Range Queries

Kuba Karpierz, Bruno Vacherot

March 4, 2016

Range query problems are of the following form:

Given an array of length \( n \), I will ask \( q \) queries. Queries may ask some form of question about a range of the array, like \( \text{sum}(i, j) \), or modify elements in the array. The hope is that by doing some precomputation, we can answer our queries faster than the naive method.

We begin by attempting the following problem:

**Problem 1**

Assume that I give you an array of integers of length \( n \). I will then ask you \( q \) queries, of the form \( \text{sum}(i, j) \), which is the sum of all of the elements of the array in the

Naively, we do no precomputation, and then whenever we encounter a query, simply sum from index \( i \) to index \( j \). However, this is \( O(n) \) work! And a lot of this work is repeated.

Suppose instead we computed an \( n \times n \) table, where \( \text{table}[i][j] \) represented the sum of \([i, j]\). If we’re clever, we can compute this in \( O(n^2) \) time, and then answer any query in \( O(1) \) time. Great!

Can we do better?

Well, we know that sums are *associative*, which means:

\[
\text{sum}(x, y) = \text{sum}(x, k) + \text{sum}(k + 1, y)
\]

In simple terms, it means we can break our problem down into smaller problems, solve those, then recombine them to get the solution to the larger problem. Conviently enough, addition also has a simple inverse, subtraction.
So we can rewrite the previous formula like so:

\[ \text{sum}(k + 1, y) = \text{sum}(x, y) - \text{sum}(x, k) \]

How does this help? Well, for \( \text{sum}(i, j) \), let \( x = 0, y = j, k = i - 1 \). Then:

\[ \text{sum}(i, j) = \text{sum}(0, j) - \text{sum}(0, i - 1) \]

Then to compute the answer to any query, we just need to know the cumulative sum, like so:

<table>
<thead>
<tr>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cumulative Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

If we’re clever about constructing this, we can construct this in \( O(n) \) time.

**Problem 2**

What if I give you an \( n \times n \) array, and ask queries of the form \( \text{sum}(x_1, x_2, y_1, y_2) \), which is the sum of all values \( A[i][j] \) where \( x_1 \leq i \leq x_2 \) and \( y_1 \leq j \leq y_2 \)?

Our two naive approaches, either precompute every range, or do no precomputation, result in taking \( O(n^4) \) time to precompute, or \( O(n^2) \) time to query respectively.

Instead, let’s take the same approach we took in the last problem. Note that:

\[
\begin{align*}
\text{sum}(x_1, x_2, y_1, y_2) &= \text{sum}(x_1, x_2, 0, y_2) - \text{sum}(x_1, x_2, 0, y_1 - 1) \\
&= (\text{sum}(0, x_2, 0, y_2) - \text{sum}(0, x_1 - 1, 0, y_2)) - (\text{sum}(0, x_2, 0, y_1 - 1) - \text{sum}(0, x_1 - 1, 0, y_1))
\end{align*}
\]
Then if we precompute every sum of the form \( \text{sum}(0, i, 0, j) \), which we can do in \( O(n^2) \) time, we can answer our queries in \( O(1) \) time!

**Problem 3**

For the rest of these problems, we’ll be working across \( n \) length arrays of integers.

Suppose that we have one kind of query we’d like to be able to handle, \( \text{min}(i, j) \), which returns the minimum value of \( A \) in \([i, j] \).

Our two naive approaches, either precompute every range, or do no precomputation, result in taking \( O(n^2) \) time to precompute, or \( O(n) \) time to query respectively.

Can we do better?

Suppose that at each index, we store the minimum for a group of arrays beginning at that index.

For each index \( i \), store the minimum value in the range \([i, i + 2^k]\), for \( k < \log(n) \).

For example, this is what would be store for \( i = 1 \). Note that the width of the yellow boxes corresponds to the range the box represents. Each box only stores a single value, which is the minimum across the range it represents.

How do we compute this quickly? Well, we can observe that:
\[ \min(i, i + 2^k) = \text{minimum}(\min(i, i + 2^{k-1}), \min(i + 1 + 2^{k-1}, i + 2^k)) \]

Thus, we can start with \( k = 0 \) (which are the ranges \([i, i]\)), and then build up from there. Since the table is of size \( O(n \log(n)) \), we can compute it in \( O(n \log(n)) \) time.

Now how do we find the minimum in the range \([i, j]\)? Well, let \( k \) be the largest integer such that \( 2^k \leq (j - i) + 1 \) (IE, the largest power of two that’s smaller than our range). Then we know that every value in \([i, j]\) is in either \([i, i + 2^k]\) or \([j - 2^k, j]\). Then the minimum value of these two ranges is the minimum of the range.

Thus, we can find the minimum of any range in \( O(n \log(n)) \) space, \( O(n \log(n)) \) precompute time, and \( O(1) \) time per query.

**Problem 4**

The three previous problems were examples of *offline queries*, where the state of the array doesn’t change with queries. But what if we can update elements of our array?

Suppose that we have two kinds of queries we’d like to be able to handle, \( \text{update(index, x)} \), which sets \( A[\text{index}] = x \), and \( \text{min}(i, j) \), which returns the minimum value of \( A \) in \([i, j]\).

Our previous approach would let us find \( \text{min}(i, j) \) in \( O(1) \) time. But how would it handle \( \text{update(index, x)} \)?

Well, since \( \text{index} \) may be contained in many precomputed intervals, we may need to update at least \( O(n) \) elements of our table! Which means that \( \text{update(index, x)} \) can take as long as \( \text{min}(i, j) \) did when we computed it naively.

So how can we do it faster?

The first approach we’ll try is called a Square Root Decomposition. The idea is to break the array up into \( \sqrt{n} \) blocks, compute the answer for each block, and then when given a query, try and build the answer from the blocks and a few elements from the original array.

For example, we have \( n = 16 \), so our buckets are each of size \( \sqrt{16} \), which is equal to 4:

What if our \( n \) isn’t a perfect square? We can choose to round up or down, but it won’t make too much of a performance difference.
What if our \( n \) isn’t perfectly divisible by our bucket size? Simply let the last bucket contain the remainder elements.

Now how do we do our queries across this structure? First, let’s try implementing \( \text{update}(index, x) \).

Well, we know that each element belongs to a single bucket, and each bucket only has at most \( \sqrt{n} \) elements. So if we update an element, we can only affect a single bucket. So we can handle \( \text{update}(index, x) \) like so:

- Set \( A[index] = x \)
- Recompute the minimum of \( index \)’s bucket.

The first step takes \( O(1) \) time, and the second step takes at most \( O(\sqrt{n}) \) time, since only need to interate through the elements of the bucket once to find the minimum.

Therefore, we can do \( \text{update}(index, x) \) in \( O(\sqrt{n}) \) time.

What about \( \text{min}(i, j) \)? Well, we can break the query up into the parts of the query entirely contained in buckets, and the edges which aren’t fully contained in a bucket. We can then take the minimum across all of these values to find the answer to the query.

For example, here is how we would break up the query for \( \text{min}(1, 12) \):

Since only the two edges are not fully contained in buckets, and each side can have at most \( \sqrt{n} \) elements (if it had more, some of those elements would comprise a bucket), we know that we have at most \( 2\sqrt{n} \) elements we need to inspect individually.

And since there are at most \( \sqrt{n} \) buckets, we only need to look at a maximum of \( 3\sqrt{n} \) values to find the minimum of any query.

Thus, we can compute both \( \text{min}(i, j) \) and \( \text{update}(index, x) \) in \( O(\sqrt{n}) \) time, only using \( O(\sqrt{n}) \) space.
Can we do this faster?

Yes! We can get this down to $O(\log(n))$ time for both queries.

How? Well, suppose for a moment that $n$ is a power of 2.

Now we’ll take a similar approach as we did in the offline case, except we won’t compute the minimum for all of the powers of 2 of each element.

Instead, we’ll construct a tree which divides the space in two at each level of the tree. For example:

Note that the width of a node corresponds the portion of the array it represents, but each node takes a constant amount of space (since it just needs to store its left and right children, and the min value). What if $n$ is not a power of 2? Then just pretend that $n$ is the next power of two, and compute the segment tree as though it was.
Here’s an example of that in action:

Now how do we do our queries across this structure? First, let’s try implementing $update(index, x)$.

We can start by updating the element sized node in the segment tree. Now, we can percolate this update upwards. To update a node, simply take its two children, and set the node’s value to the min of the two.

Since each $index$ is only contained in $\log(n)$ nodes at most, this operation takes $O(\log(n))$ time at most. For example, the nodes affected when 2 is changed have been highlighted in red:
Now, how do we find \( \min(i, j) \)? As before, let’s try breaking the interval up into piece that we’ve already solved, and then recombine them. How do we do that?

Well, let’s start by passing the query to the largest node. Then we’ll run into one of the following three cases:

- \([i, j]\) is the exact bounds of the node. We can just return the minimum of this node.

- \([i, j]\) is split by the midpoint of the bounds of the node. We can then split the interval \([i, j]\) by the midpoint, call the query on the two children, and then return the minimum of the two.
• \([i, j]\) lies entirely on one side of the midpoint. We can then just call the query on that child.

For example, these are the values of the segment tree we use to calculate the minimum for \(\text{min}(5, 12)\):

If we work it out, we end up with at most \(O(\log(n))\) values to compare, which means that we can find the minimum in \(O(\log(n))\) time!

Now how do we build a segment tree? The naive way is to compute the answer for \([0, n]\) for the largest node, and then recurse on the children. However, this takes \(O(n \log(n))\) time!
Instead, we can compute the answer for the two children, and then combine the children’s answers to form the node’s answer in constant time.

This allows us to construct the segment tree in $O(n)$ time, and because we have $O(n)$ nodes, it takes $O(n)$ space.

How do we go about representing this? Well, we could represent the tree as nodes, which works, but doesn’t take advantage of the fact that we know that our segment tree is nearly balanced.

Instead, we can treat it like a heap, where $A[0]$ represents the largest node $(0, n)$, and if a node is at index $i$, then its left child is at index $2i + 1$ and its right child is at index $2i + 2$. Then we don’t have to store two extra pointers for our children, and we get the added benefit of localized memory access.

In general, if we have a query $q(i,j)$ we’d like to compute quickly, and we want to construct a square root decomposition, or a segment tree to answer it, we only need one thing: Given the solution $q(i,k)$ and $q(k+1,j)$, can you find $q(i,j)$ quickly?

Can we go faster? Well, you’re a demanding one, aren’t you.

No.

**Problem 5**

Suppose that we have two kinds of queries we’d like to be able to handle, $rotate(i,j)$, which shifts every value in $[i,j]$ over to the left by 1 (with $A[i]$ moving to $j$), and $sum(i,j)$, which returns the sum of $A$ in $[i,j]$.

Can we do this using a Segment Tree?

Unfortunately not. $rotate(i,j)$ could change the values in most of the array ($O(n)$ values), which means that we’d need to recalculate $O(n)$ nodes in the segment tree. However, if updating a node is simple, we have another technique which has fewer nodes to update...

What about square root decomposition?

Well, again, let’s break up the array into buckets, and store the sum inside each bucket.
For example:

Now, we can implement \( \text{sum}(i, j) \) just as we computed \( \text{min}(i, j) \) earlier, except we add the elements together.

Now how do we implement \( \text{rotate}(i, j) \)? Well, when we do a rotation, most of the elements in a bucket don’t actually leave the bucket. Actually, when we rotate, at most one element leaves each bucket, and one element enters each bucket.

Then, we simply need to add the new element to the sum, and remove the exiting element.

Here’s an example of \( \text{rotate}(3, 10) \). I’ve highlighted the value that change buckets in red.

Since we’ll need to change at most \( \sqrt{n} \) buckets, and changing a bucket takes \( O(1) \) time, we can implement this in \( O(\sqrt{n}) \). This assumes that the shift in the underlying array is handled for us, but we will learn a technique to deal with that.

**Problem 6**

Suppose that we have two kinds of queries we’d like to be able to handle, \( \text{rotate}(i, j) \), which shifts every value in \([i, j]\) over to the left by 1 (with \( A[i] \) moving to \( j \)), and \( \text{min}(i, j) \), which returns the minimum of \( A \) in \([i, j]\).
Can’t we just do the same thing we did in the previous problem? No!

Why not? Well, summation has a simple inverse operation. If I remove a value from my range, I just subtract it from the total. With the minimum, if I remove the minimum value from the range, I need to find the new one, by searching through the remaining elements.

So, now I’ll need to maintain more information in my buckets. What should I store?

Well, suppose I keep an ordered set in each bucket. What is an ordered set? Formally, we don’t care. For our purposes, it’s a data structure from which I can find the minimum in \( O(\log(n)) \) time, and I can insert and remove from in \( O(\log(n)) \) time. An example would be an AVL tree, or a Red-Black tree.

Now what can we do with this? Well, now when we shift an element in or out of our bucket, instead of adding or subtracting it from our sum, we add it or remove it from our ordered set. Since we can do this in \( O(\log(\sqrt{n})) \) time, and we need to update at most \( O(\sqrt{n}) \) buckets, this takes, at worst \( O(\sqrt{n} \log(n)) \) time.

We can then extract the minimum from each bucket in \( O(\log(n)) \) time, and since there are at worst \( \sqrt{n} \) buckets in our range, it takes \( O(\sqrt{n} \log(n)) \) time, at worst, to extract the minimums of all of the buckets, and then \( O(\sqrt{n}) \) time at worst to find the minimum in the range, meaning each query takes at worst \( O(\sqrt{n} \log(n)) \) time.

We can speed this up to \( O(\sqrt{n}) \) by storing the minimums themselves in the buckets.

**Problem 7**

Suppose that we have one kind of query we’d like to be able to handle, \( \text{mode}(i, j) \), which returns the most common value in \([i, j]\). We may also assume that the array is already sorted.

Now, this is another example of an *offline query*, where the underlying structure doesn’t need to change in response to queries. So, let’s try using a segment tree.

What do we need to build our segment tree? We need to solve two problems:

1. We need to store enough information at a node to get the answer for that node’s range.
2. We also need enough information to merge two adjacent nodes into a single node.
What should we store at each node? Well, we could probably store the mode of the range. That would solve problem 1.

Now, given two adjacent nodes, what do we need to store at each node to be able to merge them? Well, since the array is sorted, we know that the only elements who can appear in both nodes are the two elements adjacent to the

Let’s call the left node $A$, and the right node $B$. Then the mode of the merged node $C$ will be one of the following:

1. The mode of $A$
2. The mode of $B$
3. The rightmost element of $A$ or the leftmost element of $B$.

The third can only happen if those elements are equal. Here are images which visualize the above scenarios. Note that $|X|$ refers to the number of times $X$ appears in the range of the node.

Figure 1: The mode of $A$ is used
There is an edge case we need to look out for, which is when the left and right elements are the same value, or in other words, every value in the range is identical. Then we need to update the edge value as well as the mode value, as shown below:
Figure 4: Note that |left| changed from the child node.

We can now construct the segment tree as before, and answer queries in $O(\log(n))$ time.

Problem 8

Suppose that we have one kind of query we’d like to be able to handle, $\text{mode}(i, j)$, which returns the most common value in $[i, j]$. Now our array is unsorted.

Well this is easy! Can’t we just take the same approach as above? Well, remember that we need to solve two problems:

1. We need to store enough information at a node to get the answer for that node’s range.
2. We also need enough information to merge two adjacent nodes into a single node.

1 is solved by simply storing the mode for that range. What about 2? Well, now we have to store a count of all of the elements within that range, because any one of them could be the mode in the final range.

Great! Except, when we merge, we may need to update the count of every element in the range. This changes our runtime for merge from $O(1)$ to $O(n)$!
So how can we solve this? Well, we’ll take advantage of the fact that our queries are offline, meaning that we can solve them in whatever order we like. Why is this helpful?

Well, suppose we have the queries \textit{mode}(0, 112) and \textit{mode}(0, 113). Then, if we keep the counts of all of the elements in the range over [0, 112], we won’t need to recompute these counts for [0, 113]. Instead, we take the data we’ve kept from [0, 112], and simply add element at 113.

So how will we store the information? We’ll use two structures: one to keep track of the count of each element, and one to be able to find the mode of the range quickly.

We’ll use a \textit{map} from \textit{element} to \textit{int} to keep track of the count, and an \textit{ordered set} of \textit{tuples} of < \textit{element}, count > to be able to find the node quickly. We’ll have the ordered set sorted by count, so that we can simply find the maximum value and return the corresponding element.

Now, suppose we have the range [a, b], and we’d like to update it to [a, b + 1]? Then we take the following procedure:

1. Find the count of \textit{A}[b + 1].
2. Remove < \textit{A}[b + 1], count(\textit{A}[b + 1]) > from our set.
3. Increase count(\textit{A}[b + 1]) by 1.
4. Add < \textit{A}[b + 1], count(\textit{A}[b + 1]) > back to our set.

Steps 1 and 3 take \(O(1)\) time, and our ordered set allows us to insert, and remove in \(O(\log(n))\) time, so adding \(b + 1\) to our range takes \(O(\log(n))\) time.

Now what if we’d like to update [a, b] to [a, b − 1]? Then we follow a very similar procedure:

1. Find the count of \textit{A}[b].
2. Remove < \textit{A}[b], count(\textit{A}[b]) > from our set.
3. Decrease count(\textit{A}[b]) by 1.
4. If count(\textit{A}[b]) is not 0, add < \textit{A}[b], count(\textit{A}[b]) > back to our set.

The runtime analysis is the same as above, so removing \(b\) from our range also takes \(O(\log(n))\) time.

We can follow similar procedures for updating [a, b] to [a + 1, b] or [a − 1, b].
Now that we have this, how can we use it to speed up our queries?

Well, we’re going to use a square root decomposition, except this time we’ll do it across our queries. This also goes by the name of Mo’s algorithm. We will sort out queries as follows:

1. Separate our total array into buckets of size $\sqrt{n}$.
   (a) For example, if $n = 9$, then we have the buckets $[0, 2]$, $[3, 5]$ and $[6, 8]$.

2. Group each query into the bucket that contains its left end point.
   For example, if we have the queries $\text{mode}(0, 1)$, $\text{mode}(2, 8)$, and $\text{mode}(5, 6)$ and $n = 9$:
   (a) $\text{mode}(0, 1)$, $\text{mode}(2, 8)$ belong in the bucket $[0, 2]$
   (b) $\text{mode}(5, 6)$ belongs in the bucket $[3, 5]$.

3. Within each bucket, sort by the endpoints.
   For example, if we have the queries $\text{mode}(1, 8)$, $\text{mode}(2, 3)$, and $\text{mode}(0, 6)$ in our first bucket, then the order is:
   (a) $\text{mode}(2, 3)$
   (b) $\text{mode}(0, 6)$
   (c) $\text{mode}(1, 8)$

Now that we have this ordering, what can we do? Well, let’s try solving a single bucket, in the order we’ve described above.

The first query $[a_1, b_1]$, we solve in our brute force fashion, by starting at $a_1$, and adding every element from $a_1$ to $b_1$. Since there are at most $n$ elements in this range, this takes $O(n \log(n))$ time (the additional $\log(n)$ is because inserting into our ordered set takes $O(\log(n))$ time).

Now, to answer the next query, we simply transform the range $[a_1, b_1]$ to $[a_2, b_2]$, first by adding elements to the back of $[a_1, b_1]$, until we get the range $[a_1, b_2]$, then adding or removing elements from the front until we get the range $[a_2, b_2]$. We can then take the maximum pair in the ordered set to find the mode.

Now, how long does it take to transform $[a_i, b_i]$ to $[a_{i+1}, b_{i+1}]$? Well, we know that all of the query’s left end points lie in the same bucket, which is of size $\sqrt{n}$, so the number of points we add or remove from the front is bounded by $\sqrt{n}$.

What about adding to the back? Well, in the worst case, the number of points we add to the back is $O(n)!$ Isn’t this a problem?
Well, note that we’ve sorted the queries within the bucket by their right end-point. Meaning that we’ll only add elements to the back of the range. And since there are only $O(n)$ elements, we’ll only need to add a total of $O(n)$ elements to the back, over all of the queries in the bucket.

So, the runtime in each bucket is $O(\log(n)(\sqrt{n} \cdot q_i + n))$, where $q_i$ is the number of queries in bucket $i$. So the runtime across the whole array is simply the sum of the work across all of the buckets, which works out to be:

$$
\sum_{i=1}^{\sqrt{n}} \log(n)(\sqrt{n} \cdot q_i + n) = \log(n)(\sqrt{n} \cdot q + \sqrt{n} \cdot n) = \log(n)\sqrt{n}(q + n)
$$

So, we can solve the problem in $O(\log(n)\sqrt{n}(q + n) + q \log(q))$, where $q$ is the number of queries. The $q \log(q)$ comes from the fact that we have to sort the queries in the first place.

**Problem 9**

Now suppose that we have one kind of query we’d like to be able to handle, $median(i, j)$, which returns the median value in $[i, j]$.

We can follow a similar approach as above, except instead of storing a count of each element, we will store a balanced binary search tree containing every element in the range.

But how do we find a median of a binary search tree? Here, I will describe how to find the $kth$ element of a binary search tree.

Suppose that each node stores its value, and its size, which is the total number of nodes contained in the tree rooted at that node. Now, let us check the number of elements smaller than this node. There are three case:

1. The number of elements smaller than this node is greater than $k$. This means that the $kth$ element lies to the left of this node.
2. The number of elements smaller than this node is equal to $k$. This means that the $kth$ element is the value of this node.
3. The number of elements smaller than this node is less than $k$. This means that the $kth$ element lies to the right of this node.
So, to find the $k^{th}$ element of a binary search tree, we follow this procedure, which will take in a $k$ and a node:

1. $numFewer = size(node.left)$
2. if ($numFewer > k$): return $findK(node.left, k)$.
3. if ($numFewer == k$): return $node.value$.
4. if ($numFewer < k$): return $findK(node.right, k - (numFewer + 1))$.

The first two cases are pretty clear, but what about the last one? Well, we need to count all of the elements to the left of the right child, and see if that’s equal to $k$. The number of elements left of the right child are the size of the right child’s left child, the current node, and the current nodes left child.

So the following two equations are equivalent:

\[
\begin{align*}
size(node.left) + 1 + size(node.right.left) &= k \quad (2) \\
(\text{size} - (size(node.right.left) + 1)) &= size(node.right.left) \quad (3)
\end{align*}
\]

Hence, instead of explicitly keeping track of how many of our parent’s nodes are to the left of the current node, we can keep track of it implicitly in the $k$ value. This lets us find the $k^{th}$ value in a binary search tree in $O(h)$ time, where $h$ is the maximum height of the tree. In the case of a balanced binary search tree, that value is $O(\log(n))$.

Now using this, we simply keep a balanced binary tree of all of the values in the range $[a, b]$, and then run $findK(\frac{b - a + 1}{2}, root)$ on our tree. This will solve our problem in $O(\log(n) + \sqrt{n}(q + n) + q\log(q))$, as before.

Can we solve it faster? That is left as an exercise to the reader.