inzva Algorithm Programme 2018-2019

Bundle 8

Data Structures - 3

Editor
Murat Ekici

Reviewers
Kayacan Vesek

Contents

1 Segment Tree With Lazy Propagation

1.1 What We Have Learned Before About Updates

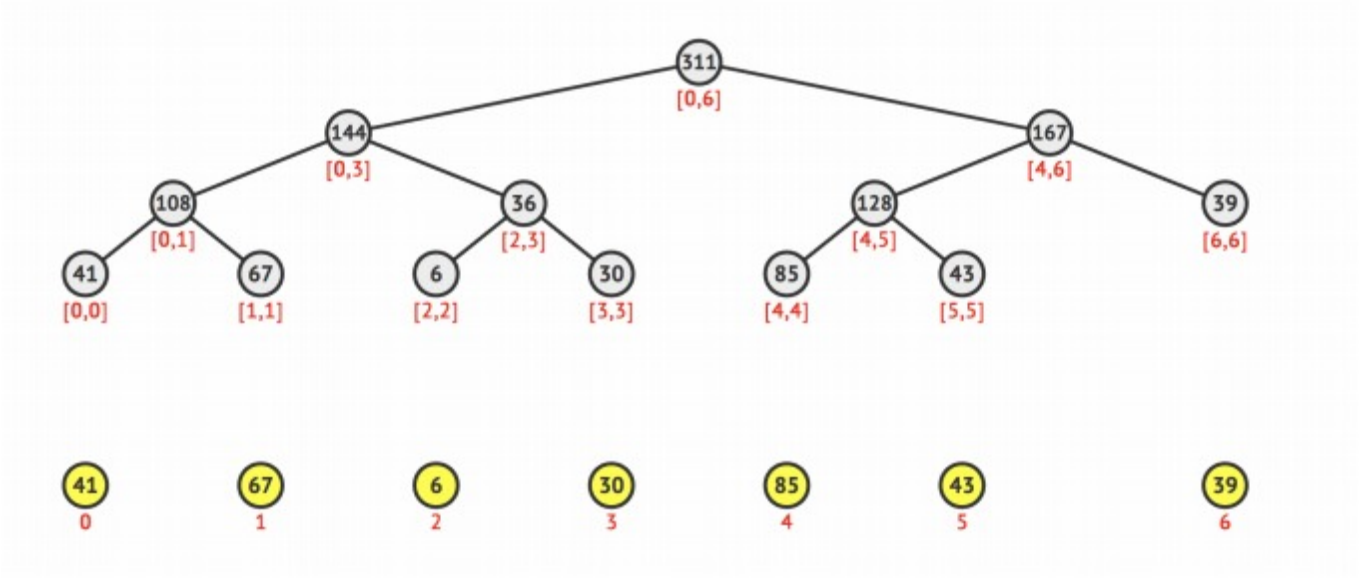

Figure 1: Segment Tree Structure

In the previous lectures, update function was called to update only a single value in array. Please note that a single value update in array may cause changes in multiple nodes in Segment Tree as there may be many segment tree nodes that have this changed single element in it's range.

Below is simple logic used in previous Segment Tree lecture.

- Start with root of segment tree.
- If array index to be updated is not in current node's range, then return.
- Else recur for children, then update this node's value.

What about updates on a range?

1.2 Lazy Propagation Algorithm

We need a structure that can perform following operations on an array $[1, N]$.

- Add *inc* to all elements in the given range $[l, r]$.
- Return the sum of all elements in the given range $[l, r]$.

Notice that if update was for single element, we could use the segment tree we have learned before. Trivial structure comes to mind is to use an array and do the operations by traversing and increasing the elements one by one. Both operations would take $O(L)$ time complexity in this structure where L is the number of elements in the given range.

Let's use segment tree's we have learned. Second operation is easy, We can do it in $O(logN)$. What about the first operation. Since we can do only single element update in the regular segment tree, we have to update all elements in the given range one by one. Thus we have to perform update operation L times. This works in $O(LlogN)$ for each range update. This looks bad, even worse than just using an array in a lot of cases.

So we need a better structure. People developed a trick called lazy propagation to perform range updates on a structure that can perform single update (This trick can be used in segment trees, treaps, k-d trees ...).

Trick is to be *lazy* i.e, do work only when needed. Do the updates only when you have to. Using Lazy Propagation we can do range updates in $O(logN)$ on standart segment tree. This is definitely fast enough.

1.3 Updates using Lazy Propagation

Let's be *lazy* as told, when we need to update an interval, we will update a node and mark its children that it needs to be updated and update them when needed. For this we need an array lazy[] of the same size as that of segment tree. Initially all the elements of the lazy[] array will be 0 representing that there is no pending update. If there is non-zero element lazy[k] then this element needs to update node k in the segment tree before making any query operation, then $lazy[2 * k]$ and $lazy[2 * k + 1]$ must be also updated correspondingly.

To update an interval we will keep 3 things in mind.

- If current segment tree node has any pending update, then first add that pending update to current node and push the update to it's children.
- If the interval represented by current node lies completely in the interval to update, then update the current node and update the lazy[] array for children nodes.
- If the interval represented by current node overlaps with the interval to update, then update the nodes as the earlier update function

```
void update(int node, int start, int end, int l, int r, int val) {
    if(lazy[node] != 0) {
        tree[node] += (end - start + 1) * lazy[node];
        if(start != end) {
            lazy[2 * node] += lazy[node];
            lazy[2 * node + 1] += lazy[node];
        }
        lazy[node] = 0;
    }
    if(start > r || end < l)
        return;
    if(l <= start && end <= r) [
        tree[node] += (end - start + 1) * val;
        if(start != end) {
            lazy[2 * node] += val;
            lazy[2 * node + 1] += val;
        }
        return;
    }
    int mid = (start + end) / 2;
    update(2 * node, start, mid, l, r, val);
    update(2 * node + 1, mid + 1, end, l, r, val);
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}
```

This is the update function for given problem. Notice that when we arrive a node, all the updates that we postponed that would effect this node will be performed since we are pushing them downwards as we go to this node. Thus this node will keep the exact values when the range updates are done without lazy. So it's seems like it is working. How about queries?

1.4 Queries using Lazy Propagation

Since we have changed the update function to postpone the update operation, we will have to change the query function also. The only change we need to make is to check if there is any pending update operation on that node. If there is a pending update operation, first update the node and then work same as the earlier query function. As we told in the last subsection, all the postponed updates that would effect this node will be performed before we reach this node. So sum value we look for will be there, correctly!

```
int query(int node, int start, int end, int l, int r) {
    if(start > r || end < l)
        return 0;
    if(lazy[node] != 0) {
        tree[node] += (end - start + 1) * lazy[node];
        if(start != end) {
            lazy[2 * node] += lazy[node];
            lazy[2 * node + 1] += lazy[node];
        }
        lazy[node] = 0;
    }
    if(l <= start && end <= r)
        return tree[node];
    int mid = (start + end) / 2;
    int p1 = query(2 * node, start, mid, l, r);
    int p2 = query(2 * node + 1, mid + 1, end, l, r);
    return (p1 + p2);
 }
```

Notice that only difference with regular query function is pushing the lazy values downwards as we travel. This is a widely used trick. You can use it for dozens of different problems. But not all range problems. You may have noticed that we used some properties of adding operation here. We used the fact that increasing updates has associative property. We merged more than one updates in *lazy* array, we didn't care about the order of this updates. This assumption must be made to use lazy propagation. Other properties that needed left as an exercise to the readers.

2 Binary Search on Segment Tree

Assume we have an array A that contains elements between 1 and M. We have to perform 2 kinds of operations.

- Change the value of the element in given index i by x.
- Return the value of the k^{th} element on the array when sorted.

2.1 How to Solve It Naively?

Let's construct a frequency array, $F[i]$ will keep how many times number i occurs in our original array. So we want to find smallest i such that $\sum_{j=1}^{j=i} F[i] \geq k$. Then the number i will be our answer for the query. And for updates we just have to change F array accordingly.

N = 5, M = 7, Q = 4	A = [1, 7, 2, 5, 2]	F = [0, 1, 2, 0, 0, 1, 0, 1]
Change A[1] by 3	A = [3, 7, 2, 5, 2]	F = [0, 0, 2, 1, 0, 1, 0, 1]
Ask for k=4	Sums are [0, 0, 2, 3, 3, 4, 4, 5]	Answer is 5
Change A[4] by 2	A = [3, 7, 2, 2, 2]	F = [0, 0, 3, 1, 0, 0, 0, 1]
Ask for k = 5	Sums are [0, 0, 3, 4, 4, 4, 4, 5]	Answer is 7
Ask for k = 3	Sums are [0, 0, 3, 4, 4, 4, 4, 5]	Answer is 2

Figure 2: An example for given problem

This is the naive algorithm. Update is $O(1)$ and query is $O(M)$.

```
1  void update(int i, int x) {
2      F[A[i]]--;
3      F[A[i] = x]++;
4  }
5
6  int query(int k) {
7      int sum = 0, ans = 0;
8      for(int i = 1; i <= M; i++) {
9          sum += F[i];
10         if(sum >= k)
11             return i;
12     }
13 }
```

2.2 How to Solve It With Segment Trees?

This is of course, slow. Let's use segment tree's to improve it. First we will construct a segment tree on F array. Segment tree will perform single element updates and range sum queries. We will use binary search to find corresponding i for k^{th} element queries.

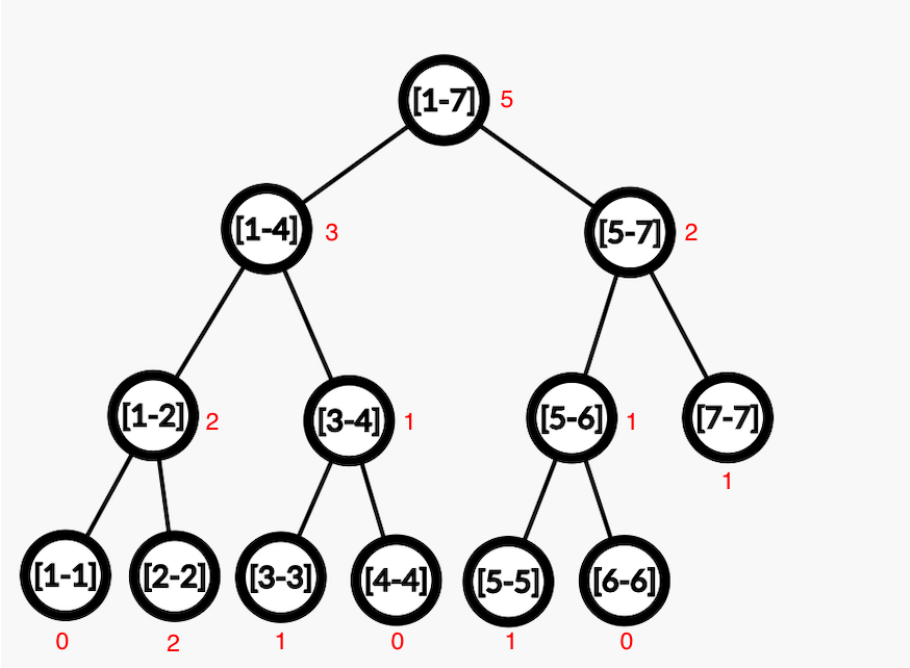

Figure 3: Segment Tree After First Update

```
void update(int i, int x) {
    update(1, 1, M, A[i], --F[A[i]]);
    A[i] = x;
    update(1, 1, M, A[i], ++F[A[i]]);
}

int query(int k) {
    int l = 1, r = m;
    while(l < r) {
        int mid = (l + r) / 2;
        if(query(1, 1, M, 1, mid) < k)
            l = mid + 1;
        else
            r = mid;
    } return l;
}
```

If you look at the code above you can notice that each update takes $O(logM)$ time and each query takes $O(log^2M)$ time, but we can do better.

2.3 Speed It Up?

If you look at the segment tree solution on preceding subsection you can see that queries are performed in $O(log^2 M)$ time. We can make is faster, actually we can reduce the time complexity to $O(logM)$ which is same with the time complexity for updates. We will do the binary search when we are traversing the segment tree. We first will start from the root and look at its left child's sum value, if this value is greater than k, this means our answer is somewhere in the left child's subtree. Otherwise it is somewhere in the right child's subtree. We will follow a path using this rule until we reach a leaf, then this will be our answer. Since we just traversed $O(logM)$ nodes (one node at each level), time complexity will be $O(logM)$. Look at the code below for better understanding.

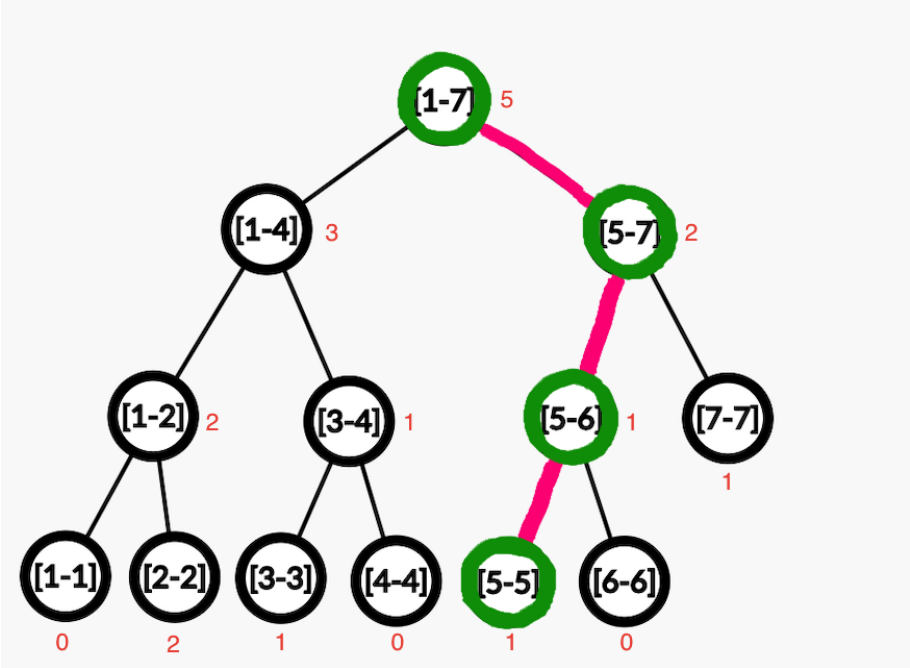

Figure 4: Solution of First Query

```
void update(int i, int x) {
    update(1, 1, M, A[i], --F[A[i]]); A[i] = x;
    update(1, 1, M, A[i], ++F[A[i]]);
}

int query(int node, int start, int end, int k) {
    if(start == end) return start;
    int mid = (start + end) / 2;
    if(tree[2 * node] >= k)
        return query(2 * node, start, mid, k);
    return query(2 * node + 1, mid + 1, end, k - tree[2 * node]);
}

int query(int k) {
    return query(1, 1, M, k);
}
```

3 Mo's Algorithm

This method will be a key for solving offline range queries on an array. By offline, we mean we can find the answers of these queries in any order we want and there are no updates. Let's introduce a problem and construct an efficient solution for it.

You have an array a with N elements such that it's elements ranges from 1 to M. You have to answer Q queries. Each is in the same type. You will be given a range $[l, r]$ for each query, you have to print how many different values are there in the subarray $[a_l, a_{l+1}..a_{r-1}, a_r]$.

First let's find a naive solution and improve it. Remember the frequency array we mentioned before. We will keep a frequency array that contains only given subarray's values. Number of values in this frequency array bigger than 0 will be our answer for given query. Then we have to update frequency array for next query. We will use $O(N)$ time for each query, so total complexity will be $O(Q \times N)$. Look at the code below for implementation.

```
class Query {
    public: int l, r, ind;
        Query(int l, int r, int ind) {
                this->l = l, this->r = r, this->ind = ind;
        }
};

void del(int ind, vector< int > &a, vector< int > &F, int &num) {
    if(F[a[ind]] == 1) num--; F[a[ind]]--;
}

void add(int ind, vector< int > &a, vector< int > &F, int &num) {
    if(F[a[ind]] == 0) num++; F[a[ind]]++;
}

vector< int > solve(vector< int > &a, vector< Query > &q) {
    int Q = q.size(), N = a.size();
    int M = *max_element(a.begin(), a.end());
    vector< int > F(M + 1, 0); // This is frequency array we mentioned before
    vector< int > ans(Q, 0);
    int l = 0, r = -1, num = 0;
       for(int i = 0; i < Q; i++) {
               int nl = q[i].l, nr = q[i].r;
        while(l < nl) del(l++, a, F, num);
        while(l > nl) add(--l, a, F, num);
        while(r > nr) del(r--, a, F, num);
        while(r < nr) add(++r, a, F, num);
        ans[q[i].ind] = num;
    } return ans;
}
```

Time complexity for each query here is $O(N)$. So total complexity is $O(Q \times N)$. Just by changing the order of queries we will reduce this complexity to $O((Q + N) \times \sqrt{N})$.

3.1 Mo's algorithm

We will change the order of answering the queries such that overall complexity will be reduced drastically. We will use following cmp function to sort our queries and will answer them in this sorted order. Block size here is $\sqrt{N}$.

```
bool operator<(Query other) const {
    return make_pair(l / block_size, r) <
        make_pair(other.l / block_size, other.r);
}
```

Why does that work? Let's examine what we do here first then find the complexity. We divide $l's$ of queries into blocks. Block number of a given l is $\frac{l}{blocksize}$ (integer division). We sort the queries first by their block numbers then for same block numbers, we sort them by their $r's$. Sorting all queries will take $O(QlogQ)$ time. Let's look at how many times we will call add and del operations to change current r. For the same block $r's$ always increases. So for same block it is $O(N)$ since it can only increase. Since there are $\frac{N}{blocksize}$ blocks in total, it will be $O(N \times N/blocksize)$ operations in total. For same block, add and del operations that changes l will be called at most $O(blocksize)$ times for each query, since if block number is same then their $l's$ must differ at most by $O(blocksize)$. So overall it is $O(Q \times blocksize)$. Also when consecutive queries has different block numbers we will perform at most $O(N)$ operations, but notice that there are at most $O(N/blocksize)$ such consecutive queries, so it doesn't change the overall time complexity. If we pick $blocksize = \sqrt{N}$ overall complexity will be $O((Q + N) \times \sqrt{N})$. Full code is given below.

N = 16, M = 10, Q = 10	a = [1, 7, 2, 5, 2, 2, 3, 4, 5, 10, 5, 7, 9, 4, 6, 1]	q = [{0, 15, 0}, {1, 5, 1}, {5, 7, 2}, [7, 11, 3}, {14, 14, 4}, {7, 15, 5}, {3, 15, 6}, {6, 13, 7}, {8, 8, 8}, {0, 11, 0}]
Sorted q array for block_size = 4: q = [{1, 5, 1}, {0, 11, 9}, {0, 15, 0}, {3, 15, 6}, {5, 7, 2}, {7, 11, 3}, {6, 13, 7}, {7, 15, 5}, {8, 8, 8}, {14, 14, 4}]		
Query	Number of Operations — Total	Answer
1 5 1	1 + 5 = 6 — 6	3
0 11 9	1 + 6 = 7 — 13	7
0 15 0	0 + 4 = 4 — 17	9
3 15 6	3 + 0 = 3 — 20	9
5 7 2	2 + 8 = 10 — 30	3
7 11 3	2 + 4 = 6 — 36	4
6 13 7	1 + 2 = 3 — 39	6
7 15 5	1 + 2 = 3 — 42	7
8 8 8	1 + 7 = 8 — 51	1
14 14 4	6 + 6 = 12 — 63	1

Figure 5: Example for the Algorithm

```cpp
int block_size;

class Query {
        public:
                int l, r, ind;
                Query(int l, int r, int ind) {
                        this->l = l, this->r = r, this->ind = ind;
                }
                bool operator<(Query other) const {
                    return make_pair(l / block_size, r) <
                        make_pair(other.l / block_size, other.r);
                }
};

void del(int ind, vector< int > &a, vector< int > &F, int &num) {
    if(F[a[ind]] == 1) num--;
    F[a[ind]]--;
}

void add(int ind, vector< int > &a, vector< int > &F, int &num) {
    if(F[a[ind]] == 0) num++;
    F[a[ind]]++;
}

vector< int > solve(vector< int > &a, vector< Query > &q) {
    int Q = q.size(), N = a.size();
    int M = *max_element(a.begin(), a.end());
    block_size = sqrt(N);
    vector< int > F(M + 1, 0); // This is frequency array we mentioned before
    vector< int > ans(Q, 0);
    int l = 0, r = -1, num = 0;
           for(int i = 0; i < Q; i++) {
                   int nl = q[i].l, nr = q[i].r;
        while(l < nl) del(l++, a, F, num);
        while(l > nl) add(--l, a, F, num);
        while(r > nr) del(r--, a, F, num);
        while(r < nr) add(++r, a, F, num);
        ans[q[i].ind] = num;
    } return ans;
}
```

4 Trie

Trie is an efficient information re*Trie*val data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to M * log N, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the penalty is on Trie storage requirements (Please refer Applications of Trie for more details)

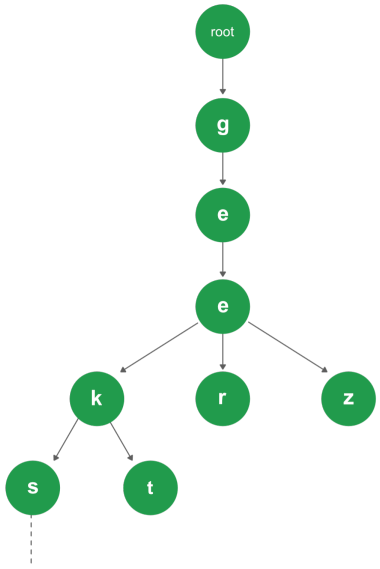

Figure 6: Trie Structure [5]

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field isEndOfWord is used to distinguish the node as end of word node. A simple structure to represent nodes of English alphabet can be as following,

```
// Trie node
class TrieNode {
    public:
        TrieNode *children[ALPHABET_SIZE];
        bool isEndOfWord;
        TrieNode() {
            isEndOfWord = false;
            for(int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = NULL;
        }
};
```

4.1 Insertion

Inserting a key into Trie is simple approach. Every character of input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next level Trie nodes. The key character acts as an index into the array children. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark end of word for last node. If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word. The key length determines Trie depth.

```
void insert(struct TrieNode *root, string key)  {
    struct TrieNode *pCrawl = root;
    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = new TrieNode;
        pCrawl = pCrawl->children[index];
    }
    pCrawl->isEndOfWord = true;
}
```

4.2 Search

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in Trie. In the former case, if the isEndofWord field of last node is true, then the key exists in Trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in Trie.

```
bool search(struct TrieNode *root, string key)  {
    TrieNode *pCrawl = root;
    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return false;
        pCrawl = pCrawl->children[index];
    }
    return (pCrawl != NULL && pCrawl->isEndOfWord);
}
```

Insert and search costs $O(key_length)$. However the memory requirements of Trie high. It is $O(ALPHABET_SIZE \times key_length \times N)$ where N is number of keys in Trie. There are efficient representation of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize memory requirements of trie.

References

[1] Lazy Propagation article on GeeksforGeeks

[2] Lazy Propagation article on HackerEarth

[3] Lazy Evaluation article on Wikipedia

[4] Wiki source for Mo's Algorithm

[5] Link to the figure 4