Range query data structures

Today we are going to explore a class of data structures for performing *range queries* and see how they can be applied to speed up algorithms. Our focus is therefore twofold. First, we would like to design and analyze a specific data structure, that we will refer to as a SegTree, and explore its power. Then, we will see how it can be used to improve other algorithms.

Objectives of this lecture

In this lecture, we will:

- Introduce the SegTree data structure
- See what kinds of problems SegTrees are capable of solving
- See how SegTrees and related data structures can be used to speed up algorithms

1 Range queries

Definition: Range queries

Given a sequence (but not strictly necessarily integers), a *range query* on that sequence asks about a property of some contiguous range of the data i to j.

For example, suppose we have a sequence of n integers $a[0], a[1], \ldots, a[n-1]$, and we want to support querying the sum between positions i to j. We will use the convention that the left index is inclusive, and the right index is exclusive. Here are two approaches to get warmed up.

Approach #1 For each query, just loop over positions i to $j-1$ and compute the sum. This takes $\Theta(j-i) = O(n)$ time. This is very simple, but not at all efficient if the sequence is long.

Approach #2 Start by *precomputing* the prefix sums

$$p[j] = \sum_{0 \le i < j} a[i],$$

then answer a query for the sum between i and j by returning $p[j] - p[i]$. This takes $O(n)$ preprocessing time, which seems reasonable, then each query can be answered in $O(1)$ time!

Approach #2 is basically optimal if we never plan to modify the elements of the array, but range queries become so much more useful if we allow modifications. So, our goal is to support an API that enables fast modifications *and* fast queries. Specifically, lets try to design a data structure that maintains an array of n integers and implements:

- **Assign**(i, x): Assign $a[i] \leftarrow x$,
- **RangeSum**(i, j): Return $\sum_{i \le k < j} a[k]$.

How would approaches #1 and #2 above fare now that we want to support modifications?

1. Approach #1 can implement **Assign** in $O(1)$ time by just assigning x to $a[i]$, then **RangeSum**s are still the same as before and take $O(n)$ time.
2. Approach #2 would require us to re-compute the prefix sums $p[j]$ for all $j < i$ whenever we perform **Assign**(i, x), which requires $\Theta(n - i) = O(n)$ time. Queries however still take $O(1)$ time which is nice.

So, in both cases we have one operation that takes $O(1)$ time, and the other which takes $O(n)$ time. If in some particular application, one of the operations is extraordinarily rare, maybe this is a good solution, but if we perform roughly half and half updates and queries, then both solutions are taking $O(n)$ time on average for each operation. This is not great at all. Can we design a data structure that makes both operations fast?

You may have already seen a data structure that can do this. In fact, we've already seen it in this course! Augmented balanced binary search trees (e.g., Splay Trees) can be used to solve this problem in just $O(\log n)$ time per operation and $O(n)$ words of space. However, they are tricky to implement, and often in practice the constant factor is quite high, making them somewhat less practical. Splay Trees also give amortized bounds rather than worst case, though you can improve this by using AVL trees or Red-Black trees.

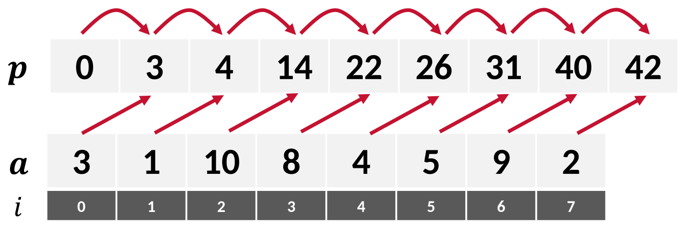
Today, we are going to design a data structure for this problem with the same bounds, $O(\log n)$ *worst-case* time per operation and $O(n)$ words of space, but that is much simpler to implement, and much faster in practice due to smaller constants hidden by the big-O. We will refer to this data structure as a SegTree[1]. We will also discuss how to generalize SegTrees to handle a much wider class of problems, where the $\sum$ operation is replaced by an arbitrary associative operator, enabling us to perform many different kinds of range queries with just one data structure!

2 Making range queries dynamic

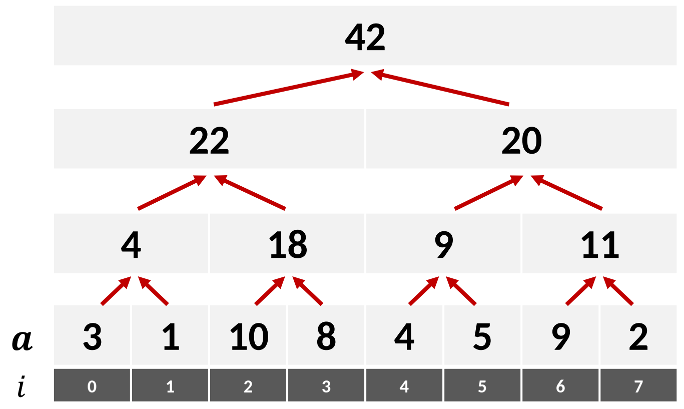
Let's take a step back and have a closer look at Approach #2 from earlier and see if we can find inspiration for a better algorithm. What the algorithm from Approach #2 is really doing is computing the sum from 1 to n in a sequential loop and just saving the results along the way.

[1] "SegTree" has become the traditional name for this data structure in 15-451, though you might not find it called that in other places. In the competition programming literature they are called "segment trees". However, this name conflicts with another specifically augmented binary search tree that represents a set of line segments in the plane. To remove this ambiguity, Danny Sleator coined the name "SegTree" and it has stuck.

The inefficiency of performing updates was due to the fact that if we edit element 0, there are n values in p that depend on it, and hence we do $O(n)$ work in updating everything.



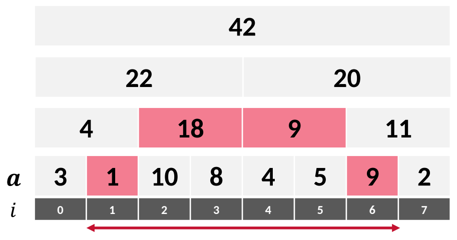
The *dependencies* here are what make the update algorithm slow. Is there instead an alternative way to break up the computation such that most of the intermediate calculations depends on fewer of the numbers? You may or may not have seen this idea before when seeking a *parallel algorithm* for computing the sum (or in general, any reduction) of a range of values. The left-to-right sequential sum is completely not parallel because of the dependencies, but a *divide-and-conquer* algorithm avoids this problem.

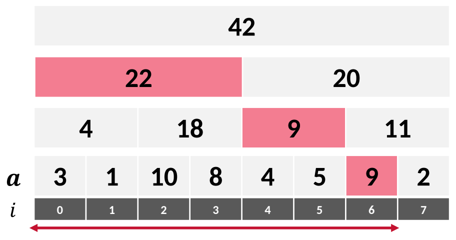


The divide-and-conquer sum has fewer dependencies because each element of the input only affects $\log_2 n$ intermediate vales produced by the computation. This means that if we update an element of the input, the output could be updated efficiently[2]. Its not clear yet though how we can actually answer **RangeSum** queries using this information, so lets figure that out now.

Doing queries The key idea is in figuring out how to build any interval $[i, j)$ that we might want to query out of some combination of the intervals represented by the divide-and-conquer tree. For example, if we wanted to query the interval $[1,7) = [1,6]$, we could add up the intervals $[1,1], [2,3], [4,5], [6,6]$ as shown below. Similarly, we can query $[0,7)$ as shown on the right.

[2]What we've actually made here is an extremely cool and powerful observation! It turns out that *parallel algorithms* are usually much easier to convert into dynamic algorithms / data structures than sequential ones, because they both share a common feature—both of them rely on having shallow dependence chains. An algorithm where all of the computations are dependent on the previous ones is hard to parallelize, and also hard to dynamize (make updatable) because changing a small amount of the input may change a large amount of the computation.



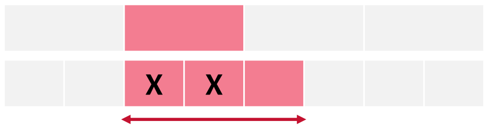


We need to somehow prove that we don't need too many intervals to make up any query interval. Because of course, we could always answer any query $[i, j)$ by summing up all of the intervals of size 1 from i to j, but that's just the first algorithm which takes $O(n)$ time.

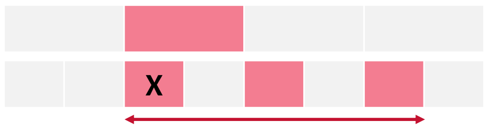
Lemma 1: Few blocks per level

Any interval $[i, j)$ can be made up of a set of disjoint intervals/blocks from the tree such that we use at most two intervals from any level.

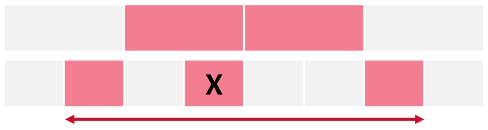
Proof. Suppose for the sake of contradiction that we use three *adjacent* blocks on the same level. Since each block is half the size of its parent, it must be the case that one of the two pairs of adjacent blocks could be replaced by its parent to cover the same interval, yielding a construction that uses fewer blocks.



Now suppose instead that we use three non-adjacent blocks on the same level. We can make a similar argument. Since the union of all the intervals gives one contiguous interval, it must be the case that the leftmost block is a right child of its parent (and symmetrically, that the rightmost block is a left child of its parent). If this were not the case, we could replace it with its parent to cover the same interval, yielding a construction that uses fewer blocks on this level.



It must be the case that the leftmost block is a right child and the rightmost block is a left child. So, the third block somewhere in between them could instead be covered by some intervals between the leftmost and rightmost block, yielding a construction that uses fewer blocks.



Applying this argument up the tree, there is a construction with at most two blocks per level. □

Since there is a construction with at most two blocks per level, we get the following corollary.

Corollary: $O(\log_2 n)$ blocks per query

Any interval $[i, j)$ can be made up of a set of at most $2\log_2 n$ disjoint blocks from the tree.

3 The data structure

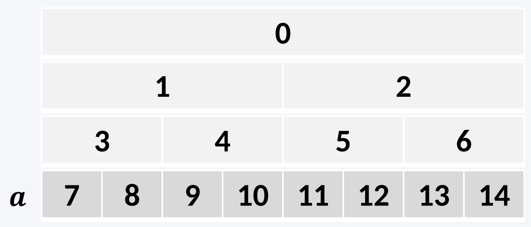
Now that we have the key ingredients, we can put together the data structure. For the moment let's assume that n is a power of two. If it is not, we can always round it up to the next one by at most doubling it, so it won't affect our asymptotic bounds. We will talk about some fairly low level implementation details today, more than we might often do so in this course.

One of the things that make common tree data structures inefficient is that traversing them requires chasing down pointers throughout the tree, each of which points to a node that might reside far away from the former in memory. To make SegTrees more efficient, we will apply the same indexing trick from the *binary heap* data structure that you may have seen before. It works because the SegTree data structure is a so-called *perfect binary tree* (every internal node has two children, and all leaves are on the same depth).

Definition: The binary heap indexing trick

Given a perfect binary tree on N nodes, we can lay it out in memory using an array of size N using the following scheme:

- Place the root node at position 0
- Given the node at position i, place its left child in position $2i+1$
- Given the node at position i, place its right child in position $2i+2$



Notice that the number of nodes $N = 2n-1$, and that when using this trick, the n input elements of a are all stored in the *last* n elements of the array. We can use these ideas to implement the data structure. We will make use of the following convenience functions:

- $\text{Parent}(i) := \lfloor (i-1)/2 \rfloor$. Returns the (index of the) parent of node i
- $\text{LeftChild}(i) := 2i+1$. Returns the (index of the) left child of i

- RightChild(i) $:= 2i+2$. Returns the (index of the) right child of i

Building the tree To build the tree, we start with the input $a[0\ldots(n-1)]$ which we copy into the leaves of the tree. We can then iterate over all of the internal nodes and fill in their values based on the values of their children. Note that the order we do this is important. We go from node $n-2$ down to node 0 (these are the indices of the internal nodes) since this corresponds to looping over the nodes on the lower levels before nodes on higher levels. If we tried to start at the root, its children might not have values yet, so this would not work!

Algorithm: Building a SegTree

```
class SegTree {
  nodes : list⟨Node⟩
  n : int }

class Node {
  val : int
  leftIdx : int
  rightIdx : int }

fun SegTree::constructor(a : list⟨int⟩) {
  n = size(a)
  nodes = array⟨Node⟩(2*n-1)
  for i in 0 to (n-1) do
    nodes[n + i - 1] = Node(a[i], i, i+1)
  for u in n-2 to 0 do {
    lNode = nodes[LeftChild(u)]
    rNode = nodes[RightChild(u)]
    nodes[u] = Node(lNode.val + rNode.val,
                    lNode.leftIdx,
                    rNode.rightIdx)
  }
}
```

Implementing updates To implement **Assign**(i,x), we just have to update the element at position i and all of its ancestor blocks in the tree. This takes $O(\log n)$ time and looks like this.

Algorithm: Updating a SegTree

```
fun Assign(i : int, x : int) {
  u := i + n - 1
  nodes[u].val = x
  while u > 0 do {
    u = Parent(u)
    nodes[u] = nodes[LeftChild(u)].val + nodes[RightChild(u)].val
  }
}
```

Implementing queries To implement **RangeSum**(i, j), we need to figure out how to identify the set of $O(\log n)$ blocks that make up $[i, j)$. Fortunately, we can do this very naturally with recursion. What we will do is essentially the same as the original divide-and-conquer sum, except we only need to recurse when we are on a block that contains some elements from $[i, j)$ and some elements not from $[i, j)$. Note, importantly, that we only recurse on both sides when necessary! Most of the time, the query will only recurse left or right but rarely both.

Algorithm: Querying a SegTree

```
fun RangeSum(i : int, j : int) {
  return sum(0, i, j)
}

fun sum(u : int, i : int, j : int) {
  node = nodes[u]
  if (i == node.leftIdx and node.rightIdx == j) return node.val
  else {
    mid = (node.leftIdx + node.rightIdx)/2
    if i >= mid then
      return sum(RightChild(u), i, j)
    else if j <= mid then
      return sum(LeftChild(u), i, j)
    else {
      return sum(LeftChild(u), i, mid) + sum(RightChild(u), mid, j)
    }
  }
}
```

4 Speeding up algorithms with range queries

One thing that we want to practice in this course is choosing the right data structure for the job when designing an algorithm. We've seen in recent lectures that applying hashing can often drastically reduce the running time of algorithms. How can range queries help us design better algorithms? If we find ourselves designing an algorithm that requires summing over, or taking the minimum or maximum of a set of numbers in a loop, then we may be able to improve it by substituting that code with a range query. Lets see an example.

Problem: Inversion count

Given a permutation p of 0 through $n-1$, the number of *inversions* in the permutation is the number of pairs i, j such that $i < j$ but $p[i] > p[j]$.

For example, the inversion count of the sorted permutation is 0, because everything is in order. The inversion count of the reverse sorted permutation is $\binom{n}{2}$ since every pair is out of order. Lets start by designing an inefficient algorithm. Per the definition, we can just loop over all pairs $i < j$ and check whether $p[i] > p[j]$.

```
fun inversions(p : list⟨int⟩) {
  n = size(p)
  result = 0
  for j in 0 to n - 1 do {
    for i in 0 to j - 1 do {
      if p[i] > p[j] then
        result = result + 1
    }
  }
  return result
}
```

This will take $O(n^2)$ time, but can we improve it somehow using range queries? Lets try to decompose what the loops of the algorithm are doing. The first one is considering each index j in order, straightforward enough. The second loop is considering all $i < j$ and counting the number of such i's that have been seen previously such that $p[i]$ is larger than $p[j]$. Okay, this is sounding like some kind of range query now because we are counting the number of things in a range... How exactly do we express this using a SegTree?

We need a range of values to correspond to the **count** of the number of elements that we have seen that are greater than a certain value. Let's store an indicator variable for each element x that contains a 1 if we have seen that element, or otherwise 0. To count the number of elements that are greater than $p[j]$ will therefore correspond to a range query of **RangeSum($p[j], n$)**. The optimized code for inversion counting therefore looks something like this.

Algorithm: Optimized inversion count using a SegTree

```
fun inversions(p : list⟨int⟩) {
  n = size(p)
  counts = SegTree(list⟨int⟩(n, 0)) // SegTree containing n zeros
  result = 0
  for j in 0 to n - 1 do {
    result = result + counts.RangeSum(p[j], n)
    counts.Assign(p[j], 1)
  }
  return result
}
```

Since each **Assign** and each **RangeSum** cost $O(\log n)$, the total cost of this algorithm is just $O(n \log n)$, which is a great improvement over the earlier $O(n^2)$ one!

5 Extensions of SegTrees

5.1 Other range queries

We just figured out how to implement SegTrees that support an **Assign** and **RangeSum** API. What makes SegTrees so versatile, though, is that they are not limited to only performing sums of integers. There was nothing particularly special about summing integers, except that it made for a good motivating example. Note that nowhere in our algorithm did we ever need to perform subtraction, which means we never made the assumption that the operation was invertible, which actually makes it even more general than the original “Approach #2” at the beginning! The exact same algorithm that we have just discussed can therefore also be used to implement range queries over **any associative operation**. In the code presented above, there are three lines that add the values computed at the child nodes. Replacing just these three lines with any other associative operation yields a correct data structure for performing range queries over that operator.

For example, we can also compute the maximum or minimum over a range by replacing $X+Y$ with $\max(X,Y)$ or $\min(X,Y)$. There's also no reason to restrict ourselves to integers. We can use any value type, such as floating-point values, or tuples of multiple values, as long as we are able to provide the corresponding associative operator.

Key Idea: SegTrees with any associative operation

SegTrees can be used with any associative operation, such as sum, min, max, but also even more complicated ones that you will see in recitation!

5.2 Other update operations

Our update operation for our vanilla SegTree is **Assign**(i,x), which sets the value of $a[i]$ to x. In some applications, instead of directly setting the value, we might want something slightly different. For example, we might want to *add* to the value instead of overwriting it. Fortunately, this can be supported with a combination of **Assign** and **RangeSum**. Note that we can always get the current value of $a[i]$ by performing **RangeSum**$(i,i+1)$, and then use that in an **Assign**. So to implement a new operation **Add**(i,x), which adds x to $a[i]$, we can just write

- **Add**(i,x): **Assign**$(i, x+$**RangeSum**$(i,i+1))$

The **Add** operation calls a constant number of SegTree operations, and hence it also runs in $O(\log n)$ time. This operation will be convenient for the next extension.

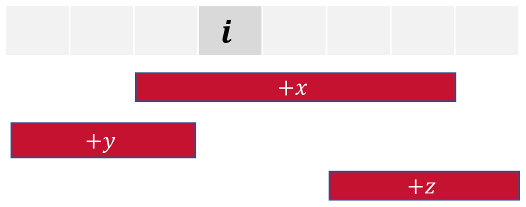
5.3 Flipping the operations

Our vanilla SegTree supports *point updates* and *range queries*, that is, we can edit the value of one element of the sequence and then query for properties (e.g., sum, min, max) of a range of values in $O(\log n)$ time. What if we want to do the opposite? Lets imagine that we want to support the following API over a sequence of n integers:

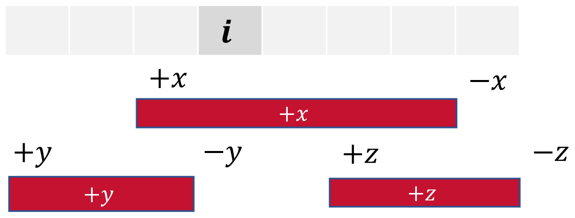
- **RangeAdd**(i,j,x): Add x to all elements $a[i],\dots,a[j-1]$
- **GetValue**(i): Return the value of $a[i]$

Rather than come up with a brand new data structure, lets try to use our original SegTree as a black box and reduce this new problem to the old one. This should always be your first choice when designing a new data structure or algorithm (can I reduce to something that I already know how to solve? This is almost always easier than designing something new from scratch!)

The idea Somehow we need to convert range additions into just a single update, and range sums into the ability to get a specific value. How might we do that? Well, notice that at a particular location i, the value $a[i]$ is equal to the *sum* of all of the **RangeAdd**s that have touched location i. That sounds like **RangeSum** should be able to help us then...



Consider the diagram above. The value of **Get**(i) is affected only by $+x$ since the ranges $+y$ and $+z$ do not touch i. Notice that more specifically, the value at i is the sum of all of the ranges whose starting point is at most i, but whose ending point is at least i. We can represent this using *prefix sums.*



Notice that the value of i is just the sum of the $+x$'s that occur at or before i, then subtract the $-x$'s that occur before at or before i (since the interval ended before it reached i). This is exactly just the prefix sums of all of these $+x$'s and $-x$'s up to position i. Therefore, we can use the **RangeSum** method to compute this prefix sum and hence implement **Get**.

Algorithm: RangeAdd and Get

We can implement the **RangeAdd** and **Get** API in terms of **Add** and **RangeSum** as follows.

- **RangeAdd**(i, j, x): **Add**(i, x); **Add**$(j, -x)$
- **Get**(i): **return RangeSum**$(0, i+1)$

Both **RangeAdd** and **Get** call a constant number of SegTree operations, and hence they both run in $O(\log n)$ time as well.

Note that in this algorithm we had to make use of *subtraction*, which means that it isn't applicable to any arbitrary associative operation anymore, since not every associative operator has an inverse. This algorithm is therefore only applicable to invertible associative operations.