Range Queries

Rares Cristian

February 6, 2018

1 Introduction

We provide a survey of existing approaches to solving range queries, mainly on static arrays, as well as provide a novel data structure to answer queries using $O(n \log n)$ preprocessing time and $O(1)$ query time. This approach will be applicable to any query type which can also be answered by a segment tree (see section 5).

From here onwards, we will assume we are given an array $A = [a_1, \ldots, a_n]$ of numbers. The most common queries are those of finding the sum of the values, or the maximum value, in a specified contiguous subsequence of a static array. More generally, 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 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 $\{f(p) : p \in P\}$.

2 Range Sum Queries

Here, we define

$$f(L, R) := \sum_{i=L}^{R} a_i$$

A trivial way to answer the query is to simply compute the summation as described above. However, we are always assuming that there will be many queries, and that each must be computed as efficiently as possible, possibly at the expense of some initial computation. For the summation problem, we have a simple solution: we can let P be the set of all prefixes of A. We may compute these values in linear time:

$$f(1, 1) = a_1 \tag{1}$$
$$f(1, r) = f(1, r - 1) + a_r \tag{2}$$

Furthermore, each query can be answered in constant time:

$$f(L, R) = f(1, R) - f(1, L - 1)$$

(We may define $f(1, 0) := 0$)

This prefix approach relies on the notion of an inverse, (subtraction in this case). However, many functions do not have an inverse and this approach is not applicable. Finding the maximum value in a range is one such example.

3 Sparse Tables

This data structure is particularly useful in answering minimum/maximum range queries, and uses $O(n \log n)$ preprocessing time and $O(1)$ query time. First, it is useful to define P as the union of sets H_i, $i = 0 \ldots \lfloor \log n \rfloor$, defined as follows:

$$H_i = \{[l, l + 2^i - 1] : l \in [n], l + 2^i \leq n\}$$

That is, H_i is the set of all contiguous subsequences of length 2^i and thus P is the set of all ranges which have length a power of two. We may compute the answer to each range in H_i as follows:

$$f(l, l) = a_l \tag{3}$$
$$f(l, l + 2^i - 1) = \min(f(l, l + 2^{i-1} - 1), f(l + 2^{i-1}, l + 2^i - 1)) \tag{4}$$

We can see that the answer to each range in H_i can determined by looking at the two ranges in H_{i-1} which partition it. Since i is at most $\log n$, and $|H_i| \leq n$ we have $O(n \log n)$ preprocessing time. We may now answer any query $[L, R]$ as follows:

Let i be the largest value such that $L + 2^i - 1 \leq R$. Now,

$$f(L, R) = \min(f(L, L + 2^i - 1), f(R - 2^i + 1, R))$$

The two ranges are completely contained within $[L, R]$ simply by construction and more importantly their union is exactly $[L, R]$ since otherwise we may have chosen a larger value of i. Additionally, note that we have precomputed the answer to each of the ranges used above since they have length 2^i (and so are contained in H_i).

Unfortunately, we cannot answer range sum queries using sparse tables. The ranges used, $[L, L + 2^i - 1]$ and $[R - 2^i + 1, R]$, may easily overlap. So,

their sum will not produce the correct value. Essentially, this approach is applicable if the query only cares about the set of values in a range, and their multiplicity is irrelevant. Finding the greatest common divisor of all elements in a range is one such example. However, we can make a slight revision to our approach in order to support summation:

$$f(L, R) = f(L, L + 2^i - 1) + f(L + 2^i, R)$$

where we recursively need to solve for $f(L + 2^i, R)$ (and we have already computed $f(L, L+2^i-1)$ as part of our preprocessing). The size of the query halves at each iteration and so this would take $O(\log n)$ time.

4 Abstractions

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

4.1 The Merging Archetype

We want our functions 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

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

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, $M(a, b) = \min(a, b)$.

4.2 Example

There are many interesting problems which have this property. For example, take the largest sum contiguous subarray problem. The problem asks us 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 prefix, and largest suffix of the range. Now, the merge can be done as follows:

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

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

4.3 High-Level Solution

We will solve the problem by determining 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, $X \subseteq P$ contains at most two elements. Moreover, the time complexity of finding the elements of X is $O(1)$. But first, we take a look at segment trees.

5 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.

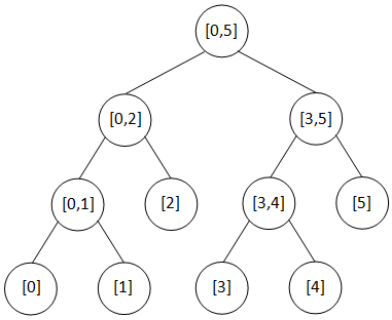


Figure 1: Segment tree constructed from array of 6 elements

We may easily construct the tree recursively. Consider the following algorithm to answer queries. We simply need to call QUERY(1, n, L, R).

```
1: function QUERY(i, k, L, R)
2:     if k < L or i > R then
3:         return NULL
4:     else if L ≤ i ≤ k ≤ R then
5:         return f(i, k)
6:     else
7:         j = ⌊(k + l)/2⌋
8:         return M(QUERY(i, j, L, R), QUERY(j + 1, k, L, R))
9:     end if
10: end function
```

5.1 Correctness

We have the invariant that QUERY(i, k, L, R) will always return $f(\max(i, L), \min(k, R))$ or more simply put, it computes f on the intersection of $[i, k]$ and $[L, R]$. Indeed, QUERY(1, n, L, R) provides the intended value.

5.2 Proof of Complexity

First, we define the notion of expansion. A range $S = [i, k]$ is said to expand $T = [L, R]$ if both $S \cap T$ and $S \setminus T$ are nonempty. We claim that at most two nodes at the same depth can be expanded.

Proof. Assume for the sake of contradiction that there are three nodes at the same depth which expand $[L, R]$. First note that any two nodes at the same depth represent two disjoint ranges. For a range $[i, k]$ to be expanding, either $L - 1 \in [i, k]$ or $R + 1 \in [i, k]$. By the pigeonhole principle, at least two of the three expanding ranges must contain the same element, a contradiction since that would imply they are not disjoint.

Since only two nodes at the same depth are expanded (also note that we only recurse into nodes which expand the query range), and the height of the tree is $\log n$, it follows that this algorithm terminates in $O(\log n)$ time.

Q.E.D.

5.3 Dynamic Arrays

Segment trees also support an additional, and often very useful, operation: updating elements of the array. When updating an element x, we must recompute f for each member of P which contains x. For segment trees, this is simple: we must only update every node on the path from root to the leaf which corresponds to x, which is also an $O(\log n)$ time operation.

5.4 Further Notes on Segment Trees

Segment trees are incredibly versatile, and here we only mention some of the things that we may do with them.

1. Lazy Propagation: We may update an entire range of the array at the same time in $O(\log n)$ time.
2. Persistency: This simply means to retain changes. This allows us to revert back to an older version of the segment tree. Creation of a new version will only take $O(\log n)$ time and space.
3. Merge Sort Tree: Within each node, we may, instead of storing a single value, store a sorted version of the range it represents. This allows to answer queries like finding the k^{th} largest element in a range in $O(\log^2 n)$ time, utilizing only $O(n \log n)$ space and preprocessing time.

There are a myriad of types of problems which may be solved by segment trees and modifications of them which is out of the scope of this paper. Instead, we will focus on a novel approach which builds off of segment trees.

6 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.

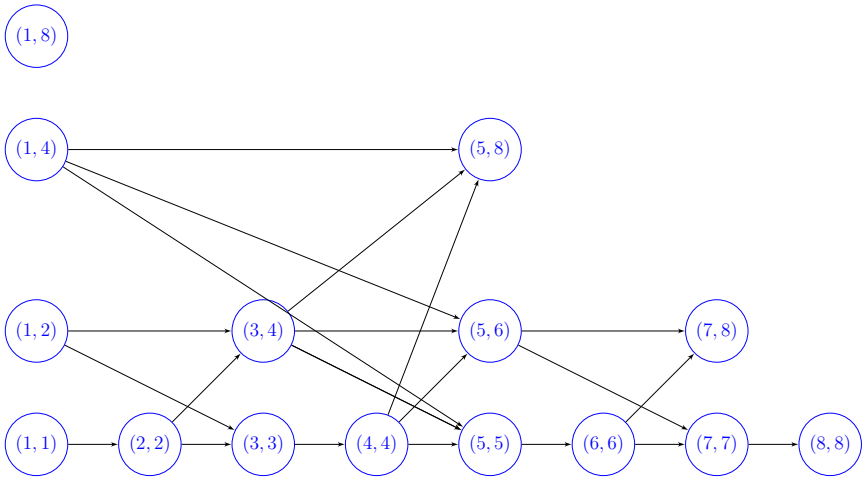


Figure 2: Adjacency graph G corresponding to an array of 8 elements.

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

7 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. A node is in column c only if c is a multiple of its length. We have that 2^i is a multiple of 2^j, which directly implies the lemma.

Q.E.D.

7.1 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 $\log n$ nodes in the path, so overall, this approach takes $O(\log^2 n)$ time.

7.2 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. Overall, this takes $O(\log n)$ time.

7.3 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'.*

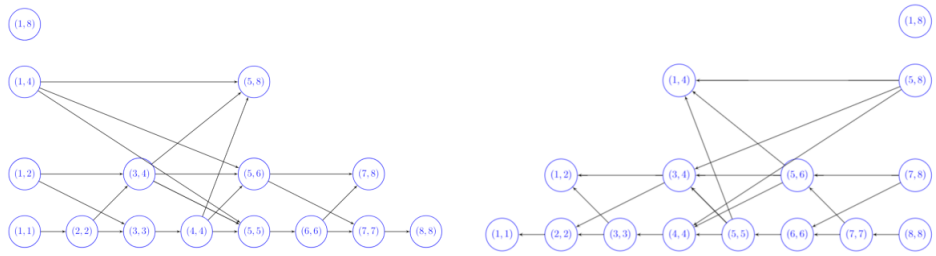
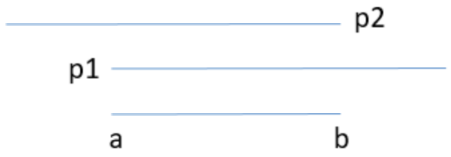


Figure 3: G on the left and G' on the right

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.



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, assume that one node, say p, of length 2^i, and in column c, in the path is not maximal. 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 is the last node in the increasing sequence. Instead, we may say that p is the first node in 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.

7.4 Reduction to $O(1)$ Query Time

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 can be 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.