Range Queries: A New Construction

Rares Cristian

November 17, 2019

1 Introduction

Range queries are a fundamental operation used in many applications. A classic example that has been deeply studied is the Range Minimum Query (RMQ) problem. We provide a data structure to efficiently answer a broader class of queries for static arrays in $O(1)$ time and $O(n \log n)$ initialization.

Let $A = [a_1, \ldots, a_n]$ be an array of n elements. Given a range $[L, R]$, we must determine the value of some given function f with input $a_L, \ldots, a_R$. We will denote $f(L, R) = f(a_L, \ldots, a_R)$. The main approach taken by most data structures, as well as ours, is to preprocess the array to determine the value of f for a particular subset P of ranges. P is chosen such that we can determine $f(L, R)$ using only values from $\{f(p) : p \in P\}$.

Instead of focusing on specific problems, let's have an abstract definition of the types of queries we want to be able to support.

1.1 The Merging Archetype

We want our functions f to have the following property: there exists a merge function M such that for all $1 \leq i \leq j \leq k \leq n$, we have that

$$f(i,k) = M(f(i,j), f(j+1,k))$$

1.2 Examples

As a concrete example, we could define f to be the sum of the elements in a range and $M(a,b) = a + b$. Similarly, for a minimum range query, $M(a,b) = \min(a,b)$.

There are many other interesting problems which have this property. For example, we may wish to find, within any specified range, the contiguous subsequence with greatest sum. The merging process is slightly more complicated. In addition to storing the solution to a particular range, we also need to store three more auxiliary values: the sum over the entire range, largest sum of a prefix, and largest suffix of the range. Now, the merge can be done as follows:

$$\begin{aligned}
\text{sum}[i,k] &= \text{sum}[i,j] + \text{sum}[j+1,k] \\
\text{prefix}[i,k] &= \max\{\text{sum}[i,j] + \text{prefix}[j+1,k], \text{prefix}[i,j]\} \\
\text{suffix}[i,k] &= \max\{\text{sum}[j+1,k] + \text{suffix}[i,j], \text{suffix}[j+1,k]\} \\
\text{answer}[i,k] &= \max\{\text{answer}[i,j], \text{answer}[j+1,k], \\
&\qquad \text{suffix}[i,j] + \text{prefix}[j+1,k]\}
\end{aligned}$$

for any $i \leq j \leq k$.

1.3 Our Approach

We will determine a set P consisting of ranges in A, and computing $f(p)\ \forall p \in P$. Any range $[i, k]$ in A must be capable of being represented by the union of some pairwise disjoint elements, $X = \{p_1, p_2, \ldots, p_N\} \subseteq P$. We say X is a partition of $[i, k]$. To answer an arbitrary query $f(i, k)$, we can then iteratively merge the pre-computed solutions of X together.

We will show that there exists a set P which contains $O(n \log n)$ elements such that for any query, there exists a partition $X \subseteq P$ containing at most two elements. Moreover, we may find the elements in $O(1)$ time. But first, we review some key ideas of segment trees as we will make use of them later.

2 Segment Trees

We will define P recursively:

$[1, n] \in P$
If $[i, k] \in P, i \neq k$, then $[i, j], [j+1, k] \in P$ where $j = \lfloor (i+k)/2 \rfloor$.

We say that $[1, n]$ is the root of the tree, and it has two children, $[i, j]$, and $[j+1, k]$. See figure 1.

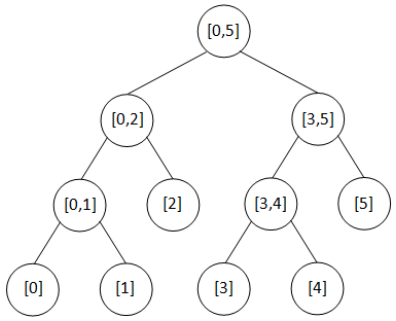


Figure 1: Segment tree constructed from array of 6 elements

2.1 Properties

1. The height of the tree is $\log n$, where n is the length of the array.
2. For any subarray, there exists a partition into $O(\log n)$ elements from P.

3 A Graph Construction

Let $G = (P, E)$ where P is the same vertex set as in segment trees, and where $E = \{([a, b], [b + 1, c]) \in P \times P : a \leq b \leq c, a \neq c\}$. Essentially, we have a directed graph with edges from one range p_1 to another range p_2 if and only if p_1 and p_2 are adjacent and p_1 comes before p_2. For much of the remainder of the paper we will assume that n is a power of two. We will consider the general case at the end.

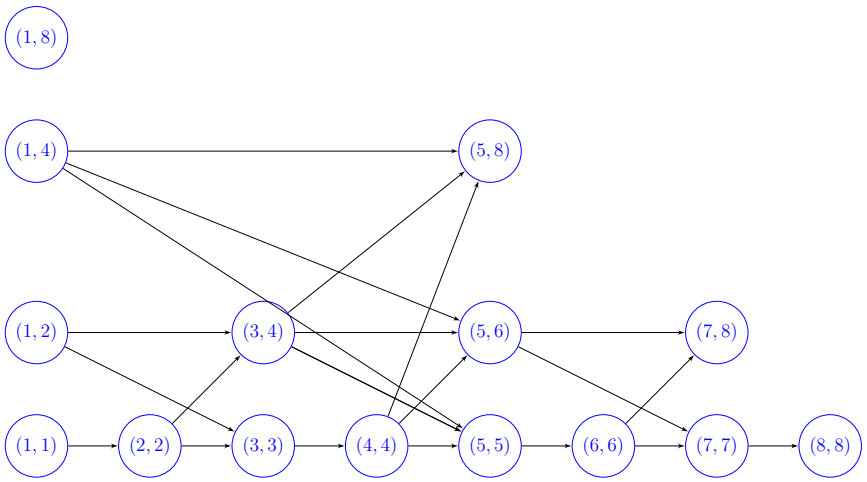


Figure 2: Adjacency graph G corresponding to an array of 8 elements.

We say that a node is in column i if the range that node represents has leftmost element at index i. Additionally, we say that the length of a node is the number of nodes in the range it represents. Finally, a maximal node is one which is the longest in its columm.

4 Constructing Partitions

A path in G corresponds to a partition of a range. Indeed, any range may be partitioned using some path (using only nodes of length 1, for example). In order to answer a query $[i, k]$, we need to find the shortest path from a node in column i to a node whose right endpoint is k. Something like BFS would work here, but we have a very structured graph with some nice properties which will make the task much simpler.

Lemma 1. *If a node of length 2^i is in some column, then so is a node of length 2^j for any $j < i$.*

Proof. By construction all nodes of length 2^k are in a column c if and only if $c = 1 + d \cdot 2^k$ for some d. Since 2^i is a multiple of 2^j, this directly implies the lemma.

Q.E.D.

4.1 A Naive Approach

Step 1: Find the longest node in column i completely contained in the range query $[i, k]$. Say it has length 2^j. Put it in the path.

Step 2: Solve the sub-problem of finding the optimal partition for the query $Q' = [i + 2^j, k]$ by going to Step 1 until Q' is empty.

Finding the longest admissible node in a column will take us at most $O(\log n)$ time. There may be $O(\log n)$ nodes in the path (why? this finds the same set of nodes as we would for a segment tree, so we know only $O(\log n)$ are needed), so overall, this approach takes $O(\log^2 n)$ time.

4.2 Reduction to $O(\log n)$

Lemma 2. *: The optimal partition will consist of two sequences; the first contains nodes of increasing length, and the second contains nodes of decreasing length. This is equivalent to saying that if we choose some node of shorter length than the previous one, then we may never choose a larger one.*

Proof. Assume for the sake of contradiction that this is not true. Say we choose a node p_1 of length 2^i followed by a node p_2 of length 2^j, $j < i$ which is itself followed by some node p_3 of length 2^k, $k > j$.

Case 1: $i < k$

First, note that a node of length 2^i also lies in the same column as p_2 since p_2 directly follows p_1. Additionally, a node of length 2^i also lies in the same column as p_3 (by Lemma 1), which is a contradiction since the length between p_1 and p_3 is $2^j < 2^i$.

Case 2: $i \geq k$

Again, there is a node length 2^i which lies in the same column as p_2. Thus there is a node of length 2^k in the same column as p_2, by Lemma 1. This implies that there is no node of length 2^k which begins after p_2 ends.

Q.E.D.

Now, searching for the largest node in a column which is contained in the query is much simpler. We no longer need to search every node in a given column, but rather begin where we ended in the previous node's column. This way, we only inspect a node of length 2^i at most twice for any i. Overall, this takes $O(\log n)$ time.

4.3 Finding the Increasing and Decreasing Sequences

The problem of finding the decreasing sequence turns out to be identical to finding an increasing sequence. Consider the graph G' with the same vertex set as G and if $(u, v) \in E(G)$, then $(v, u) \in E(G')$. Essentially, we reverse the direction of edges. We will also define columns differently. We say that a node $[i, k]$ is in column k, not in column i as before. The increasing sequence in G' directly corresponds to the decreasing sequence in G. So, instead of finding the decreasing sequence in G, we may find the increasing sequence in G'.

Lemma 3. *If a node is not maximal in G, then it is maximal in G'.*

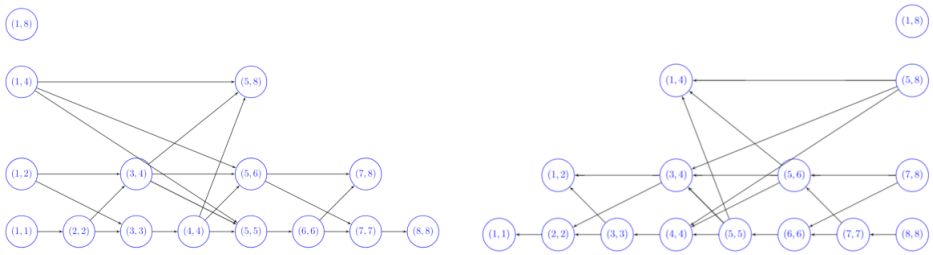
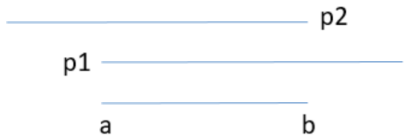


Figure 3: G on the left and G' on the right

Proof. For the sake of contradiction, assume that there exists a node $[a, b]$ of length 2^i which is not maximal in either G or G'. Then, there exist nodes p_1, p_2, one starting at a and the other ending at b, respectively, both of greater length than $[a, b]$.



From lemma 1, a contradiction easily follows.

Q.E.D.

Lemma 4. *All nodes in the increasing sequence must be maximal.*

Proof. For the sake of contradiction, let p be the first node in the path which is not maximal. Suppose it has length 2^i, and lies in column c. Therefore, there exists a node of length 2^{i+1} in column c as well. There can be no node of length greater than or equal to 2^i following p in the path since otherwise, we may have chosen the 2^{i+1} length node in column c instead. Thus, p must be the last node in the increasing sequence. Recall, we assumed that p was the first non-maximal node in the increasing sequence, so it is actually unique.

To simplify, we can remove this node from the increasing sequence, and place it at the beginning of the decreasing sequence - and thus maximal in G' by lemma 3.

Q.E.D.

From now on, we will only consider trying to solve the problem of finding the increasing sequence. We know our path must consist only of maximal nodes in G, so, let's precompute the answer to all of these paths. There are n maximal nodes at which a path may begin, and each successive element in a maximal path has increasing length, so a path has $O(\log n)$ nodes. Thus, we have $O(n \log n)$ paths we need to precompute the answer for.

4.4 Queries

Let P_i be the set of paths which begin with a node in column i. For each $j = 0, \ldots, \lfloor \log n \rfloor$, keep a pointer to the longest path in P_i which has its longest node of length less than or equal to 2^j. The idea here is similar to that of sparse tables. For any query $[L, R]$, we find the greatest integer c such that 2^c is smaller than or equal to $R - L + 1$. Then, we inspect the path which begins in column i and is pointed to by 2^c.

Case 1: This path is completely contained in the query range. In this case, that's our answer.

Case 2: It's not completely contained. Let's look at the path with this last node removed. It has length less than 2^c (The last node would have length at most 2^{c-1} so the path's length is upper bounded by $2^0 + 2^1 + \cdots + 2^{c-1} < 2^c$). Therefore, this path is completely contained in our query. So, that is our answer.

Both cases above can be carried out in $O(1)$ time. Proceed similarly for G'.

Finally, if we have an array of length which is not a power of two, we may "pad" it with neutral values to reach the smallest power of two greater than its original length. Thus, we have $O(n \log n)$ preprocessing time, and $O(1)$ query time.

5 Maximal Node Subgraph: A Binomial Forest

Since we only care about the maximal nodes in the graph, we may remove all other nodes. This produces a forest of binomial trees.

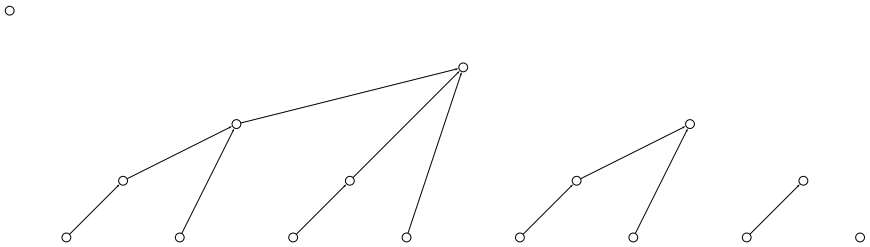


Figure 4: Binomial Forest from Maximal Node Subgraph

So, we have reduced the problem from range queries on arrays of length n, to path queries on binomial trees of height $\log n$. More specifically, paths from a node to one of its ancestors.