Hands-on #02: Segment tree

Competitive Programming and Contests

Jan Krejčí

November 23, 2022

1 Min and Max problem

1.1 Parsing input

First the program verifies validity of input. If the input does not meet the task conditions (line format, data types, etc) the program exits with the appropriate message: `panic!("Invalid input! type of error")`. New test set has been created for this purpose of input validity testing. Final implementation uses parts of this article[1].

1.2 Segment tree

Full understanding of the functioning and implementation of the Segment tree was based on articles on the GeeksforGeeks website[2] and youtube video[3].

1.2.1 Creation

After input is validated the Segment tree is constructed from the parsed input array. Segment tree is represented and stored in an array. Length of this array is equal to $N * 2 - 1$, where N is always power of two based on the length of input array (same value if it is already power of two or the next power of two). The implementation using the code from stackoverflow[4].

Elements of the array are `Nodes` – struct that stores positive integers. The array is first initialized with all zeros – minimal value. Then algorithm goes recursively through segment tree (root-to-leaves). Always calculates middle by dividing the current range into two halves, and then call the same construct function on both halves. The calculation of the index of the children in the array is as follows:

- $2 * i + 2$ for left child,
- $2 * i + 1$ for right child.

If proper position of element of an input array is found, the value is stored in segment tree. Parents of these nodes store the max value of their two children so the whole segment tree is prepared for *Max* operation, described in next subsection 1.2.2.

1.2.2 Queries

When going through the tree, in each node there is checked if the node's range overlaps with target range. It is always total, partial or no overlap. Depending on the type of operation, the necessary action is performed in these states.

Update operation replaces value if element of an array in that range is found. On partial overlap it recursively calls update function for both children with adapted range. Since it's recursive function,

[1] https://www.becomebetterprogrammer.com/rust-read-user-input-stdin
[2] https://www.geeksforgeeks.org/segment-tree-sum-of-given-range
[3] https://youtu.be/ZBHKZF5w4YU2
[4] https://stackoverflow.com/questions/66253909/how-do-i-find-the-next-power-of-2-for-a-u64-integer-in-rust-efficiently

parent node's value is also updated so possible value change is reflected in the entire tree (path from updated node to the root).

The segment tree is ready for the *Max* operation. When there is a total overlap it stops the recursion and return value of current node. On partial overlap it recursively go to both children and from their results it picks up the maximum and return it. In the case of no overlap the minimum value is returned.

1.3 Time and Space complexity

The time to construct the segment tree array that is described in subsection 1.2.1 is $O(n)$, because Segment tree can contain a maximum of $4 * n + 1$ nodes. *Max* and *Update* operations are the range queries. Segment tree allows answering these queries over an array very efficiently in $O(\log n)$ time. The extra space required is $O(n)$ to store the segment tree as an array.

2 Queries of operations problem

The second problem is described very briefly, because main functional parts have been described in a previous one.

2.1 Implementation

Input validation is done in a similar way to the first problem described in subsection 1.1. Then two segment trees are constructed in the same way as in a previous problem 1.2.1. Struct `Node` is used again and also its modified version struct `NodeOp` for storing operations.

2.1.1 Queries

First tree is to process queries. After queries execution there is known which operations and how many times are to be executed. Segment tree is initialized with all zeros – how many times should be each operation executed. Then in a loop through all queries the required range of operations is incremented. After, the sum of all nodes (on a path from the leaf that represents operation to root node) is calculated. Each operations value d is multiplied by this value.

2.1.2 Operations

Second tree is constructed from the parsed input array. Zeros are stored into parent nodes during initialization. Similar to the previous tree, ranges are updated based on operations. After all operations values on the path leaf-root are added to nodes of input array. This modified array is printed to the standard output as a result.

2.2 Time and Space complexity

Since execution of queries and operations to each segment tree are again range queries, time complexities are $O(\log n)$ time, which meets the task specification $O((k+m)\log(n))$. Plus there is a $O(n)$ time for sums of root-to-leaf paths described in previous subsection 2.1. Space complexity is again $O(n)$.