On the Range Maximum-Sum Segment Query Problem

Kuan-Yu Chen and Kun-Mao Chao*

Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106

June 11, 2004

* Corresponding Author:

Kun-Mao Chao
Professor
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106
Email: kmchao@csie.ntu.edu.tw
Fax: 886-2-23628167

Abstract

We are given a sequence A of n real numbers which is to be preprocessed. In the Range Maximum-Sum Segment Query (RMSQ) problem, a query is comprised of two intervals $[i, j]$ and $[k, l]$ and our goal is to return the maximum-sum segment of A whose starting index lies in $[i, j]$ and ending index lies in $[k, l]$. We propose the first known algorithm for this problem in $O(n)$ preprocessing time and $O(1)$ time per query under the unit-cost RAM model.

Keywords: RMQ, maximum sum interval, sequence analysis

1 Introduction

Sequence analysis in bioinformatics has been studied for decades. One important line of investigation in sequence analysis is to locate the biologically meaningful segments, like conserved regions or GC-rich regions in DNA sequences. A common approach is to assign a real number (also called scores) to each residue, and then look for the maximum-sum or maximum-average segment [3, 4, 9].

Huang [8] extended the well-known recurrence relation used by Bentley [2] for solving the maximum sum consecutive subsequence problem, and derived a linear-time algorithm for computing the optimal segments of lengths at least L. Lin, Jiang, and Chao [9] and Fan *et al.* [4] studied the maximum-sum segment problem of length at least L and at most U. Ruzzo and Tompa [10] gave a linear time algorithm for finding all maximal-sum subsequences.

In this paper, we consider a more general problem in which we wish to find the maximum-sum segment whose starting and ending indices lie in given intervals. By preprocessing the sequence in $O(n)$ time, each query can be answered in $O(1)$ time. This also yields alternative linear-time algorithms both for finding the maximum-sum segment with length constraints and finding all maximal-sum subsequences.

The rest of the paper is organized as follows. Section 2 gives a formal definition of the RMSQ problem and introduces the RMQ techniques [5]. Section 3 considers a the special case of the RMSQ problem (called the SRMSQ problem). Section 4 gives an optimal algorithm for the RMSQ problem. Section 5 uses the RMSQ techniques to solve two relevant problems in linear time.

2 Preliminaries

The input is a nonempty sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ of real numbers. The maximum-sum segment of A is simply the contiguous subsequence having the greatest total sum. For simplicity, throughout the paper, the term "subsequence" will be taken to mean "contiguous subsequence". To avoid ambiguity, we disallow nonempty, zero-sum prefix or suffix (also called ties) in

the maximum-sum segments. For example, consider the input sequence $A = \langle 4, -5, 2, -2, 4, 3, -2, 6\rangle$. The maximum-sum segment of A is $M = \langle 4, 3, -2, 6\rangle$, with a total sum of 11. There is another subsequence tied for this sum by appending $\langle 2, -2\rangle$ to the left end of M, but this subsequence is not the maximum-sum segment since it has a nonempty zero-sum prefix.

Let $A(i, j)$ denote the subsequence $\langle a_i, \ldots, a_j\rangle$ and $S(i, j)$ denote the sum of $A(i, j)$, i.e. $S(i, j) = \sum_{i \leq k \leq j} a_k$. Let $c[i]$ denote the cumulative sum of A, i.e. $c[i] = \sum_{1 \leq k \leq i} a_k$ for all $1 \leqslant i \leqslant n$ and $c[0] = 0$. It's easy to see that $S(i, j) = c[j] - c[i-1]$ for $1 \leqslant i \leqslant n$.

2.1 Problem Definitions

We start with a special case of the RMSQ problem, called the SRMSQ problem.

Problem 1. *A Special Case of the RMSQ problem (SRMSQ)*
Input to be preprocessed: *A nonempty sequence of n real numbers $A = \langle a_1, a_2, \ldots, a_n\rangle$.*
Online query: *For an interval $[i, j]$, $1 \leqslant i \leqslant j \leqslant n$, $SRMSQ_A(i, j)$ returns a pair of indices (x, y), $i \leqslant x \leqslant y \leqslant j$, such that $A(x, y)$ is the maximum-sum segment of $A(i, j)$. (When the context is clear, we drop the subscript A of the SRMSQ.)*

A naïve algorithm is to build an $n \times n$ table storing the answers for all possible queries. Each entry (i, j) in the table represents a querying interval $[i, j]$. Since $i \leq j$, we only have to fill in the upper-triangular part of the table. By applying Bentley's linear time algorithm for finding the maximum-sum segment of a sequence, we have an $O(n^3)$-time preprocessing algorithm. Notice that answering an SRMSQ query now requires just one lookup to the table. To achieve $O(n^2)$-time preprocessing rather than the $O(n^3)$-time naïve preprocessing, we use the online manner of the algorithm, filling in the table row-by-row. The total time required is therefore equivalent to the size of the table since each entry can be computed in constant time. In the paper, we give an algorithm that achieves $O(n)$ preprocessing time, and $O(1)$ time per query.

Problem 2. *the Range Maximum-Sum Segment Query problem (RMSQ)*
Input to be preprocessed: *A nonempty sequence of n real numbers $A = \langle a_1, a_2, \ldots, a_n\rangle$.*
Online query: *For two intervals $[i, j]$ and $[k, l]$, $1 \leqslant i \leqslant j \leqslant n$ and $1 \leqslant k \leqslant l \leqslant n$, $RMSQ(i, j, k, l)$ returns a pair of indices (x, y), such that $S(x, y)$ is maximized for $i \leqslant x \leqslant j$ and $k \leqslant y \leqslant l$.*

This is a generalized version of the SRMSQ problem because when $i = k$ and $j = l$, we are actually querying SRMSQ(i, j). A naïve algorithm is to build a 4-dimensional table and the time for preprocessing is $\Omega(n^4)$. We give an algorithm that achieves $O(n)$ preprocessing time and $O(1)$ time query.

2.2 The RMQ Techniques

Now we describe an important technique, called RMQ, used in our algorithm. We are given a sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ to be preprocessed. A Range Minima Query (RMQ) specifies an interval I and the goal is to find the index k with minimum value a_k for $k \in I$.

Lemma 1. *The RMQ problem can be solved in $O(n)$ prerprocessing time and $O(1)$ time per query under the unit-cost RAM model. [1, 5]*

The well known algorithm for the RMQ problem is to first construct the Cartesian tree (defined by Vuillemin in 1980 [12]) of the sequence, then be preprocessed for LCA (Least Common Ancestor) queries [11, 7]. This algorithm can be easily modified to output the index k for which a_k achieves the minimum or the maximum. We let RMQ_{min} denote the minimum query and RMQ_{max} denote the maximum query. That is, $\text{RMQ}_{min}(A, i, j)$ will return index k such that a_k achieves the minimum for $i \leqslant k \leqslant j$, and $\text{RMQ}_{max}(A, i, j)$ will return index k such that a_k achieves the maximum. For the correctness of our algorithm if there are more than one minimum (maximum) in the querying interval, it always outputs the rightmost (leftmost) index k for which a_k achieves the minimum (maximum). This can be done by constructing the Cartesian tree in a particular order.

3 Coping with the SRMSQ Problem

The SRMSQ problem is to answer queries comprised of a single interval. Our strategy for preprocessing is to first find the "good partner" for each index of the sequence such that every such pair will constitute a candidate solution for the maximum-sum segment. Then by applying RMQ techniques one can retrieve the pair with greatest total sum of any given range in constant time. Our first attempt for preprocessing is described as follows.

1. Intuitively, one may pick index l as a partner of k such that $c[l-1]$ is minimized for $1 \leqslant l \leqslant k$ since the sum $S(l, k) = c[k] - c[l-1]$ will then be maximized. Record the partner of each index k in an array of length n, say $p[\cdot]$, and the sum $S(l, k)$ in an array of length n, say $m[\cdot]$.

2. Apply RMQ_{max} preprocessing to array $m[\cdot]$ for later retrieve.

It's not hard see that the maximum-sum segment within interval $[1, j]$ for all $1 \leqslant j \leqslant n$ can be retrieved by simply querying $RMQ_{max}(m, 1, j)$. But when it comes to an arbitrary interval $[i, j]$, $1 \leqslant i \leqslant j \leqslant n$, the partners we found might go beyond the left end of $[i, j]$. In this case, we have to "update" the partners since they no longer constitute candidate solutions for the maximum-sum segment of the subsequence $A(i, j)$. Such updates may cost linear per query in the worst case. Hence, the challenge now is to find a somehow "better" partner such that updates for each query can be done in constant time.

Definition 1. *We define $l[j]$ for each index j of A to be the largest index k such that $c[k] \geq c[j]$ and $1 \leqslant k \leqslant j-1$. But if no such k exists, i.e. $c[j] > c[k]$ for all $1 \leqslant k \leqslant j-1$, we assign $l[j] = 0$.*

Such largest index $l[\cdot]$ for each index and the cumulative sum $c[\cdot]$ can be computed by PREPROCESS1 in Fig. 1.

Algorithm PREPROCESS1
Input: An nonempty array of n real numbers $A[1 \ldots n]$.
Output: An array $l[\cdot]$ of length n and an array $c[\cdot]$ of length $n+1$.

```
1  c[0] ← 0;
2  for j ← 1 to n do
3      c[j] ← c[j − 1] + A[j];
4      l[j] ← j − 1;
5      while c[l[j]] < c[j] and l[j] > 0 do
6          l[j] ← l[l[j]];
7      end while
8  end for
```

Figure 1: Algorithm for computing $c[\cdot]$ and $l[\cdot]$

Lemma 2. *The algorithm PREPROCESS1 runs in $O(n)$ time.*

Proof. The total number of operations of the algorithm is clearly bounded by $O(n)$ except for the while-loop body of Steps 5-7. In the following, we show that the amortized cost of the while-loop is a constant. Therefore, the overall time required by the loop is $O(n)$. Let $\Phi(j)$ be the number of "proceedings" of $l[j]$ before it reaches 0, i.e. $\Phi(j)$ is the minimal integer such that $\overbrace{l[\ldots l[l[j]]\ldots]}^{\Phi(j)\ times} = 0$. In each iteration $\Phi(j)$ is increased by 1 and then possibly decreased a bit; however since $\Phi(j)$ can at most be increased by n in total, and can never be negative, it cannot be decreased by more than n times. Thus the while loop is bounded by $O(n)$. □

Definition 2. *We define the "good partner", $p[j]$, of A at index j as the largest index k such that $c[k-1]$ is minimized for $l[j] \leqslant k-1 \leqslant j-1$. And segment $A(p[j], j)$ is called a "candidate segment" of A at index j. We also let $m[j]$ be the sum of the candidate segment, i.e. $m[j] = S(p[j], j)$ for $1 \leqslant j \leqslant n$.*

The relationship between $l[j]$ and $p[j]$ as defined above is illustrated in Fig. 2. And $p[\cdot]$ and $m[\cdot]$ can be computed by PREPROCESS2 in Fig. 3.

The following lemmas show that each pair $(p[j], j)$ constitutes a candidate solution, i.e. $A(p[j], j)$, for the maximum-sum segment of A.

Lemma 3. *If $A(i, j)$ is the maximum-sum segment of A, then $i = p[j]$.*

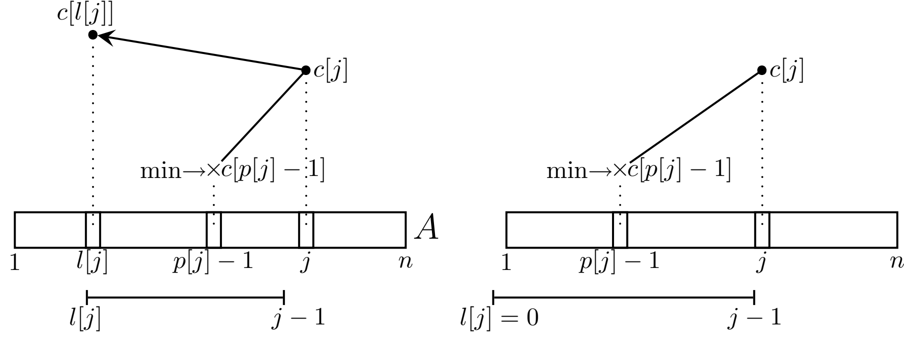


Figure 2: An illustration for $l[\cdot]$ and $p[\cdot]$. Note that y-axis is the value $c[j]$ for the various j's. The left side shows the case that there exists a largest index k such that $c[k] \geq c[j]$ and $1 \leqslant k \leqslant j-1$. And the right side shows the case that $c[j]$ is the unique maximum of $c[k]$ for all $1 \leqslant k \leqslant j$.

Proof. Suppose not, then either (1) index $i-1$ lies in the interval $[0, l[j]-1]$ or (2) index $i-1$ lies in the interval $[l[j], j-1]$ but $c[i-1]$ is not the rightmost minimum of $c[k]$ for all $l[j] \leqslant k \leqslant j-1$. We discuss both cases in the following.

(1) Suppose index $i-1$ lies in the interval $[0, l[j]-1]$. Then $S(i, l[j]) = c[l[j]] - c[i-1] \geq c[j] - c[i-1] = S(i, j)$. The equality must hold for otherwise $A(i, j)$ cannot be the maximum-sum segment of A. Hence $S(l[j]+1, j) = c[j] - c[l[j]] = 0$. It follows $A(l[j]+1, j)$ would be a zero-sum suffix of $A(i, j)$, which contradicts to the definition of the maximum-sum segment.

(2) Suppose index $i-1$ lies in the interval $[l[j], j-1]$. Then $c[i-1]$ must be minimized for $l[j] \leqslant i-1 \leqslant j-1$. Otherwise, $A(i, j)$ cannot be the maximum-sum segment. If $c[i-1]$ is minimized but not the rightmost minimum, i.e. there exists an index $k' > i$ such that $c[k'-1] = c[i-1]$ is also a minimum, then $S(i, k'-1) = c[k'-1] - c[i-1] = 0$, which means $A(i, j)$ has a zero-sum prefix $A(i, k'-1)$.

Hence, by (1) and (2), index i must be the largest index k such that the cumulative sum $c[k-1]$ is minimized for $l[j] \leqslant k-1 \leqslant j-1$, i.e. $i = p[j]$. □

Lemma 4. *If index r satisfies $m[r] \geq m[k]$ for all $1 \leqslant k \leqslant n$, then $A(p[r], r)$ is the maximum-sum segment of A.*

Proof. Suppose on the contrary that segment $A(s, t)$ is the maximum-sum segment and $(s, t) \neq (p[r], r)$. By Lemma 3, we have $s = p[t]$. So

$$m[t] = S(p[t], t) = S(s, t) > S(p[r], r) = m[r]$$

which contradicts to $m[r]$ is the maximum value. □

Algorithm PREPROCESS2
Input: An array of n real numbers $A[1 \dots n]$, array $c[\cdot]$ and array $l[\cdot]$.
Output: Two arrays $p[\cdot]$ and $m[\cdot]$ of length n.

```
1   Apply RMQ_min preprocess to array c[·].
2   for j ← 1 to n do
3       p[j] ← RMQ_min(c, l[j], j − 1) + 1;
4       m[j] ← c[j] − c[p[j] − 1];
5   end for
```

Figure 3: Algorithm for computing $p[\cdot]$ and $m[\cdot]$

Therefore, once we have computed $m[\cdot]$ and $p[\cdot]$ for each index of A, to find the maximum-sum segment of A, we only have to retrieve index r such that $m[r] \geq m[k]$ for all $1 \leqslant k \leqslant n$. Then, candidate segment $A(p[r], r)$ is the maximum-sum segment of A.

Lemma 5. *If $p[j]$ is the good partner of A at index j and $p[j] < j$, then $c[p[j]-1] < c[k] < c[j]$ for all $p[j] \leqslant k \leqslant j-1$.*

Proof. Suppose not. That is, there exists an index k', $p[j] \leqslant k' \leqslant j-1$, such that $c[k'] \leq c[p[j]-1]$ or $c[k'] \geq c[j]$. By the definition of $p[j]$, we know that $c[k'] \leq c[p[j]-1]$ cannot hold. If $c[k'] \geq c[j]$, then again by the definition of $p[j]$, we know $p[j]-1$ must lie in $[k', j-1]$. Thus, $k' \leq p[j]-1 < k'$. A contradiction occurs. □

In other words, $c[p[j]-1]$ is the unique minimum of $c[k]$ for all $p[j]-1 \leqslant k \leqslant j$ and $c[j]$ is the unique maximum. The following key lemma shows the nesting property of the candidate segments. (See Fig. 5 for an illustration of the nesting property.) This important property will make "update in constant time" possible as we will show in later proof.

Lemma 6. *For two indices i and j, $i < j$, if $p[i]$ is the good partner of A at i and $p[j]$ is the good partner of A at j, then it cannot be the case that $p[i] < p[j] \leq i < j$.*

Proof. Suppose $p[i] < p[j] \leq i < j$ holds. By Lemma 5, we have

$$c[p[i]-1] < c[k'] < c[i] \qquad \forall \quad p[i] \leqslant k' \leqslant i-1 \tag{1}$$
$$c[p[j]-1] < c[k''] < c[j] \qquad \forall \quad p[j] \leqslant k'' \leqslant j-1 \tag{2}$$

Since $p[j]-1 < i$, we can substitute $p[j]-1$ for k' in (1) $\Rightarrow c[p[i]-1] < c[p[j]-1] < c[i]$. Similarly, we can substitute i for k'' in (2) $\Rightarrow c[p[j]-1] < c[i] < c[j]$. Then we have

$$c[p[i]-1] < c[p[j]-1] < c[k''] < c[j] \qquad \forall \quad p[j] \leqslant k'' \leqslant j-1 \tag{3}$$
$$c[p[i]-1] < c[k'] < c[i] < c[j] \qquad \forall \quad p[i] \leqslant k' \leqslant i-1 \tag{4}$$

By (3) and (4), we have

$$c[p[i]-1] < c[k'''] < c[j] \qquad \forall \quad p[i] \leqslant k''' \leqslant j-1 \tag{5}$$

By (5), we conclude that $l[j] < p[i-1]$. So $c[p[i]-1] < c[p[j]-1]$ is a contradiction to that $c[p[j]-1]$ minimizes $c[k]$ for all $l[j] \leqslant k \leqslant j-1$. □

Now, we are about to establish the relationship between sequence A and its subsequence $A(i,j)$. The following lemma shows that some good partners of A, say $p[r]$, do not need to be "updated" for the subsequence $A(i,j)$ if $A(p[r],r)$ doesn't go beyond $[i,j]$.

Lemma 7. *If $p[r]$ is the good partner of A at index r and $i \leq p[r] \leq r \leq j$, then $p[r]$ is still the good partner of $A(i,j)$ at index r.*

Proof. Let $c^*[k]$ be the cumulative sum of $A(i,j)$. Then $c^*[k] = c[k] - c[i-1]$ for $i-1 \leqslant k \leqslant j$. Let $l^*[k]$ be the largest index and $p^*[k]$ be the good partner of $A(i,j)$ for $i \leqslant k \leqslant j$, defined similarly as before. Our goal is therefore to prove that $p[r]$ is the largest index k that minimizes $c^*[k-1]$ for $l^*[r] \leqslant k-1 \leqslant r-1$, i.e. $p^*[r] = p[r]$.

If $l[r] \geq i$, we have $l^*[r] = l[r]$ since $l[r]$ is the largest index satisfying $c^*[l[r]] = c[l[r]] - c[i-1] \geq c[r] - c[i-1] = c^*[r]$. Otherwise, i.e. $l[r] < i$, we have $l^*[r] = i-1$. Moreover, since minimizing $c[k-1]$ minimizes $c^*[k-1] = c[k-1] - c[i-1]$, it's not hard to see that $p[r]$ is still the largest index k that minimizes $c^*[k-1]$ for $l^*[j] \leqslant k-1 \leqslant j-1$. □

Corollary 1. *If $p[r]$ is the good partner of A at index r and $i \leq p[r] \leq r \leq j$ and $m[r]$ is the sum of the candidate segment of A at index r, then $m[r]$ is also the sum of the candidate segment of $A(i,j)$ at index r.*

Proof. A direct result of Lemma 7. □

Now, we are ready to present our main algorithm for the SRMSQ problem, which is given in Fig. 4.

For instance, in Fig. 5, the input sequence A has 15 elements. Suppose we are querying SRMSQ$(3,7)$. The algorithm QUERY OF SRMSQ in Fig. 4 first queries the index r such that $m[r]$ is maximized for $3 \leqslant r \leqslant 7$ (line 1). In this case, $r = 5$, which means candidate segment $A(p[5],5)$ has the largest sum compared to other candidate segments whose ending indices lie in $[3,7]$. Since the left end of $A(p[5],5)$, i.e. $p[5] = 3$, lies in the interval $[3,7]$, the algorithm executes line 11, outputs $(p[5],5)$, which means segment $A(3,5)$ is the maximum-sum segment of the subsequence $A(3,7)$.

Suppose we are querying SRMSQ$(6,12)$, RMQ$_{max}(m,6,12)$ returns index 9 (line 1). Since $p[9] = 3 < 6$, lines 3-9 are executed. In line 3, RMQ$_{min}(c,5,8)$ returns index 8. In line 4, RMQ$_{max}(m,10,12)$ will return index 11. In line 5, since $c[9]-c[8] = 6 < m[11] = 8$, the algorithm outputs $(p[11],11)$, which means $A(11,11)$ is the maximum-sum segment of the subsequence $A(6,12)$.

Algorithm PREPROCESS OF SRMSQ(i, j)

1 Run algorithm PREPROCESS1 to compute array $c[\cdot], l[\cdot]$ of A.
2 Run algorithm PREPROCESS2 to compute array $p[\cdot], m[\cdot]$ of A.
3 Apply RMQ$_{max}$ preprocessing to array $m[\cdot]$.
4 Apply RMQ$_{min}$ preprocessing to array $c[\cdot]$.

Algorithm QUERY OF SRMSQ(i, j)

1 r $\leftarrow$ RMQ$_{max}(m, i, j)$
2 **if** $p[r] < i$ **then**
3 $\quad r_1 \leftarrow$ RMQ$_{min}(c, i-1, r-1) + 1$;
4 $\quad r_2 \leftarrow$ RMQ$_{max}(m, r+1, j)$;
5 $\quad$**if** $c[r] - c[r_1 - 1] < m[r_2]$ **then**
6 $\quad\quad$OUTPUT ($p[r_2]$,r_2);
7 $\quad$**else**
8 $\quad\quad$OUTPUT (r_1, r);
9 $\quad$**end if**
10 **else**
11 $\quad$OUTPUT $(p[r], r)$;
12 **end if**

Figure 4: Algorithm for the SRMSQ problem.

Lemma 8. *Algorithm QUERY OF SRMSQ(i, j) outputs the maximum-sum segment of the subsequence $A(i, j)$.*

Proof. Let $c^*[k]$ be the cumulative sum of $A(i, j)$ for $i - 1 \leqslant k \leqslant j$. We have $c^*[k] = c[k] - c[i-1]$, $i - 1 \leqslant k \leqslant j$. Let $p^*[k]$ be the good partner of $A(i, j)$ at index k for $i \leqslant k \leqslant j$. Let $m^*[k] = S(p^*[k], k)$ for $i \leqslant k \leqslant j$. Besides, we let index r satisfy $m[r] \geq m[k]$ for all $i \leqslant k \leqslant j$ (line 1).
(1) If $p[r] \geq i$ (lines 10-11):

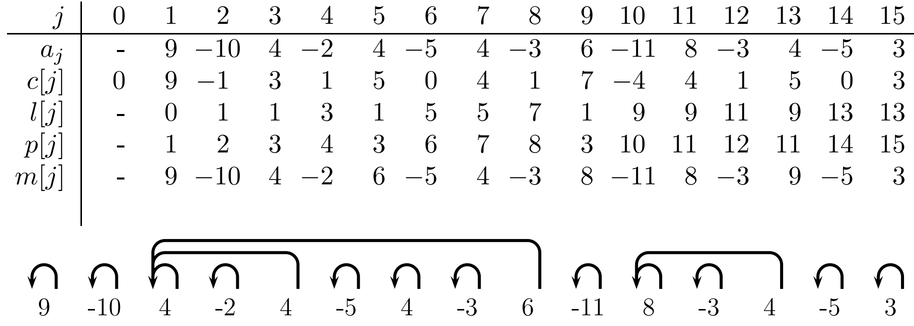
Our goal is to show that $m^*[r] \geq m^*[k]$ for all $i \leqslant k \leqslant j$, and then apply Lemma 4 to prove that $A(p[r], r)$ is the maximum-sum segment of $A(i, j)$.

(a) First, we consider each index k' where $i \leqslant k' \leqslant j$ and $i \leq p[k'] \leq k' \leq j$. By corollary 1, we have $m^*[k'] = m[k'] \leq m[r] = m^*[r]$.

(b) Next, we consider each index k'' where $i \leqslant k'' \leqslant j$ and $p[k''] < i$. By Lemma 5 we have $c[p[k''] - 1] < c[k] < c[k'']$ for all $p[k''] \leqslant k \leqslant k'' - 1$. Since $p^*[k'']$ lies in $[i, k'' - 1]$, we have $c[p[k''] - 1] < c[p^*[k''] - 1]$. Hence, we can deduce that

$$m^*[k''] = c^*[k''] - c^*[p^*[k''] - 1] = c[k''] - c[p^*[k''] - 1] < c[k''] - c[p[k''] - 1] = m[k''].$$

Moreover, because $m[k''] \leq m[r] = m^*[r]$ we have $m^*[k''] < m^*[r]$. Thus, we have shown that $m^*[r] \geq m^*[k]$ for all $i \leqslant k \leqslant j$.
(2) If $p[r] < i$ (lines 2-9):



j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a_j	-	9	-10	4	-2	4	-5	4	-3	6	-11	8	-3	4	-5	3
$c[j]$	0	9	-1	3	1	5	0	4	1	7	-4	4	1	5	0	3
$l[j]$	-	0	1	1	3	1	5	5	7	1	9	9	11	9	13	13
$p[j]$	-	1	2	3	4	3	6	7	8	3	10	11	12	11	14	15
$m[j]$	-	9	-10	4	-2	6	-5	4	-3	8	-11	8	-3	9	-5	3

Figure 5: The candidate segment $A(p[j], j)$ of A for each index j. Notice that, the pointer of each index j points to the position of $p[j]$.

(a) First, we consider each index k' where $i \leqslant k' \leqslant r-1$. By Lemma 5, we have $c[p[r]-1] < c[k] < c[r]$ for all $p[r] \leqslant k \leqslant r-1$. For any index i', $i \leqslant i' \leqslant k'$, since $S(i', k') = c[k'] - c[i'] < c[r] - c[i'] = S(i', r)$, it's not hard to see that k' cannot be the right end of the maximum-sum segment.

(b) Next, we consider each index k'' where $r+1 \leqslant k'' \leqslant j$. By Lemma 6, we know that it cannot be the case $p[r] < p[k''] \leq r < k''$. If $p[k''] \leq r$, then it must be the case $p[k''] \leq p[r] < r < k''$. If so, by Lemma 5 we have $c[p[k''] - 1] < c[r] < c[k'']$ and $c[p[k''] - 1] < c[p[r] - 1] < c[k'']$. Hence, $m[k''] = c[k''] - c[p[k''] - 1] > c[r] - c[p[r] - 1] = m[r]$ which contradicts to that $m[r] \geq m[k]$ for all $i \leqslant k \leqslant j$. Thus, $p[k''] \nleq r$, that is, $p[k''] > r$. By Corollary 1, we have $m^*[k''] = m[k'']$. Let index r_2 satisfy $m[r_2] \geq m[k]$ for all $r+1 \leqslant k \leqslant j$ (line 4). It's not hard to see by Lemma 4 that either $A(r_1, r)$ or $A(p[r_2], r_2)$ is the maximum-sum segment of $A(i, j)$ (lines 5-9). $\square$

As you can see, if $A(p[r], r)$ does not go beyond i, then we are done. Otherwise, it turns out that for each index k', where k' lies in $[i, r-1]$, k' cannot be the right end of the maximum-sum segment of $A(i, j)$. And for each index k'', where k'' lies in $[r+1, j]$, the good partner of k'' doesn't need to be updated. Hence, only the good partner of index r have to be updated. The time required for each query can therefore achieve constant. We summarize our main result in the following theorem.

Theorem 1. *The SRMSQ problem can be solved in $O(n)$ preprocessing time and $O(1)$ time per query.*

4 Coping with the RMSQ Problem

The RMSQ problem is to answer queries comprised of two intervals $[i, j]$ and $[k, l]$, where $[i, j]$ specifies the range of the starting index of the maximum-sum segment, and $[k, l]$ specifies the range of the ending index. Since it is meaningless if the range of the starting index is in front of the range of the ending index,

and vice versa. We assume, without loss of generality, that $i \leq k$ and $j \leq l$. The algorithm for the RMSQ problem is given in Fig. 6.

Algorithm PREPROCESS OF RMSQ
1 Apply SRMSQ preprocessing.
2 Apply RMQ_{max} preprocessing to $c[\cdot]$.

Algorithm QUERY OF RMSQ(i, j, k, l)
1 **if** $j \leq k$ **then**
2 OUTPUT ($\text{RMQ}_{min}(c, i-1, j-1)+1$,$\text{RMQ}_{max}(c, k, l)$)
3 **else**
4 $(r_1,r'_1) \leftarrow (\text{RMQ}_{min}(c, i-1, k-1)+1, \text{RMQ}_{max}(c, k, l))$;
5 $(r_2,r'_2) \leftarrow (\text{RMQ}_{min}(c, k, j-1)+1, \text{RMQ}_{max}(c, j, l))$;
6 $(r_3,r'_3) \leftarrow \text{SRMSQ}(k, j)$;
7 OUTPUT (r_s,r'_s) such that $c[r_s] - c[r'_s]$ is maximized for $1 \leqslant s \leqslant 3$;
8 **end if**

Figure 6: Algorithm for the RMSQ problem.

Lemma 9. *Algorithm QUERY OF RMSQ outputs the correct answer.*

Proof. We discuss it under two possible conditions.
(1) Nonoverlapping case ($j \leq k$):
Suppose the intervals $[i, j]$ and $[k, l]$ do not overlap. Since $S(x, y) = c[y] - c[x-1]$, maximizing $S(x, y)$ is equivalent to maximizing $c[y]$ and minimizing $c[x-1]$ for $i \leq x \leq j$ and $k \leq y \leq l$. By applying RMQ techniques to preprocess $c[\cdot]$, the maximum-sum segment can be easily located (line 2).
(2) Overlapping case ($j > k$):
When it comes to the overlapping case, just to find the maximum cumulative sum and the minimum cumulative sum might go wrong if the minimum is on the right of the maximum. There are three possible cases for the maximum-sum segment $A(x, y)$. (a) Suppose $i \leq x \leq k$ and $k \leq y \leq l$, which is an nonoverlapping case. We find the minimum cumulative sum and the maximum cumulative sum (line 4). (b) Suppose $k+1 \leq x \leq j$ and $j \leq y \leq l$, which is also an nonoverlapping case. We find the minimum cumulative sum and the maximum cumulative sum (line 5). (c) Otherwise, i.e. $k+1 \leq x \leq j$ and $k+1 \leq y \leq j$, which is exactly the same as an $\text{SRMSQ}(k+1, j)$ query (line 6). The maximum sum segment $A(x, y)$ must be one of these three possible cases which have the maximum sum (line 7). □

Theorem 2. *The RMSQ problem can be solved in $O(n)$ preprocessing time and $O(1)$ time per query.*

5 Solving Two Relevant Problems in Linear Time

Given a sequence of n numbers, a lower bound L, and a upper bound U, the first problem is to find the maximum-sum segment of length at least L and at most U [9, 4]. It's not hard to see that it suffices to find for each index j the maximum-sum segment whose starting index lies in $[j-U+1, j-L+1]$ and ending index is j. Since our RMSQ algorithm can answer each such query in $O(1)$ time, the total running time is therefore linear.

The second problem is to find those nonoverlapping, contiguous subsequences having greatest total sum. The highest maximal-sum subsequence is simply the maximum-sum segment of the sequence. The kth maximal-sum subsequence is defined to be the maximum-sum segment disjoint from the $k-1$ maximal-sum subsequences. Additionally, we stop the process when the next best sum is nonpositive. By applying Bentley's linear time algorithm, all maximal-sum subsequences can be found in $O(n^2)$ time in the worst case. Ruzzo and Tompa [10] proposed a genius linear time algorithm for this problem. In the paper, our SRMSQ techniques immediately suggest an alternative divide-and-conquer algorithm for finding all maximal-sum subsequences in linear time: query the maximum-sum segment of the sequence, remove it, and then apply the SRMSQ query recursively to the left of the removed portion, and then to the right. Since our SRMSQ algorithm can answer each such query in $O(1)$ time, the total running time is therefore linear.

Acknowledgements. We thank Yu-Ru Huang, Rung-Ren Lin, Hsueh-I Lu, and An-Chiang Chu for helpful conversations. Kuan-Yu Chen and Kun-Mao Chao were supported in part by an NSC grant 92-2213-E-002-059.

References

[1] M. A. Bender, and M. Farach-Colton. The LCA Problem Revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical INformatics*, 17: 88–94, 2000.

[2] J. Bentley. Programming Pearls - Algorithm Design Techniques, *CACM*, 865–871, 1984.

[3] K. Chung and H.-I. Lu. An Optimal Algorithm for the Maximum-Density Segment Problem. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA 2003)*, LNCS 2832, 136–147, 2003.

[4] 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. *CIAA*, LNCS 2759, 251–257, 2003.

[5] H. Gabow, J. Bentley, and R. Tarjan. Scaling and Related Techniques for Geometry Problems. *Proc. Symp Theory of Computing*(STOC), 135–143, 1984.

[6] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1999.

[7] D. Harel and R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J Comput.*, 13:338–355, 1984.

[8] X. Huang. An algorithm for identifying regions of a DNA sequence that satisfy a content requirement. *CABIOS*, 10:219–225, 1994.

[9] Y.-L. Lin, T. Jiang, and K.-M. Chao. Efficient Algorithms for Locating the Length-constrained Heaviest Segments with Applications to Biomolecular Sequence Analysis. *Journal of Computer and System Sciences*, 65: 570–586, 2002.

[10] W. L. Ruzzo and M. Tompa. A Linear Time Algorithm for Finding All Maximal Scoring Subsequences. In *7th Intl. Conf. Intelligent Systems for Molecular Biology, Heidelberg, Germany*, 234–241, 1999.

[11] B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM J Comput.*, 17:1253–1262, 1988.

[12] J. Vuillemin. A Unifying Look at Data Structures. *CACM*, 23:229–239, 1980.