Efficient Algorithms for the Sum Selection Problem and K Maximum Sums Problem*

Tien-Ching Lin** and D.T. Lee**

Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan
{kero, dtlee}@iis.sinica.edu.tw

Abstract. Given a sequence of n real numbers $A = a_1, a_2, \ldots, a_n$ and a positive integer k, the Sum Selection Problem is to find the segment $A(i,j) = a_i, a_{i+1}, \ldots, a_j$ such that the rank of the sum $s(i,j) = \sum_{t=i}^{j} a_t$ is k over all $\frac{n(n-1)}{2}$ segments. We present a deterministic algorithm for this problem that runs in $O(n \log n)$ time. The previously best known randomized algorithm for this problem runs in expected $O(n \log n)$ time. Applying this algorithm we can obtain a deterministic algorithm for the k Maximum Sums Problem, i.e., the problem of enumerating the k largest sum segments, that runs in $O(n \log n + k)$ time. The previously best known randomized and deterministic algorithms for the k Maximum Sums Problem run respectively in expected $O(n \log n + k)$ and $O(n \log^2 n + k)$ time in the worst case.

Keywords: k maximum sums problem, sum selection problem, maximum sum problem, maximum sum subarray problem.

1 Introduction

Given a sequence of n real numbers $A = a_1, a_2, \ldots, a_n$, the Maximum Sum Problem is to find the segment $A(i,j) = a_i, a_{i+1}, \ldots, a_j$ whose sum $s(i,j) = \sum_{t=i}^{j} a_t$ is the maximum among all possible $1 \leq i \leq j \leq n$. This problem was first introduced by Bentley [6,7] and can be easily solved in $O(n)$ time [7,13].

Given an $m \times n$ matrix of real numbers (assuming that $m \leq n$), the Maximum Sum Subarray Problem is is to find the submatrix, the sum of whose entries is the maximum among all $O(m^2 n^2)$ submatries. The problem can be solved in $O(m^2 n)$ time [7,13,18]. Tamaki and Tokuyama [19] gave the first sub-cubic time algorithm for this problem and Takaoka [20] later gave a simplified algorithm achieving sub-cubic time as well. Many parallel algorithms under different parallel models of computation were also obtained [3,16,17,18].

* Research supported in part by the National Science Council under the Grants No. NSC-94-2213-E-001-004, NSC-95-2221-E-001-016-MY3, and NSC 94-2752-E-002-005-PAE, and by the Taiwan Information Security Center (TWISC), National Science Council under the Grant No. NSC94-3114-P-001-001-Y.

** Also with Institute of Information Science, Academia Sinica, Nankang, Taipei 115, Taiwan.

T. Asano (Ed.): ISAAC 2006, LNCS 4288, pp. 460–473, 2006.

The Maximum Sum Problem can find many applications in pattern recognition, image processing and data mining [1,12]. A natural generalization of the above Maximum Sum Problem is the k Maximum Sums Problem which is to find the k segments such that their sums are the k largest over all $\frac{n(n-1)}{2}$ segments. Bae and Takaoka [4] presented an $O(kn)$ time algorithm for this problem. Bengtsson and Chen [5] gave an $O(\min\{k+n\log^2 n, nk^{\frac{1}{2}}\})$ time algorithm, or $O(n\log^2 n + k)$ time in the worst case. Cheng et al. [8] recently gave an $O(n + k\log(min\{n,k\}))$ time algorithm for this problem which is superior to Bengtsson and Chen's when k is $o(n\log n)$, but it runs in $O(n^2\log n)$ time in the worst case. Lin and Lee [14] recently gave an expected $O(n\log n + k)$ time randomized algorithm based on a randomized algorithm which finds in expected $O(n\log n)$ time the segment whose sum is the k-th smallest, for any given positive integer $1 \le k \le \frac{n(n-1)}{2}$. The latter problem is referred to as the Sum Selection Problem. In this paper we will give a deterministic $O(n\log n + k)$ time algorithm for the k Maximum Sums Problem based on a deterministic $O(n\log n)$ time algorithm for the Sum Selection Problem as well.

The rest of the paper is organized as follows. Section 2 give a deterministic algorithm for the Sum Selection Problem. Section 3 gives a deterministic algorithm for the k Maximum Sums Problem. Section 4 gives some conclusion.

2 Algorithm for the Sum Selection Problem

We define the *rank* $r(x,P)$ of an element x in a set $P \subseteq \mathbf{R}$ of real numbers to be the number of elements in P no greater than x, i.e. $r(x,P) = |\{y|y \in P, y \le x\}|$. Given a sequence A of real numbers $a_1, a_2, \ldots, a_n$, and a positive integer $1 \le k \le \frac{n(n-1)}{2}$, the Sum Selection Problem is to find the segment $A(i^*, j^*)$ over all $\frac{n(n-1)}{2}$ segments such that the rank of the sum $s(i^*, j^*) = \sum_{t=i^*}^{j^*} a_t$ in the set of possible subsequence sums is k. That is, we would like to find $s^* = s(i^*, j^*)$ for some $i^* < j^*$ such that $r(s^*, P) = k$ where $P = \{s(i,j) \mid s(i,j) = \sum_{t=i}^{j} a_t, 1 \le i \le j \le n\}$.

We will transform the Sum Selection Problem into a problem of arrangements of lines in computational geometry in $O(n)$ time as follows. We first define the set $S = \{s_0, s_1, \ldots, s_n\}$ according to the prefix sums of the sequence A, where $s_i = \sum_{t=1}^{i} a_t$, $i = 1, 2, \ldots, n$ and $s_0 = 0$. We then define two sets of lines $H = \{h_i \mid h_i : y = -s_i, i = 0, 1, \ldots, n\}$ and $V = \{v_i \mid v_i : y = x - s_i, i = 0, 1, \ldots, n\}$ in the plane respectively. For any two lines $h_i \in H$ and $v_j \in V$ with $i < j$, they intersect at the point $p_{ij} = (x_{ij}, y_{ij})$ with abscissa $x_{ij} = s_j - s_i$. It means that the abscissa of the intersection point of any two lines $h_i \in H$ and $v_j \in V$ with $i < j$ is equal to the sum $s(i+1, j)$ of the segment $A(i+1, j)$. We say that an intersection point of two lines $h_i \in H$ and $v_j \in V$ is *feasible* if $i < j$. Note that there are totally n^2 intersection points in the arrangements of lines $\mathcal{A}(H \cup V)$ and it contains $\frac{n(n-1)}{2}$ feasible intersection points and $\frac{n(n+1)}{2}$ non-feasible intersection points. An example of the arrangements of lines $\mathcal{A}(H \cup V)$ is shown in Figure 1. Let $X_f = \{x_{ij} \mid p_{ij} = (x_{ij}, y_{ij})$ is a feasible intersection point of

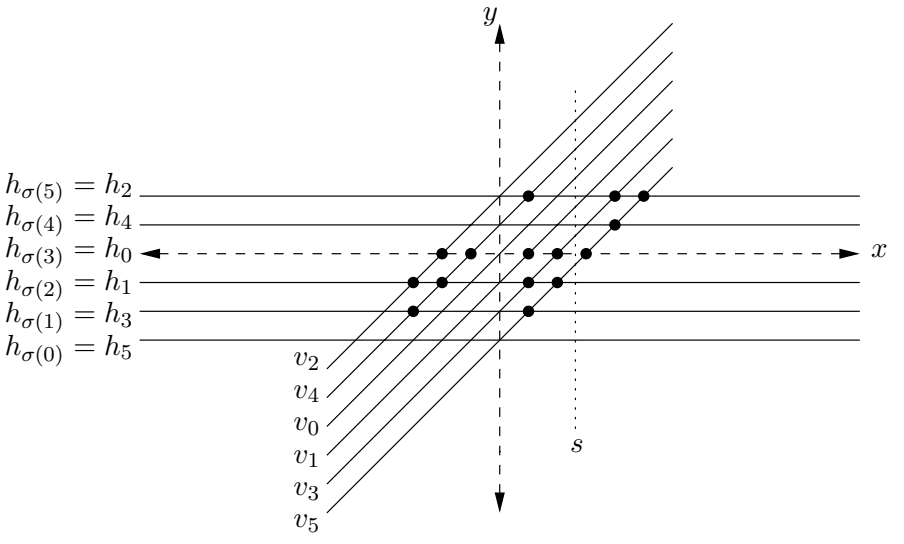

Fig. 1. Given $A = a_1, a_2, a_3, a_4, a_5 = 1, -3, 4, -3, 4$, we have $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\} = \{0, 1, -2, 2, -1, 3\}$, $H = \{h_i \mid h_i : y = -s_i, i = 0, 1, \ldots, 6\}$ and $V = \{v_i \mid v_i : y = x - s_i, i = 0, 1, \ldots, 6\}$ respectively. The intersection points shown in dark solid dots are the feasible intersection points and others are the non-feasible intersection points.

$\mathcal{A}(H \cup V)\}$. The Sum Selection Problem now is equivalent to the following problem.

> Given a set of lines $L = H \cup V = \{h_0, v_0, h_1, v_1, \ldots, h_n, v_n\}$ in $\mathbf{R}^2$, where $h_i : y = -s_i$ and $v_j : y = x - s_j$, find the feasible intersection point $p_{i^*j^*} = (x_{i^*j^*}, y_{i^*j^*})$ such that $r(x_{i^*j^*}, X_f) = k$.

Given a set of n lines in the plane and an integer k, $1 \leq k \leq \frac{n(n-1)}{2}$, the well-known dual problem of the Slope Selection Problem[1] in computational geometry is to find the intersection point whose x-coordinate is the k-th smallest among all intersection points of these n lines. Cole et al. [10] develop an approximate counting scheme combining the AKS sorting network and parametric search to obtain an optimal $O(n \log n)$ algorithm for this problem. Brönnimann and Chazelle [9] modify their approximate counting scheme combining ε-net to obtain another optimal algorithm for this problem. The Sum Selection Problem can be viewed as a variant of the Slope Selection Problem. Since we don't know how many non-feasible intersection points of $\mathcal{A}(L)$ are to the left of the k-th feasible intersection point, and thus we don't know the actual rank of

[1] Given a set of n points in the plane and an integer k, $1 \leq k \leq \frac{n(n-1)}{2}$, the slope selection problem is to select the pair of points determining the line with the k-th smallest slope.

the k-th feasible intersection point in the set of all intersection points of $\mathcal{A}(L)$. The actual rank of the k-th feasible intersection point may lie between k and $k + \frac{n(n+1)}{2}$ in the set of all intersection points of $\mathcal{A}(L)$. Therefore, we can not solve the SUM SELECTION PROBLEM by fixing some specific rank and applying the slope selection algorithms [9,10] directly. We will give a deterministic algorithm for this problem that runs in $O(n \log n)$ time based on the ingenious parametric search technique of Megiddo [15], AKS sorting network [2] and a new approximate counting scheme. This new approximate counting scheme is a generalization of the approximate counting schemes developed by Cole et al. [10] and Brönnimann and Chazelle [9].

Given a vertical line $x = s$, the number of intersection points of $\mathcal{A}(L)$ on it or to its left is denoted $\mathcal{I}(L, s)$ and the number of feasible intersection points of $\mathcal{A}(L)$ on it or to its left is denoted $\mathcal{I}_f(L, s)$. The vertical order of the lines of L at $x = s$ defines a *permutation* $\pi(s)$ of L at s with $\pi(-\infty)$ being the identity permutation. An example of $\pi(s) = (h_5, h_3, h_1, v_5, h_0, v_3, h_4, v_1, h_2, v_0, v_4, v_2)$ is shown in Figure 1. An *inversion* of a permutation $(p_1, p_2, \ldots, p_n)$ of $\{1, 2, \ldots, n\}$ is a pair of indices $i < j$ with $p_i > p_j$. It is easy to see that the number of inversions, denoted by $I(\pi(s))$, of a permutation $\pi(s)$ is exactly $\mathcal{I}(L, s)$. We define the number of feasible inversions, denoted by $I_f(\pi(s))$, of $\pi(s)$ to be $\mathcal{I}_f(L, s)$. Therefore, the SUM SELECTION PROBLEM is also equivalent to finding some s^* such that $I_f(\pi(s^*)) = k$.

The problem for finding s^* can be viewed as an unusual sorting problem attempting to sort the set of lines L at $x = s^*$ without knowing the value of s^*, i.e. to sort $h_0(s^*), v_0(s^*), h_1(s^*), v_1(s^*), \ldots, h_n(s^*), v_n(s^*)$ in vertical order without knowing the value of s^*. We know that this sort may be achieved in $O(n \log n)$ comparisons. In particular, the questions of the forms "$h_i(s^*) \leq h_j(s^*)$" and "$v_i(s^*) \leq v_j(s^*)$" can be solved in $O(n \log n)$ time by any usual optimal sorting algorithm, since the ordering of h_i's, which is identical to that of v_j's, is independent of s^*. However, the question, q_{ij} of the form "$h_i(s^*) \leq v_j(s^*)$", can be answered by a counting subroutine that given any vertical line $x = s$ it can quickly compute $\mathcal{I}_f(L, s)$, the number of feasible intersection points of $\mathcal{A}(L)$ that lie on it or to its left. There is a simple way to perform this task in $O(n \log n)$ time by Lemma 1 with $s_\ell = -\infty$ and $s_r = s$. Even though we don't know s^*, we can answer the question q_{ij} by finding the x_{ij}, the x-coordinate of intersection point of h_i and v_j in constant time and call the counting subroutine at $x = x_{ij}$. If the return of the subroutine is less than or equal to k, we get $h_i(s^*) \leq v_j(s^*)$. Otherwise we get $h_i(s^*) > v_j(s^*)$. After solving the unusual sorting problem we can obtain the permutation $\pi(s^*)$ without knowing the value of s^*. Then, we can obtain $s^* = \max\{x_{\pi(s^*)[i]\pi(s^*)[i+1]}\}$.

Lemma 1. *([14], Lemma 2) Given a sequence A of n real numbers $a_1, a_2, \ldots, a_n$ and two real numbers s_ℓ, s_r with $s_\ell \leq s_r$, it takes $O(n)$ space and $O(n \log n)$ time to count the total number of segments $A(i, j)$, $1 \leq i \leq j \leq n$, among all $\frac{n(n-1)}{2}$ segments such that their sums $s(i, j)$ satisfy $s_\ell \leq s(i, j) \leq s_r$.*

How can we solve the unusual sorting problem? We will use the parametric search approach running a sequential simulation of a generic parallel sorting

algorithm, which attempts to sort the lines along the line $x = s^*$, where s^* is the x-coordinate of the desired k-th leftmost feasible intersection point, without knowing the value of s^*. A naive algorithm is to use a parallel sorting algorithm of depth $O(\log n)$ and $O(n)$ processors developed by Ajtai, Komlós, and Szemerédi [2], and at each parallel step we may perform $\frac{n}{2}$ comparisons between pairs of lines. Since each comparison can be solved in $O(n \log n)$ time and $O(n)$ space following Lemma 1, it takes $O(n^2 \log n)$ time at each parallel step, and $O(n^2 \log^2 n)$ time overall.

However, we can improve it by the following slightly complicated algorithm. That is, we compute the median x_m of the x-coordinates of all the intersection points of these $\frac{n}{2}$ pairs of lines in each parallel step, and call the counting subroutine at x_m, which can answer half of the questions in $O(n \log n)$ time. For the $\frac{n}{4}$ unresolved questions at the same step, we again find the median among the $\frac{n}{4}$ x-coordinates and call the counting subroutine at the median, which can answer half of these $\frac{n}{4}$ unresolved questions in $O(n \log n)$ time. Repeating the above *binary search* process $O(\log n)$ times we can answer all $\frac{n}{2}$ comparisons in $O(n \log^2 n)$ time in each parallel step. We thus obtain an algorithm that runs in $O(n \log^3 n)$ time.

We can further improve $O(n \log^3 n)$ to $O(n \log^2 n)$ by using a technique due to Cole [11] as follows. Instead of invoking $O(\log n)$ counting subroutine calls at each parallel step to resolve all comparisons at this step, we call the counting subroutine only a constant number of times. This of course does not resolve all comparisons of this parallel step, but it does resolve a large fraction of them. All the unresolved comparisons at this step will be deferred to the next parallel step. Suppose that each of the unresolved comparisons can affect only a constant number of comparisons executed at the next parallel step. Each parallel step is now a mixture of many parallel steps. Cole shows that if it is implemented carefully by assigning an appropriate time-dependent weight to each unresolved comparison and choosing the weighted median at each step of the binary search, the number of the parallel steps of the algorithm increases only by an additive $O(\log n)$ steps. Since each of these steps uses only a constant number of counting subroutine calls, the whole running time improves to $O(n \log^2 n)$.

The final step to improve the sum selection algorithm from $O(n \log^2 n)$ to $O(n \log n)$ is to develop an approximate counting scheme. Note that the expensive counting subroutine, Lemma 1, can be used not only to find $\mathcal{I}_f(L, s)$ for each point s given by the sorting network but also to determine the relative ordering of s and s^* in $O(n \log n)$ time. Instead of invoking the expensive counting subroutines $O(\log n)$ times, we shall develop an approximate counting scheme, that counts the number of inversions of desired permutations only approximately, with an error that gets smaller and smaller as we get closer to the desired s^*. The idea of the approximate counting scheme is to use an approximation algorithm in $O(n)$ time for each point s chosen by the sorting network. If the error for the approximation algorithm is small enough, then we can decide the relative ordering of s and s^* directly. Otherwise, we will refine the approximation until we can decide the relative ordering of s and s^*. It turns out that an amortized

$O(n \log n)$ extra time is sufficient to refine approximations throughout the entire course of the algorithm.

We first define an m-block *left-compatible* (resp. *right-compatible*) permutation $\pi_l(s)$ (resp. $\pi_r(s)$) of permutation $\pi(s)$ such that it satisfies $I(\pi_l(s)) \leq I(\pi(s)) \leq I(\pi_l(s)) + mn$ (resp. $I(\pi_r(s)) - mn \leq I(\pi(s)) \leq I(\pi_r(s))$). Let $(\sigma(0), \sigma(1), \ldots, \sigma(n))$ denote the permutation of $\{0, 1, \ldots, n\}$ such that $h_{\sigma(0)}$, $h_{\sigma(1)}, \ldots, h_{\sigma(n)}$ are in the ascending vertical order, i.e. $-s_{\sigma(0)} \leq -s_{\sigma(1)} \leq \ldots \leq -s_{\sigma(n)}$. Let $G_0, G_1, \ldots, G_{\frac{n}{m}}$ be an *m-block* of H for some fixed size m, where group $G_i = \{h_{\sigma(i \cdot m)}, h_{\sigma(i \cdot m+1)}, \ldots, h_{\sigma(i \cdot m+m-1)}\}$. For any i, j, we say that v_i is greater than group G_j at s, denoted by $v_i(s) \succ G_j$, if $v_i(s) = x - s_i > h_{\sigma(j \cdot m+m-1)}(s) = -s_{\sigma(j \cdot m+m-1)}$ where $h_{\sigma(j \cdot m+m-1)}$ is the largest element in group G_j, and we say that v_i is in group G_j at s, denoted by $v_i(s) \sqsubset G_j$, if $h_{\sigma((j-1) \cdot m+m-1)}(s) < v_i(s) \leq h_{\sigma(j \cdot m+m-1)}(s)$.

We define a permutation $\pi_l(s)$ (resp. $\pi_r(s)$) as an m-block *left-compatible* (resp. *right-compatible*) permutation of $\pi(s)$ as follows: Given $h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)}$ sorted in ascending σ-order, $\pi_l(s)$ and $\pi_r(s)$ will be obtained by inserting $v_{\sigma(i)}$, $i = 0, 1, \ldots, n$, one by one in between $h_{\sigma(q_i)}$ and $h_{\sigma(q_i+1)}$ for some q_i such that $v_{\sigma(i)}$'s, $i = 0, 1, \ldots, n$, are also in ascending σ-order. For each $v_{\sigma(i)} \in V$, if $v_{\sigma(i)}(s) \sqsubset G_j$ for some j then we insert $v_{\sigma(i)}$ in between $h_{\sigma((j-1) \cdot m+m-1)}$ and $h_{\sigma(j \cdot m)}$, where $h_{\sigma((j-1) \cdot m+m-1)}$ is the largest element in group G_{j-1} and $h_{\sigma(j \cdot m)}$ is the smallest element in group G_j. (resp. For each $v_{\sigma(i)} \in V$, if $v_{\sigma(i)}(s) \sqsubset G_j$ for some j then we insert $v_{\sigma(i)}$ in between $h_{\sigma(j \cdot m+m-1)}$ and $h_{\sigma((j+1) \cdot m)}$, where $h_{\sigma(j \cdot m+m-1)}$ is the largest element in group G_j and $h_{\sigma((j+1) \cdot m)}$ is the smallest element in group G_{j+1}.) For example, the 2-block left-compatible permutation $\pi_l(s) = (h_5, h_3, v_5, h_1, h_0, v_3, v_1, h_4, h_2, v_0, v_4, v_2)$ and right-compatible permutation $\pi_r(s) = (h_5, h_3, h_1, h_0, v_5, h_4, h_2, v_3, v_1, v_0, v_4, v_2)$ in Figure 1. Therefore, we have

$$I(\pi_l(s)) \leq I(\pi(s)) \leq I(\pi_l(s)) + mn,\ I_f(\pi_l(s)) \leq I_f(\pi(s)) \leq I_f(\pi_l(s)) + mn.$$

$$I(\pi_r(s)) - mn \leq I(\pi(s)) \leq I(\pi_r(s)),\ I_f(\pi_r(s)) - mn \leq I_f(\pi(s)) \leq I_f(\pi_r(s)).$$

Thus, we see that maintaining left-compatible (right-compatible) permutation with $\pi(s)$ gives a good approximation on the number of inversions of the permutation: the smaller the block size m, the finer the approximation.

We now give an $O(n \log n)$ algorithm for the Sum Selection Problem as follows. We will first sketch the algorithm and then explain and analyze it in detail in subsequent paragraphs. We assume, for simplicity, that $n = 2^g$ for some integer g and the fractions in this algorithm are integers taken by floor or ceiling functions. We define $sign(s)$ to be 1 if s is a positive real number, 0 if s is zero and -1 if s is a negative real number. The algorithm maintains an interval (s_l, s_r) containing s^*, an m_l-block left-compatible permutation $\pi_l(s_l)$ at s_l and an m_r-block right-compatible permutation $\pi_r(s_r)$ at s_r such that they satisfy invariant conditions $(I1)$ and $(I2)$. An example of an interval (s_l, s_r) containing s^* is shown in Figure 2.

$(I1)$ $I_f(\pi_l(s_l)) + m_l n \leq I_f(\pi(s^*)) \leq I_f(\pi_r(s_r)) - m_r n.$

$(I2)$ $I_f(\pi_r(s_r)) - 2m_r n \leq I_f(\pi(s^*)) \leq I_f(\pi_l(s_l)) + 2m_l n.$

$(I1)$ means that s^* lies within the interval (s_l, s_r). Since $I_f(\pi(s_l)) \leq I_f(\pi_l(s_l)) + m_l n \leq I_f(\pi(s^*)) \leq I_f(\pi_r(s_r)) - m_r n \leq I_f(\pi(s_r))$, we have $s_l \leq s^* \leq s_r$. $(I2)$ means that the left-compatible and right-compatible permutations are no finer than needed.

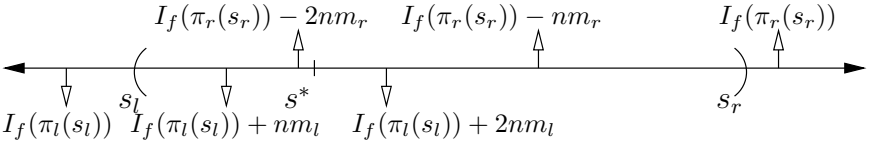

Fig. 2. The sum selection algorithm maintains an interval (s_l, s_r) containing s^* satisfying invariant conditions $(I1)$ and $(I2)$

If $k < n$, then we can solve the sum selection problem by using the algorithm due to Cheng et al. [8]. Let us assume $k \geq n$ in the following. To initialize the algorithm we set $m_l = \frac{k}{n}$, $s_l = -\infty$, $\pi_l(s_l) = (v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)}, h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)})$, $I_f(\pi_l(s_l)) = 0$, $m_r = \frac{(n-1)}{4} - \frac{k}{2n}$, $s_r = \infty$, $\pi_r(s_r) = (h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)}, v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)})$, $I_f(\pi_r(s_r)) = \frac{n(n-1)}{2}$ and $I_f(\pi(s^*)) = k$. It is easy to check that this initial condition satisfies $(I1)$ and $(I2)$.

After coming in a new point s from AKS network combining Cole's technique, we will decide an interval, called *winning interval*, which contains s^* between (s_l, s) and (s, s_r) and maintain invariant conditions $(I1)$ and $(I2)$ for the winning interval. In order to decide the winning interval and maintain $(I1)$ and $(I2)$, we need the following four subroutines, each costing $O(n)$ time. The left reblocking subroutine allows us to construct an m_l-block left-compatible permutation $\pi_l(s)$ at s when $I_f(\pi_l(s_l)) + 2m_l n \geq I_f(\pi_l(s))$ holds. We will show later that if $I_f(\pi_l(s_l)) + 2m_l n < I_f(\pi_l(s))$ then (s, s_r) can not be the winning interval so we don't need to construct $\pi_l(s)$. The right reblocking subroutine allows us to construct an m_r-block right-compatible permutation $\pi_r(s)$ at s when $I_f(\pi_r(s)) \geq I_f(\pi_r(s_r)) - 2m_r n$ holds. We will also show later that if $I_f(\pi_r(s)) < I_f(\pi_r(s_r)) - 2m_r n$ then (s_l, s) can not be the winning interval so we don't need to construct $\pi_r(s)$. The left halving subroutine is to construct a $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$ at s. The right halving subroutine is to construct a $\frac{m_r}{2}$-block right-compatible $\pi_r(s)$ at s.

After coming in a new point s, we first do left reblocking m_l and right reblocking m_r at s to construct $\pi_l(s)$ and $\pi_r(s)$ respectively. It divides into three cases: For the case 1: If $I_f(\pi_l(s_l)) + 2m_l n < I_f(\pi_l(s))$ then (s_l, s) will be the winning interval. But if the winning interval (s_l, s) doesn't satisfy $(I1)$ and $(I2)$, then we will do the right halving $\frac{m_r}{2^1}$, $\frac{m_r}{2^2}$, ... until $\frac{m_r}{2^t}$ such that $(I1)$ and $(I2)$ hold for (s_l, s). For the case 2: If $I_f(\pi_r(s)) < I_f(\pi_r(s_r)) - 2m_r n$ then (s, s_r) will

be the winning interval. But if the winning interval (s, s_r) doesn't satisfy $(I1)$ and $(I2)$, then we will do the left halving $\frac{m_l}{2^1}$, $\frac{m_l}{2^2}$, ... until $\frac{m_l}{2^t}$ such that $(I1)$ and $(I2)$ hold for (s, s_r). For the case 3: If $I_f(\pi_l(s_l)) + 2m_l n \geq I_f(\pi_l(s))$ and $I_f(\pi_r(s)) \geq I_f(\pi_r(s_r)) - 2m_r n$ then we can not decide the winning interval yet. We will do the left halving and the right halving interleavingly $\frac{m_l}{2^1}$, $\frac{m_r}{2^1}$, ... until $\frac{m_l}{2^t}$ ($\frac{m_r}{2^t}$) such that $(I1)$ and $(I2)$ hold, then (s, s_r) $((s_l, s))$ will be the winning interval and it satisfies $(I1)$ and $(I2)$ automatically.

After deciding the winning interval, we can decide the relative order of s and s^*. Therefore, we can answer the comparison question at s and the relevant s_i's of which s was the weighted median such that $sign(s - s_i) = sign(s^* - s)$. Then another new point will come in and repeat above procedure again. The algorithm will continue above procedure to make approximations until $m_l < 10$ and $m_r < 10$. When $m_l < 10$ and $m_r < 10$, we know that the winning interval (s_l, s_r) will contain s^* and $O(n)$ feasible intersection points. Let $k^{'}$ be the total number of feasible intersection points in $(-\infty, s_l]$ which can be obtained by the counting subroutine in Lemma 1. Then, we can enumerate all feasible intersection points in the winning interval (s_l, s_r) in $O(n \log n + n) = O(n \log n)$ time by the enumerating subroutine in Lemma 2, and select from those feasible intersection points the $(k - k^{'})$-th feasible intersection point with sum s^* by using any standard selection algorithm in $O(n)$ time. If after the algorithm ends we have either $m_l \geq 10$ or $m_r \geq 10$, at this moment we have solved the unusual sorting problem to obtain $\pi(s^*)$ without knowing the value of s^*. Therefore, we can obtain $s^* = \max\{x_{\pi(s^*)[i]\pi(s^*)[i+1]}\}$.

Lemma 2. *([14], Lemma 1) Given a sequence A of n real numbers $a_1, a_2, \ldots, a_n$ and two real numbers s_ℓ, s_r with $s_\ell \leq s_r$, it costs $O(n)$ space and $O(n \log n + h)$ time, where h is the output size, to find all segments $A(i, j)$, $1 \leq i \leq j \leq n$, among all $\frac{n(n-1)}{2}$ segments such that their sums $s(i, j)$ satisfy $s_\ell \leq s(i, j) \leq s_r$.*

We now develop the left reblocking, right reblocking, left halving and right halving subroutines and then explain the algorithm and analyze its complexity in detail. We develop left reblocking subroutine as an example since the right reblocking subroutine can be done similarly. The left reblocking subroutine will either construct an m_l-block left-compatible permutation $\pi_l(s)$ when $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$ holds or output "fail" otherwise. Given an m_l-block left-compatible permutation $\pi_l(s_l)$, the left reblocking subroutine is to find the m_l-block left-compatible permutation $\pi_l(s)$ at s for some $s > s_l$ only if $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$. At the beginning of the subroutine we just know $s > s_l$, but we don't know whether $I_f(\pi_l(s)) - I_f(\pi_l(s_l))$ is greater than $2m_l n$ or not. But once we found that $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) > 2m_l n$ during running the left reblocking subroutine, we will halt the subroutine immediately and output "fail". Assume that we have had an m_l-block $G_1, G_2, \ldots, G_{\frac{n}{m_l}}$ of H, an m_l-block left-compatible permutation $\pi_l(s_l)$ and $I_f(\pi_l(s_l))$ and maintained an array $d_l[i] = j$ at s_l such that $v_i(s_l) \sqsubset G_j$ for each i, we want to find an m_l-block left-compatible permutation $\pi_l(s)$ and $I_f(\pi_l(s))$ and maintain an array $d_l[i] = j$ at s such that $v_i(s) \sqsubset G_j$ for each i. Let us process the lines of L one by one

according the order $v_0, h_0, v_1, h_1, \ldots, v_n, h_n$ to construct $\pi_l(s)$ at s. Initially we set $I_f(\pi_l(s))$ to be $I_f(\pi_l(s_l))$ and current size $c[j] = 0$ for each group G_j, where $c[j]$ denotes the total number of lines in G_j processed so far. While processing v_i we will do the following steps until $v_i(s) \sqsubset G_{d_l[i]}$. If $v_i(s) \succ G_{d_l[i]}$, then $I_f(\pi_l(s))$ is increased by $c[d_l[i]]$ and $d_l[i]$ is increased by 1. While processing h_i, if it is in group G_j then the current size $c[j]$ in group G_j is increased by 1. A detailed description of the left reblocking subroutine is shown in the pseudo code. The whole procedure can be done in $O(n)$ time if $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$. This can be easily seen by the fact that the total processing time is proportional to the number of times each v_i steps up the groups. But doing so increases rank m_l, and we know that there are at most $2m_l n$ rank between $I(\pi_l(s))$ and $I(\pi_l(s_l))$. Therefore, going up the groups cannot happen more than $O(n)$ times. And once we have found that $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) > 2m_l n$, we will halt the subroutine immediately and output "fail". Therefore, it also costs $O(n)$ time if $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) > 2m_l n$. Thus we have Lemma 3.

Lemma 3. (*Reblocking*) *Given an m_l-block left-compatible permutation $\pi_l(s_l)$ with approximation rank $I_f(\pi_l(s_l))$, we can compute in $O(n)$ time an m_l-block left-compatible permutation $\pi_l(s)$ with approximation rank $I_f(\pi_l(s))$ for any $s > s_l$ when $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$ holds.*

Subroutine LeftReblocking(s, s_l, m_l, $d_l[\cdot]$).
Input: An m_l-block left-compatible permutation $\pi_l(s_l)$
Output: An m_l-block left-compatible permutation $\pi_l(s)$

1. **for** $i = 0$ **to** n **do** $t[i] \leftarrow d_l[i]$;
2. $I_f(\pi_l(s)) \leftarrow I_f(\pi_l(s_l))$; $g \leftarrow 0$;
3. **for** $i = 0$ **to** $\frac{n}{m_l}$ **do** $c[i] \leftarrow 0$;
4. **for** $i = 0$ **to** n **do**
5. **while** $v_i(s) \succ G_{t[i]}$
6. $I_f(\pi_l(s)) \leftarrow I_f(\pi_l(s)) + c[t[i]]$; $g = g + m_l$; $t[i] \leftarrow t[i] + 1$;
7. **if** $g > 2m_l n$ **then return** fail;
8. **if** $h_i(s)$ is in group G_j **then** $c[j] \leftarrow c[j] + 1$;
9. **for** $i = 0$ **to** $\frac{n}{m_l}$ **do** insert $h_{\sigma(i \cdot m_l)}, h_{\sigma(i \cdot m_l + 1)}, \ldots, h_{\sigma(i \cdot m_l + m_l - 1)}$ one by one into list $B[i]$;
10. **for** $i = 0$ **to** n **do** insert $v_{\sigma(i)}$ into list $B[t[i] - 1]$;
11. Concatenate the lists $B[0], B[1], \ldots, B[\frac{n}{m_l}]$ to obtain $\pi_l(s)$;
12. **for** $i = 0$ **to** n **do** $d_l[i] \leftarrow t[i]$;
13. **return** $\pi_l(s)$;

We develop the left halving subroutine as an example as follows. Given an m_l-block left-compatible permutation $\pi_l(s)$, the left halving subroutine is to find an $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$. Assume that we have had an m_l-block $G_1, G_2, \ldots, G_{\frac{n}{m_l}}$ of H, an m_l-block left-compatible permutation $\pi_l(s)$ and $I_f(\pi_l(s))$ and maintained an array $d_l[i] = j$ at s such that $v_i(s) \sqsubset G_j$ for each i, we want to find a $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$ and $I_f(\pi_l(s))$ and maintain the array $d_l[i]$ at s for each i.

Let $G_1^{'}, G_2^{'}, \ldots, G_{\frac{2n}{m_l}}^{'}$ be a $\frac{m_l}{2}$-block of H. Let us process the lines of L one by one according the order $v_0, h_0, v_1, h_1, \ldots, v_n, h_n$ to construct an $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$ at s. It is easy to see that v_i is either in the group $G_{2d_l[i]}^{'}$ or $G_{2d_l[i]+1}^{'}$. Initially we set $d_l[i]$ to be $2d_l[i]$ for each i, and current size $c[j] = 0$ for each group $G_j^{'}$, where $c[j]$ denotes the total number of lines in $G_j^{'}$ processed so far. While processing v_i, if $v_i(s) \succ G_{d_l[i]}^{'}$, then $I_f(\pi_l(s))$ is increased by $c[d_l[i]]$ and $d_l[i]$ is increased by 1. While processing h_i, if it is in group $G_j^{'}$ then the current size $c[j]$ of group $G_j^{'}$ is increased by 1. A detailed description of the left halving subroutine is shown in the pseudo code. The whole procedure can be done in $O(n)$ time since each v_i steps up at most one group. Thus we have Lemma 4.

Lemma 4. (*Halving*) *Given an m_l-block left-compatible permutation $\pi_l(s)$ with approximation rank $I_f(\pi_l(s))$ for some s, we can compute in $O(n)$ time a $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$ with approximation rank $I_f(\pi_l(s))$.*

Subroutine LeftHalving(s, m_l, $d_l[\cdot]$).
Input: An m_l-block left-compatible permutation $\pi_l(s)$
Output: A $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$

1. $m_l^{'} \leftarrow \frac{m_l}{2}$;
2. **for** $i = 0$ **to** n **do** $d_l[i] \leftarrow 2d_l[i]$;
3. **for** $i = 0$ **to** $\frac{n}{m_l^{'}}$ **do** $c[i] \leftarrow 0$;
4. **for** $i = 0$ **to** n
5. **if** $v_i(s) \succ G_{d_l[i]}^{'}$ **then** $I_f(\pi_l(s)) \leftarrow I_f(\pi_l(s)) + c[d_l[i]]$; $d_l[i] \leftarrow d_l[i] + 1$;
6. **if** h_i is in group $G_j^{'}$ **then** $c[j] \leftarrow c[j] + 1$;
7. **for** $i = 0$ **to** $\frac{n}{m_l^{'}}$ **do** insert $h_{\sigma(i \cdot m_l^{'})}, h_{\sigma(i \cdot m_l^{'}+1)}, \ldots, h_{\sigma(i \cdot m_l^{'}+m_l^{'}-1)}$ one by one into list $B[i]$;
8. **for** $i = 0$ **to** n **do** insert $v_{\sigma(i)}$ into list $B[d_l[i] - 1]$;
9. Concatenate the lists $B[0], B[1], \ldots, B[\frac{n}{m_l^{'}}]$ to obtain $\pi_l(s)$;
10. **return** $\pi_l(s)$;

We now explain the algorithm and analyze its complexity. After coming in a new point s from sorting network, we first do left reblocking m_l and right reblocking m_r at s. If left blocking fails, we have $I_f(\pi(s)) > I_f(\pi_l(s)) > I_f(\pi_l(s_l)) + 2m_l n > I_f(s^*)$. It implies $s > s^*$. Therefore, we can decide (s_l, s) to be the winning interval. But m_r may not be small enough such that (s_l, s) satisfies $(I1)$ and $(I2)$. If so, we do right halving at s until both $(I1)$ and $(I2)$ hold. Similarly, if right blocking fails, we have $I_f(\pi(s)) < I_f(\pi_r(s)) < I_f(\pi_r(s_r)) - 2m_r n < I_f(s^*)$. It implies $s < s^*$. Therefore, we can decide (s, s_r) to be the winning interval. But m_l may not be small enough such that (s, s_r) satisfies $(I1)$ and $(I2)$. If so, we do left halving at s until both $(I1)$ and $(I2)$ hold. If both left blocking and right blocking don't fail then we have $I_f(\pi_l(s_l)) + 2m_l n \geq I_f(\pi_l(s))$ and $I_f(\pi_r(s)) \geq I_f(\pi_r(s_r)) - 2m_r n$. It means that both m_l and m_r are not fine enough to decide the winning interval, so we can not decide the winning interval

yet. We will do left halving and right halving at s interleavingly $\frac{m_l}{2^1}$, $\frac{m_r}{2^1}$, ... until $\frac{m_l}{2^t}$ ($\frac{m_r}{2^t}$) such that both ($I1$) and ($I2$) hold, then (s, s_r) ((s_l, s)) will be the winning interval and it will satisfy ($I1$) and ($I2$) automatically. After deciding the winning interval, we can decide the relative order of s and s^*. Therefore, we can answer the comparison question at s and the relevant s_i's of which s was the weighted median such that $sign(s - s_i) = sign(s^* - s)$. Then another new point will come in and repeat above procedure again.

Since our sorting network is an AKS sorting network combining Cole's technique, the algorithm will invoke $O(1)$ left blocking and right blocking subroutines at each parallel step to resolve all comparisons at this step, each costing $O(n)$, it totally costs $O(n)$ time at each step. The sorting network has depth $O(\log n)$, each parallel step requires $O(n)$, so the algorithm totally costs $O(n \log n)$ time to do left blocking and right blocking. But during the execution of the algorithm the approximation sometimes is not small enough to distinguish the relative ordering of s and s^*, we will refine the approximation until we can decide relative ordering of s and s^*. The algorithm will at most invoke $O(\log n)$ left halving and right halving subroutines, each costing $O(n)$. It turns out that an amortized $O(n \log n)$ extra time will be done to refine approximations throughout the entire course of the algorithm. The correctness of this algorithm follows from the above discussion. Thus, we conclude with the following theorem.

Theorem 1. *The* Sum Selection Problem *can be solved in $O(n)$ space and $O(n \log n)$ time.*

The complete pseudo code of the algorithm follows.

Algorithm Sum Selection Problem.
Input: A set of lines $L = H \cup V = \{h_0, v_0, h_1, v_1, \ldots, h_n, v_n\}$ in $\mathbf{R}^2$, where $h_i : y = -s_i$ and $v_j : y = x - s_j$.
Output: The feasible intersection pt. $p_{i^*j^*} = (x_{i^*j^*}, y_{i^*j^*})$ s.t. $r(x_{i^*j^*}, X_f) = k$.

1. $m_l \leftarrow \frac{k}{n}$; $s_l \leftarrow -\infty$; $\pi_l(s_l) \leftarrow (v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)}, h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)})$;
2. $m_r \leftarrow \frac{(n-1)}{4} - \frac{k}{2n}$; $s_r \leftarrow \infty$; $\pi_r(s_r) \leftarrow (h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)}, v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)})$;
3. **for** $i = 0$ **to** n **do** $d_l[i] \leftarrow 0$; $d_r[i] \leftarrow 0$;
4. **while** $m_l > 10$ **or** $m_r > 10$
5. get next s from AKS network
6. **if** s is not in (s_l, s_r)
7. **then** resolve s and the relevant s_i's such that $sign(s - s_i) = sign(s^* - s)$;
8. **else**
9. **for** $i = 0$ **to** n **do** $t_l[i] \leftarrow d_l[i]$; $t_r[i] \leftarrow d_r[i]$;
10. $m_l' \leftarrow m_l$; $m_r' \leftarrow m_r$;
11. $\pi_l(s) \leftarrow$ LeftBlocking(s, s_l, m_l', $t_l[\cdot]$);
12. $\pi_r(s) \leftarrow$ RightBlocking(s, s_r, m_r', $t_r[\cdot]$);
13. **if** LeftBlocking subroutine outputs "fail"
14. **then**

15. **if** (s_l, s) doesn't satisfy $(I1)$ and $(I2)$
16. **then do** $\pi_r(s) \leftarrow$ RightHalving(s, $m_r^{'}$, $t_r[\cdot]$); $m_r^{'} \leftarrow \frac{m_r^{'}}{2}$; **until** $(I1)$, $(I2)$ hold
17. **if** RightBlocking subroutine outputs "fail"
18. **then**
19. **if** (s, s_r) doesn't satisfy $(I1)$ and $(I2)$
20. **then do** $\pi_l(s) \leftarrow$ LeftHalving(s, $m_l^{'}$, $t_l[\cdot]$); $m_l^{'} \leftarrow \frac{m_l^{'}}{2}$; **until** $(I1)$, $(I2)$ hold
21. **if** LeftBlocking **and** RightBlocking subroutines don't output "fail"
22. **then do**
23. $\pi_l(s) \leftarrow$ LeftHalving(s, $m_l^{'}$, $t_l[\cdot]$); $m_l^{'} \leftarrow \frac{m_l^{'}}{2}$;
24. $\pi_r(s) \leftarrow$ RightHalving(s, $m_r^{'}$, $t_r[\cdot]$); $m_r^{'} \leftarrow \frac{m_r^{'}}{2}$;
25. **until** (s_l, s) satisfies $(I1)$ and $(I2)$ **or** (s, s_r) satisfies $(I1)$ and $(I2)$
26. **if** (s, s_r) satisfies $(I1)$ and $(I2)$
27. **then** $s_l \leftarrow s$; $m_l \leftarrow m_l^{'}$; $d_l[\cdot] \leftarrow t_l[\cdot]$; resolve s and the relevant s_i's such that $sign(s - s_i) = sign(s^* - s)$
28. **else** $s_r \leftarrow s$; $m_r \leftarrow m_r^{'}$; $d_r[\cdot] \leftarrow t_r[\cdot]$; resolve s and the relevant s_i's such that $sign(s - s_i) = sign(s^* - s)$
29. **if** $m_l \leq 10$ **and** $m_r \leq 10$
30. **then**
31. $k^{'} \leftarrow$ total number of feasible points in $(-\infty, s_l]$ by Lemma 1
32. $S \leftarrow$ the set of all feasible points in (s_l, s_r) by Lemma 2
33. **return** $s^* \leftarrow (k - k^{'})$-th element in S by any optimal selection alg.
34. **return** $s^* \leftarrow \max\{x_{\pi(s^*)[i]\pi(s^*)[i+1]}\}$

3 Algorithm for k Maximum Sums Problem

After obtaining the algorithm for the Sum Selection Problem, we can use it to obtain the algorithm for k Maximum Sums Problem directly. We have the following result.

Theorem 2. *The* k Maximum Sums Problem *can be solved in* $O(n)$ *space and* $O(n \log n + k)$ *time.*

Proof. Let $\ell = \frac{n(n-1)}{2} - k + 1$ and $r = \frac{n(n-1)}{2}$. We can run the algorithm of the Sum Selection Problem to obtain the ℓ-th smallest segment s_ℓ and r-th smallest segment s_r respectively in $O(n \log n)$ time and then we can enumerate them by the enumerating subroutine Lemma 2 in the interval $[s_\ell, s_r]$ in $O(n \log n + k)$ time.

4 Conclusion

In the paper we have presented an algorithm for the Sum Selection Problem that runs in $O(n \log n)$ time. We then use it to give a more efficient algorithm for the k Maximum Sums Problem that runs in $O(n \log n + k)$ time. It is

better than the previously best known result for the problem, but whether or not one can prove a $\Omega(n \log n)$ lower bound for the SUM SELECTION PROBLEM is of great interest.

References

1. Agrawal, R., Imielinski, T. Swami, A. Data mining using two-dimensional optimized association rules: scheme, algorithms, and visualization. *Proceedings of the 1993 ACM SIGMOD international conference on management of data*, 207-216, 1993.
2. M. Ajtai, J. Komlós, E. Szemerédi. An $O(n \log n)$ sorting networks. *Combinatorica*, 3:1–19, 1983.
3. Alk, S., Guenther, G. Application of broadcasting with selective reduction to the maximal sum subsegment problem. *International journal of high speed computating*, 3:107-119, 1991.
4. Bae, S. E., Takaoka, T. Algorithms for the problem of k maximum sums and a VLSI algorithm for the k maximum subarrays problem. *2004 International Symposium on Parallel Architectures, Algorithms and Networks*, 247-253, 2004.
5. Bengtsson, F., Chen, J. Efficient Algorithms for K Maximum Sums. *Algorithms and Computation, 15th International Symposium, ISAAC 2004*, 137-148.
6. Bentley, J. Programming perals: algorithm design techniques. *Commun. ACM*, 27, 9:865-873, 1984.
7. Bentley, J. Programming perals: algorithm design techniques. *Commun. ACM*, 27, 11:1087-1092, 1984.
8. Chih-Huai Cheng, Kuan-Yu Chen, Wen-Chin Tien, and Kun-Mao Chao Improved Algorithms for the k Maximum-Sums Problems. *Algorithms and Computation, 16th International Symposium, ISAAC 2005.*
9. H. Brönnimann, B. Chazelle. Optimal slope selection via cuttings. *Computational Geometry*, 10:23–29, 1998.
10. R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemeredi. An optimal-time algorithm for slope selection. *SIAM Journal on Computing*, 18(4):792–810, 1989.
11. R. Cole. Slowing down sorign networks to obtain faster sorting algorithm. *Journal of the Association for Computing Machinery*, Vol. 34, No. 1:200–208, 1987.
12. Fukuda, T., Morimoto, Y., Morishita, S. Tokuyama, T. Mining association rules between sets of items in large databases. *Proceedings of the 1996 ACM SIGMOD international conference on management of data*, 13-23, 1996.
13. Gries, D. A note on the standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207-214, 1982.
14. Tien-Ching Lin, D. T. Lee Randomized algorithm for the Sum Selection Problem. *Algorithms and Computation, 16th International Symposium, ISAAC 2005*, 515-523.
15. N. Megiddo. Applying parallel computation algorithms in the design of serial algorithm. *Journal of the Association for Computing Machinery*, Vol. 30, No. 4:852–865, 1983.
16. Perumalla, K., Deo, N. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5:367-373, 1995.
17. Qiu, K., Alk, S. Parallel maximum sum algorithms on interconnection networks. *Technical Report No. 99-431, Jodrey School of Computer Science, Acadia University, Canada*, 1999.

18. Smith, D. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213-229, 1987.
19. Tamaki, H., Tokuyama, T. Algorithms for the maximum subarray problem based on matrix multiplication. *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, 446-452, 1998.
20. Takaoka, T. Efficient algorithms for the maximum dubarray problem by fistance matrix multiplication. *Proceedings of the 2002 australian theory symposium*, 189-198, 2002.