On the Range Maximum-Sum Segment Query Problem

Kuan-Yu Chen[1] and Kun-Mao Chao[1,2]

[1]Department of Computer Science and Information Engineering
[2]Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106
{r92047, kmchao}@csie.ntu.edu.tw

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. We also use the RMSQ techniques to solve three relevant problems in linear time. These variations on the basic theme demonstrate the utilities of the techniques developed in this paper.

Keywords: Algorithm, 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, 10].

Huang [8] used the expression $x - p \cdot l$ to measure the GC richness of a region, where x is the C+G count of the region, p is a positive constant ratio, and l is the length of the region. 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 [10] and Fan *et al.* [4] studied the maximum-sum segment problem of length at least L and at most U. Ruzzo and Tompa [11] gave a linear time algorithm for finding all maximal-sum subsequences. Wang and Xu [14] proposed a linear time algorithm for finding the longest segment with lower-bound average.

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. We also show that it can be easily utilized to yield alternative linear-time algorithms for finding the maximum-sum segment with length constraints, finding all maximal-sum subsequences, and finding the longest segment with lower-bound average. These variations on the basic theme demonstrate the utilities of the techniques developed here.

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 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 three 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.

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

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) with $i \leqslant x \leqslant y \leqslant j$ such that $A(x, y)$ is the maximum-sum segment of $A(i, j)$.

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 $i \le j$, 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 take advantage of 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$(A, i, j, k, l)$ returns a pair of indices (x, y) with $i \leqslant x \leqslant j$ and $k \leqslant y \leqslant l$ that maximizes $S(x, y)$.

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 per query for this problem.

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 $[i, j]$ and the goal is to find the index k with $i \leqslant k \leqslant j$ such that a_k achieves minimum.

Lemma 1. *The RMQ problem can be solved in $O(n)$ preprocessing 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 [13]) of the sequence, then be preprocessed for LCA (Least Common Ancestor) queries [12, 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, RMQ$_{min}(A, i, j)$ returns index k with $i \leqslant k \leqslant j$ such that a_k achieves the minimum, and RMQ$_{max}(A, i, j)$ returns index k such that a_k achieves the maximum. For 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 index and its partner constitute a candidate solution for the maximum-sum segment. Then by applying RMQ techniques one can retrieve the pair with the greatest total sum within any given interval in constant time. Our first attempt for preprocessing is described as follows.

1. Intuitively, one may pick index p with $1 \leqslant p \leqslant k$ that minimizes $C[p-1]$ as a partner of k, since the sum $S(p,k) = C[k] - C[p-1]$ will then be maximized. Record the partner of each index k in an array of length n, say $P[\cdot]$, and the sum $S(p,k)$ 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 to see that the maximum-sum segment of subsequence $A(1,j)$ for all $1 \leqslant j \leqslant n$ can be retrieved by simply querying $\mathrm{RMQ}_{max}(M,1,j)$. But when it comes to an arbitrary subsequence $A(i,j)$ for all $1 \leqslant i \leqslant j \leqslant n$, the partners we found might go beyond the left end of $[i,j]$. In this case, we have to "update" these 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 arbitrary queries can be done in constant time.

Definition 2. *We define the "left bound" $L[k]$ of A at index k to be the largest index l with $1 \leqslant l \leqslant k-1$ such that $C[l] \geq C[k]$. But if no such l exists, i.e. $C[k] > C[k']$ for all $1 \leqslant k' \leqslant k-1$, we assign $L[k] = 0$.*

Definition 3. *We define the "good partner" $P[k]$ of A at index k to be the largest index p with $L[k]+1 \leqslant p \leqslant k$ that minimizes $C[p-1]$.*

Definition 4. *$A(P[k],k)$ is called a "candidate segment" of A at index k. We define $M[k]$ to be the sum of the candidate segment $A(P[k],k)$, i.e. $M[k] = S(P[k],k)$ for $1 \leqslant k \leqslant n$.*

Instead of finding the partner for each index k of A within the range $[1,k]$, we slightly narrow down the range from $[1,k]$ to $[L[k]+1,k]$. The relationship between $L[\cdot]$ and $P[\cdot]$ defined above is illustrated in Fig. 1. The three arrays $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$ can be computed by COMPUTE-CPM in Fig. 2.

Lemma 2. *The algorithm COMPUTE-CPM correctly computes $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$ in $O(n)$ time.*

Proof. The correctness of the values of $C[\cdot]$ is trivial, and the values of $M[\cdot]$ are correct if the values of $P[\cdot]$ are correct. We first show by induction on k that the algorithm correctly computes array $L[\cdot]$. The basis, $k = 0$, is immediate. In the algorithm, $L[k]$ is the current working pointer scanning from right to left, searching for the largest index whose cumulative sum is greater than or equals to $C[k]$. Since by induction $L[1]$, $L[2]$,..., $L[k-1]$ have been computed correctly, the while-loop examines monotonically increasing values, $C[L[k]]$, $C[L[L[k]]]$,...,$C[L[\ldots L[L[k]]\ldots]]$, until it finally finds one or reaches 0.

Now we discuss the correctness of the values of $P[\cdot]$. By definition, $P[k]$ is the largest index lying in $[L[k]+1,k]$ that minimizes $C[P[k]-1]$. In each iteration of the while-loop, the range where $P[k]$ lies in, i.e. $[L[k]+1,k]$, is about to be extended to $[L[L[k]]+1,k]$. Since by induction $P[L[k]]$ is the largest index lying

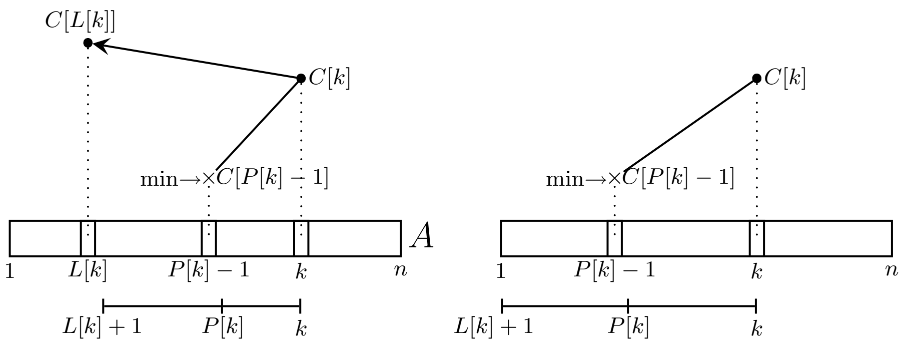

Fig. 1. An illustration for $L[\cdot]$ and $P[\cdot]$. Note that y-axis is the value $C[k]$ for the various k's. The left side shows the case that there exists a largest index l with $1 \leqslant l \leqslant k-1$ such that $C[l] \geq C[k]$. And the right side shows the case that $C[k]$ is the unique maximum of $C[k']$ for all $1 \leqslant k' \leqslant k-1$.

Algorithm COMPUTE-CPM
Input: An nonempty array of n real numbers $A[1 \ldots n]$.
Output: An array $C[\cdot]$ of length $n+1$ and two arrays $P[\cdot]$ and $M[\cdot]$ of length n.

```
1  $C[0] \leftarrow 0$;
2  for $k \leftarrow 1$ to $n$ do
3      $C[k] \leftarrow C[k-1] + A[k]$;
4      $L[k] \leftarrow k-1$;   $P[k] \leftarrow k$;
5      while $C[L[k]] < C[k]$ and $L[k] > 0$ do
6          if $C[P[L[k]]-1] < C[P[k]-1]$ then $P[k] \leftarrow P[L[k]]$;
7          $L[k] \leftarrow L[L[k]]$;
8      end while
9      $M[k] \leftarrow C[k] - C[P[k]-1]$;
10 end for
```

Fig. 2. Algorithm for computing $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$

in $[L[L[k]]+1, L[k]]$ that minimizes $C[P[L[k]]-1]$, the algorithm checks if $P[k]$ needs to be updated in line 6 to guarantee that $C[P[k]-1]$ is minimized for the extended range. Hence, when the while-loop terminates both the values of $L[k]$ and $P[k]$ will be correctly computed.

The total number of operations of the algorithm is clearly bounded by $O(n)$ except for the while-loop body of lines 5-7. In the following, we show that the amortized cost of the while-loop is a constant. Let $\Phi(k)$ be the number of times $L[k]$ may advance forward before it reaches 0, i.e. $\Phi(k)$ is the minimal integer such that $\overbrace{L[...L[L[k]]...]}^{\Phi(k)\ times} = 0$. In each iteration $\Phi(k)$ is increased by one and then possibly decreased a bit; however since $\Phi(k)$ 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 $O(n)$. □

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

Lemma 3. *For two indices p and k, $p \leq k$, if $A(p, k)$ is the maximum-sum segment of A then $p = P[k]$.*

Proof. Suppose not, then either p lies in $[1, L[k]]$ or p lies in $[L[k] + 1, k]$ but $C[p-1]$ is not the rightmost minimum of $C[k']$ for all $L[k] \leqslant k' \leqslant k-1$. We discuss both cases in the following.

1. Suppose index p lies in the interval $[1, L[k]]$. Then $S(p, L[k]) = C[L[k]] - C[p-1] \geq C[k] - C[p-1] = S(p, k)$. The equality must hold for otherwise $A(p, k)$ cannot be the maximum-sum segment of A. It follows $S(L[k]+1, k) = C[k] - C[L[k]] = 0$. Hence $A(L[k]+1, k)$ would be a zero-sum suffix of $A(p, k)$, which contradicts to the definition of the maximum-sum segment.
2. Suppose index p lies in the interval $[L[k] + 1, k]$. We know $C[p-1]$ must be minimized for otherwise $A(p, k)$ cannot be the maximum-sum segment. If $C[p-1]$ is not the rightmost minimum, i.e. there exists an index $k' > p$ such that $C[k'-1] = C[p-1]$ is also a minimum, then $S(p, k'-1) = C[k'-1] - C[p-1] = 0$, which means $A(p, k)$ has a zero-sum prefix $A(p, k'-1)$.

Hence, index p must be the largest index with $L[k] + 1 \leqslant p \leqslant k$ that minimizes $C[p-1]$, i.e. $p = P[k]$. □

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 $A(p, k)$ is the maximum-sum segment of A and $(p, k) \neq (P[r], r)$. By Lemma 3, we have $p = P[k]$. So

$$M[k] = S(P[k], k) = S(p, k) > S(P[r], r) = M[r]$$

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

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 $M[r]$ is the maximum value of $M[k']$ for all $1 \leqslant k' \leqslant n$. Then, candidate segment $A(P[r], r)$ is the maximum-sum segment of A.

Lemma 5. *For an index k, if $P[k] < k$ then $C[P[k]-1] < C[k'] < C[k]$ for all $P[k] \leqslant k' \leqslant k-1$.*

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

In other words, $C[P[k]-1]$ is the unique minimum of $C[k']$ for all $P[k]-1 \leqslant k' \leqslant k$ and $C[k]$ is the unique maximum. The following key lemma shows the nesting property of the candidate segments. (See Fig. 4 for an illustration of the nesting property. This important property makes "update in constant time" possible as we will show in later proof.)

Lemma 6. *For two indices k and l, $k < l$, it cannot be the case that $P[k] < P[l] \leq k < l$.*

Proof. Suppose $P[k] < P[l] \leq k < l$ holds. By Lemma 5, we have $C[P[k]-1] < C[k'] < C[k]$ for all $P[k] \leqslant k' \leqslant k-1$ and $C[P[l]-1] < C[k''] < C[l]$ for all $P[l] \leqslant k'' \leqslant l-1$. Since the two intervals $[P[k]-1, k]$ and $[P[l]-1, l]$ overlap, it's not hard to see that $C[P[k]-1] < C[k'''] < C[l]$ for all $P[k] \leqslant k''' \leqslant l-1$. It follows that $L[l] < P[k]-1$. Thus, $C[P[k]-1] < C[P[l]-1]$ with $L[l]+1 \leqslant P[k] \leqslant l$ is a contradiction to that $C[P[l]-1]$ is minimized with $L[l]+1 \leqslant P[l] \leqslant l$. □

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 $P[s]$, do not need to be "updated" for the subsequence $A(i,j)$ if $[P[s], s]$ doesn't go beyond $[i,j]$.

Lemma 7. *For an index s, if $i \leq P[s] \leq s \leq j$, then $P[s]$ is still the good partner of $A(i,j)$ at index s.*

Proof. Let $C^*[k]$ be the cumulative sum of $A(i,j)$. Then we have $C^*[k] = C[k] - C[i-1]$ for $i-1 \leqslant k \leqslant j$. Let $L^*[k]$ be the left bound of $A(i,j)$ at index k for $i \leqslant k \leqslant j$ and $P^*[k]$ be the good partner of $A(i,j)$ at index k.

If $L[s] \geq i$, we have $L^*[s] = L[s]$ since $L[s]$ is the largest index l with $i \leq l \leq s-1$ such that $C^*[L[s]] = C[L[s]] - C[i-1] \geq C[s] - C[i-1] = C^*[s]$. Otherwise, i.e. $L[s] < i$, we have $L^*[s] = i-1$. Therefore we can conclude that $L[s] \leq L^*[s]$. Moreover, since minimizing $C[P[s]-1]$ minimizes $C^*[P[s]-1] = C[P[s]-1] - C[i-1]$, it's not hard to see that $P[s]$ is still the largest index with $L^*[s]+1 \leqslant P[s] \leqslant s$ that minimizes $C^*[P[s]-1]$, i.e. $P^*[s] = P[s]$. □

Corollary 1. *For an index s, if $i \leq P[s] \leq s \leq j$ then $M[s]$ is still the sum of the candidate segment of $A(i,j)$ at index s.*

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. 3. As an example, in Fig. 4, the input sequence A has 15 elements. Suppose we are querying SRMSQ$(A, 3, 7)$. QUERY OF SRMSQ in Fig. 3 first retrieves index r such that $M[r]$ is maximized with $3 \leqslant r \leqslant 7$ (line 1). In this case, $r = 5$, which means candidate segment $A(P[5], 5)$ has the largest sum compared with other candidate segments whose ending indices lie in $[3, 7]$. Since $A(P[5], 5)$ doesn't go beyond interval $[3, 7]$, the algorithm 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$(A, 6, 12)$. Since $[P[9], 9]$ goes beyond the left end of $[6, 12]$, lines 3-9 are executed. In line 3, RMQ$_{min}(c, 5, 8)$ retrieves index 8. In line 4, RMQ$_{max}(m, 10, 12)$ retrieves 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($A$)
1  Run COMPUTE-CPM to compute $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$ of $A$.
2  Apply RMQ$_{min}$ preprocessing to array $C[\cdot]$.
3  Apply RMQ$_{max}$ preprocessing to array $M[\cdot]$.

Algorithm QUERY OF SRMSQ($A, i, j$)
1  $r \leftarrow$ RMQ$_{max}(M, i, j)$
2  if $P[r] < i$ then
3      $p \leftarrow$ RMQ$_{min}(C, i-1, r-1) + 1$;
4      $s \leftarrow$ RMQ$_{max}(M, r+1, j)$;
5      if $C[r] - C[p-1] < M[s]$ then
6          OUTPUT $(P[s], s)$;
7      else
8          OUTPUT $(p, r)$;
9      end if
10 else
11     OUTPUT $(P[r], r)$;
12 end if
```

Fig. 3. Algorithm for the SRMSQ problem.

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

Proof. Let $C^*[k]$, $P^*[k]$, and $M^*[k]$ be the cumulative sum, the good partner, and the sum of candidate segment of $A(i, j)$ at index k for $i \leqslant k \leqslant j$, respectively. In addition, we let index r satisfy $M[r] \geq M[k]$ for all $i \leqslant k \leqslant j$ (line 1).

1. If $i \leq P[r] \leq r \leq j$ (lines 10-11): Our goal is to show that $M^*[r] \geq M^*[k]$ for all $i \leqslant k \leqslant j$, and then by Lemma 4 $A(P[r], r)$ is the maximum-sum segment of $A(i, j)$.
   (a) First, we consider each index k' where $i \leq P[k'] \leq k' \leq j$. By corollary 1, we have $M^*[k'] = M[k'] \leq M[r] = M^*[r]$.
   (b) Next, we consider each index k'' where $P[k''] < i \leqslant k'' \leqslant j$. By Lemma 5 we have $C[P[k''] - 1] < C[k] < C[k'']$ for all $P[k''] \leqslant k \leqslant k'' - 1$. Since by definition $P^*[k''] - 1$ must lie in $[i-1, k''-1]$, we have $C[P[k''] - 1] <$

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a_k	-	9	-10	4	-2	4	-5	4	-3	6	-11	8	-3	4	-5	3
$C[k]$	0	9	-1	3	1	5	0	4	1	7	-4	4	1	5	0	3
$L[k]$	-	0	1	1	3	1	5	5	7	1	9	9	11	9	13	13
$P[k]$	-	1	2	3	4	3	6	7	8	3	10	11	12	11	14	15
$M[k]$	-	9	-10	4	-2	6	-5	4	-3	8	-11	8	-3	9	-5	3

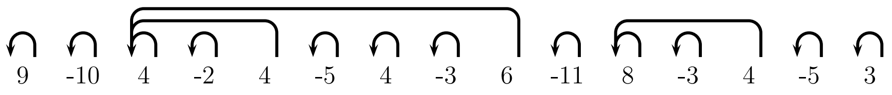

Fig. 4. The candidate segment $A(P[k], k)$ of A at each index k. Notice that, the pointer at each index k points to the position of $P[k]$.

$C[P^*[k'']-1]$. Hence, we can deduce that $M^*[k''] = C^*[k'']-C^*[P^*[k'']-1] = C[k''] - C[P^*[k''] - 1] < C[k''] - C[P[k''] - 1] = M[k''] \leq M[r] = M^*[r]$.

Thus, by (a) and (b) we conclude that $M^*[r] \geq M^*[k]$ for all $i \leqslant k \leqslant j$.

2. If $P[r] < i \leq r \leq j$ (lines 2-9):
   (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 p' with $i \leqslant p' \leqslant k'$, since $S(p', k') = C[k'] - C[p'] < C[r] - C[p'] = S(p', r)$, k' cannot be the right end of the maximum-sum segment of $A(i, j)$.
   (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 $P[r] < P[k''] \leq r < k''$. Suppose $P[k''] \leq P[r] < r < k''$, then by Lemma 5 we have $C[P[k'']-1] < C[r] < C[k'']$ and $C[P[k''] - 1] \leq C[P[r] - 1] < C[k'']$. It follows that $M[k''] = C[k''] - C[P[k''] - 1] > C[r] - C[P[r] - 1] = M[r]$ which contradicts to that $M[r] \geq M[k]$ for all $i \leqslant k \leqslant j$. Thus, it must be the case $r < P[k''] \leq k'' \leq j$. Therefore by Corollary 1 we have $M^*[k''] = M[k'']$.

   Let p be the largest index that minimizes $C[p - 1]$ with $i \leqslant p \leqslant r$ (line 3). Let index s satisfy $M[s] \geq M[k]$ for all $r + 1 \leqslant k \leqslant j$ (line 4). One can easily deduce by (a), (b), and Lemma 4 that either $A(p, r)$ or $A(P[s], s)$ is the maximum-sum segment of $A(i, j)$ (lines 5-9). □

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' which lies in $[i, r - 1]$, k' cannot be the right end of the maximum-sum segment of $A(i, j)$. On the other hand, for each index k'' which lies in $[r + 1, j]$, the good partner of k'' doesn't need to be updated. Hence, only the good partner of index r needs 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 under the unit-cost RAM model.*

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. It is meaningless if the range of the starting index is in front of the range of the ending index, and vice versa. Therefore we assume, without loss of generality, that $i \leq k$ and $j \leq l$. We give our main result of the RMSQ problem as follows.

Theorem 2. *The RMSQ problem can be solved in $O(n)$ preprocessing time and $O(1)$ time per query under the unit-cost RAM model.*

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 $C[x-1]$ with $i \leq x \leq j$ and $k \leq y \leq l$. By applying RMQ techniques to preprocess $C[\cdot]$, the maximum-sum segment can be located by simply querying $\text{RMQ}_{min}(C, i-1, j-1)$ and $\text{RMQ}_{max}(C, k, l)$.
2. Overlapping case ($j > k$): When it comes to the overlapping case, just to retrieve the maximum cumulative sum and the minimum cumulative sum might go wrong if the minimum is on the right of the maximum in the overlapping region. 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 a nonoverlapping case. To maximize $S(x, y)$, we retrieve the minimum cumulative sum by querying $\text{RMQ}_{min}(C, i-1, k-1)$ and the maximum cumulative sum by querying $\text{RMQ}_{max}(C, k, l)$.
   (b) Suppose $k+1 \leq x \leq j$ and $j \leq y \leq l$. This is also a nonoverlapping case.
   (c) Otherwise, i.e. $k+1 \leq x \leq j$ and $k+1 \leq y \leq j$. This is exactly the same as an $\text{SRMSQ}(A, k+1, j)$ query.

   The maximum-sum segment $A(x, y)$ must be the one of these three possible cases which has the greatest sum. $\square$

5 Solving Three 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 [10, 4]. It's not hard to see that it suffices to find for each index $k \geq L$ the maximum-sum segment whose starting index lies in $[max(1, k-U+1), k-L+1]$ and ending index is k. 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 k^{th} 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 [11] 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.

The third problem is to identify the longest segment with lower-bound average [14]. Given a sequence of n numbers and a lower bound N, the goal is to find the longest consecutive subsequence with the average value of those numbers in the subsequence at least N. First, we obtain a new sequence of n numbers $B = \langle b_1, b_2, \ldots, b_n \rangle$ by subtracting N from each of the numbers in the given sequence. It's not hard to see that it suffices to find the longest segment in B with nonnegative sum. Our strategy is to compute for each index k of B the longest nonnegative segment starting at position k. The key idea is that if $A(k,l)$ has nonnegative sum, then by appending the maximum-sum segment starting at index $l+1$ and ending in $[l+1,n]$ to the right end of $A(k,l)$ we get a longer segment $A(k,l')$. Again, our RMSQ techniques can achieve this in constant time. We repeat the process until the sum of the next extended segment is negative. In this way we can find $A(k,R[k])$, the longest segment starting at k with nonnegative sum. Another trick to achieve linear running time is the key observation proposed by Wang and Xu in [14]. Let k and l be the current working pointer scanning from left to right. Assume that we have computed $R[k_1]$ for some index k_1. For each index k_2 lying in $[k_1+1, R[k_1]]$, if $C[k_2] \geq C[k_1]$ then we have $R[k_2] \leq R[k_1]$ since otherwise we would get a longer nonnegative segment starting at k_1 by appending $A(R[k_1]+1, R[k_2])$ to $A(k_1, R[k_1])$. Therefore we can bypass the computation of k_2 and move k forward from k_1 until $C[k] < C[k_1]$. Next we can move l forward from $max(R[k_1]+1, k)$ by appending the maximum-sum segment, since $A(k, R[k_1])$ certainly has nonnegative sum when $C[k] < C[k_1]$ with $k < R[k_1]$. The procedure terminates when k or l reaches the end of B. It is clear that the total running time is linear since either the value of k or l increases by at least one in each iteration, and never decreases.

Acknowledgments. 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, X. Huang, T. Jiang, and K.-M. Chao, MAVG: Locating Non-Overlapping Maximum Average Segments in a Given Sequence, *Bioinformatics*, 19:151-152, 2003.
10. 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.
11. 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.
12. B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM J Comput.*, 17: 1253–1262, 1988.
13. J. Vuillemin. A Unifying Look at Data Structures. *CACM*, 23: 229–239, 1980.
14. L. Wang and Y. Xu. SEGID:Identifying Interesting Segments in (Multiple) Sequence Alignments. *Bioinformatics*, 19: 297–298, 2003.