On the Range Maximum-Sum Segment Query Problem*

Kuan-Yu Chen[1] and Kun-Mao Chao[1,2,3,†]
[1]Department of Computer Science and Information Engineering
[2]Graduate Institute of Biomedical Electronics and Bioinformatics
[3]Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106

March 16, 2007

Abstract

The range minimum query problem, RMQ for short, is to preprocess a sequence of real numbers $A[1 \ldots n]$ for subsequent queries of the form: "Given indices i, j, what is the index of the minimum value of $A[i \ldots j]$?" This problem has been shown to be linearly equivalent to the LCA problem in which a tree is preprocessed for answering the lowest common ancestor of two nodes. It has also been shown that both the RMQ and LCA problems can be solved in linear preprocessing time and constant query time under the unit-cost RAM model. This paper studies a new query problem arising from the analysis of biological sequences. Specifically, we wish to answer queries of the form: "Given indices i and j, what is the maximum-sum segment of $A[i \ldots j]$?" We establish the linear equivalence relation between RMQ and this new problem. As a consequence, we can solve the new query problem in linear preprocessing time and constant query time under the unit-cost RAM model. We then present alternative linear time solutions for two other biological sequence analysis problems to demonstrate the utilities of the techniques developed in this paper.

Keywords: range minimum query, least common ancestor, maximum-sum segment

1 Introduction

The range minimum query problem, RMQ for short, and the least common ancestor problem, LCA for short, have both been studied intensively for decades [3, 4, 9, 11, 16]. The linear equivalence relation between RMQ and LCA was established by Gabow, Bentley, and Tarjan [9]. Harel and Tarjan [11] gave a linear preprocessing time and constant query time algorithm for LCA under the unit-cost RAM model, which makes RMQ solvable in the same time bound. Here a RAM model stands for the Random Access Machine model in which the memory is an array of words [1]. The uniform cost

*A preliminary version of this work appeared in *Proceedings of the 15th International Symposium on Algorithms and Computation*, Hong Kong, 2004.

†Corresponding author, kmchao@csie.ntu.edu.tw

measure is used for measuring time on a random access machine where each operation on a word or a pair of words requires only constant time provided that a word is of size $O(\log n)$ bits. Several studies on both problems in an on-line or off-line setting, and in various models of computation followed (for details see [3]). More recently, Bender *et al.* [4] showed the implementability of LCA and RMQ problems. These two problems have also shown to be related to many string problems [10], such as the longest common extension problem.

On the other hand, the problem of finding the maximum-sum segment of a given number sequence plays an important role in sequence analysis. The maximum-sum segment of a sequence is simply the contiguous subsequence having the greatest total sum. Bentley's linear-time algorithm [5] for finding such a segment is by now a folklore example considered in algorithm classes. There are many kinds of variants of the maximum-sum segment problem that impose extra constraints on the input or on the output [2, 6, 7, 8, 12, 13, 14, 15, 18]. For example, Ruzzo and Tompa [15] studied the problem of finding all maximal-sum segments. All maximal-sum segments are defined recursively. The first maximal-sum segment is simply the maximum-sum segment of the whole input sequence. The ith maximal-sum segment is defined to be the maximum-sum segment of the disjoint intervals obtained by removing $1^{st}, 2^{nd}, \ldots, i-1^{th}$ maximal-sum segments from the input sequence. The goal is to find all the maximal-sum segments having nonnegative sums. If we apply Bentley's algorithm directly, all maximal-sum segments can be found in quadratic time in the worst case. Ruzzo and Tompa [15] gave a novel linear-time algorithm.

Now let us consider an interesting query problem similar to RMQ: "Can we preprocess a number sequence to efficiently answer queries that ask for the maximum-sum segment of any given interval?" If so, then by applying it iteratively, we have a divide-and-conquer algorithm for finding all maximal-sum segments that works in this way: query the maximum-sum segment of the whole input sequence, remove it, and then query to the left of the removed portion, and then to the right. Continue in this manner until all the nonnegative maximal-sum segments are found. If the sequence has n numbers, such queries are carried out no more than n times. Suppose this new query problem has an $f(n)$ preprocessing time and $g(n)$ query time solution. Then the divide-and-conquer approach runs in $O(f(n) + n \cdot g(n))$ time in total. This paper studies this new query problem and proves that it is linearly equivalent to RMQ, which means that it can also

be solved in $O(n)$ preprocessing time and $O(1)$ query time under the unit-cost RAM model [9, 4]. Thus, as an immediate application the above divide-and-conquer approach for finding all maximal-sum segments works in $O(n)$ time.

We call this new query problem the RMSQ problem, standing for the Range Maximum-Sum Segment Query problem, defined formally in Section 2. We then give a linear preprocessing time and constant query time algorithm for RMSQ by establishing the linear equivalence relation between RMQ and RMSQ in Section 3. Section 4 extends this result to a more general case, and Section 5 solves two other related problems in linear time by applying the RMSQ techniques. These variants demonstrate the utilities of the RMSQ techniques developed in this paper.

2 Preliminaries

The input is a nonempty sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ of real numbers. We adopt the following notation. Let $[i, j]$ denote the sets $\{i, i+1, \ldots, j\}$. Let $A[i \ldots j]$ denote the subsequence $\langle a_i, \ldots, a_j \rangle$ and $A[i]$ denote a_i. Let $S(i, j)$ denote the sum of $A[i \ldots j]$, defined by $S(i, j) = \sum_{i \le k \le j} a_k$ for $1 \leqslant i \leqslant j \leqslant n$. Let $C[i]$ denote the cumulative sum of A, defined by $C[i] = \sum_{1 \le k \le i} a_k$ 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 $1 \leqslant i \leqslant j \leqslant n$. If an algorithm has preprocessing time $f(n)$ and query time $g(n)$, we say that the algorithm runs in $\langle f(n), g(n) \rangle$-time.

2.1 The Range Minimum Query Problem (RMQ)

We are given a sequence $A[1 \ldots n]$ to be preprocessed. A Range Minimum Query (RMQ) specifies an interval $[i, j]$ and the goal is to find index $k \in [i, j]$ such that $A[k]$ achieves minimum. The well known algorithm for RMQ is to first construct the Cartesian tree (defined by Vuillemin in 1980 [17]) of the sequence, which is then preprocessed for LCA (Least Common Ancestor) queries [11, 16]. This algorithm can be easily modified to output the index k for which $A[k]$ achieves maximum. We let $\text{RMQ}_{\min}$ and $\text{RMQ}_{\max}$ denote the minimum query and the maximum query, respectively. That is, $\text{RMQ}_{\min}(A, i, j) = \arg\min_{k \in [i,j]} A[k]$ and $\text{RMQ}_{\max}(A, i, j) = \arg\max_{k \in [i,j]} A[k]$, where $\arg\min$ stands for the argument of the minimum and $\arg\max$ is defined analogously.

For correctness of our algorithm, if there are more than one minimum (maximum) in the query 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.

Theorem 1: The Range Minimum Query problem can be solved in $\langle O(n), O(1)\rangle$ under the unit-cost RAM model [9, 4].

2.2 The Range Maximum-Sum Segment Query Problem (RMSQ)

Now we consider a new query problem similar to RMQ. For simplicity, throughout the paper, the terms "subsequence" and "contiguous subsequence" are used interchangeably. To avoid ambiguity, we disallow a nonempty, zero-sum prefix or suffix (also called a tie) in the maximum-sum segments. For example, consider $A = \langle 4, -5, 2, -2, 4, 3, -2, 6\rangle$. 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 we wish to find since it has a nonempty zero-sum prefix. The RMSQ problem is formally defined as follows.

Input to be preprocessed: A nonempty sequence $A[1 \dots n]$ of real numbers.

Online query: For an interval $[i, j]$, $1 \leqslant i \leqslant j \leqslant n$, RMSQ$(A, i, j)$ returns a pair of indices (x, y), $i \leqslant x \leqslant y \leqslant j$, such that $A[x \dots y]$ is the maximum-sum segment of $A[i \dots j]$.

3 RMSQ is Linearly Equivalent to RMQ

In this section, we establish the linear equivalence relation between RMQ and RMSQ by proving that both problems can be transformed into each other in time linear to the size of the input.

3.1 Reduction from RMQ to RMSQ

Theorem 2: If there is an $\langle f(n), g(n)\rangle$-time solution for RMSQ, then there is an $\langle f(2n-1) + O(n), g(2n-1) + O(1)\rangle$-time solution for RMQ.

Proof: Suppose sequence $A = \langle a_1, a_2, \dots, a_n\rangle$ is the input for RMQ. We modify it in $O(n)$ time as follows. For any two consecutive numbers, we augment a negative number c, where $|c| > |a_k|$ for $k \in [1, n]$. Let $B = \langle a_1, c, a_2, \dots, c, a_n\rangle$ denote the new sequence. Then preprocess B for RMSQ queries, which costs $f(2n-1)$ time since $|B| = 2n-1$. The total time for preprocessing is therefore $O(n) + f(2n-1)$.

Now we show that an RMQ query to A can be answered by querying an RMSQ query to B in $g(2n-1)$ time and transforming the answer into the corresponding index of A in $O(1)$ time. More specifically, we show that $\text{RMQ}(A, i, j) = x$ if and only if $\text{RMSQ}(B, 2i-1, 2j-1)$ is $B[2x-1] = \langle a_x \rangle$. The "if" direction is obvious. As for the "only if" direction, since a_x is the global maximum of $A[i \ldots j] = \{a_i, a_{i+1}, \ldots, a_j\}$, for any segment C of $B[2i-1 \ldots 2j-1] = \langle a_i, c, a_{i+1}, \ldots, c, a_j \rangle$, where $|C| = \ell \geq 1$, we have the sum of C is less than $a_x \times \lceil \ell/2 \rceil + \lfloor \ell/2 \rfloor \times c \leq a_x$. We conclude that the maximum-sum segment of $B[2i-1 \ldots 2j-1]$ is an atomic element $\langle a_x \rangle$. □

3.2 Reduction from RMSQ to RMQ

The reduction from RMSQ to RMQ is much more complicated. The result is summarized in the following theorem.

Theorem 3: If there is an $\langle f(n), g(n) \rangle$-time solution for RMQ, then there is a $\langle 2f(n) + O(n), 3g(n) + O(1) \rangle$-time solution for RMSQ.

Before going through the details, we sketch the basic ideas as follows.

1. For each index i, find an index $p \leq i$, called a "good partner" of i, such that $A[p \ldots i]$ forms a "candidate segment" of segments that end at i. We shall give a formal definition of the "good partner" in Section 3.2.1.

2. Use an array $P[1 \ldots n]$ to record the "good partners" and another array $M[1 \ldots n]$ to record the sum of each "candidate segment." That is, if p is the "good partner" of i, then $P[i] = p$ and $M[i] = S(p, i)$.

3. Preprocess array M for $\text{RMQ}_{\max}$ queries.

To answer $\text{RMSQ}(A, i, j)$, we query $\text{RMQ}_{\max}(M, i, j)$, which returns the right end of the "candidate segment" that has the largest sum among those "candidate segments" that end at positions in $[i, j]$. With the help of array P, we can output its both ends, which are the answer of $\text{RMSQ}(A, i, j)$ if the good partner falls in $[i, j]$. However, exceptions happen when the "candidate segment" with the largest sum goes beyond interval $[i, j]$. We show in Section 3.2.2 that this can be remedied by one more RMSQ query to array M and an $\text{RMQ}_{\min}$ query to cumulative-sum array C.

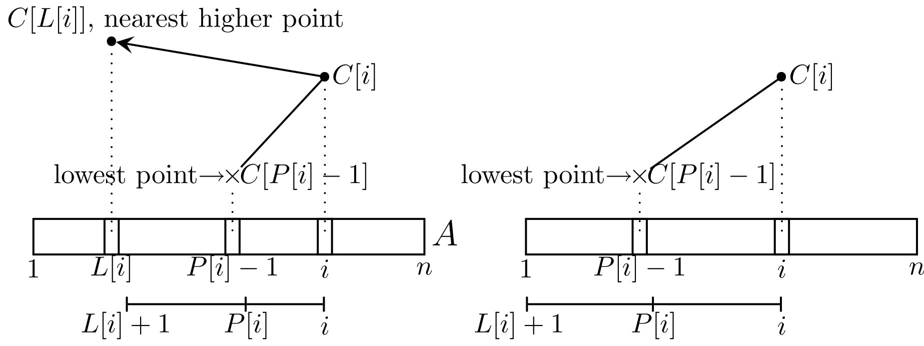

Figure 1: An illustration for $L[\cdot]$ and $P[\cdot]$. Note that y-axis is the value $C[i]$ for various i's. The left figure shows the case that there exists an index $k \in [1, i-1]$ such that $C[k] \geq C[i]$, whereas the right figure shows the case that $C[i] > C[k]$ for all $k \in [1, i-1]$.

3.2.1 A Good Partner

In the following, we formally define the "good partner" of each index and show how they can be found in $O(n)$ time.

Definition 1: Define the *left bound* of $A[1 \ldots n]$ at index i to be

$$L[i] = \begin{cases} \max\{k \mid C[k] \geq C[i] \textit{ and } k \in [1, i-1]\} & \text{if such } k \text{ exists;} \\ 0 & \text{otherwise.} \end{cases}$$

Definition 2: Define the *good partner* of $A[1 \ldots n]$ at index i to be

$$P[i] = \max\{k \mid k \in [L[i]+1, i] \textit{ and } C[k-1] \leq C[l] \ \forall\, l \in [L[i], i-1]\}.$$

To find the good partner of index i, we first find the nearest higher cumulative-sum point in $[1, i-1]$, denoted $L[i]$, and then find the rightmost lowest cumulative-sum point in $[L[i], i-1]$, denoted $P[i]-1$. The relationship between $L[\cdot]$ and $P[\cdot]$ is illustrated in Figure 1.

Definition 3: Each index i together with its good partner $P[i]$ constitute a "candidate segment" $A[P[i] \ldots i]$. We let $M[i]$ denote the sum of that segment, that is $M[i] = S(P[i], i)$ for $i \in [1, n]$.

The three arrays $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$ can be computed by COMPUTE-CPM in Figure 2. An example will be given later in Figure 4.

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

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

Figure 2: Algorithm for computing $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$

Lemma 4: Algorithm Compute-cpm correctly computes arrays $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$ in linear time.

Proof: The correctness of $C[\cdot]$ is trivial, and $M[\cdot]$ are correct if $P[\cdot]$ are correctly computed. Thus, below we show by induction on i that the algorithm correctly computes arrays $L[\cdot]$ and $P[\cdot]$. The basis, $i = 0$, is immediate. As for $i \geq 1$, observe that $L[i]$ is a working pointer scanning from right to left, searching for the nearest higher cumulative-sum point of $C[i]$. The algorithm begins by checking $C[i-1]$. If $C[i-1] < C[i]$, it next checks $C[L[i-1]]$, which by induction is the nearest higher cumulative point of $C[i-1]$. Continuing in this manner, the algorithm examines a list of increasing cumulative-sum values, $C[L[i-1]]$, $C[L[L[i-1]]]$,…,$C[L[\dots L[L[i-1]]\dots]]$. Since by induction $L[1]$, $L[2]$,…, $L[i-1]$ have been computed correctly, the algorithm finally finds the nearest higher cumulative-sum point of $C[i]$ or reaches 0. As for the value of $P[i]$, observe that $P[i]$ is updated correspondingly as the value of $L[i]$ changes. A similar argument over the correctness of $P[\cdot]$ using mathematical induction can be made.

Now we analyze the time complexity of algorithm Compute-cpm. The total number of operations of the algorithm is clearly bounded by $O(n)$ except for the while-loop body of lines 5-7. We show that the amortized cost of the while-loop is a constant. Let $\Phi(i)$ be an integer such that in the beginning of iteration i, $i \in [1, n]$, we have $\overbrace{L[...L[L[i-1]]...]}^{\Phi(i)\ times} = 0$. Observe that before the while-loop is executed, we have

$\Phi(i+1) = \Phi(i) + 1$ since $L[i] = i - 1$. But if the while-loop body is executed c times, then $\Phi(i+1) = \Phi(i) + 1 - c$. Thus, throughout the execution of the algorithm the value of $\Phi(i)$ is increased by one and then possibly decreased a bit; however since $\Phi(i)$ can at most be increased by n in total, and can never be negative, it cannot be decreased by more than n times. The time of while-loop body is therefore bounded by $O(n)$. □

3.2.2 A Formal Proof

Before introducing our RMSQ algorithm, we first prove several properties of good partners and candidate segments.

Lemma 5: For two indices i and j, $i \leq j$, if $A[i \ldots j]$ is the maximum-sum segment of $A[1 \ldots n]$, then $i = P[j]$.

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

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

2. Suppose index i lies in the interval $[L[j]+1, j]$. We know $C[i-1]$ must be minimized since otherwise $A[i \ldots j]$ cannot be the maximum-sum segment. If $C[i-1]$ is not the rightmost minimum, i.e. there exists an index $k \in [i+1, j]$ 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 $A[i \ldots j]$ has a zero-sum prefix $A[i \ldots k-1]$.

Hence, index i must be the rightmost index in $[L[j]+1, j]$ that minimizes $C[i-1]$, i.e. $i = P[j]$. □

Lemma 6: If index x satisfies $M[x] \geq M[k]$ for all $k \in [1, n]$, then $A[P[x] \ldots x]$ is the maximum-sum segment of $A[1 \ldots n]$.

Proof: Suppose on the contrary that segment $A[i \dots j]$ is the maximum-sum segment of A and $(i, j) \neq (P[x], x)$. By Lemma 5, we have $i = P[j]$. So

$$M[j] = S(P[j], j) = S(i, j) > S(P[x], x) = M[x]$$

which contradicts the assumption that $M[x]$ 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 x such that $M[x]$ is the maximum value of $M[k]$ for all $k \in [1, n]$. Then, segment $A[P[x] \dots x]$ is the maximum-sum segment of A.

Lemma 7: For an index $i \in [1, n]$, if $P[i] < i$ then $C[P[i] - 1] < C[k] < C[i]$ for all $k \in [P[i], i - 1]$.

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

In other words, $C[P[i]-1]$ is the unique minimum of $C[k]$ for all $k \in [P[i]-1, k]$ while $C[i]$ is the unique maximum. The following lemma shows that a candidate segment will contain the other candidate segment properly, or be contained the other way, or they are disjoint with each other. That is, overlapping between two candidate segments is not allowed.

Lemma 8: For two indices i and j, $i < j$, 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 7, we have $C[P[i]-1] < C[k_1] < C[i]$ for all $k_1 \in [P[i], i - 1]$ and $C[P[j] - 1] < C[k_2] < C[j]$ for all $k_2 \in [P[j], j - 1]$. Since the two intervals $[P[i] - 1, i]$ and $[P[j] - 1, j]$ overlap, it's not hard to see that $C[P[i] - 1] < C[k_3] < C[j]$ for all $k_3 \in [P[i], j - 1]$. It follows that $L[j] < P[i] - 1$. Thus, $C[P[i] - 1] < C[P[j] - 1]$ together with $L[j] + 1 \leqslant P[i] \leqslant j$ is a contradiction to that $C[P[j] - 1]$ is minimized for $P[j] \in [L[j] + 1, j]$. □

Let us now establish the relationship between sequence A and its subsequence $A[i \ldots j]$. The following lemma shows that some good partners of A, say $P[s]$, do not need to be "updated" for the subsequence $A[i \ldots j]$ if $[P[s], s]$ doesn't go beyond $[i, j]$.

Lemma 9: For an index r, if $[P[r], r] \subseteq [i, j]$, then $P[r]$ is still the good partner of $A[i \ldots j]$ at index r.

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

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

Corollary 10: For an index r, if $[P[r], r] \subseteq [i, j]$ then $M[r]$ is still the sum of the candidate segment of $A[i \ldots j]$ at index r.

Proof: A direct result of Lemma 9. □

Now, we are ready to present our main algorithm based on RMQ for RMSQ (see Figure 3). After finding the good partners and the sums of all candidate segments, we preprocess cumulative-sum array C for $\text{RMQ}_{\min}$ queries and array M for $\text{RMQ}_{\max}$ queries. The preprocessing time is $2f(n) + O(n)$ in total, where $f(n)$ is the preprocessing time for RMQ. To answer an $\text{RMSQ}(A, i, j)$, we first query $\text{RMQ}_{\max}(M, i, j)$ which returns x as the answer. As will be shown in Lemma 11, if $A[P[x] \ldots x]$ does not go beyond i then we are done. Otherwise, it turns out that for each index $k_1 \in [i, x-1]$, k_1 cannot be the right end of the maximum-sum segment of $A[i \ldots j]$. On the other hand, for each index $k_2 \in [x+1, j]$, the good partner of k_2 doesn't need to be updated. Hence, only the good partner of index x needs to be updated. So one more $\text{RMQ}_{\max}(M, x+1, j)$

query and one more $\text{RMQ}_{\min}(C, i-1, x-1)$ query are needed. The time required for an RMSQ query is therefore $3g(n) + O(1)$.

Algorithm PREPROCESS-OF-RMSQ(A)
1 Run COMPUTE-CPM to compute $C[\cdot]$, $P[\cdot]$, and $M[\cdot]$ of A.
2 Preprocess array $C[\cdot]$ for $\text{RMQ}_{\min}$.
3 Preprocess array $M[\cdot]$ for $\text{RMQ}_{\max}$.

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

Figure 3: Algorithm for the RMSQ problem.

As an example, in Figure 4, the input sequence A has 15 elements. Suppose we are querying RMSQ$(A, 3, 7)$. QUERY-OF-RMSQ in Figure 3 first retrieves index $x \in [3, 7]$ such that $M[x]$ is maximized (line 1). In this case, $x = 5$, which means candidate segment $A[P[5] \ldots 5]$ has the largest sum compared with other candidate segments whose ending indices lie in $[3, 7]$. Since $A[P[5] \ldots 5]$ doesn't go beyond interval $[3, 7]$, the algorithm outputs $(P[5], 5)$, which means segment $A[3 \ldots 5]$ is the maximum-sum segment of the subsequence $A[3 \ldots 7]$.

Suppose we are querying RMSQ$(A, 6, 12)$. Since $[P[9], 9]$ goes beyond the left end of $[6, 12]$, lines 3-9 are executed. In line 3, $\text{RMQ}_{\max}(M, 10, 12)$ retrieves index 11. In line 4, $\text{RMQ}_{\min}(C, 5, 8)$ retrieves index 8. In line 5, since $C[9] - C[8] = 6 < M[11] = 8$, the algorithm outputs $(P[11], 11)$, which means $A[11]$ is the maximum-sum segment of the subsequence $A[6 \ldots 12]$.

Lemma 11: Algorithm QUERY-OF-RMSQ(A, i, j) outputs the maximum-sum segment

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

9 -10 4 -2 4 -5 4 -3 6 -11 8 -3 4 -5 3

Figure 4: The candidate segment $A[P[k] \dots k]$ of A at each index k. Notice that the pointer at each index k points to the position of $P[k]$.

of the subsequence $A[i \dots j]$.

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

Case 1: $i \leq P[x] \leq x \leq j$ (lines 10-11). Our goal is to show that $M^*[x] \geq M^*[k]$ for all $k \in [i, j]$, and then, by Lemma 5, $A[P[x] \dots x]$ is the maximum-sum segment of $A[i, j]$.

(a) First, we consider each index k_1 where $[P[k_1], k_1] \subseteq [i, j]$. By Corollary 10, we have $M^*[k_1] = M[k_1] \leq M[x] = M^*[x]$.

(b) Next, we consider each index k_2 where $P[k_2] < i \leqslant k_2 \leqslant j$. By Lemma 7 we have $C[P[k_2] - 1] < C[k] < C[k_2]$ for all $k \in [P[k_2], k_2 - 1]$. Since by definition $P^*[k_2] - 1$ must lie in $[i - 1, k_2 - 1]$, we have $C[P[k_2] - 1] < C[P^*[k_2] - 1]$. Hence, we can deduce that $M^*[k_2] = C^*[k_2] - C^*[P^*[k_2] - 1] = C[k_2] - C[P^*[k_2] - 1] < C[k_2] - C[P[k_2] - 1] = M[k_2] \leq M[x] = M^*[x]$.

Combining (a) with (b), we conclude that $M^*[x] \geq M^*[k]$ for all $k \in [i, j]$.

Case 2: $P[x] < i \leq x \leq j$ (lines 2-9).

(a) First, we consider each index $k_1 \in [i, x - 1]$. By Lemma 7, we have $C[P[x] - 1] < C[k] < C[x]$ for all $k \in [P[x], x - 1]$. For any index $p \in [i, k_1]$, since $S(p, k_1) = C[k_1] - C[p] < C[x] - C[p] = S(p, x)$, k_1 cannot be the right end of the maximum-sum segment of $A[i \dots j]$.

(b) Next, we consider each index $k_2 \in [x+1, j]$. By Lemma 8, we know that it cannot be the case $P[x] < P[k_2] \leq x < k_2$. Suppose $P[k_2] \leq P[x] < x < k_2$, then by Lemma 7

we have $C[P[k_2]-1] < C[x] < C[k_2]$ and $C[P[k_2]-1] \leq C[P[x]-1] < C[k_2]$. It follows that $M[k_2] = C[k_2] - C[P[k_2]-1] > C[x] - C[P[x]-1] = M[x]$ which contradicts to that $M[x] \geq M[k]$ for all $k \in [i,j]$. Thus, it must be the case $x < P[k_2] \leq k_2 \leq j$. Therefore by Corollary 10 we have $M^*[k_2] = M[k_2]$.

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

Combining Theorems 1, 2 and 3, we can conclude the following theorem.

Theorem 12: The RMSQ problem can be solved in $\langle O(n), O(1)\rangle$-time under the unit-cost RAM model.

4 RMSQ with Two Query Intervals

An extension of RMSQ is to answer queries comprised of two intervals $[i,j]$ and $[k,\ell]$, where $[i,j]$ specifies the range of the start index of the maximum-sum segment, and $[k,\ell]$ specifies the range of the end index. We call this generalized version "RMSQ with two query intervals." It is meaningless if the range of the start index is in front of the range of the end index, and vice versa. Therefore we assume, without loss of generality, that $i \leq k$ and $j \leq \ell$. The RMSQ problem with two query intervals is formally defined as follows.

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 $[k,\ell]$, $1 \leqslant i \leqslant j \leqslant n$ and $1 \leqslant k \leqslant \ell \leqslant n$, RMSQ$(A, i, j, k, \ell)$ returns a pair of indices (x, y) with $i \leqslant x \leqslant j$ and $k \leqslant y \leqslant \ell$ that maximizes $S(x, y)$.

This is a generalized version of RMSQ because when $i = k$ and $j = \ell$, we are actually querying RMSQ(i, j). Our algorithm for the RMSQ problem with two query intervals is given in Fig. 5.

Theorem 13: The RMSQ problem with two query intervals can be solved in $\langle O(n), O(1)\rangle$-time under the unit-cost RAM model.

Proof: The two query intervals are either nonoverlapping or overlapping.

Algorithm PREPROCESS-OF-RMSQ2(A)
1 Apply RMSQ preprocessing to A.
2 Apply $\text{RMQ}_{\min}$ and $\text{RMQ}_{\max}$ preprocessing to $C[\cdot]$.

Algorithm QUERY-OF-RMSQ2(A, i, j, k, ℓ)
1 **if** $j \leq k$ **then**
2 OUTPUT ($\text{RMQ}_{\min}(C, i-1, j-1)+1$,$\text{RMQ}_{\max}(C, k, \ell)$);
3 **else**
4 $(x_1,y_1) \leftarrow (\text{RMQ}_{\min}(C, i-1, k-1)+1, \text{RMQ}_{\max}(C, k, \ell))$;
5 $(x_2,y_2) \leftarrow (\text{RMQ}_{\min}(C, k, j-1)+1, \text{RMQ}_{\max}(C, j, \ell))$;
6 $(x_3,y_3) \leftarrow \text{RMSQ}(k, j)$;
7 OUTPUT (x_r,y_r) such that $C[x_r] - C[y_r - 1]$ is maximized for $r = 1, 2, 3$;
8 **end if**

Figure 5: Algorithm for the RMSQ problem with two intervals.

Suppose the intervals $[i, j]$ and $[k, \ell]$ do not overlap, *i.e.*, $j \leq k$. 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 $k \leq y \leq \ell$. Preprocessing $C[\cdot]$ for $\text{RMQ}_{\max}$ and $\text{RMQ}_{\min}$, the maximum-sum segment can be located by simply querying $\text{RMQ}_{\min}(C, i-1, j-1)$ and $\text{RMQ}_{\max}(C, k, \ell)$.

Now we consider the overlapping case, *i.e.*, $j > k$. It might go wrong if we just retrieve the maximum cumulative sum and the minimum cumulative sum because the minimum could be on the right of the maximum in the overlapping region. There are three possible subcases for the maximum-sum segment $A[x \ldots y]$.

1. Suppose $i \leq x \leq k$ and $k \leq y \leq \ell$, 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 $\text{RMQ}_{\max}(C, k, l)$.

2. Suppose $k+1 \leq x \leq j$ and $j \leq y \leq \ell$. This is also a nonoverlapping case.

3. Otherwise, i.e. $k+1 \leq x \leq j$ and $k+1 \leq y \leq j$. This is exactly the same as an $\text{RMSQ}(A, k+1, j)$ query.

The maximum-sum segment $A[x \ldots y]$ is the one of these three subcases which has the greatest sum.

□

5 Applications of RMSQ

As discussed in the introduction, there exists a linear time algorithm based on RMSQ for finding all maximal-sum segments. That is, we first preprocess input sequence $A[1 \dots n]$ for RMSQ queries. Suppose $\text{RMSQ}(A, 1, n) = (x, y)$. Then, all the maximal-sum segments of $A[1 \dots n]$ are the maximal-sum segments of $A[1 \dots x-1]$ and $A[y+1 \dots n]$ together with segment $A[x \dots y]$. This divide-and-conquer approach takes $O(n)$ time since the total number of RMSQ queries are $O(n)$. In the following, we use RMSQ to solve two other related sequence analysis problems in linear time.

5.1 Finding the Maximum-Sum Segment Satisfying Length Constraints

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 with length at least L and at most U [14, 8]. It's not hard to see that it suffices to find for each index $i \geq L$ the maximum-sum segment whose start index lies in $[\max(1, i-U+1), i-L+1]$ and end index is i. Remembering the best segment while scanning the input sequence gives us the answer. Since our RMSQ algorithm can answer each such query in constant time, the total running time is therefore linear.

5.2 Finding the Longest Segment Satisfying an Average Constraint

Given a sequence of n numbers $A = \langle a_1, a_2, \cdots, a_n \rangle$ and a lower bound L, the second problem is to find the longest segment satisfying the average of those numbers in that segment is at least L [6, 18]. This problem is equivalent to finding the longest segment of $B = \langle a_1 - L, a_2 - L, \cdots, a_n - L \rangle$ whose sum is nonnegative, since segment $A[i \dots j]$ with average at least L corresponds to segment $B[i \dots j]$ with sum at least 0.

To find the longest segment with nonnegative sum in $B[1 \dots k]$, we compare the length of the longest segment with nonnegative sum in $B[1 \dots k-1]$ and the longest segment ending at k with nonnegative sum. Then the one with a longer length is clearly the answer of $B[1 \dots k]$. The former segment, denoted by $B[x \dots y]$, is obtained by induction, where initially $x = 0$ and $y = -1$. To find the latter segment, we proceed in the safest manner to avoid unnecessary steps. Since our goal is to find the longest segment ending at k satisfying a sum lower bound, we retrieve by RMSQ the maximum-sum segment ending at k with a longer length than the best segment known so far. If this maximum-sum segment doesn't satisfy the sum lower bound, then we have the conclusion that no other

Algorithm LONGEST-SEGMENT(A, L)
Input: A nonempty array $A = \langle a_1, a_2, \cdots, a_n \rangle$ and an average lower bound L.
Output: The longest segment with average at least L.

```
1   B = ⟨a_1 − L, a_2 − L, ···, a_n − L⟩
2   PREPROCESS-OF-RMSQ2(B);
3   x ← 0; y ← −1;
4   for k ← 1 to n do
5       (i, j) ← QUERY-OF-RMSQ2(B, 1, k − y + x − 1, k, k);
6       while S(i, j) ≥ 0 do
7           x ← i; y ← j;
8           (i, j) ← QUERY-OF-RMSQ2(B, 1, x − 1, k, k);
9       end while
10  end for
11  if x > 0 then OUTPUT (x, y);
12  else OUTPUT "No segment with average at least L.";
13  end if
```

Figure 6: Algorithm for finding the longest segment satisfying an average constraint.

segment does. Specifically, we query the maximum-sum segment whose start index lies in $[1, k - y + x - 1]$ and end index is k. Suppose RMSQ returns $A[i, j]$. If $S(i, j) \geq 0$, we let $x = i$ and $y = j$, and repeat the above query to find a longer segment. Otherwise, we terminate the procedure and return $B[x \dots y]$ as the answer. Figure 6 gives the algorithm for finding the longest segment satisfying an average lower bound.

We show that the above procedure runs in linear-time by observing that the total times of RMSQ queries are bounded by the length of the sequence plus the length of the longest segment grown throughout the execution, which is the value of $y - x + 1$ at the end. Since the segment cannot grow longer than the input sequence and each RMSQ query can be answered in constant time, the total running time of LONGEST-SEGMENT(A, L) is $O(n)$.

Acknowledgments. We thank the anonymous reviewers, Yu-Ru Huang, Rung-Ren Lin, Hsueh-I Lu, An-Chiang Chu and Hsiao-Fei Liu for their helpful comments that improve the presentation of the paper. Kuan-Yu Chen and Kun-Mao Chao were supported in part by NSC grants 92-2213-E-002-059 and 95-2221-E-002-126-MY3 from the National Science Council, Taiwan.

References

[1] A. V. Aho, J. E. Hopcroft, and J.D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA, 1974.

[2] L. Allison. Longest Biased Interval and Longest Non-Negative Sum Interval. *Bioinformatics*, 19:1294–1295, 2003.

[3] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest Common Ancestors: A Survey and a New Algorithm for a Distributed Environment. *Theory of Computing System*, 37: 441–456, 2004.

[4] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest Common Ancestors in Trees and Directed Acyclic Graphs. *Journal of Algorithms*, 57: 75–94, 2005.

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

[6] K.-Y. Chen and K.-M. Chao, Optimal Algorithms for Locating the Longest and Shortest Segments Satisfying a Sum or an Average Constraint, *Information Processing Letters*, 96: 197–201, 2005.

[7] K. Chung and H.-I. Lu. An Optimal Algorithm for the Maximum-Density Segment Problem. *SIAM Journal on Computing*, 34(2):373–387, 2004.

[8] 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.

[9] H. Gabow, J. Bentley, and R. Tarjan. Scaling and Related Techniques for Geometry Problems. *STOC*, 135–143, 1984.

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

[11] D. Harel and R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13: 338–355, 1984.

[12] X. Huang. An Algorithm for Identifying Regions of a DNA Sequence that Satisfy a Content Requirement. *CABIOS*, 10: 219–225, 1994.

[13] 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.

[14] 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.

[15] 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.

[16] B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM Journal on Computing*, 17: 1253–1262, 1988.

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

[18] L. Wang and Y. Xu. SEGID: Identifying Interesting Segments in (Multiple) Sequence Alignments. *Bioinformatics*, 19: 297–298, 2003.