsc{MAX\_SUM\_CROSS-1} finds the \( k \) largest elements in \( A + B \) (and thus the \( k \) largest crossing sums) in \( O(k \log^3 k) \) time. The \( k \) sums are implicitly represented and can be report with an additional \( O(k) \) time.

Recall that there is no need to sort the elements in the \textsc{max-cross} since the arrays contributing in the combine step are \textsc{max-left} and \textsc{max-right}. Therefore, instead of using \textsc{MERGE}, we can use the \textsc{SELECT} algorithm in line 9 of algorithm \textsc{Max-}\( k \) to find \( k \) \textsc{max-sub} in \( O(k) \) time.

Plugging in the new procedure for finding crossing sums, \textsc{MAX\_SUM\_CROSS-1}, the running time of the divide and conquer algorithm \textsc{Max-}\( k \) is now described by the recurrence \( T(n, k) = 2T(n/2, k) + O(k) \), with \( T(\sqrt{k}, k) = O(k) \) which solves for \( T(n) = O(n \sqrt{k}) \).

**Theorem 2.** Algorithm \textsc{MAX-}\( k \) finds the \( k \) largest subarrays of an array \( A \) of size \( n \) in \( O(n \sqrt{k}) \) time, where \( 1 \leq k \leq n(n + 1)/2 \).

## 6 Implementation and Experiments

We have implemented our algorithms and performed multiple experiments, reporting excellent results. For comparison, we also implemented the algorithm presented in [12] which is used in the combine phase in [3]. The implementation is in JAVA on macOS High Sierra with 3.1 GHz intel i5 processor and 8 GB of RAM, while the data sets have been randomly generated.

For experimentation, we generated two random sorted arrays of size \( n \) which consist of integer values and defined an integer \( k \). For comparison both \( n \) and \( k \) are power of 4 as assumed in [12]. Table 1 shows comparison results between the two algorithms. It is clear that our algorithm outperformed. The time complexity of our algorithm depends upon the value of \( p \leq \sqrt{k} \). At each step of our algorithm, we eliminate many elements which are not candidates for the \( k \)th largest element. For example, while computing a staircase matrix we have knowledge about each row and column (elements in submatrix \( M[1, m; 1, r] \) are greater than or equal to \( M[m, r] \)).

For better assessing our algorithm, we run more experiments for general values of \( n \) and \( k \), where we variate the size of the input array from \( 10^1 \) to \( 10^6 \). For each size, we generated \( 10^4 \) test cases, and computed the average running time over \( 10^2, 10^3 \) and \( 10^4 \) test cases. For each input, the value of \( k \) is set to the size of the array \( (k = n) \) for an input of size \( n \) which is case during combine phase where we have arrays of size \( k \) and we need to find \( k \) largest elements. Results of runs of the \textsc{MAX\_SUM\_CROSS-1} procedure are shown in Table 2. As it
Table 1. Comparison between our algorithm and [12] \((k=n)\).

| Size of input array | Average time for 10\(^2\) test cases (in milliseconds) | Algorithm in [12] | Our Algorithm |
|---------------------|------------------------------------------------------|-------------------|---------------|
| 4\(^4\)             | 0.18                                                 | 0.03              |
| 4\(^5\)             | 0.91                                                 | 0.12              |
| 4\(^6\)             | 1.2                                                  | 0.14              |
| 4\(^7\)             | 2.6                                                  | 0.17              |

Table 2. Average time taken to find the \(k\)-maximum values of \(A + B\) \((k=n)\).

| Size of input array | Average Time (in milliseconds) |
|---------------------|--------------------------------|
|                     | Number of tests = 10\(^2\)  | Number of tests = 10\(^3\) | Number of tests = 10\(^4\) |
| 10                  | 0.03                          | 0.01                      | 0.02                      |
| 10\(^2\)            | 0.07                          | 0.03                      | 0.04                      |
| 10\(^3\)            | 0.18                          | 0.08                      | 0.09                      |
| 10\(^4\)            | 0.37                          | 0.23                      | 0.27                      |
| 10\(^5\)            | 1.35                          | 1.44                      | 1.39                      |
| 10\(^6\)            | 16.21                         | 15.78                     | 15.85                     |

Table 3. Average time taken to find the \(k\)-maximum subarrays when \(k=n\).

| Size of input array | Average Time (in milliseconds) |
|---------------------|--------------------------------|
|                     | Number of tests = 10\(^2\)  | Number of tests = 10\(^3\) | Number of tests = 10\(^4\) |
| 10                  | 0.08                          | 0.03                      | 0.01                      |
| 10\(^2\)            | 0.44                          | 0.363                     | 0.10                      |
| 10\(^3\)            | 1.19                          | 3.63                      | 1.60                      |
| 10\(^4\)            | 91.28                         | 98.56                     | 93.43                     |
| 10\(^5\)            | 1729.78                       | 1890.03                   | 1780.13                   |
| 10\(^6\)            | 52101.62                      | -                         | -                         |

Table 4. Average time taken to find the \(k\)-maximum subarrays for small \(k\).

| Size of input array | Average Time (in milliseconds) |
|---------------------|--------------------------------|
|                     | k=5                            | k=15                       | k=25                       | k=35                       | k=45                       |
| 10                  | 0.07                           | 0.04                       | 0.02                       | 0.03                       | 0.02                       |
| 10\(^2\)            | 0.31                           | 0.33                       | 0.24                       | 0.31                       | 0.28                       |
| 10\(^3\)            | 0.81                           | 2.03                       | 1.45                       | 1.21                       | 0.88                       |
| 10\(^4\)            | 4.03                           | 4.78                       | 6.2                        | 5.39                       | 5.02                       |
| 10\(^5\)            | 34.63                          | 45.11                      | 46.33                      | 51.58                      | 46.94                      |

Table 5. Average time taken to find the \(k\)-maximum subarrays.

| Size of input array | Average Time (in milliseconds) |
|---------------------|--------------------------------|
|                     | k=105                          | k=205                      | k=405                      | k=605                      | k=805                      | k=1005                     |
| 10\(^4\)            | 0.54                           | 0.4                        | 0.69                       | 0.24                       | 0.31                       | 0.93                       |
| 10\(^5\)            | 2.52                           | 1.63                       | 2.06                       | 1.61                       | 1.39                       | 2.68                       |
| 10\(^6\)            | 10.29                          | 8.4                        | 9.96                       | 14.24                      | 13.65                      | 15.16                      |
| 10\(^7\)            | 71.06                          | 76.49                      | 114.97                     | 123.29                     | 160.73                     | 192.22                     |
can be seen, even for very large $n$, our algorithm takes only a few milliseconds. Results of runs of the overall MAX-$k$ divide and conquer algorithm are shown in Table 3 to Table 5. As it can be seen, our algorithm is very fast: for arrays of size $10^6$, with $k = 10^6$, we can find the $k$ maximum subarrays in about 52 seconds.

7 Conclusion

In this paper, we studied the $k$-maximum subarray problem and proposed a simple divide-and-conquer algorithm for small values of $k$. Our algorithm matches the best known divide-and-conquer algorithm, while considerably simplifying the combine step. As part of our solution, we provided a simple prune-and-search procedure for finding the largest $k$ values of $X + Y$, where $X$ and $Y$ are sorted arrays of size $n$ each. These values are computed and stored implicitly in $O(\sqrt{k} \log^2 k)$ time, and can be reported in additional $O(k)$ time. Our solutions benefit from simplicity and very fast execution time, even for large values of $n$ and $k$. We implemented our algorithms and reported excellent results.

References

1. Bentley, J. "Algorithm design techniques." Communications of the ACM 27.9 (1984): 865-871.
2. Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2009.
3. Bengtsson, Fredrik, and Jingsen Chen. "Efficient algorithms for k maximum sums." Algorithmica 46.1 (2006): 27-41.
4. Bae, S. E., & Takaoka, T. (2006). Improved algorithms for the k-maximum subarray problem. The Computer Journal, 49(3), 358-374.
5. Cheng, Chih-Huai, et al. "Improved algorithms for the k maximum-sums problems." Theoretical Computer Science 362.1-3 (2006): 162-170.
6. Bengtsson, Fredrik, and Jingsen Chen. "Ranking k maximum sums." Theoretical Computer Science 377.1-3 (2007): 229-237.
7. Grenander, Ulf. Lectures in Pattern Theory: Volume 2: Pattern Analysis. Vol. 24. Springer Science & Business Media, 2012.
8. Fukuda, Takeshi, et al. "Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization." ACM SIGMOD Record 25.2 (1996): 13-23.
9. Bentley, Jon. "Programming pearls: algorithm design techniques." Communications of the ACM 27.9 (1984): 865-873.
10. Agrawal, Rakesh, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases." Acm sigmod record. Vol. 22. No. 2. ACM, 1993.
11. Frederickson, Greg N., and Donald B. Johnson. "The complexity of selection and ranking in $X+Y$ and matrices with sorted columns." Journal of Computer and System Sciences 24.2 (1982): 197-208.
12. Frederickson, Greg N., and Donald B. Johnson. "Generalized selection and ranking: sorted matrices." SIAM Journal on computing 13.1 (1984): 14-30.
13. Blum, Manuel, et al. "Time bounds for selection." J. Comput. Syst. Sci. 7.4 (1973): 448-461.