Selecting Sums in Arrays

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. In an array of n numbers each of the $\binom{n}{2}+n$ contiguous subarrays define a sum. In this paper we focus on algorithms for selecting and reporting maximal sums from an array of numbers. First, we consider the problem of reporting k subarrays inducing the k largest sums among all subarrays of length at least l and at most u. For this problem we design an optimal $O(n+k)$ time algorithm. Secondly, we consider the problem of selecting a subarray storing the k'th largest sum. For this problem we prove a time bound of $\Theta(n \cdot \max\{1, \log(k/n)\})$ by describing an algorithm with this running time and by proving a matching lower bound. Finally, we combine the ideas and obtain an $O(n \cdot \max\{1, \log(k/n)\})$ time algorithm that selects a subarray storing the k'th largest sum among all subarrays of length at least l and at most u.

1 Introduction

In an array, $A[1,\ldots,n]$, of numbers each subarray, $A[i,\ldots,j]$ for $1 \le i \le j \le n$, defines a sum, $\sum_{t=i}^{j} A[t]$. There are $\binom{n}{2}+n$ different subarrays each inducing a sum. Locating a subarray $A[i,\ldots,j]$ maximizing $\sum_{t=i}^{j} A[t]$ is known as the *maximum sum* problem, and it was formulated by Ulf Grenander in a pattern matching context. Algorithms solving the problem also have applications in areas such as Data Mining [12, 13] and Bioinformatics [1]. In [5] Bentley describes the problem and an optimal linear time algorithm.

The problem can be extended to any number of dimensions. In the two dimensional version of the problem the input is an $m \times n$ matrix of numbers, and the task is to locate the connected submatrix storing the largest aggregate. This problem can be solved by a reduction to $\binom{m}{2}+m$ one-dimensional instances of size n, or a single one-dimensional instance in one array of length $O(m^2 n)$ created by separating each of the $\binom{m}{2}+m$ instances mentioned before by dummy $-\infty$ elements. However, this solution is not optimal since faster algorithms are known [22, 21]. The currently fastest algorithm is due to Takaoka who describes an $O(m^2 n\sqrt{\log\log m/\log m})$ time algorithm in [21][1]. The only known

[⋆] 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.

[1] For simplicity of exposition by $\log x$ we denote the value $\max\{1, \log_2 x\}$.

lower bound for the problem is the trivial $\Omega(mn)$ bound. The two-dimensional version was the first problem studied, introduced as a method for maximum likelihood estimations of patterns in digitized images [5].

A natural extension of the maximum sum problem, introduced in [4], is to compute the k largest sums for $1 \leq k \leq \binom{n}{2} + n$. The subarrays are allowed to overlap, and the output is k triples of the form $(i, j, \sum_{t=i}^{j} A[t])$. An $O(nk)$ time algorithm is given in [4]. An algorithm solving the problem in optimal $O(n+k)$ time using $O(k)$ additional space is described in [7].

Another generalization of the maximal sum problem is to restrict the length of the subarrays considered. This generalization is considered in [15, 18, 9] mainly motivated by applications in Bioinformatics such as finding tandem repeats [23], locating GC-rich regions [14], and constructing low complexity filters for sequence database search [2]. In [15] Huang describes an $O(n)$ time algorithm locating the largest sum of length at least l, while in [18] an $O(n)$ time algorithm locating the largest sum of length at most u is described. The algorithms can be combined into at linear time algorithm finding the largest sum of length at least l and at most u [18]. In [9] it is shown how to solve the problem in $O(n)$ time when the input elements are given online one by one.

The *length constrained k maximal sums* problem is defined as follows. Given an array A of length n, find the k largest sums consisting of at least l and at most u numbers. The k maximal sums problem is the special case of this problem where $l = 1$ and $u = n$. Lin and Lee solved the problem using a randomized algorithm with an expected running time of $O(n \log(u-l) + k)$ [17]. Their algorithm is based on a randomized algorithm that selects the k'th largest length constrained sum from an array in $O(n \log(u-l))$ expected time. The authors state as an open problem whether this is optimal. Furthermore, in [16] Lin and Lee describe a deterministic $O(n \log n)$ time algorithm that selects the k'th largest sum in an array of size n. They propose as an open problem whether this bound is tight. This problem is known as the *sum selection* problem.

Our Contribution. In this paper we settle the time complexity for the sum selection problem and the length constrained k maximal sums problem. First, we describe an optimal $O(n+k)$ time algorithm for the length constrained k maximal sums problem in Section 2. This algorithm is an extension of our optimal algorithm solving the k maximal sums problem from [7]. Secondly, we prove a time bound of $\Theta(n \log(k/n))$ for the sum selection problem in Section 3. This is the main result of the paper. An $O(n \log(k/n))$ time algorithm that selects the k'th largest sum is described in Section 3.1, and in Section 3.2 we prove a matching lower bound using a reduction from the cartesian sum problem [11]. Finally, in Section 4 we combine the ideas from the two algorithms we have designed and obtain an $O(n \log(k/n))$ time algorithm that selects the k'th largest sum among all sums consisting of at least l and at most u numbers. This bound is always as good as the previous randomized bound of $O(n \log(u-l))$ by Lin and Lee [17], since there are $\sum_{t=l}^{u} n - t + 1 \leq n(u-l+1)$ subarrays of length between l and u in an array of size n and thus $k/n \leq u - l + 1$. Due to lack of

Table 1. Overview of results on reporting and selecting sums in arrays.

Problem	Previous Work	This Paper
Length Const. k Maximal Sums	$O(n \log(u-l)+k)$ exp. [17]	$O(n+k)$
Sum Selection	$O(n \log n)$ [16]	$\Theta(n \log(k/n))$
Length Const. Sum Selection	$O(n \log(u-l))$ exp. [17]	$O(n \log(k/n))$

space the details are deferred to the full version which will combine the results of this paper and the results in [7]. The results are summarized in Table 1.

2 The Length Constrained k Maximal Sums Problem

In this section we present an optimal $O(n+k)$ time algorithm that reports the k largest sums of an array A of length n with the restriction that each sum is an aggregate of at least l and at most u numbers. We reuse the idea from the k maximal sums algorithm in [7], and construct a heap[2] that implicitly represents all $\sum_{t=l}^{u} n-t+1 = O(n(u-l))$ valid sums from the input array using only $O(n)$ time and space. The k largest sums are then retrieved from the heap using Fredericksons heap selection algorithm [10] that extracts the k largest elements from a heap in $O(k)$ time. We note that the k maximal sums algorithm from [7] can be altered to use a heap supporting deletions to obtain an $O(n \log(u-l)+k)$ algorithm solving the problem without randomization. The difference between our new $O(n+k)$ time algorithm and the algorithm solving the k maximal sums problem [7] is in the way the sums are grouped in heaps. This change enables us to solve the problem without deleting elements from a heap. In the following we assume that $l < u$. If $l = u$ the problem can be solved in $O(n)$ time using a linear time selection algorithm [6].

2.1 A Linear Time Algorithm

For each array index j, for $j = 1, \ldots, n-l+1$, we build data structures representing all sums of length between l and u ending at index $j+l-1$. This is achieved by constructing all sums ending at $A[j]$ with length between 1 and $u-l+1$, and then adding the sum of the $l-1$ elements, $A[j+1, \ldots, j+l-1]$, following $A[j]$ in the input array to each sum. To construct these data structures efficiently, the input array is divided into *slabs* of $w = u-l$ consecutive elements, and the sums are grouped in disjoint sets, $\hat{Q}_j$ and $\bar{Q}_j$ for $j = 1, \ldots, n$, depending on the slab boundaries.

Let a be the first index in the slab containing index j, i.e. $a = 1 + \left\lfloor \frac{j-1}{w} \right\rfloor w$. The set $\hat{Q}_j$ contains all sums of length between l and u ending at index $j+l-1$ that start in the slab containing index j and is defined as follows:

$$\hat{Q}_j = \{(i, j+l-1, sum) \mid a \leq i \leq j,\ sum = c + \textstyle\sum_{t=i}^{j} A[t]\} ,$$

[2] For simplicity of exposition, by heap we denote a heap ordered binary tree where the largest element is placed at the root.

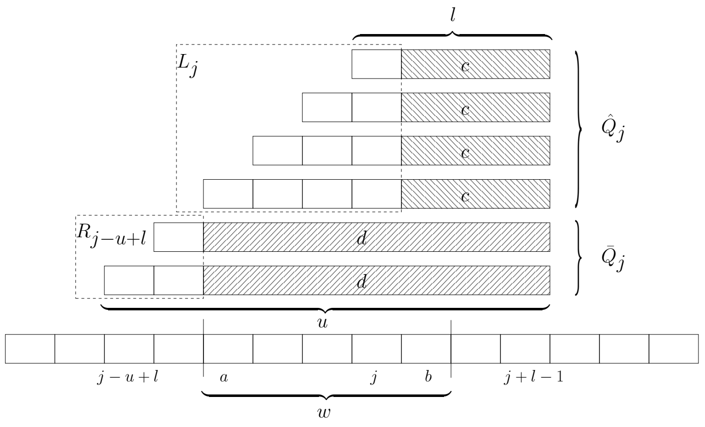

Fig. 1. Overview of the sets, $l = 4, u = 9, c = \sum_{t=j+1}^{j+l-1} A[t]$ and $d = \sum_{t=a}^{j+l-1} A[t]$. A slab is starting at index a and ending at index b.

where $c = \sum_{t=j+1}^{j+l-1} A[t]$ is the sum of $l-1$ numbers in $A[j+1,\dots,j+l-1]$. The set $\bar{Q}_j$ contains the $(u-l+1)-(j-a+1) = u-l-j+a$ valid sums ending at index $j+l-1$ that start to the left of index a, thus:

$$\bar{Q}_j = \{(i, j+l-1, sum) \mid j-u+l \le i < a,\ sum = d + \textstyle\sum_{t=i}^{a-1} A[t]\}\ ,$$

where $d = \sum_{t=a}^{j+l-1} A[t]$ is the result of summing the $j-a+l$ numbers in $A[a,\dots,j+l-1]$. The sets are illustrated in Figure 1. By construction, the sets $\hat{Q}_j$ and $\bar{Q}_j$ are disjoint and their union is the $u-l+1$ sums of length between l and u ending at index $j+l-1$.

With the sets of sums defined we continue with the representation of these. The sets $\hat{Q}_j$ and $\bar{Q}_j$ are represented by pairs $\langle\hat{\delta}_j, \hat{H}_j\rangle$ and $\langle\bar{\delta}_j, \bar{H}_j\rangle$ where $\hat{H}_j$ and $\bar{H}_j$ are partially persistent heaps and $\hat{\delta}_j$ and $\bar{\delta}_j$ are constants that must be added to all elements in $\hat{H}_j$ and $\bar{H}_j$ respectively to obtain the correct sums. For the heaps we use the Iheap from [7] which supports insertions in amortized constant time. Partial persistence is implemented using the node copying technique [8].

We construct representations of two sequences of sets, L_j and R_j for $j = 1,\dots,n$, that depend on the slab boundaries. Consider the slab $A[a,\dots,j,\dots,b]$ containing index j. The set L_j contains the $j-a+1$ sums ending at $A[j]$ that start between index a and j. The set R_j contains the $b-j+1$ sums ending at $A[b]$ starting between index j and b, see Figure 1.

Each set L_j is represented as a pair $\langle\delta_j^L, H_j^L\rangle$ where δ_j^L is an additive constant as above and H_j^L is a partially persistent Iheap. The pairs are incrementally constructed while scanning the input array from left to right as follows:

$$\begin{aligned}&\langle\delta_a^L, H_a^L\rangle = \langle A[a], \{0\}\rangle \wedge\\ &\langle\delta_j^L, H_j^L\rangle = \langle\delta_{j-1}^L + A[j], H_{j-1}^L \cup \{-\delta_{j-1}^L\}\rangle\ .\end{aligned}$$

This is also the construction equations used in [7]. Constructing a representation of L_a is simple, and creating a representation for L_j can be done efficiently given a representation of L_{j-1}. The representation of L_j is constructed by implicitly adding $A[j]$ to all elements from L_{j-1} by setting $\delta_j^L = \delta_{j-1}^L + A[j]$ and inserting an element to represent the sum $A[j]$. Since $\delta_{j-1}^L + A[j]$ needs to be added to all elements in the representation of L_j, an element with $-\delta_{j-1}^L$ as key is inserted into H_{j-1}^L, yielding H_j^L ending the construction. Partial persistence ensures that the Iheap H_{j-1}^L used to represent L_{j-1} is not destroyed. By the above description and the cost of applying the node copying technique [8] the amortized time needed to construct a pair is $O(1)$.

The R_j sets are represented by partially persistent Iheaps H_j^R, and these representations are built by scanning the input array from right to left. We get the following incremental construction equations:

$$\begin{aligned} H_b^R &= \{A[b]\} \wedge \\ H_j^R &= H_{j+1}^R \cup \{\textstyle\sum_{t=j}^{b} A[t]\} \ . \end{aligned}$$

Similar to the $\langle \delta_j^L, H_j^L \rangle$ pairs, constructing a partial persistent Iheap H_j^R also takes $O(1)$ time amortized. Therefore, the time needed to build the representation of the $2n$ sets L_j and R_j for $j = 1, \dots, n$ is $O(n)$.

We represent the sets $\hat{Q}_j$ and $\bar{Q}_j$ using the representations of the sets L_j and R_{j-u+l}. Figure 1 illustrates the correspondence between $\hat{Q}_j$ and L_j and $\bar{Q}_j$ and R_{j-u+l}. Consider any index $j \in \{1, \dots, n-l+1\}$, and let $A[a, \dots, j, \dots, b]$ be the current slab containing j. The set $\hat{Q}_j$ contains the sums of length between l and u that start in the current slab and end at index $j+l-1$. The set L_j contains the $j-a+1$ sums that start in the current slab and end at $A[j]$. Therefore, adding the sum of the $l-1$ numbers in $A[j+1, \dots, j+l-1]$ to each element in L_j gives $\hat{Q}_j$ and thus:

$$\hat{Q}_j = \langle c + \delta_j^L, H_j^L \rangle \ ,$$

where $c = \sum_{t=j+1}^{j+l-1} A[t]$.

Similarly, the set $\bar{Q}_j$ contains the $u - l + 1 - (j - a + 1) = u - l - j + a$ sums of length between l and u ending at $A[j+l-1]$ starting in the previous slab. The set R_{j-u+l} contains the $u-l-j-a$ shortest sums ending at the last index in the previous slab. Therefore, adding the sum of the $j+l-a$ numbers in $A[a, \dots, j+l-1]$ to each element in R_{j-u+l} gives $\bar{Q}_j$ and thus:

$$\bar{Q}_j = \langle d, H_{j-u+l}^R \rangle \ ,$$

where $d = \sum_{t=a}^{j+l-1} A[t]$.

Lemma 1. *Constructing the $2(n-l+1)$ pairs that represent $\hat{Q}_j$ and $\bar{Q}_j$ for $j = 1, \dots, n-l+1$ takes $O(n)$ time.*

Proof. Constructing all $\langle \delta_j^L, H_j^L \rangle$ pairs and all H_j^R partial persistent Iheaps takes $O(n)$ time, and calculating sums c and d takes constant time using a prefix array. Constructing the prefix array takes $O(n)$ time. Therefore, constructing $\hat{Q}_j$ and $\bar{Q}_j$ for $j = 1, \dots, n-l+1$ takes $O(n)$ time. □

After constructing the $2(n-l+1)$ pairs, they are assembled into one large heap using $2(n-l+1)-1$ dummy ∞ keys as in [7]. The largest $2(n-l+1)-1+k$ elements are then extracted from the assembled heap in $O(n+k)$ time using Fredericksons heap selection algorithm. The implicit sums given by adding δ values are explicitly computed while Fredericksons algorithm explores the final heap top down in the way described in [7]. The $2(n-l+1)-1$ dummy elements are discarded.

Theorem 1. *The algorithm described reports the k largest sums with length between l and u in an array of length n in $O(n+k)$ time.*

3 Sum Selection Problem

In this section we prove a $\Theta(n\log(k/n))$ time bound for the sum selection problem by designing an $O(n\log(k/n))$ time algorithm that selects the k'th largest sum in an array of size n and by proving a matching lower bound.

The idea of the algorithm is to reduce the problem to selection in a collection of sorted arrays and weight balanced search trees [19, 3]. The trees and the sorted arrays are constructed using the ideas from Section 2 and [7]. Selecting the k'th largest element from a set of trees and sorted arrays is done using an essential part of the sorted column matrix selection algorithm of Frederickson and Johnson [11]. The part of Frederickson and Johnsons algorithm that we use is an iterative procedure named *Reduce*. In a round of the Reduce algorithm each array, A, is represented by the $1+\lfloor\alpha|A|\rfloor$ largest element stored in the array, and a constant fraction of the elements in each array may be eliminated. This can be approximated in weight balanced search trees and the complexity analysis from [11] remains valid.

The lower bound is proved using a reduction from the $X+Y$ cartesian sum selection problem [11].

We note that if $k\leq n$ then the k maximal sums algorithm from [7] can be used to solve the problem optimally in $O(n)$ time.

To construct the sorted arrays efficiently, we use a heap data structure, that is a generalization of the Iheap, which we name *Bheap*. The Bheap is a heap ordered binary tree where each node of the tree contains a sorted array of size B. By heap order, we mean that all elements in a child of a node v must be smaller than the smallest element stored in v. Sorted arrays of B elements are required to be inserted in $O(B)$ time amortized. Our Bheap implementation is based on ideas from the functional random access lists in [20] and simple bubble up/down procedures based on merging sorted arrays.

3.1 An $O(n\log(k/n))$ Time Algorithm

In this section we reduce the sum selection problem to selection in a set of trees and sorted arrays. We use the weight balanced B-trees of Arge and Vitter [3] with degree $B=O(1)$. Similar to the grouping of sums in Section 2, each index

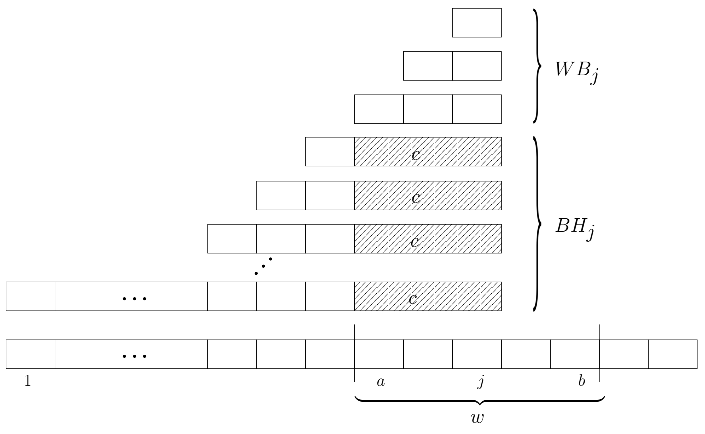

Fig. 2. Overview of the sets. Slab size $w = 5$, and $A[a, \ldots, b]$ is the slab containing index j.

j, for $j = 1, \ldots, n$, is associated with data structures representing all possible sums ending at $A[j]$. The set representing all sums ending at index j is defined as follows:

$$Q_j = \left\{ (i, j, sum) \mid 1 \le i \le j,\ sum = \textstyle\sum_{t=i}^{j} A[t] \right\} .$$

The input array is divided into slabs of size $w = \lceil k/n \rceil$, and the set Q_j is represented by two disjoint sets WB_j and BH_j that depend on the slab boundaries. The set WB_j contains the sums ending at index j beginning in the current slab, and BH_j contains the sums ending at index j not beginning in the current slab. Let $a = 1 + \left\lfloor \frac{j-1}{w} \right\rfloor w$, i.e. the first index in the slab containing index j, then:

$$\begin{aligned} WB_j &= \left\{ (i, j, sum) \mid a \le i \le j,\ sum = \textstyle\sum_{t=i}^{j} A[t] \right\} \wedge \\ BH_j &= \left\{ (i, j, sum) \mid 1 \le i < a,\ sum = c + \textstyle\sum_{t=i}^{a-1} A[t] \right\} , \end{aligned}$$

where $c = \sum_{t=a}^{j} A[t]$ is the sum of the $j - a + 1$ numbers in $A[a, \ldots, j]$. The sets WB_j and BH_j are disjoint, and $WB_j \cup BH_j = Q_j$ by construction. The sets are illustrated in Figure 2.

The set WB_j is represented as a pair $\langle \tau_j, T_j \rangle$ where T_j is a partial persistent weight balanced B-tree and τ_j is an additive constant that must be added to all elements in T_j to obtain the correct sums. The set BH_j is represented as a pair $\langle \delta_j, H_j \rangle$ where δ_j is an additive constant and H_j is a partial persistent Bheap with $B = w$.

The pairs $\langle \tau_j, T_j \rangle$ are constructed as follows. If j is the first index of a slab, i.e. $j = 1 + tw$ for some natural number t, then:

$$\langle \tau_j, T_j \rangle = \langle A[j], \{0\} \rangle .$$

This is the start of a new slab, and a new partial persistent weight balanced B-tree representing $A[j]$, the first element in the slab, is created. If j is not the first index in a slab then:

$$\langle \tau_j, T_j \rangle = \langle \tau_{j-1} + A[j], T_{j-1} \cup \{-\tau_{j-1}\} \rangle \ ,$$

i.e. we change the additive constant and insert $-\tau_{j-1}$ into the weight balanced tree T_{j-1}. These construction equations are identical to the construction equations from Section 2, and partial persistence ensures that T_{j-1} is not destroyed by constructing T_j.

For the $\langle \delta_j, H_j \rangle$ pairs representing the sets $\hat{Q}_j$, we observe that if $j \leq w$ then $BH_j = \emptyset$, thus:

$$\langle \delta_j, H_j \rangle = \langle 0, \emptyset \rangle \ .$$

If $j > w$ and j is not the first index in a slab, then adding $A[j]$ to all elements from the previous set yields the new set, thus:

$$\langle \delta_j, H_j \rangle = \langle \delta_{j-1} + A[j], H_{j-1} \rangle \ .$$

If j is the first index of a slab, i.e. $j = 1 + tw$ for some integer $t \geq 1$, all w sums represented in $\langle \tau_{j-1}, T_{j-1} \rangle$ are inserted into a sorted array S and each sum explicitly calculated. This sorted array then contains all sums starting in the previous slab ending at index $j-1$. For each element in S the additive constant δ_{j-1} is subtracted and S is inserted into the Bheap H_{j-1}. The construction equation becomes:

$$\langle \delta_j, H_j \rangle = \langle \delta_{j-1} + A[j], H_{j-1} \cup S \rangle \ ,$$

where

$$S = \left\{ (i, j, s - \delta_{j-1}) \mid j - w \leq i < j, s = \textstyle\sum_{t=i}^{j-1} A[t] \right\} \ .$$

Again, partial persistence ensures that the previous version of the Bheap, H_{j-1}, is not destroyed.

Lemma 2. *Constructing the pairs $\langle \delta_j, H_j \rangle$ and $\langle \tau_j, T_j \rangle$ for $j = 1, \ldots, n$ takes $O(n \log(k/n))$ time.*

Proof. The Bheap and the weight balanced B-trees have constant in and out-degree. Therefore, partial persistence can be implemented for both using the node copying technique [8].

For the Bheap, amortized $O(1)$ pointers and arrays are changed per insertion. The extra cost for applying the node copying technique is $O(B) = O(w)$ time amortized per insert operation. Constructing the sorted array S from a weight balanced B-tree takes $O(w)$ time. An insert in a Bheap is only performed every w'th step, and calculating additive constants in each step takes constant time. Therefore, the time used for constructing all $\langle \delta_j, H_j \rangle$ pairs is $O(n + \frac{n}{w} w) = O(n)$.

Each insert in a weight balanced B-tree is performed on a tree containing at most w elements using $O(\log w)$ time. Therefore, the extra cost of using the node copying technique is $O(\log w)$ time amortized per insert operation. Calculating an additive constant τ_j takes constant time, thus, constructing all $\langle \tau_j, T_j \rangle$ pairs takes $O(n \log(k/n))$ time. □

After the n pairs, $\langle \delta_i, H_i \rangle$, storing Bheaps are constructed, they are assembled into one large heap in the same way as in Section 2. That is, we construct a complete heap on top of the pairs using $n-1$ dummy nodes storing the same array of w dummy ∞ elements. We then use Fredericksons heap selection algorithm in the same way as in Section 2 where the representative for each node is the smallest element in the sorted array stored in it. Using Fredericksons heap selection algorithm the $2n-1$ nodes with the maximal smallest element and their $2n$ children are extracted. This takes $O(n)$ time and the nodes extracted from the Bheap gives $3n$ sorted arrays by discarding the $n-1$ dummy nodes.

Lemma 3. *The $3n$ nodes found as described above contain the k largest sums contained in the n pairs $\langle \delta_i, H_i \rangle$.*

Proof. The $4n-1$ nodes found by the heap selection algorithm forms a connected subtree T of the heap rooted at the root of the heap. Any element e stored in a node $v_e \notin T$ is smaller than all elements stored in any internal node $v \in T$ since, by heap order, e is smaller than the smallest element in the leaf of T that is on the path from v_e to the root. The smallest element in a leaf is smaller than the smallest element in any internal node since the leaf was not picked by the heap selection algorithm. There are $2n-1$ internal nodes in T and n of these does not store dummy elements. Therefore, for each element not residing in T there at least $nw = n\lceil \frac{k}{n} \rceil \geq k$ larger elements in the $3n$ found nodes. $\square$

These $3n$ sorted arrays of size w and the n pairs $\langle \tau_i, T_i \rangle$ storing weight balanced B-trees of size at most w contain at most $4nw = 4n\lceil \frac{k}{n} \rceil \leq 4(k+n)$ sums. The $3n$ arrays and the n weight balanced B-trees are given as input to the adapted sorted column matrix selection algorithm, which extracts the k'th largest element from these in $O(n \log(k/n))$ time. The fact that the weight balanced B-trees are partially persistent versions of the same tree and contain additive constants is handled by expanding the trees and computing the sums explicitly during the top down traversals performed by the selection algorithm as in Section 2 and [7].

Theorem 2. *The algorithm described selects the k'th largest sum in an array of size n in $O(n \log(k/n))$ time.*

3.2 Lower Bound

In this section we prove a matching lower bound of $\Omega(n \log(k/n))$ time for the sum selection problem via a reduction from the $X+Y$ cartesian sum selection problem. In the $X+Y$ cartesian sum selection problem as defined in [11], the input is two unsorted arrays X and Y and an integer k, and the task is to select the k'th largest element in the cartesian sum $\{x+y \mid x \in X,\ y \in Y\}$.

Given an instance of the $X+Y$ cartesian sum selection problem, $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$, and k, construct the following array A :

$x_1 - x_2$	$\cdots$	$x_i - x_{i+1}$	$\cdots$	$x_{n-1} - x_n$	$x_n + \infty + y_1$	$y_2 - y_1$	$\cdots$	$y_m - y_{m-1}$

where ∞ is a number larger than $(n+m) \cdot \max\{|x| \mid x \in X\} \cup \{|y| \mid y \in Y\}$. The sums in A have the following properties:

- A sum ranging from i to j where $i \leq n \leq j$ represents the sum $(\sum_{t=i}^{n-1} A[t]) + x_n + \infty + y_1 + (\sum_{t=n+1}^{j} A[t]) = x_i + y_{j-n+1} + \infty$.
- A sum including $A[n] = x_n + \infty + y_1$ is larger than any sum that does not

There are more sums in the sum selection instance than there are in the $X+Y$ cartesian sum instance since any sum not containing $A[n]$ does not correspond to an element in the cartesian sum. However, the k'th largest sum does contain $A[n]$ and corresponds to the k'th largest sum in the cartesian sum instance. Therefore, any algorithm that selects the k'th largest sum in an array can be used to select the k'th largest element in the cartesian sum.

The lower bound for selecting the k'th largest element in the cartesian sum $(X+Y)$ is $\Omega(m + p\log(k/p))$ comparisons where $|X| = n, |Y| = m$ with $n \leq m$ and $p = \min\{k, m\}$ [11]. In the reduction the size of the array A is $n+m-1$, which is $\Theta(n+m) = \Theta(m)$, and it can be built in $O(m)$ time.

Theorem 3. *Any algorithm that selects the k'th largest sum in an array of size n uses $\Omega(n\log(k/n))$ comparisons.*

4 Length Constrained Sum Selection

In this section we sketch how to select the k'th largest sum consisting of at least l and at most u numbers from an array of size n in $O(n\log(k/n))$ time. The algorithm combines the ideas from Section 2 and Section 3. Similar to Section 3 the algorithm works by reducing the problem to selection in a collection of weight balanced search trees and sorted arrays. It should be noted that a deterministic algorithm with running time $O(n\log(u-l))$ can be achieved by using weight balanced B-trees instead of Iheaps in the algorithm from Section 2, and using these as input to the adapted sorted column matrix selection algorithm instead of the heap selection algorithm.

To achieve $O(n\log(k/n))$ time, we constrain the lengths of the sums considered and divide the input array into slabs of size $u-l$ as in Section 2. Subsequently, we efficiently construct representations of the sets $\hat{Q}_j$ and $\bar{Q}_j$ defined in Section 2 using weight balanced trees and Bheaps by subdividing each slab into sub-slabs of size $\lceil \frac{k}{n} \rceil$ as in Section 3, recall $k/n \leq u-l+1$. Weight balanced B-trees are used to represent sums residing inside a sub-slab, and Bheaps are used to represent sums covering multiple sub-slabs. The sums are illustrated in Figure 3. The Bheaps and the weight balanced B-trees are constructed efficiently as in Section 3 using partial persistence.

After the representations of the sets $\hat{Q}_j$ and $\bar{Q}_j$ are constructed, the algorithm continues as in Section 3. The sorted arrays storing the k largest sums stored in the Bheaps are extracted using Fredericksons heap selection algorithm. The sorted arrays and the weight balanced B-trees are then given as input to the adapted sorted column matrix selection algorithm that selects the k'th largest sum.

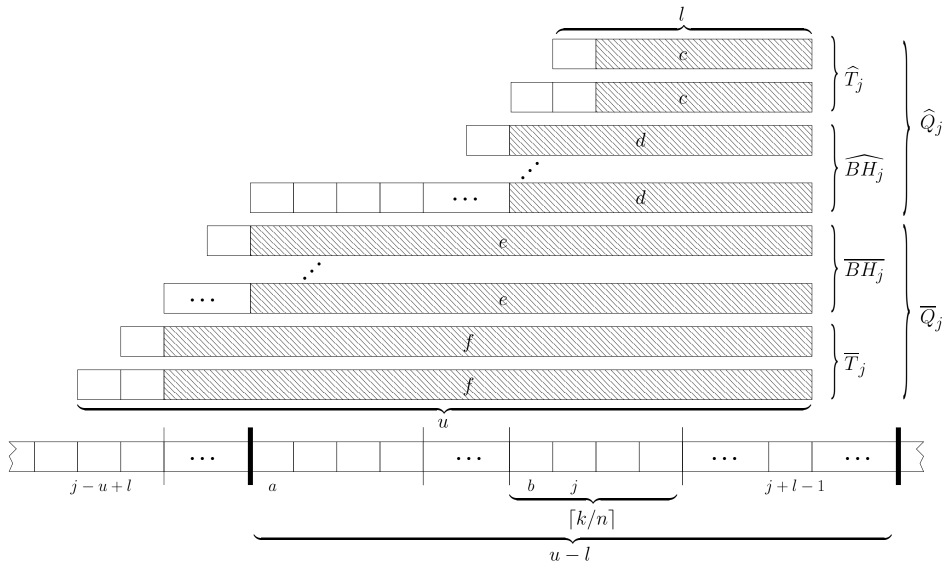

Fig. 3. Combining ideas - The sums associated with index j. A new slab of length $u-l$ starts at index a and a new subslab of length $\lceil k/n \rceil = 4$ starts at index b. $c = \sum_{t=j+1}^{j+l-1} A[t]$, $d = \sum_{t=b}^{j+l-1} A[t]$, $e = \sum_{t=a}^{j+l-1} A[t]$ and $f = \sum_{t=x}^{j+l-1} A[t]$ where x is the first index in the subslab following the subslab containing index $j-u+l$. The set $\hat{Q}_j$ is split into $\hat{T}_j$, represented by a weight balanced tree, and $\widehat{BH_j}$, represented by a Bheap. The set $\bar{Q}_j$ is split similarly.

Theorem 4. *The k'th largest sum of length between l and u in an array of size n can be selected in $O(n \log(k/n))$ time.*

References

1. L. Allison. Longest biased interval and longest non-negative sum interval. *Bioinformatics*, 19(10):1294–1295, 2003.
2. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
3. L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.
4. S. E. Bae and T. Takaoka. Algorithms for the problem of k maximum sums and a vlsi algorithm for the k maximum subarrays problem. In *Proc. 7th International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 247–253. IEEE Computer Society, 2004.
5. J. Bentley. Programming pearls: algorithm design techniques. *Commun. ACM*, 27(9):865–873, 1984.
6. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.
7. G. S. Brodal and A. G. Jørgensen. A linear time algorithm for the k maximal sums problem. In *Proc. 32nd International Symposium on Mathematical Foundations of*

Computer Science, volume 4708 of *Lecture Notes in Computer Science*, pages 442–453. Springer Verlag, Berlin, 2007.

8. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
9. T.-H. Fan, S. Lee, H.-I. Lu, T.-S. Tsou, T.-C. Wang, and A. Yao. An optimal algorithm for maximum-sum segment and its application in bioinformatics. In *Proc. 8th International Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science, pages 46–66. Springer Verlag, Berlin, 2003.
10. G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.
11. G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in X+Y and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982.
12. T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining with optimized two-dimensional association rules. *ACM Trans. Database Syst.*, 26(2):179–213, 2001.
13. T. Fukuda, Y. Morimoto, S. Morishta, and T. Tokuyama. Interval finding and its application to data mining. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E80-A(4):620–626, 1997.
14. S. Hannenhalli and S. Levy. Promoter prediction in the human genome. *Bioinformatics*, 17:S90–96, 2001.
15. X. Huang. An algorithm for identifying regions of a dna sequence that satisfy a content requirement. *Computer Applications in the Biosciences*, 10(3):219–225, 1994.
16. T.-C. Lin and D. T. Lee. Efficient algorithms for the sum selection problem and k maximum sums problem. In *The 17th International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science, pages 460–473. Springer Verlag, Berlin, 2006.
17. T.-C. Lin and D. T. Lee. Randomized algorithm for the sum selection problem. *Theor. Comput. Sci.*, 377(1-3):151–156, 2007.
18. Y.-L. Lin, T. Jiang, and K.-M. Chao. Efficient algorithms for locating the length-constrained heaviest segments, with applications to biomolecular sequence analysis. In *Proc. 27th International Symposium of Mathematical Foundations of Computer Science 2002*, Lecture Notes in Computer Science, pages 459–470. Springer Verlag, Berlin, 2002.
19. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 137–142, New York, NY, USA, 1972. ACM.
20. C. Okasaki. Purely functional random-access lists. In *Functional Programming Languages and Computer Architecture*, pages 86–95, 1995.
21. T. Takaoka. Efficient algorithms for the maximum subarray problem by distance matrix multiplication. *Electr. Notes Theor. Comput. Sci.*, 61, 2002.
22. H. Tamaki and T. Tokuyama. Algorithms for the maximum subarray problem based on matrix multiplication. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 446–452, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.
23. R. Y. Walder, M. R. Garrett, A. M. McClain, G. E. Beck, T. M. Brennan, N. A. Kramer, A. B. Kanis, A. L. Mark, J. P. Rapp, and V. C. Sheffield. Short tandem repeat polymorphic markers for the rat genome from marker-selected libraries. *Mammalian Genome*, 9(12):1013–1021, December 1998.