Other Problems

We learned how to build the segment tree. Let's figure out how to solve the following problems.

The segment with the maximum sum

Now we consider the problem of finding a segment with a maximum sum. Our data structure must support two operations on the array:

- `set(i, v)`: set the element with index i to v.
- `max_segment()`: find the segment of the array with the maximum sum.

Let's try to build a segment tree that calculates the required function.

Consider the segment x, which is divided into two halves. We want for the segment x to find the value seg: the sum on the subsegment with the maximum sum. Note that knowing only seg_1 and seg_2 (answers for halves) we cannot get seg, because the answer for x can intersect both segments. But in case of intersection, the optimal segment consists of the suffix of the left half and the prefix of the right half. Let's record for each segment two more values: $pref$ and suf: prefix and suffix with the maximum sum. Then you can calculate seg as follows:
$seg = \max(seg_1, seg_2, suf_1 + pref_2)$.

How we need to recalculate $pref$ and suf. Consider $pref$, suf will be considered similarly. The maximum prefix is either the maximum prefix of the left half, or consists of the entire left half and the maximum prefix of the right half. Add to each node another value sum, equal to the sum on the segment. Then
$pref = \max(pref_1, sum_1 + pref_2)$, similarly
$suf = \max(suf_2, sum_2 + suf_1)$. Finally, the sum can be calculated using the formula $sum = sum_1 + sum_2$.

Similarly, we can construct a data structure with the additional operation `max_subsegment(l, r)`: find the subsegment of the segment from l to r with the maximum sum. To do this, you need to learn how to merge answers for segments, and this is what we just learned to do.

K-th one

Consider the problem of finding the k-th one. Our data structure must support two operations on the array:

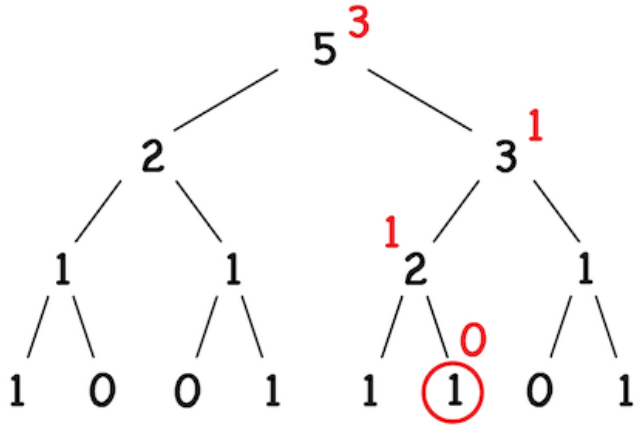
- `set(i, v)`: set element i to $v \in \{0, 1\}$.
- `find(k)`: find the index of the k-th one.

The main idea: we maintain a segment tree with the operation sum. Changing an element is done in a standard way. Finding the k-th one is equivalent to finding the leftmost prefix with the sum $k + 1$. The algorithm is quite simple. Suppose we need to find the k-th one on the segment $[l, r)$. If $r = l + 1$, then we found the desired

one. Otherwise, we look at the sum s on the left subsegment. If $k < s$, then the k-th one is in the left subtree, otherwise, we need to start the search for the one with index $k - s$ in the right subtree.

Obviously, the time of `find` is $O(\ logn)$.

Consider a small example. We construct a segment tree with the operation sum. And let's get `find(3)` query. We start at the root, the segment $[0, 8)$, and search of the third one. We look at the left subsegment $[0, 4)$ and see that it has a sum of 2, which is less than $k + 1 = 4$. Therefore, we go down to the right subsegment $[4, 8)$ and look for $k - 2 = 3 - 2 = 1$-th one on it. In the left subsegment $[4, 6)$, the sum is 2, which is less than or equal to $k + 1 = 1 + 1 = 2$, so our unit lies in the subsegment $[4, 6)$. And finally, in the left subsegment $[4, 5)$, the sum is 1, which is less than 2, which means our one is in the right subsegment $[5, 6)$.



The first element greater than or equal to x Consider the task of finding the first element greater than or equal to x. Our structure should support two operations on the array:

- `set(i, v)`: set element i to v.
- `first_above(x)`: find the first item greater than or equal to x.

Let's do the same as the previous task: we construct a tree of segments with the operation max. To search for an element in a segment, we go down to the left subtree, if there is a maximum of at least x, otherwise, we go down to the right subtree.

Now let's complicate the task: we need to respond to requests `first_above(`x`,` l`)`: find the first element to the right of l greater than or equal to x. To process this request, we will act recursively as follows. Suppose we want to find such an element on the segment. If the maximum on the left subsegment is greater than or equal to x and the segment intersects with $[l, n)$, then we recursively start from the left subsegment, if we did not find an element in the left subsegment, then we start from the right subsegment. The request will be processed int $O(\log n)$ time. The proof is similar to the proof of the statement about finding the sum on a segment: the number of "bad" segments, i.e. the segments for which we went down to the left and to the right do not exceed the number of segments containing the position l, which is $\log n$.