# Improved Algorithms for the $K$-Maximum Subarray Problem for Small $K$

Sung E. Bae and Tadao Takaoka

Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand
{seb43,tad}@cosc.canterbury.ac.nz

**Abstract.** The maximum subarray problem for a one- or two-dimensional array is to find the array portion that maiximizes the sum of array elements in it. The $K$-maximum subarray problem is to find the $K$ subarrays with largest sums. We improve the time complexity for the one-dimensional case from $O(min\{K + n\log^2 n, n\sqrt{K}\})$ for $0 \leq K \leq n(n-1)/2$ to $O(n \log K + K^2)$ for $K \leq n$. The latter is better when $K \leq \sqrt{n}\log n$. If we simply extend this result to the two-dimensional case, we will have the complexity of $O(n^3 \log K + K^2 n^2)$. We improve this complexity to $O(n^3)$ for $K \leq \sqrt{n}$.

## 1 Introduction

The maximum subarray problem was first described by Bentley in his literature *Programming Pearls* [4, 5] 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 data set 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 $O(m^2 n)$ time for an $m \times n$ array ($m \leq n$), which is cubic when $m = n$ [4, 5]. In this paper, if only $n$ appears in complexities for the two-dimensional case, we assume $m = n$. The sub-cubic time solution based on Takaoka's sub-cubic *distance matrix multiplication algorithm* [14] is given by Tamaki and Tokuyama [17], which is further simplified by Takaoka [15]. In the context of parallel computations, time and cost optimal PRAM and mesh algorithms for the one-dimensional case are described in [10]. For the two-dimensional case, EREW PRAM solutions achieving $O(\log n)$ time with $O(n^3/\log n)$ processors are given in [11, 18] and comparable result on interconnection networks is given in [12]. The EREW PRAM version of the subcubic algorithm in [15, 17] is given in [1], which also features a VLSI algorithm based on the technique introduced

L. Wang (Ed.): COCOON 2005, LNCS 3595, pp. 621–631, 2005.

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 [2] and [3]. The former provides $O(Kn)$ and $O(Km^2n)$ time solutions for the one- and two-dimensional cases in the course of development of a systolic array algorithm of $O(n)$ 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.

This paper reviews the former solution and tunes it up for greater speed. Specifically we achieve $O(n \log K + K^2)$ time for the one-dimensional case. This is better than [3] when $K \leq \sqrt{n} \log n$.

If we use the above algorithm directly for the two-dimensional maximum subarray problem with an $(n, n)$-array, we have $O(n^3 \log K + n^2K^2)$ time complexity. We improve this time complexity to $O(n^3 + n^2K^2)$, which is $O(n^3)$ when $K \leq \sqrt{n}$.

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 of concluding remarks.

## 2 Review of the Maximum Subarray Problem

We give a two-dimensional array $a[1..m, 1..n]$ as 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 data set. We suppose the upper-left corner has coordinates (1,1).

*Example 1.* : Let $a$ be given by

$$a = \begin{bmatrix} -1 & 2 & -3 & 5 & -4 & -8 & 3 & -3 \\ 2 & -4 & -6 & -8 & 2 & -5 & 4 & 1 \\ 3 & -2 & 9 & -9 & \left[\begin{matrix} -1 & 10 \\ 8 & -2 \end{matrix}\right] & & -5 & 2 \\ 1 & -3 & 5 & -7 & & & 2 & -6 \end{bmatrix}$$

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 [4], and extended it to two-dimensions.

We use another $O(n)$ algorithm given in [2]. It 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

**Algorithm 1** Prefix Sum

```
sum[0]←0;
for i←1 to n do sum[i]←sum[i-1]+a[i];
```

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

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

To yield the maximum sum from a one-dimensional array, we have to find indices $x, y$ that maximize $\sum_{i=x}^{y} a[i]$. The notations $min$ and $max$ are used for variables and MAX and MIN are used for operations.

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

**Lemma 1.** *For all $x, y \in [1..n]$, and $x \leq y$,*
$\mathrm{MAX}_{1\leq x\leq y\leq n}\{\sum_{i=x}^{y} a[i]\} = \mathrm{MAX}_{1\leq x\leq y\leq n}\{sum[y] - sum[x-1]\}$
$= \mathrm{MAX}_{1\leq y\leq n}\{sum[y] - \mathrm{MIN}_{1\leq x\leq y}\{sum[x-1]\}\} = \mathrm{MAX}_{1\leq y\leq n}\{sum[y] - min_y\}$

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

**Algorithm 2** Maximum Sum in a one-dimensional array

```
min←0; //minimum prefix sum
M←0; //current maximum sum, initially 0 for empty subarray
sum[0]←0;
for i←1 to n do begin
    sum[i]←sum[i-1]+a[i];
    cand←sum[i]-min; //min=min_i
    M←MAX{M,cand};
    min←MIN{min,sum[i]};//min=min_{i+1}
end.
```

While we accumulate $sum[i]$, the prefix sum, we also maintain $min$, the minimum of the preceding prefix sums. By subtracting $min$ from $sum[i]$, we have a candidate for the maximum sum. At the end, $M$ is the maximum sum.

## 3 Finding the $K$ Maximum Sums in $O(Kn)$ Time

Based on the algorithm in Section 2, let us proceed to discuss the $K$-maximum subarray problem, again for the one-dimensional case.

Instead of having a single variable that book-keeps the minimum prefix sum, 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 $min_i$ is given by $min = \{0, +\infty\cdots, +\infty\}$. We also maintain the list of candidate sums produced from $sum[i]$, sorted in non-decreasing order. This list $cand_i$ is given by $\{sum[i]-min_i[1], sum[i]-min_i[2]\cdots, sum[i]-min_i[K]\}$ . Let $max_i$ be the list of $K$ maximum sums for $a[1..i]$. This list is maintained in $M$ in Algorithms 3 and 4 sorted in non-increasing order. When the algorithm ends, $M$ contains the final solution $max_n$. The merged list of two sorted sequences $x$ and $y$ are denoted by $merge(x, y)$. We have the following lemma.

**Lemma 2.** $max_{i+1}$ is the list of the $K$ maximum elements of $merge(max_i, cand_{i+1})$

Array names are used to denote sets, lists, etc. in the subsequent descriptions. We maintain the list of $K$ minimum prefix sums in $min$. 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 largest $K$ 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 8 in the algorithm preserves the loop-invariant from step $i$ to step $i+1$ as stated in Lemma 2. At the end, $M$ is the solution.

**Algorithm 3** $K$ maximum sums in a one-dimensional array

```
 1: for k←1 to K do begin
 2:    min[k]←∞; M[k]←−∞;
 3: end;
 4: sum[0]←0; min[1]←0; M[1]←0;
 5: for i←1 to n do begin
 6:    sum[i]←sum[i-1]+a[i];
 7:    for k←1 to K do cand[k]←sum[i]-min[k];
 8:    M←K largest elements of merge(M,cand);
 9:    insert sum[i] into min;
10: end.
```

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 line 7 and 8 where we anyway need to spend $O(K)$ time generating the candidate list and the list of $K$ maximum sums 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 2.

## 4 Improved Algorithm for $K$ Maximum Sums for Small $K$

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. It will now be discussed that we do not need to generate such a number of candidates when $K \leq n$.

Let us assume we have in $min_i[1..K]$ the list of $K$ minimum prefix sums to be subtracted from $sum[i]$. This list is sorted in non-decreasing order. In Algorithm 4, $min_i$ is given by $min$ at the end of the $i$-th iteration.

Let $cand_i[k] = sum[i] - min_i[k]$ for $k = 1 \cdots, K$. As $min_i$ is sorted, the produced list of candidates $cand_i$ is sorted in non-increasing order, and has the first item $cand_i[1]$ being the largest candidate produced from $sum[i]$.

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

$$sample[i] = cand_i[1] = sum[i] - min_i[1], (i = 1 \cdots, n)$$

We then select $K$ largest values of $sample$. Let us denote the list containing them by $Ksamples$, which is sorted in non-increasing order, given by

$$Ksamples = \{sample[x_1], sample[x_2] \cdots, sample[x_K]\},$$

where $x_1 \cdots, x_K$ are the indices of those selected samples.

It is easily observed that if $sample[w]$, the largest candidate produced from $sum[w]$, does not even qualify for $Ksamples$, no candidate produced from $sum[w]$ can become one of the final $K$ maximum sums as we know there are already at least $K$ sums greater than or equal to them.

When $Ksamples$ does not include $sample[w]$, the full generation of $cand_w[1..K]$ is thus avoided. In such a case, we can skip to the next iteration saving $O(K)$ time. We generate candidates only from $sum[x_i]$, which produced selected candidates for $Ksamples$.

### 4.1 Pre-process

We note that we do not need $min_i[1..K]$ for all $i \in [1..n]$ before the sampling and selection process. We only need $min_i[1]$ for $i = 1 \cdots, n$.

During the pre-process, we traverse 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 $1 \leq j \leq i-1$. 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 is selected by a linear time selection algorithm. Then we filter out values smaller than the $K$-th maximum, being left with the $K$ largest samples. We sort and store those samples in $Ksamples$. We can test whether an item is in $Ksamples$ by comparing it with $sample[x_K]$, the last element in the list.

### 4.2 Candidate Generation and Selection

Inside the "for" loop starting at line 10 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 $cand_i$ is performed when the $i$-th sample is included in $Ksamples$. Thus Part I is performed $K$ times.

The following routine, Part II, is the insertion of prefix sum into the sorted list of minimum prefix sums. Unlike Part I, all $n$ prefix sums should be considered. We first examine if a new prefix sum ever needs to be inserted, and if so we need to find an appropriate position for the new entry in $min$. This $min$ contains $K$ minimum prefix sums, and if there are more than $K$ items, we may need to drop the largest item. The choice of an appropriate data structure for $min$ is important to determine the total complexity. Besides $min$, all other lists, $cand$ and $M$, may be simple one-dimensional arrays. We assume $min[k]$ is the $k$-th smallest element of $min$ when $min$ is regarded as a set regardless of the actual data structure of $min$. We choose a 2-3 tree with *level-link* as a suitable data structure.

A 2-3 tree keeps all the leaf nodes sorted. A 2-3 tree with level-link described in [7] has all the internal nodes at the same depth connected, enabling *finger* searches. Finger search trees with constant update time are discussed in [6, 9], but they both require logarithmic time for positioning and do not improve overall time complexity. Now we analyze each part.

*Part I.* For Part I, generating the candidate list involves access to the list of minimum prefix sums. If an ordinary 2-3 tree is used, accessing each of $min[1..K]$ costs $O(\log K)$ time. Since we need to access all $min[1..K]$ sequentially to generate candidates, this access cost seems rather expensive. However if a 2-3 tree with level-link is used, after initial search for $min[1]$ spending $O(\log K)$, subsequent elements are found in $O(1)$ time due to finger search. As actual generation of $K$ candidates requires $O(K)$ time, this initial $O(\log K)$ access time is absorbed. The total time for Part I over $K$ iterations is therefore $O(K^2)$.

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

**Algorithm 4** Improved algorithm for $K$ maximum sums in a one-dimensional array

```
//[INITIALIZATION]
 1: for k←1 to K do begin min[k]←∞; M[k]←−∞; end;
 2: sum[0]←0; min[1]←0; M[1]←0;
//[PRE-PROCESS]
 3: for i←1 to n do begin
 4:     sum[i]←sum[i-1]+a[i];
  //sample for initial K large values
 5:     sample[i]←sum[i]-min[1];
 6:     if sum[i] < min[1] then min[1]←sum[i];
 7: end;
 8: Ksamples←K largest sorted values of sample[1..n];
//[CANDIDATE GENERATION and SELECTION]
 9: min[1]←0;
10: for i←1 to n do begin
11:     if sum[i]-min[1] > sample[x_K] then begin
            //PART I: Generate cand and update M
12:         for k←1 to K do cand[k]←sum[i]-min[k];
13:         M←K largest values of merge(M,cand);
14:     end;
        //PART II: Update min
15:     insert sum[i] into min;
16: end.
```

### 4.3 Total Time

Using the data structure for $min$ described above, the overall time including Part I and Part II is thus $O(n \log K + K^2)$. The time for the preprocessing (sampling, selection and screening) is $O(n)$ which is absorbed. Compared with $O(min\{K + n \log^2 n, n\sqrt{K}\})$ time by [3], this algorithm is faster when $K \leq \sqrt{n} \log n$.

We can organize the if-statement at line 11 into a while-statement. We keep computing candidates as $sum[i] - min[k]$ for $k = 1, 2 \cdots, K$, while the condition $sum[i] - min[k] > sample[x_K]$ is satisfied, and only those candidates can be inserted into $M$ with unqualified sums being deleted from $M$. Also the right-hand side of the condition can be replaced by the minimum of $M$ instead of the fixed $sample[x_K]$. This modification can improve the average performance, but the worst case behavior is not clear at present.

## 5 Speed-Up for Two Dimensions

If we use the algorithm in the previous section for an $(n, n)$-array, we have an $O(n^3 \log K)$ time algorithm. We speed up the algorithm for small $K$ in the two-dimensional case based on the divide-and-conquer method. To remove the factor of $\log K$ from the complexity, we do not maintain sorted order for $K$-tuples.

### 5.1 Generalization of Distance Matrix Multiplication

The distance matrix multiplication 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.

$$c_{ij} = \mathrm{MIN}_{k=1}^{n}\{a_{ik} + b_{kj}\}, (i, j = 1 \cdots, n) \cdots (1)$$

The operation in the right-hand side of (1) is called distance matrix multiplication 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.

Now we divide $A$, $B$, and $C$ into $(K, K)$-submatrices for $N = n/K$ as follows:

$$\begin{pmatrix} A_{1,1} & \cdots & A_{1,N} \\ \cdots & & \\ A_{N,1} & \cdots & A_{N,N} \end{pmatrix} \begin{pmatrix} B_{1,1} & \cdots & B_{1,N} \\ \cdots & & \\ B_{N,1} & \cdots & B_{N,N} \end{pmatrix} = \begin{pmatrix} C_{1,1} & \cdots & C_{1,N} \\ \cdots & & \\ C_{N,1} & \cdots & C_{N,N} \end{pmatrix}$$

Matrix $C$ can be computed by

$$C_{ij} = \mathrm{MIN}_{k=1}^{N}\{A_{ik}B_{kj}\}(i, j = 1 \cdots, N) \cdots (2)$$

where the product of submatrices is defined similarly to (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 (2).

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 $\{a_{ik} + b_{kj} | k = 1 \cdots, n\}$. We call this definition $K$-distance matrix multiplication, or simply $K$-matrix multiplication.

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 $\mathbf{a} = (a_1 \cdots, a_K)$. The MIN operation on the two $K$-tuples $\mathbf{a}$ and $\mathbf{b}$ is defined by $\mathrm{MIN}\{\mathbf{a}, \mathbf{b}\} = (c_1 \cdots, c_K)$, where $(c_1 \cdots, c_K)$ is the list of the $K$ smallest elements of $\mathbf{a} \cup \mathbf{b}$. If there are equal values in $\mathbf{a}$ or $\mathbf{b}$, the union operation here is for multisets. Similarly we can define $\mathrm{MAX}\{\mathbf{a}, \mathbf{b}\} = \mathbf{a} \cup \mathbf{b} - (c_1 \cdots, c_K)$. 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 with the median selection algorithm and filtering process in a similar way to those described in Section 4.1. 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 $K$-matrix multiplication, where each element in (1) is a $K$-tuple, we use the extended MIN operation in (2), where the elements of matrix $A_{ik}B_{kj}$ are $K$-tuples.

Let us rename $A_{ik}$ and $B_{kj}$ in the above by $A$ and $B$, and consider the multiplication. This time we can return all $\{a_{i1} + b_{1j} \cdots, a_{iK} + b_{Kj}\}$ as candidate $K$-tuples, taking $O(K^3)$ time, and use the extended MIN operations in (2). Then the time for $N^3$ products in (2) is $O((n/K)^3K^3) = O(n^3)$. The time for extended MIN operations in (2) is $O(Nn^2K) = O(n^3)$. Thus the total time is $O(n^3)$ for $K$-matrix multiplication.

### 5.2 Application to the $K$-Maximum Subarray Problem

We review the divide-and-conquer approach given in [15]. 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 $m \leq n$ 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.

The central algorithmic concept in this section is again that of prefix sum. We use distance matrix multiplications 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 $s[i, 0] = s[0, j] = 0$. Obviously this can be done in $O(mn)$ time. The outer framework of the algorithm is given below. Note that the prefix sums once computed are used throughout recursion.

**Algorithm M: Maximum subarray**
1. If the array becomes one element, return its value.
2. Otherwise, if $m > n$, rotate the array 90 degrees.
3. Thus we assume $m \leq n$.
4. Let $A_{left}$ be the solution for the left half.
5. Let $A_{right}$ be the solution for the right half.
6. Let $A_{column}$ be the solution for the column-centered problem.
7. Let the solution be the maximum of those three.

Here the column-centered 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.

$$A_{column} = \mathrm{MAX}_{k=0,l=0,i=1,j=n/2+1}^{i-1,n/2-1,m,n}\{s[i,j]-s[i,l]-s[k,j]+s[k,l]\}$$

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 $i = 1\cdots, m$ and $k = 0\cdots, i-1$.

$$A_{column}[i,k] = \mathrm{MAX}_{l=0,j=n/2+1}^{n/2-1,n}\{-s[i,l]+s[k,l]+s[i,j]-s[k,j]\}$$

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

$$A_{column}[i,k] = -\mathrm{MIN}_{l=0}^{n/2-1}\{s[i,l]+s^*[l,k]\}+\mathrm{MAX}_{j=n/2+1}^{n}\{s[i,j]+s^*[j,k]\}\cdots(3)$$

The first part in the above is distance matrix multiplication 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 $i = 1\cdots, m; j = 1\cdots, n/2$. 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 multiplying $S_1$ and $S_1^*$ by the MIN-version and taking the lower triangle, multiplying $S_2$ and $S_2^*$ by the MAX-version and taking the lower triangle, and finally subtracting the former from the latter and taking the maximum from the resulting triangle. We call the operation of transforming a matrix into a triangle $triangulation$.

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

Now let us analyze the time for the work at level 0. We 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 (3). Let $M(n)$ be the time for multiplying two $(n/2,n/2)$ matrices. At level 0, we obtain an $A_{column}$ and two smaller $A_{column}$'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 3.** *Let $c$ be an arbitrary constant such that $c > 0$. Suppose $M(n)$ satisfies the condition $M(n) \geq (4+c)M(n/2)$. Then the above $T(n)$ satisfies $T(n) \leq 12(1+4/c)M(n)$.*

Clearly the complexity of $O(n^3)$ for $M(n)$ satisfies the condition of the lemma with some constant $c > 0$. Thus the maximum subarray problem can be solved in $O(n^3)$ 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 $M(n)$.

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. Similar considerations can be made on $K$ in the following.

Now we describe the $K$-maximum subarray problem. When the recursion hits a $(K, K)$ array, we select $K$ largest sums from possible $K^4$ subarrays. This can easily be done by changing the top-left and bottom-right co-ordinates on the prefix sum array. 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 $K$. First we change line 1 in Algorithm M as follow:

1. If the array becomes $K \times K$, return the solution by Algorithm A.

Next we describe how to compute $A_{column}$ at each recursion. We first define $\mathbf{a} - \mathbf{b}$ for two $K$-tuples, **a** and **b**, to be the $K$ values that are made by subtracting elements of **b** from those of **a** component-wise. To compute distance matrix multiplication by $S_2S_2^* - S_1S_1^*$ in (4), we use the $K$-matrix multiplication of MAX and MIN version. To compute the subtraction, we follow the above operation of $\mathbf{a}-\mathbf{b}$ component-wise. As we assume $K \leq n$, this complexity $O(Kn^2)$ of triangulation and subtraction is absorbed in the main complexity. The initial condition for $T$ becomes $T(K) = O(K^4)$. As there are $n/K \times n/K$ subarrays at the bottom of recursion, the total time spent by Algorithm A is $O((n/K)^2K^4)$ = $O(n^2K^2)$. If we use the $O(n^3)$ time algorithm for $K$-matrix multiplication in Algorithm M, the total time before hitting the bottom of recursion is $O(n^3)$. Thus the total time is $O(n^3 + n^2K^2)$. This time complexity is $O(n^3)$ when $K \leq \sqrt{n}$. The $K$ maximum sums can be sorted with additional $O(K \log K)$ time.

## 6 Concluding Remarks

In the previous section, we improved the complexity from $O(n^3 \log K)$ to $O(n^3)$ for small $K$. If we use a sub-cubic algorithm for DMM with time complexity $O(n^3\sqrt{\frac{\log\log n}{\log n}})$ in [14], we can achieve a sub-cubic complexity for the the two-dimensional case for even smaller $K \leq O(\sqrt{\frac{\log n}{\log\log n}})$, using the same frame work of divide-and-conquer and $K$-tuples. Recent developments for DMM [16, 19] can also be incorporated. Details are omitted here.

If we find $K$-maximum subarray 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 Algorithm 2, 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 [13]. We can extend the $O(Kn)$ time algorithm to two dimensions with $O(Kn^3)$ time. 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 [8]. To the authors' knowledge, the two-dimensional case has not been solved.

## References


1. Bae, S.E., Takaoka, T.: Parallel approaches to the maximum subarray problem. Japan-Korea Workshop on Al. and Comp. (2003) 94–104
2. Bae, S.E., Takaoka, T.: Algorithms for the problem of K maximum sums and a VLSI algorithm for the K maximum subarrays problem. ISPAN 2004 (2004) 247–253
3. Bengtsson, F., Chen, J.: Efficient algorithms for the k maximum sums. ISAAC 2004 LNCS, Vol. 3341 Springer (2004) 137–148
4. Bentley, J.: Programming pearls: algorithm design techniques. Commun. ACM, Vol. 27(9) (1984) 865–873
5. Bentley, J.: Programming pearls: perspective on performance. Commun. ACM, Vol. 27(11) (1984) 1087–1092
6. Brodal, G.S.: Finger search trees with constant insertion time. SODA 1998 (1998) 540–549
7. Brown, M.R., Tarjan, R.E.: The design and analysis of a data structure for representing sorted lists. SIAM Jour. on Comp., Vol. 9(3) (1980) 594–614
8. Csürös, M.: Algorithms for finding maxima-scoring segment sets. WABI 2004 LNCS, Vol. 3240 Springer (2004) 62–73
9. Dietz, P.F., Raman, R.: A constant update time finger search tree. Inf. Process. Lett., Vol. 52(3) (1994) 147–154
10. Miller, R., Boxer, L.: Algorithms Sequential & Parallel- A Unified Approach. Prentice Hall, (2000)
11. Perumalla, K., Deo, N.: Parallel algorithms for maximum subsequence and maximum subarray. Parallel Process. Lett., Vol. 5(3) (1995) 367–373
12. Qui, K., Akl, S.G.: Parallel maximum sum algorithms on intercommenction networks. Queen's Uni. Dept. of Com. and Info. Sci. Technical Report 99-431 (1999)
13. Ruzzo, W.L., Tompa, M.: A linear time algorithm for finding all maximal scoring subsequences. Intelligent Sys. in Molecular Biology (1999) 234–241
14. Takaoka, T.: A new upper bound on the complexity of the all pairs shortest paths problem. Inf. Process. Lett., Vol. 43(4) (1992) 195–199
15. Takaoka, T.: Efficient algorithms for the maximum subarray problem by distance matrix multiplication. Elec. Notes in Theoretical Computer Sci., Vol. 61 Elsevier (2002)
16. Takaoka, T.: A faster algorithm for the all-pairs shortest path problem and its application. COCOON 2004, LNCS, Vol. 4106 Springer (2004) 278–289
17. Tamaki, H., Tokuyama, T: Algorithms for the maximum subarray problem based on matrix multiplication. SODA 1998 (1998) 446–452
18. Wen, Z.: Fast parallel algorithms for the maximum sum problem. Parallel Computing, Vol. 21(3) (1995) 461–466
19. Zwick, U.: A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. ISAAC 2004, LNCS, Vol. 3341 Springer (2004) 921–932