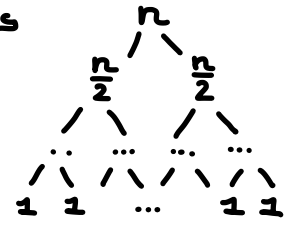
Problem 2. Given the inversion vector of a permutation of $\{1, \ldots, n\}$, reconstruct this permutation.

1) To solve our problem, we'll use a segment tree.

Building of a segment tree:

divide in: [left, right)

- From a segment array (arr[0, 1, …, n-1]), we divide at each step the segment until we reach length = 1. On each half, we call our "rule" that is at the basis of the structure.
  So, at all levels excluding the leaves, the segment tree will be filled.
- In our case, the leaves get value 1 (to represent that they have not been visited yet. If they get visited, their values will become 0).
  ↳ ex: our possible segment tree looks something like this



- In the parent nodes, we store the sum of the values in the left and right subtrees of the node.
- To build the tree, we need O(n), where n is the size of the input segment.

2) Updating the tree:

- BASE CASE:

  if (right - left) == 1:
    tree[current_node] == 0

  } if the difference between the right and left interval == 1 (so, we are in a leaf), we set its value to 0 ⇒ this means we are looking/have looked at this node and won't look at it again.

- if we aren't in a leaf, we take the middle = $\frac{\text{left} + \text{right}}{2}$ ⇒ ceiling function.

  index we want to update = i

  if i < middle (so we are on the left side):

    update(i, current_node · 2 + 1, left, middle)

  else (so we are on the right side):

    update(i, current_node · 2 + 2, middle, right)

  tree[current_node] = tree[current_node · 2 + 1] + tree[current_node · 2 + 2]
  ↳ we are simply looking at the left / right child of the parent and updating at index i. We then propagate the change all the way up.

3) MAIN IDEA: starting at i = n - 1, we traverse from right to left and construct the tree. For each element in the given inverse function, the algorithm identifies the K^{th} value in the original permutation. This will be achieved by recursively querying the segment tree, adjusting for the cumulative count of ones in the left subtrees. The original permutation will be then generated by recording the indices of the identified ones. To get the correct answer, we have to do a final reverse.

This works because:

- the segment tree efficiently maintains information about the sum of ones in different ranges, making it fast to find the K^{th} one
- We prevent duplicates by setting visited nodes = 0.
- We take advantage of the structure of the tree to find the K^{th} one in logarithmic time.

* When we are talking about the k^{th} value, we are referring to the k^{th} occurance of an element "x" in the original permutation.
  - ex: k = 1 ⇒ we find the index of the first occurance of "1" in the original permutation.

    k = 2 ⇒ we find the index of the first occurance of "2" in the original permutation.

    ..

```
int getans(int x, int lx, int rx,
           int k, int n)
{
    // Base Condition
    if (rx - lx == 1) {
        if (st[x] == k)
            return lx;
        return n;
    }

    // Split into two halves
    int m = (lx + rx) / 2;

    // Check if kth one is in left subtree
    // or right subtree of current node
    if (st[x * 2 + 1] >= k)
        return getans(x * 2 + 1,
                      lx, m, k, n);
    else
        return getans(x * 2 + 2, m,
                      rx, k - st[x * 2 + 1],
                      n);
}

// Function to generate the original
// permutation
void getPermutation(int inv[], int n)
{
    // Build segment tree
    build(0, 0, n);

    // Stores the original permutation
    vector<int> ans;

    for (int i = n - 1; i >= 0; i--) {

        // Find kth one
        int temp = getans(0, 0, n,
                          st[0] - inv[i], n);

        // Answer for arr[i]
        ans.push_back(temp + 1);

        // Setting found value back to 0
        update(max(0, temp), 0, 0, n);
    }
```