Learning Data Structures: Segment Trees

Lecturers Notes - CPC@UTEC

November 2020

1 Introduction

Welcome to the CPC & IEEE CS *Learning Data Structures and algorithms* workshop! Today we will discuss a very important data structure in competitive programming: Segment trees.

1.1 Problem

Suppose we have and array A of length n with elements $A_1, A_2, A_3 \ldots A_n$ and we need to find the same of elements in a range. More specifically, lets say we need to answer some queries of the form $\text{query}(l, r)$, such that:

$$\text{query}(l, r) = \sum_{i=l}^{r} A_i$$

- Brute force solution in $O(n)$ per query.
- Fast $O(1)$ solution using *prefix-sum.*

Now we want to support another type of operation: $\text{update}(i, x)$, adding x to the element A_i. How can we do this?

- Updating in $O(1)$ and querying in $O(n)$.
- Updating in $O(n)$ and querying in $O(1)$.

Both of this solutions have a worst case complexity of $O(qn)$. This is too slow! We need a better solution.

Now we can even complicate things more, what if we also needed to support another operation: $\text{update}(l, r, x)$, adding x to all elements A_i such that $l \leq i \leq r$.

All of this are very common scenarios when solving problems, so having a general and efficient solution for solving all of these would be a great addition to our problem-solving algorithmic toolbox.

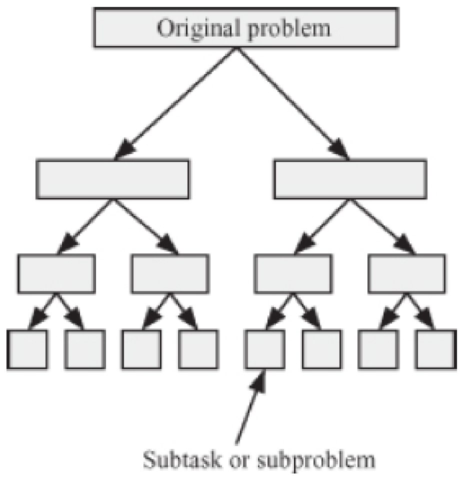
1.2 Divide & Conquer

Divide and conquer is one of the most popular problem solving paradigms in computer science. The key idea is that we can solve a problem by dividing it into smaller similar problems. We can recursively do this until the solution is trivial, and then join the solutions of this *sub-problems* in order to find the general solution. In general divide and conquer has 3 steps:

1. **Divide:** Splitting the original problem in smaller sub-problems.
2. **Conquer:** Solve each of this smaller problems (recursively).
3. **Merge:** Joining the smaller solutions to find the global solution.

Examples: Merge Sort, Sum

An interesting observation is that we can represent this problem / sub-problem structure as a tree, in which each node represents a problem and its children the sub-problems that need to be solved in order to find its solution:

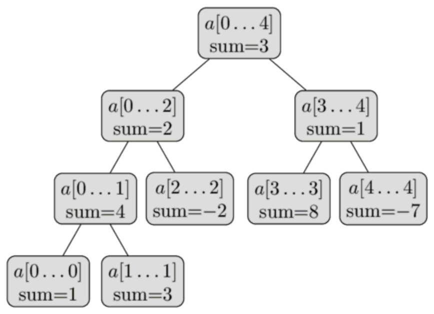


Now, what if we not only applied the divide and conquer algorithm, but also created this divide and conquer tree?

2 Segment Tree

The solution to all the problems presented above is this *divide and conquer tree* or as it is popularly known: *segment tree*. This data structure will allow us to efficiently answer range updates and queries.

In general we can think of a segment tree as a *binary tree* in which each node represents a range of values in an array, and each of the children represent a different half of this range:



In this lecture we will explore how we can use a segment tree to do the following operations efficiently:

- **Point Update:** Updating a single value.
- **Range Query:** Solving a query for a range of values.
- **Range Update:** Updating a range of values.

The segment tree will allow us to realize all of this operations in $O(log(n))$ time with just $O(n)$ preprocessing. This will allow for an overall complexity of $O(n + qlgn)$, way better than the naive $O(qn)$ brute force complexity.

2.1 Building a segment tree

The first thing we need to do in order to use the *segment tree* data structure, is to understand how we can transform a simple array into a segment tree. Lets say we have a node v that represents the range $[l, r]$ of the original array, then it is clear that before finding the value of $st[v]$, we need to first find the value that will be stored in the two children of node v.

As a segment tree is a binary tree we can use a smart indexing tree to easily maintain a tree structure, we will say that our tree is rooted at node 1, and for every node v its two children are nodes $2v$ and $2v + 1$.

We can combine these two ideas and arrive at the following algorithm:

```
void build($v, l, r$)
    if $l = r$ then
        $st[v] \leftarrow A[l]$;
    else
        $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
        build($2v, l, m$) ;
        build($2v + 1, m + 1, r$) ;
        $st[v] \leftarrow st[2v] + st[2v + 1]$
```

We can see that the time complexity of the construction of a segment tree obeys the following recurrence:

$T(n) = 2T\left(\frac{n}{2}\right) + O(1) \implies T(n) = O(n)$

Therefore, in order to set up our segment tree we only need $O(n)$ preprocessing.

2.2 Point Update

Probably the easiest operation to understand is the point query operation. We can see the update operation as looking for the node that represents the i-th element of the array, updating it, and then updating the $O(log(n))$ nodes that contain that element according to the changes in the tree.

We can easily implement this algorithm using recursion:

```
void update($v, l, r, i, x$)
    if $l = r$ then
        $st[v] \leftarrow st[v] + x$;
    else
        $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
        if $i \leq m$ then
            update($2v, l, m, i, x$) ;
        else
            update($2v + 1, m + 1, r, i, x$) ;
        $st[v] \leftarrow st[2v] + st[2v + 1]$
```

Evaluating the complexity of this solution is very simple. As in the general case at most one recursive call is made, we can express the complexity of the algorithm with the following recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$
$$\implies T(n) = O(log(n))$$

2.3 Range Query

The range query operation, even though it can be intimidating at first, has a very similar idea to the point update operation at heart. Here instead of looking for the node that represents our index, we will need to look for the *minimal set of nodes* that represent our range.

In order to better deal with these cases we will talk about three main situations:

- Total coverage
- Partial coverage
- No coverage

The easy cases are the total and null coverage, as we can simply return the value stored in our node, or zero respectively. The tricky case is when there is partial coverage of our range: The node contains the points that we care about, but also have some additional points. In this case we can separate the problem in two sub-problems: looking for a solution in our left child and a solution in our right child.

Using recursion and some implementation tricks to deal with edge cases, we can reach the following algorithm:

```
int query($v, l, r, ql, qr$)
    if $ql > qr$ then
        return 0 ;
    if $l = ql \wedge r = qr$ then
        return $st[v]$;
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
    $vl \leftarrow$ query($2v, l, m, ql, min(m, qr)$);
    $vr \leftarrow$ query($2v+1, m+1, r, max(m+1, ql), qr$);
    return $vl + vr$;
```

The time complexity of this algorithm is harder to analyze as with a naive glance it would appear that the recurrence this function obeys is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

As there are two recursive calls. This would be bad, as it would imply that $T(n) = O(n)$. So why does this solution work?

The key observation is that in the general case, at least one of this recursive calls should be a base case, which would mean that in reality we only have one recursive call, giving us the same recurrence as point updates and therefore a complexity of $O(log(n))$.

2.4 Range Update

We already know how to range query, so range updating shouldn't be that different, right? Wrong. We could naively think that we can recycle our code for querying (with some minor changes), and have a working code for updating. At the end of the day we can efficiently edit the $O(logn)$ nodes that cover our range. For example, we could try something like the following algorithm:

```
void update($v, l, r, ql, qr$)
    if $ql > qr$ then
        return;
    if $l = ql \wedge r = qr$ then
        $st[v] \leftarrow st[v] + x(r - l + 1)$;
        return;
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$;
    update($2v, l, m, ql, min(m, qr)$);
    update($2v + 1, m + 1, r, max(m + 1, ql), qr$);
    $st[v] \leftarrow st[2v] + st[2v + 1]$;
```

However, there is a big problem with this solution: we are only updating the *"topmost"* nodes that cover our range of interest, not all of them. What is worst is that we can show that our range is represented by $O(n)$ nodes, which means that in order to update all of them we should realize at least $O(n)$ operations. There must be a way of solving this problem

2.5 Lazy Propagation

The solution to the problem stated above is called *lazy propagation*, and it allow us to update the $O(n)$ nodes that represent our range while keeping the $O(log(n))$ time per query.

Lazy propagation is a technique we can add to our data structures to amortize the complexity of some of the update operations in our data structure. Lazy propagation is **not a technique that can be exclusively used on segment trees**, *Van Emde Boas trees*, *tries*, and other tree-like structures.

The key idea of lazy propagation is that we can procrastinate updating some nodes until it is absolutely necessary. In practice this can be achieved by only updating some of the nodes that cover our range of interest in the original update operation, and then amortizing the rest of updates in other operations such as queries and other updates. We will only update a set of nodes, and then *lazily* propagate the changes when we need to access lower nodes.

Example: Lazy Propagation in Segment Tree

When properly used, lazy propagation should have a minor impact on our runtime complexity. In general, the complexity of all our operations will now be multiplied by a factor $T_{push}(n)$: the time complexity of our propagation. For this reason we need to ensure that our propagation is as efficient as possible.

3 Segment Tree Variations

Segment trees are a very broad topic for research and algorithmic development, thus they have many variations and techniques. Some of them that have gained popularity in the competitive programming sphere are:

- Persistent segment tree
- Merge sort tree
- Segment tree beats
- Li-Chao tree

References

[1] CP-Algorithms: Segment Tree,
`https://cp-algorithms.com/data_structures/segment_tree.html`

[2] Algorithm Gym: Everything About Segment Trees,
`https://codeforces.com/blog/entry/15890`

[3] Algorithms Live: Segment Tree,
`https://youtu.be/Tr-xEGoByFQ`

[4] MIT Open Coursware, Persistent Data Structures,
`https://youtu.be/T0yzrZL1py0`