NOI.PH Training: DS 3

Advanced Data Structures

Kevin Charles Atienza

Contents

1 Introduction

By now, you should already know how to answer range queries on an array. You've already learned that *segment trees* allow you to answer many kinds of range queries and point updates efficiently, and that the *lazy propagation* technique allows you to handle some range updates efficiently as well.

Problems involving range queries and updates are pretty common nowadays, so you are expected to know how to deal with them, and at least master the basics. Such problems are usually easy to recognize since they explicitly tell you to perform certain operations on some given data. That's not always true, though; there are some problems where using a segment tree (or a similar structure) is part of a computation you wish to perform efficiently. The rule of thumb is to consider a segment tree as a general-purpose, black-box structure that you can use whenever you need to perform range operations, kinda like how you think of using a `map` whenever you need some kind key-value storage; you don't usually think of the implementation details.

I must admit that even knowing that you're supposed to use segment trees, some problems still take some creativity to solve. The creativity part can only be learned with more practice, but this document will discuss the *standard* part, which is much easier to learn.

Here are a few things to keep in mind:

- I will be using the terms *query*, *update* and *operation* in specific ways:
  - A **query** is an operation that asks for a certain value from the structure, but doesn't modify the structure in any way.
  - An **update** is an operation that modifies the structure in some way, but doesn't ask for a value.
  - An **operation** is either a query or an update.[1]
- I will also be using the following terms:
  - A **point operation** is an operation that only concerns a single location/index in the structure.
  - A **range operation** is an operation that concerns a (usually contiguous) set of indices in the structure.

  Terms like *point query*, *range update*, etc., should be self-explanatory.
- It's worth noting that a point operation can be considered a special kind of range operation, though we consider them separately since they can usually be solved with simpler techniques.
- We also distinguish between *static* and *dynamic* versions of problems/data structures:
  - A **static** problem/data structure means there won't be any updates.
  - A **dynamic** problem/data structure means there will updates.
- There's also *offline* and *online*:
  - We say our solution is **offline** if we know all the operations beforehand and we can process them in any order.

[1] This is not standard terminology; other sources might just call operations *queries* as well.

  - We say our solution is **online** if we don't know all the operations beforehand and we have to process them in the given order, without necessarily knowing what operations will follow.
  - There are occasions where we know all the operations beforehand but we still need to process them in the given order. We don't have a special term for it.

- One thing to remember here is that whenever you're implementing a new kind of data structure, then there's a good chance you won't get it completely right the first time. Debugging is inevitably a part of the task. To help with debugging, one advice I could give you is to write a brute-force solution and a simple random data generator and then use those to verify the output of your fast solution, at least for small-ish cases. Fortunately, the problems we'll be tackling here will usually have a brute-force solution that's easy to write.[2]

- In the following, sections marked with an asterisk (*) are considered optional. I recommend skipping them on the first reading.

- We will use zero-indexing unless otherwise specified.

- All of the implementations below can also be seen in a Google Drive folder that will be shared with you in Google classroom/Discord.

[2]This is not always the case; for example, geometry problems are notoriously hard to debug because of the difficulty in even writing the brute-force solution! Thus, you usually find geometry problems among the harder problems in a set.

2 Point manipulations

A segment tree with lazy propagation allows you to perform range operations quickly, but only on a fixed-size array. What if we also allow some modifications to the array, say insertion and deletion of elements?

To illustrate better, let's consider a particular problem with particularly simple queries.[3] Suppose we have an array A, and we want to implement these types of operations:

- **Range sum query.** Given i and j, find the sum of all elements from index i to index j; in other words, find $\sum_{i \le k \le j} A_k$.
- **Range increase update.** Given i, j and v, increase every element from index i to index j by v.
- **Point insert.** Given i and v, insert the value v so that v becomes the value at index i. (Thus, inserting at $i = 0$ and at $i = |A|$ is the same as inserting in front and at the back, respectively.)
- **Point delete.** Given i, remove the element at index i.

The new operations here are *point insert* and *point delete*, which we'll collectively call **point manipulations**. Without them, we only have range queries and updates which can easily be implemented using segment trees as usual, like this:

```
typedef long long ll;

class Segtree {
    int i, j;
    ll total, add = 0;
    Segtree *l, *r;

    void combine() {
        total = l->total + r->total;
    }
    void visit() {
        if (add) {
            total += add * (j - i + 1);
            if (l != NULL) l->add += add, r->add += add;
            add = 0;
        }
    }

public:
    Segtree(vector<ll>& a, int i, int j): i(i), j(j) {
        if (i == j) {
            l = r = NULL;
            total = a[i];
        } else {
            int k = i + j >> 1;
            l = new Segtree(a, i, k);
            r = new Segtree(a, k + 1, j);
            combine();
        }
    }

    void increase(int I, int J, ll v) {
```

[3] More complicated queries can be handled similarly. I will leave the reader to figure out the details.

```
        visit();
        if (I <= i && j <= J) {
            add += v;
            visit();
        } else if (!(J < i || j < I)) {
            l->increase(I, J, v);
            r->increase(I, J, v);
            combine();
        }
    }
    ll sum(int I, int J) {
        visit();
        if (I <= i && j <= J) {
            return total;
        } else if (J < i || j < I) {
            return 0LL;
        } else {
            return l->sum(I, J) + r->sum(I, J);
        }
    }
};

int main() {
    int n, q;
    cin >> n >> q;
    vector<ll> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    Segtree *tree = new Segtree(a, 0, a.size() - 1);
    for (int qq = 0; qq < q; qq++) {
        string typ;
        cin >> typ;
        if (typ == "increase") {
            int i, j; ll v;
            cin >> i >> j >> v;
            tree->increase(i, j, v);
        } else if (typ == "sum") {
            int i, j;
            cin >> i >> j;
            cout << tree->sum(i, j) << '\n';
        } else {
            cerr << "unknown command: " << typ << '\n';
            assert(false);
        }
    }
}
```

I invite you to come up with solutions yourself first before proceeding.[4]

[4] I will be making this invitation often in this document. When you see this, it's best to spend several hours yourself first before proceeding, since it gives you some preliminary intuition on the problem and also gives you an idea of the level of novelty/ingenuity of the solutions. The problems that will be discussed can usually be solved using techniques that you've already learned from the previous modules, although I will also be introducing new techniques.

2.1 Offline solution

First, let's discuss an easier variant of the problem. Let's assume that we know all the operations ahead of time (also known as offline operations). It turns out that this information will let us reduce the problem to the simpler variant—the one without point manipulations—which we already know how to solve!

To see why this works, notice that the difficulty behind point manipulations is that we need to allocate space between initial elements to account for future insertions, but we don't know how much to allocate! We also can't allocate arbitrarily big gaps between elements since that would be slow and would take up too much space. But if we know all the future manipulations, then we can predict precisely how much space to allocate beforehand!

So the key is to use the information from the operations to determine ahead of time how much extra space we need to allocate. To illustrate, consider the initial array

$$A = [10, 20, 30, 40, 50, 60, 70, 80, 90, 99]$$

(with $n = 10$) and the following sequence of operations:

- **insert** $i = 5$, $v = 17$
- **insert** $i = 9$, $v = 27$
- **insert** $i = 3$, $v = 37$
- **insert** $i = 10$, $v = 47$
- **insert** $i = 4$, $v = 57$

The array will look like this after every operation:

initial array	10	20	30	40	50	60	70	80	90	99					
after *insert*$(5, 17)$	10	20	30	40	50	17	60	70	80	90	99				
after *insert*$(9, 27)$	10	20	30	40	50	17	60	70	80	27	90	99			
after *insert*$(3, 37)$	10	20	30	37	40	50	17	60	70	80	27	90	99		
after *insert*$(10, 47)$	10	20	30	37	40	50	17	60	70	80	47	27	90	99	
after *insert*$(4, 57)$	10	20	30	37	57	40	50	17	60	70	80	47	27	90	99

If these are all the insertion operations and we know them all beforehand, then we can initially just allocate an array of length 15 and the strategically place the initial values at the right locations so that there is a spot for every future insertion, like so:

initial array	10	20	30	_	_	40	50	_	60	70	80	_	_	90	99
after *insert*$(5, 17)$	10	20	30	_	_	40	50	17	60	70	80	_	_	90	99
after *insert*$(9, 27)$	10	20	30	_	_	40	50	17	60	70	80	_	27	90	99
after *insert*$(3, 37)$	10	20	30	37	_	40	50	17	60	70	80	_	27	90	99
after *insert*$(10, 47)$	10	20	30	37	_	40	50	17	60	70	80	47	27	90	99
after *insert*$(4, 57)$	10	20	30	37	57	40	50	17	60	70	80	47	27	90	99

Initially, we declare the locations marked _ as *inactive* and the initial values *active*. Then each point insert is converted into a point update that turns an inactive cell active. This converts point manipulations into simple point updates which we know can already be handled with normal segment trees!

The remaining part is actually knowing how to do this initial *strategic placing*. This turns out to be a little tricky, but we can still solve it quickly by taking full advantage of the fact that we're doing offline operations. We will perform a preprocessing step that computes the eventual index of each operation. We also assume that there are no point deletes for simplicity. There are quite a few ways to do this, but we will show a particularly simple one.

To simplify things a bit, we consider the array to be initially empty, and treat the initial n elements as n inserts at the beginning. Also, let's ignore the queries for now, and assume that there are q' inserts. Then we can determine the eventual index of the inserts by doing the following procedure: (We will let (i_k, v_k) be the kth insert)

1. Let P be an initially empty array.
2. For each k in increasing order, we insert the number k at the i_kth position in P.
3. The eventual index of i_k (in the kth insert) will then be the final position of k in the array P.

Please ensure that you understand why this is correct.

Unfortunately, this procedure runs in $\mathcal{O}((n+q')^2)$ time in the worst case if implemented naively. We will need a data structure built on top of P that can insert elements at any position efficiently. There is, in fact, a structure that can do that, and it allows you to perform this procedure in $\mathcal{O}((n+q')\log(n+q'))$ time! (What is it?) But here, we will discuss a different way of doing it.

It turns out that things will be much easier if we perform the steps above in reverse!

1. Let P be an boolean array of size $n+q'$, all initially *true*.
2. For each k in *decreasing* order, find the index of the i_kth *true* value in P, say j. The eventual index of i_k (in the kth insert) will then be j. Then set $P[j]$ to be *false*.

Thus, we can implement this if we can find a structure that can do the following operations on a fixed boolean array:

- **Point update**. Given k, set $P[k]$ to *false*.
- **ith value query**. Given i, find the index of the ith *true* value.

The crucial insight here is that they only need to be performed on a *fixed* boolean array, so a segment tree can be used for it!

The following is an implementation. The idea is that the `add_X` methods should be called first, and after that, `compute_answers` will return the answers to all queries in the correct order. The implementation of the segment tree is omitted for clarity.

```
class Segtree {
    int i, j, size;
    ll total = 0, add = 0;
    Segtree *l, *r;
    // Implementation omitted for clarity.
    // You may view the full implementation at point_manip_offline.cpp
};

enum OpType { Insert, Increase, Sum };

```

```cpp
struct Operation {
    OpType type;
    int i, j;
    ll v;
};

class OfflineSums {
    vector<Operation> ops;
    int insertc = 0, queryc = 0;

public:
    void add_insert(int i, ll v) {
        ops.push_back({Insert, i, i, v});
        insertc++;
    }
    void add_increase(int i, int j, ll v) {
        ops.push_back({Increase, i, j, v});
    }
    void add_sum(int i, int j) {
        ops.push_back({Sum, i, j, 0LL});
        queryc++;
    }
    vector<ll> compute_answers() {
        // compute the eventual indices of all operations, in reverse.
        Segtree *tree = new Segtree(0, insertc - 1);
        for (auto it = ops.rbegin(); it != ops.rend(); it++) {
            Operation& o = *it;
            o.i = tree->get_kth_active(o.i);
            o.j = tree->get_kth_active(o.j);
            if (o.type == Insert) tree->deactivate(o.i);
        }

        // now, process the operations in order
        vector<ll> answers;
        answers.reserve(queryc);
        for (Operation& o: ops) {
            switch (o.type) {
                case Insert:
                    tree->activate(o.i, o.v);
                    break;
                case Increase:
                    tree->increase(o.i, o.j, o.v);
                    break;
                case Sum:
                    answers.push_back(tree->sum(o.i, o.j));
                    break;
            }
        }

        // cleanup and return
        delete tree;
        return answers;
    }
};
```

The implementation of the `Segtree` class is omitted since it's quite long. If you wish to view the full implementation, which includes a working `main()` function that takes input/output, see `point_manip_offline.cpp`.[5]

Each range query can then be computed in $\mathcal{O}(\log(n+q))$ time since the array could be up to $n+q$ elements long but is otherwise a normal segment tree. Overall, the algorithm runs in

[5] A slightly different implementation can be seen in `point_manip_offline2.cpp`.

$\mathcal{O}((n+q)\log(n+q))$ time.

Unfortunately, this solution doesn't support point deletions. Supporting point deletes is harder and is left to the reader as an exercise.

2.2 Sqrt decomposition

While the above is quite fast, unfortunately, we can't always use it; there are times when we are forced to process the operations online. The online version is harder since we can't pre-allocate the bigger array beforehand without knowing where we'll insert/delete future elements. Fortunately, there are still ways of solving it.

As a rule of thumb, problems that can be solved with advanced tree techniques (with logarithmic query times) often can also be solved using some form of sqrt decomposition.[6] It is your responsibility to determine whether a sqrt decomposition technique will run fast enough.

So let's try solving this with sqrt decomposition. The first thing that comes to mind is splitting the array into blocks of size, say, u. Then we somehow implement the operations, taking this decomposition in mind.

Range queries and updates can be handled as usual by considering each block separately. Blocks that are completely contained in the query range $[i, j]$ must be handled quickly, ideally in $\mathcal{O}(1)$ so that this runs in $\mathcal{O}(\frac{n}{u} + 2u) = \mathcal{O}(\frac{n}{u} + u)$ time. Since we're dealing with range sums, we can simply keep track of the sum of each block, and update it as needed.

But what about point insertions and deletions? Well, since each block is "small", we can simply rebuild the block that was updated. This runs in $\mathcal{O}(u)$ time. However, there's a catch. While deletion is fine since it reduces the size of the block, insertion makes the block larger, and if there are too many insertions done in a single block, then the size of that block could potentially be much larger than u, ruining the speed of our algorithm!

So we need a way to ensure that the size of each block remains small. Well, in this case, we can be simplistic: Whenever a block's size exceeds $2u$, then we simply split it into two blocks of sizes $\approx u$, and continue on as usual. This keeps the size of every block at most $2u$.

This technique keeps the block size manageable, but a consequence is that the number of blocks may now exceed $\Theta(n/u)$, which is also part of the running time of our operations. But how large can the number of blocks really get? Well, the only way we can end up with b blocks at the end is when we split blocks at least $b - \frac{n}{u}$ times, but splitting a block entails at least u inserts since each fresh block always starts at size $\approx u$. Therefore, there must be at least $u \cdot (b - \frac{n}{u})$ inserts, and the number of inserts can't exceed the number of queries, so we have $u \cdot (b - \frac{n}{u}) \leq q$, which is equivalent to $b \leq \frac{n+q}{u}$. This shows that there are at most $\Theta(\frac{n+q}{u})$ blocks!

So to summarize,

- Range updates and queries run in $\mathcal{O}(\frac{n+q}{u} + u)$ time.
- Point insertions and deletions run in $\mathcal{O}(\frac{n+q}{u} + u)$ time, the first term representing the occasion when we have to split a block into two, which requires moving the other blocks one place to the right, and the second term representing the cost of actually splitting the block.

We still have free choice on u, and we should choose this with the goal of minimizing $\mathcal{O}(\frac{n+q}{u} + u)$. To do so, we note that $\frac{n+q}{u}$ is decreasing and u is increasing as u increases. This, and using the fact that $\Theta(f(n)+g(n)) = \Theta(\max(f(n), g(n)))$ (why?), we find that the minimum

[6] The sqrt decomposition solution is usually easier to think of, but the running time is usually worse.

value of $\mathcal{O}(\frac{n+q}{u} + u)$ is when $\Theta(\frac{n+q}{u}) = \Theta(u)$, i.e., when $u = \Theta(\sqrt{n+q})$. This gives an overall complexity of $\mathcal{O}(\sqrt{n+q})$ per query, and $\mathcal{O}(n + q\sqrt{n+q})$ overall.[7]

Exercise 2.1. Show that $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$.

Here's one possible implementation. Here,

- `s` denotes the default size of a block. `u` is just `2*s` and represents the maximum size of a block.
- `block[b]` contains the elements at the bth block.
- `total[b]` represents the sum of the elements of the bth block.
- `toadd[b]` represents a lazy value that is added to each element of the block.

```
class SqrtDecomp {
    int s, u;
    vector<vector<ll>*> block;
    vector<ll> total, toadd;

    void visit(int b) { // propagate
        if (toadd[b]) {
            for (int i = 0; i < block[b]->size(); i++) {
                (*block[b])[i] += toadd[b];
            }
            toadd[b] = 0;
        }
    }

    void try_splitting(int b) { // try splitting the block if too large
        if (block[b]->size() >= u) {
            // insert the latter half of this block as a new block
            block.insert(block.begin() + b + 1,
                new vector<ll>(block[b]->begin() + s, block[b]->end()));
            // erase the latter half from this block
            block[b]->erase(block[b]->begin() + s, block[b]->end());
            block[b + 1]->reserve(u + 1);
            // update total and toadd as well.
            total.insert(total.begin() + b + 1,
                accumulate(block[b + 1]->begin(), block[b + 1]->end(), 0LL));
            total[b] = accumulate(block[b]->begin(), block[b]->end(), 0LL);
            toadd.insert(toadd.begin() + b + 1, 0LL);
        }
    }

public:
    SqrtDecomp(vector<ll>& a, int s): s(s), u(2*s) {
        // allocate n / s + 1 blocks
        block.resize(a.size() / s + 1);
        total.resize(block.size());
        toadd.resize(block.size());
        // initialize all blocks
        for (int b = 0; b < block.size(); b++) {
            block[b] = new vector<ll>;
            block[b]->reserve(u + 1);
```

[7] Strictly speaking, this only works when $q = \mathcal{O}(n^2)$; otherwise, we just set $u = n$ and conclude that the complexity is $\mathcal{O}(qn)$.

```
        }
        // push all elements to the blocks
        for (int i = 0; i < a.size(); i++) {
            block[i / s]->push_back(a[i]);
            total[i / s] += a[i];
        }
    }
    ~SqrtDecomp() {
        for (int b = 0; b < block.size(); b++) {
            delete block[b];
        }
    }

    void insert(int i, ll v) {
        for (int b = 0; b < block.size(); b++) {
            if (i > block[b]->size()) { // skip
                i -= block[b]->size();
            } else { // insert
                visit(b);
                block[b]->insert(block[b]->begin() + i, v);
                total[b] += v;
                try_splitting(b);
                return;
            }
        }
        throw "Out of bounds exception";
    }

    void erase(int i) {
        for (int b = 0; b < block.size(); b++) {
            if (i >= block[b]->size()) { // skip
                i -= block[b]->size();
            } else { // erase
                visit(b);
                total[b] -= (*block[b])[i];
                block[b]->erase(block[b]->begin() + i);
                return;
            }
        }
        throw "Out of bounds exception";
    }

    void increase(int I, int J, ll v) {
        for (int b = 0, i = 0; b < block.size(); i += block[b++]->size()) {
            int j = i + block[b]->size() - 1;
            if (I <= i && j <= J) { // contained
                toadd[b] += v;
                total[b] += v * block[b]->size();
            } else if (!(J < i || j < I)) { // partially overlaps
                visit(b);
                for (int k = 0; k < block[b]->size(); k++) {
                    if (I <= i + k && i + k <= J) {
                        (*block[b])[k] += v;
                        total[b] += v;
                    }
                }
            }
        }
    }

    ll sum(int I, int J) {
        ll ans = 0;
```

```
        for (int b = 0, i = 0; b < block.size(); i += block[b++]->size()) {
            int j = i + block[b]->size() - 1;
            if (I <= i && j <= J) { // contained
                ans += total[b];
            } else if (!(J < i || j < I)) { // partially overlaps
                visit(b);
                for (int k = 0; k < block[b]->size(); k++) {
                    if (I <= i + k && i + k <= J) {
                        ans += (*block[b])[k];
                    }
                }
            }
        }
        return ans;
    }
};
```

Keep in mind that this implementation is not terribly optimized, but I hope it gets the idea across.

Here's some sample usage:

```
int main() {
    int n, q;
    cin >> n >> q;
    int s = int(1 + sqrt(n + q));
    vector<ll> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];
    SqrtDecomp sd(a, s);
    while (q--) {
        string typ;
        cin >> typ;
        if (typ == "insert") {
            int i; ll v;
            cin >> i >> v;
            sd.insert(i, v);
        } else if (typ == "delete") {
            int i;
            cin >> i;
            sd.erase(i);
        } else if (typ == "increase") {
            int i, j; ll v;
            cin >> i >> j >> v;
            sd.increase(i, j, v);
        } else if (typ == "sum") {
            int i, j;
            cin >> i >> j;
            cout << sd.sum(i, j) << '\n';
        } else {
            cerr << "unknown command: " << typ << '\n';
            assert(false);
        }
    }
}
```

Note that we can also solve this problem using a different kind of sqrt decomposition, namely the "lazy" sqrt decomposition discussed in an earlier module. I recommend reviewing it. I leave solving this using lazy sqrt decomposition as an exercise to the reader.

Exercise 2.2. Use lazy sqrt decomposition to solve the problem.

Another thing to note is that in this implementation, the partition size $1+\sqrt{n+q}$ is chosen. This is just for simplicity; to get the most out of this technique, you just try to find the value that gives the best runtime especially for large n and q. This is highly dependent on implementation details. Although we set $s = \Theta(\sqrt{n+q})$ for the analysis to work, this freedom to choose s anyway is justified by the fact that there's a hidden constant behind the Θ notation.

2.3 Augmented binary trees

We can actually do much better! This problem can be solved with segment-tree-like techniques, but with a few important changes.

To begin, note that a *segment tree* is really just a binary tree where each node represents a subarray of an array. More specifically, it has the following properties:

- Each node represents a subarray $[i, j]$ of the array A and also contains an *aggregate value* for that subarray. (This aggregate value could be the sum, min, etc., and depends on the operation the segment tree is trying to implement.)
- Each node has either zero or two children.
- For each internal node, if its left child represents the subarray $[i_L, j_L]$ and its right child represents the subarray $[i_R, j_R]$, then the node itself represents the subarray $[i_L, j_R]$ and we also have $i_R = j_L + 1$.

Now, we usually use a perfectly balanced binary tree for this purpose to make the operations fast, i.e., $\mathcal{O}(\log n)$, but we can actually do this for *any* binary tree with the above properties. And the same sort of analysis shows that each tree operation can be performed in $\mathcal{O}(h)$ time where h is the height of the tree.

The usual way, which involves using perfectly balanced trees, enjoys the fact that h is as small as possible, but suffers from the fact that the shape of the tree is fixed. This means that we can't make any sort of insertions and deletions since they require changing the shape of the tree.

To fix this, we simply relax our requirement of using perfectly balanced binary trees, and simply use self-balancing binary trees as our segment tree! You've already learned two self-balancing binary trees, namely the AVL tree and the treap, and there's nothing stopping us from *augmenting* these trees with aggregate data and turning them into segment trees as well. And since these trees support insertion and deletion, we also gain that ability for our segment tree! And since their height is $\mathcal{O}(\log n)$, we are still guaranteed that the required operations run in $\mathcal{O}(\log n)$ time![8] We can even add lazy propagation to this generalized segment tree!

One important caveat is that whenever we change the shape of the tree, we must also take care to update the aggregate values as needed. But if you're already comfortable with implementing balancing trees, then this shouldn't be a problem.

Here's an implementation: `point_manip_avl.cpp`. Compare it with the normal segment tree implementation shown at the beginning of this section.

Here are a few remarks:

[8] Note: For treaps, this is $\mathcal{O}(\log n)$ *expected time*, while for AVL trees, this is $\mathcal{O}(\log n)$ *worst-case* time.

- The balancing rules are all implemented in the `balance` method. It tries to ensure that the node satisfies the AVL tree property by performing some rotations if necessary. It returns the new root.
- The `combine` and `visit` methods are still present and are used for recomputing aggregates and for lazy propagation.
- Unlike in the normal segment tree code, we don't keep track of the endpoints $[i, j]$ of each subtree anymore since insertions will mess those values up. Instead, we simply keep track of the size of the subtree, and then adjust the query indices/intervals as we go down the tree.
- On delete operations, it doesn't actually delete nodes; it just deactivates them by setting their size to 0. This is just for simplicity; you may also choose to actually delete nodes, which is simple in this case since we will always delete leaves.

As I mentioned, you can also use a treap here. I will leave that for you to implement. Later on, you'll encounter other types of self-balancing trees, and you may use those as well.

3 Range operations on enormous arrays

As I have mentioned in the introduction, using segment trees with lazy propagation, you can usually perform range queries and updates on an array, even if the array is long and there are many, many queries. But what if the array is even longer, like, *really* long?

Let's be more specific and consider a problem which is usually simple on a normal-size array. Suppose we have an array A of length n, all elements initially 0. There are two types of queries:

- **Range sum query.** Given i and j, find the sum of the elements from index i to index j.
- **Range increase update.** Given i, j and d, increment each element from index i to index j by d.

If n is small, say $\leq 10^5$, then we can simply use a segment tree with lazy propagation and solve the problem in $\mathcal{O}(n + q \log n)$ time, where q is the number of queries. However, what if the constraints say that $n \leq 10^{12}$? Unfortunately, we can't build a segment tree anymore since we will need up to a trillion leaves! This will definitely exceed the memory limit.

As usual, I invite you to come up with solutions yourself first before proceeding.

3.1 Sqrt decomposition

As usual, some form of sqrt decomposition works!

In this case, what can we do? Clearly, we can't simply do the usual sqrt decomposition step of chopping the array into blocks of size $\sqrt{n}$; since n is too large, this would mean 10^6 blocks, each of size 10^6.

But we can take advantage of the fact that there are far fewer queries/updates than elements of the array. This means that many parts of the array will contain equal values! More formally, I invite you to verify that after q queries, the array can be partitioned into $\leq 2q + 1$ subarrays where each subarray contains equal values.

Exercise 3.1. Show that after q range sum queries and range increase updates on an array with initially equal values, the array can be partitioned into $\leq 2q + 1$ subarrays where each subarray contains equal values.

Armed with this observation, we can represent the array A in a different way: as a list of subarrays $S = [S_1, S_2, \ldots, S_k]$, each of which contains just a single value. And since each subarray contains just one value, we can represent each subarray with just three numbers (i, j, v) where i and j are the leftmost and rightmost index of the subarray (relative to A), respectively, and v is the common value of all its elements. Initially, our array is represented as $S = [(0, n - 1, 0)]$.

Now, let's see how we can implement the operations.

- First, consider a range sum query (i, j). We don't have to do anything fancy here. We can simply consider each subarray (i', j', v) in S and compute its intersection with the interval $[i, j]$ (in $\mathcal{O}(1)$). Then we simply add v that many times to our running total. This runs in $\mathcal{O}(|S|)$ time.
- Next, consider a range update (i, j, d). In this operation, we still consider each subarray (i', j', v) and update each one separately. There are three cases, though:
  - If the intervals $[i, j]$ and $[i', j']$ are disjoint, then we don't have to do anything to this subarray. This runs in $\mathcal{O}(1)$ time.
  - If $[i, j]$ completely contains $[i', j']$, then we simply increase v by d. This runs in $\mathcal{O}(1)$ time.
  - Otherwise, $[i, j]$ partially overlaps with $[i', j']$. In this case, we have to *split* the subarray into two or three parts such that each resulting part is either completely contained in $[i, j]$ or disjoint with $[i, j]$, and in those cases, we already know what to do. Splitting a subarray into two requires moving the subarrays to its right one step to the right, but that can be done in $\mathcal{O}(|S|)$ time.

  Since there are $|S|$ subarrays, it seems like this operation runs in $\mathcal{O}(|S|^2)$ time. However, in fact, the last case only happens at most two times (why?), so the running time is, in fact, $\mathcal{O}(|S| + 2 \cdot |S|) = \mathcal{O}(|S|)$.

So what's the overall running time? Note that, as mentioned before, $|S| \leq 2q + 1$, and so, each operation runs in $\mathcal{O}(q)$ time. Overall, we get a running time of $\mathcal{O}(q^2)$. This is slightly better in that it doesn't contain an $\mathcal{O}(n)$ term, but it's not quite fast enough.

But this is where sqrt decomposition comes in. We place a limit on the size of S, say u, and whenever $|S|$ exceeds the threshold u, then we "start over" somehow. This technique should feel

familiar since it has been discussed in a previous module and is called "lazy sqrt decomposition". However, there's a slight problem since it's not yet clear how to do this "start over" step, since n is large.

One important observation here is that we can usually compute static range sums in $\mathcal{O}(1)$ time by doing an $\mathcal{O}(n)$ preprocessing step, namely prefix sums. In this case, though, we need to modify this step since $\mathcal{O}(n)$ is infeasible. We need to take advantage of the fact that there are $\leq 2q+1$ pieces of the array, and precompute a *compressed* version of the prefix sums. We can then do a binary search to compute static range sums in the compressed array in $\mathcal{O}(\log q)$ time.

We will use this compressed prefix sums trick as our "start over" step. Remember that it runs in $\mathcal{O}(q)$ time. Since we will be doing this start-over step at most $\mathcal{O}(\frac{2q}{u}) = \mathcal{O}(\frac{q}{u})$ (why?), the overall complexity becomes $\mathcal{O}(\frac{q^2}{u} + q(u + \log q))$.

We still have free choice on u, and our goal is to minimize this. To do so, we note that $\frac{q^2}{u}$ is decreasing and qu is increasing as u increases, and so using arguments we've used before, we want to solve $\Theta(\frac{q^2}{u}) = \Theta(qu)$, and we get $u = \Theta(\sqrt{q})$. This gives an overall complexity of $\mathcal{O}(q\sqrt{q})$.[9] This is much better than $\mathcal{O}(q^2)$!

Here's an implementation: `enormous_sqrt.cpp`.

We can also apply this technique to cases where the range query does an operation other than a sum, say a min or product. In those cases, we can't do the prefix-sum-like technique since not all operations are invertible. But in those cases, we can simply replace prefix sums with other known techniques like sparse tables or segment trees. However, note that the complexity could be slightly worse than $\mathcal{O}(q\sqrt{q})$.

As mentioned earlier, you still have freedom on the actual u value; select the one that gives the best runtime in practice.

3.2 Coordinate compression

We can actually improve the previous algorithm slightly in the case where we know all the queries ahead of time (i.e., *offline*). Instead of only splitting the array when we need, we can just pre-split the array into $\leq 2q+1$ parts based on all the queries. By doing so, we don't have to split the array again anymore!

More specifically, we consider all range operations, take all the endpoints of the ranges, sort them, and split the array at those points. This gives us $\leq 2q+1$ subarrays $S = [S_1, S_2, \ldots]$, but after doing so, we are now guaranteed that every operation will will never partially overlap with any subarray S_i. Thus, we can simply consider each element of S as virtually a single element on each operation!

The idea, then, is to construct a segment tree on S instead of on A. Since $|S| \leq 2q+1$, this can be done quickly. Furthermore, each range query and update can be done in $\mathcal{O}(\log |S|) = \mathcal{O}(\log q)$ time (by using the same analysis as basic segment trees).

Thus, overall, we get an $\mathcal{O}(q \log q)$ time algorithm! (This includes the $\mathcal{O}(q \log q)$ sorting step at the beginning.) Note the independence of the running time from n.

```
1  class Segtree {
2      ll i, j;
3      ll total, add = 0;
```

[9] Question: where did the $\mathcal{O}(\log q)$ part (binary search) go?

```
    Segtree *l, *r;

    // Implementation omitted for clarity.
    // You may view the full implementation at enormous_compress.cpp

public:
    Segtree(vector<ll>& bounds, int i, int j)
            : i(bounds[i])
            , j(bounds[j + 1] - 1) {
        if (i == j) {
            l = r = NULL;
            total = 0;
        } else {
            int k = i + j >> 1;
            l = new Segtree(bounds, i, k);
            r = new Segtree(bounds, k + 1, j);
            combine();
        }
    }
    // Implementation omitted for clarity.
};

enum OpType { Increase, Sum };

struct Operation {
    OpType type;
    ll i, j, v;
};

class OfflineSums {
    vector<Operation> ops;
    int queryc;
    ll n;

public:
    OfflineSums(ll n): n(n), queryc(0) {}
    void add_increase(ll i, ll j, ll v) {
        ops.push_back({Increase, i, j, v});
    }
    void add_sum(ll i, ll j) {
        ops.push_back({Sum, i, j, 0LL});
        queryc++;
    }
    vector<ll> compute_answers() {
        // compute boundaries
        set<ll> boundset({0, n});
        for (Operation& o: ops) {
            boundset.insert(o.i);
            boundset.insert(o.j + 1);
        }
        vector<ll> bounds(boundset.begin(), boundset.end());

        // compute the answers using the compressed segment tree
        Segtree *tree = new Segtree(bounds, 0, bounds.size() - 2);
        vector<ll> answers;
        answers.reserve(queryc);
        for (Operation& o: ops) {
            switch (o.type) {
                case Increase:
                    tree->increase(o.i, o.j, o.v);
                    break;
                case Sum:
```

```
                    answers.push_back(tree->sum(o.i, o.j));
                    break;
                }
            }

            // cleanup and return
            delete tree;
            return answers;
        }
};
```

As before, the implementation of the `Segtree` class is (mostly) omitted; it's actually the same as a regular segment tree except for slight changes in the constructor, that's why the constructor is shown.

The usage is similar to the offline solution shown in subsection 2.1. The full implementation (including sample usage) can be found here: `enormous_compress.cpp`.[10]

3.3 Implicit segment trees

Obviously, the previous technique only works when the queries are known beforehand. In some cases, this isn't true, but luckily, there are also ways of handling the *online* version of the problem.

This next technique can handle online queries and is a simple modification of the traditional segment tree solution. First, recall the standard segment tree solution to the problem, ignoring the fact that n is large:

```
class Segtree {
    ll i, j;
    ll total, add = 0;
    Segtree *l, *r;

    void combine() {
        total = l->total + r->total;
    }
    void visit() {
        if (add) {
            total += add * (j - i + 1);
            if (l != NULL) l->add += add, r->add += add;
            add = 0;
        }
    }

public:
    Segtree(ll i, ll j): i(i), j(j) {
        if (i == j) {
            l = r = NULL;
            total = 0;
        } else {
            ll k = i + j >> 1;
            l = new Segtree(i, k);
            r = new Segtree(k + 1, j);
            combine();
        }
```

[10] I've also provided an implementation with a slightly different flavor: `enormous_compress2.cpp`. The main difference is that this one uses an inclusive-exclusive scheme for intervals (i.e., $[i, j)$ instead of the usual inclusive-inclusive (i.e., $[i, j]$.)

```
    }

    void increase(ll I, ll J, ll v) {
        visit();
        if (I <= i && j <= J) {
            add += v;
            visit();
        } else if (!(J < i || j < I)) {
            l->increase(I, J, v);
            r->increase(I, J, v);
            combine();
        }
    }

    ll sum(ll I, ll J) {
        visit();
        if (I <= i && j <= J) {
            return total;
        } else if (J < i || j < I) {
            return 0;
        } else {
            return l->sum(I, J) + r->sum(I, J);
        }
    }
};
```

This runs in $\mathcal{O}(n + q \log n)$ time and uses $\mathcal{O}(n)$ memory, which is obviously too large.

But we can reduce this complexity by making our segment tree *lazy*. What I mean by this is that we don't construct the whole segment tree at the beginning, but rather we only construct them as needed. We still *imagine* that all the nodes are there, but we only construct a node when we actually need to visit it.

Here's a rough idea of what we want to do. First, we perform one simple change in our code:

```
class Segtree {
    ll i, j;
    ll total, add = 0;
    Segtree *_l, *_r;

    // ...

    Segtree *l() { return _l; }
    Segtree *r() { return _r; }

public:
    Segtree(ll i, ll j): i(i), j(j) {
        if (i == j) {
            _l = _r = NULL;
            total = 0;
        } else {
            ll k = i + j >> 1;
            _l = new Segtree(i, k);
            _r = new Segtree(k + 1, j);
            combine();
        }
    }
    // ...
};
```

And then we replace all instances of `l` and `r` with method calls `l()` and `r()`. The rest

implementation should stay the same.[11]

But now, we can easily make this *lazy* by only creating `_l` and `_r` when needed, like so:

```
class Segtree {
    ll i, j;
    ll total = 0, add = 0;
    Segtree *_l, *_r;

    // ...

    void construct_children() {
        if (i == j) {
            _l = _r = NULL;
        } else if (_l == NULL) {
            ll k = i + j >> 1;
            _l = new Segtree(i, k);
            _r = new Segtree(k + 1, j);
        }
    }

    Segtree *l() {
        construct_children();
        return _l;
    }
    Segtree *r() {
        construct_children();
        return _r;
    }

public:
    Segtree(ll i, ll j): i(i), j(j) {}

    // ...
}
```

Note that the creation of `_l` and `_r` is not in the constructor anymore; rather, it's inside the `construct_children` method which only gets called when either child is actually accessed. Furthermore, they are only created at most once.

The full implementation can be found here: `enormous_implicit.cpp`. Notice the minimal changes compared with the explicit version: `enormous_explicit.cpp`![12]

So, what's the complexity of this technique? Well, the $\mathcal{O}(n)$ preprocessing step is now gone because we've become lazy at the beginning, and the query time is still $\mathcal{O}(\log n)$, so the running time is $\mathcal{O}(q \log n)$. Also, the space complexity is $\mathcal{O}(q \log n)$ since we can only create $\mathcal{O}(\log n)$ new nodes per query. Since $\log n$ is small (even if n itself is large), this is (usually) acceptable![13]

3.4 Augmented binary trees

There are occasions where $\mathcal{O}(q \log n)$ space complexity is too much. Luckily, there's another technique that uses far less memory and time and that also works with online queries, and can actually do even more operations, in effect making it the best of all worlds! (Maybe the only downside is the potentially trickier implementation.)

[11] The `l` and `r` methods here are usually called *getters*.

[12] This is one of the attractive features of this technique; making it lazy gives us a huge benefit with little cost. This should give you an idea of why some programming languages are purely lazy by default.

[13] This gives you a taste of the power of lazy techniques. The "lazy" in "lazy propagation" operates on the same principle.

The idea is to use a self-balancing binary tree as before, and also augmenting each node with aggregate data. The difference is that this time, a leaf don't necessarily represent a single cell of the array, rather, it represents a whole subarray (containing equal values). And then, when we need to, we can split such leaves into two (e.g., during range updates). The split doesn't have to be balanced, unlike in the implicit segment tree technique, but rather, we will want to split it in just the right way so that the query can be decomposed as nice as possible.

This ensures that there are $2(2q + 1) - 1 = \mathcal{O}(q)$ nodes in our tree. Unfortunately, by not ensuring a balanced split, we can't necessarily guarantee that the height of the tree is $\mathcal{O}(\log n)$, but then, this is where the self-balancing nature of the tree helps us! We can consider splitting a node as an operation similar to inserting a new node in the tree, and we can then rebalance the tree (e.g., tree rotations in the case of AVL trees) after such an operation. This ensures that the height is always $\mathcal{O}(\log q)$!

Here's an implementation: `enormous_avl.cpp`. Note that the `balance` method is identical to that in the code presented in subsection 2.3.

Overall, this requires $\mathcal{O}(q \log q)$ time and $\mathcal{O}(q)$ memory. This also has the advantage that we can perform point manipulations (as discussed in subsection 2.3) just fine!

4 Range manipulations

We will now consider a harder variant of the problem considered in section 2.

Suppose we have an array A of length n, and we have the following types of operations:[14]

- **Range sum query.** Given i and j, find the sum of all elements from index i to index j.
- **Range increase update.** Given i, j and v, increase every element from index i to index j by v.
- **Range reversal.** Given i and j, reverse the subarray from index i to index j.
- **Range surgery.** Given i, j and k, move the subarray from index i to index j to position k. For this operation, let's assume that k is not contained in the interval $[i, j]$.

There are two types of surgery, depending on which side k is located with respect to the interval $[i, j]$.

We can also consider range inserts and deletes, and yes, there are ways to implement them, but I don't include them to keep this section simple.

The new operations *range reversal* and *range surgery* seem intimidating, and yes they kinda are. However, it turns out that reversals are quite powerful and they allow us to simplify things a bit: we can reduce all operations to their *prefix* counterparts!

The most obvious way to see this is converting range operations into a range surgery that brings the range in question to the front, performing a prefix operation, and then doing another surgery to put the subarray back to its original position.

Exercise 4.1. Reduce a general range operation into its equivalent prefix operation plus four prefix reversals.

Also, under what instances can we use just two prefix reversals instead?

As another example, verify that the operation "**reversal(i, j)**" is equivalent to the following sequence of operations: **prefixReversal(j)**, **prefixReversal($j-i$)**, **prefixReversal(j)**. Thus, if we can implement the slightly simpler *prefix reversal* operation, then we get range reversals for free, with only a constant overhead!

Finally, *range surgery* can also be reduced into prefix reversals. This reduction is left to the reader as exercise.

Exercise 4.2. Reduce range surgery into an $\mathcal{O}(1)$ number of prefix reversals.

Thus, we're now looking at the simplified problem with only the following operations:

- **Prefix sum query.**
- **Prefix increase update.**
- **Prefix reversal.**

However, even the prefix version looks hard!

As usual, I invite you to come up with solutions yourself first before proceeding.

[14] As usual, we will only consider range sum queries and range increments, and it will be left to you to implement other kinds of range operations.

4.1 Offline solution

Since the full problem is hard, let's try to consider the offline version first, where we know all the queries beforehand. We usually expect the offline version to be easier. What we hope to achieve here is a way to pre-partition the initial array so that range manipulations can be done much more quickly.

Unfortunately, in this problem, there doesn't seem to be an obvious way to do that! It looks like the offline version isn't really that much easier this time. The main issue is that prefix reversals make huge changes in the array, and so we can't really predict how to pre-partition the array, short of actually solving the problem with other techniques.

And so, unfortunately, I will end this section without giving any offline solution, since I don't have any![15]

4.2 Sqrt decomposition

As usual, our first bet is to use some form of sqrt decomposition. The first thing we consider is if we can split the initial array into blocks of $\sqrt{n}$ in size; unfortunately, that doesn't work since prefix reversals will mess those blocks up! Specifically, each reversal will most likely split one of the blocks into two, and so the number of blocks could possibly increase to $\Theta(n)$. We can kinda get around the issue by splitting the blocks as needed, but that would just increase the number of blocks to n in the worst case, which makes it slow.

So we abandon the pretense that we can somehow predict how to pre-partition the array, and instead use the lazy version of sqrt decomposition. Again, we can start by representing the array as a list of blocks $S = [S_1, S_2, \ldots, S_k]$ where each S_i represents a contiguous subarray of A. More specifically, we represent each S_i with three numbers, (l_i, r_i, rev_i), which says that S_i represents the contiguous subarray $A[l_i \ldots r_i]$. The rev_i thing is new, and it is a boolean representing whether the subarray is reversed or not. This is important since we're doing prefix reversals.

Initially, we have $S = [(0, n-1, \mathit{false})]$. And then, on prefix reversals, we will simply split one of these subarrays as needed.

Since we're doing range sums, we can keep track of prefix sums of the original array A. This allows us to compute the sum of any S_i in $\mathcal{O}(1)$ time.[16]

This representation allows us to perform a prefix reversal of size p as follows.

1. First, find the prefix of S such that there are at least p elements.
2. Next, if this prefix has strictly more than p elements, split the last subarray in that prefix to get a prefix with *exactly* p elements.
3. Reverse each subarray individually, i.e., flip the *rev* flag of each.
4. Reverse the prefix as a whole.

Any other prefix operation can be computed in a similar way, so the details will be left to the reader. Thus, we now have a way to perform any operation in $\mathcal{O}(|S|)$ time! Unfortunately, again, $|S|$ could be up to n, but this can be mitigated by setting a threshold on $|S|$, say u, and then whenever $|S|$ exceeds the threshold u, then we "start over" somehow. In this

[15] If you can come up with a simple offline solution, please let me know; I would be very interested in hearing it!

[16] As usual, more complicated range queries might require structures like sparse tables or segment trees.

case, "starting over" means actually rearranging the elements of the array since S represents a particular permutation (and possibly some insertions), but this can be done in $\mathcal{O}(n)$.

Analyzing everything as usual, we get the total complexity as $\mathcal{O}(n+\frac{nq}{u}+qu)$, and by setting $u=\Theta(\sqrt{n})$, we get a time complexity of $\mathcal{O}(n+q\sqrt{n})$.

Here's an implementation:[17]

```cpp
class SqrtDecomp {
    vector<ll> a;
    ll *total;
    int s;

    class Block {
        // Implementation omitted for clarity
    };

    vector<Block> block;

    void init_blocks() {
        for (int i = 0; i < a.size(); i++) {
            total[i + 1] = total[i] + a[i];
        }
        block.clear();
        block.reserve(s + 1);
        block.emplace_back(this, 0, a.size(), false);
    }

    void try_rebuild() { // rebuild if too large
        if (block.size() > s) {
            vector<ll> temp;
            temp.reserve(a.size());
            for (Block& b: block) b.push_values(temp);
            a = temp;
            init_blocks();
        }
    }

    int get_prefix(int i) {
        try_rebuild();
        int b;
        for (b = 0; i > 0 && b < block.size(); b++) {
            if (i >= block[b].size()) {
                i -= block[b].size();
            } else {
                block.insert(block.begin() + b + 1, block[b].split(i));
                return b + 1;
            }
        }
        return b;
    }

public:
    SqrtDecomp(vector<ll> &a, int s): a(a), s(s) {
        total = new ll[a.size() + 1];
        init_blocks();
    }
    ~SqrtDecomp() {
        delete[] total;
    }
```

[17] The full implementation can be seen in `range_manip_sqrt.cpp`

```
53
54      void prefix_reverse(int i) {
55          int pref = get_prefix(i);
56          for (int b = 0; b < pref; b++) block[b].reverse();
57          reverse(block.begin(), block.begin() + pref); // also reverse the blocks
58      }
59
60      void prefix_increase(int i, ll v) {
61          int pref = get_prefix(i);
62          for (int b = 0; b < pref; b++) block[b].increase(v);
63      }
64
65      ll prefix_sum(int i) {
66          int pref = get_prefix(i);
67          ll ans = 0;
68          for (int b = 0; b < pref; b++) ans += block[b].sum();
69          return ans;
70      }
71  };
```

Note that:

- The `Block` class represents a contiguous block of the array, where `Block(i, j)` represents the block $[i, j)$, i.e., it includes i but excludes j. I prefer it this way because in my opinion, it makes the implementation nicer. Of course, you can alternatively choose to have it represent $[i, j]$ with some slight changes in the implementation details.
- Much of the logic is found in the `get_prefix` function, which returns the number of blocks needed to cover `i` cells. It splits one of the blocks if needed, so it covers exactly `i` cells.
- The function `init_blocks` is called if we want to "start over".
- The function `try_rebuild` starts over if the list of blocks exceeds `s`. It computes the updated permutation by calling `push_values` for each block in order.

Here's the implementation of the inner class `Block`.

```
1   class SqrtDecomp {
2       vector<ll> a;
3       ll *totals;
4       int s;
5
6       class Block {
7           SqrtDecomp *src;
8           int i, j;
9           bool rev;
10          ll add;
11      public:
12          Block(SqrtDecomp *src, int i, int j, bool rev, ll add = 0)
13              : src(src), i(i), j(j), rev(rev), add(add) {}
14          ll sum() {
15              return src->total[j] - src->total[i] + add * (j - i);
16          }
17          void reverse() {
18              rev = !rev;
19          }
20          void increase(ll v) {
21              add += v;
22          }
```

```
        int size() {
            return j - i;
        }
        Block split(int k) {
            if (rev) {
                Block b(src, i, j - k, true, add);
                i = j - k;
                return b;
            } else {
                Block b(src, i + k, j, false, add);
                j = i + k;
                return b;
            }
        }
        void push_values(vector<ll>& target) {
            if (rev) {
                for (int k = j; k --> i;) {
                    target.push_back(src->a[k] + add);
                }
            } else {
                for (int k = i; k < j; k++) {
                    target.push_back(src->a[k] + add);
                }
            }
        }
    };

    // ...
};
```

In particular, note the differences in the behavior of `split` and `push_values` depending on whether the array is reversed or not.

Here's some sample usage.

```

int main() {
    int n, q;
    cin >> n >> q;
    vector<ll> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    int s = int(1 + sqrt(n));
    SqrtDecomp sd(a, s);
    for (int qq = 0; qq < q; qq++) {
        string typ;
        cin >> typ;
        if (typ == "reverse") {
            int i;
            cin >> i;
            sd.prefix_reverse(i);
        } else if (typ == "increase") {
            int i; ll v;
            cin >> i >> v;
            sd.prefix_increase(i, v);
        } else if (typ == "sum") {
            int i;
            cin >> i;
            cout << sd.prefix_sum(i) << '\n';
```

```
27          } else {
28              cerr << "unknown command: " << typ << '\n';
29              assert(false);
30          }
31      }
32  }
```

We can also handle point insertions and deletions easily; the running time then becomes $\mathcal{O}(n + q\sqrt{n+q})$. This is left as an exercise.

> **Exercise 4.3.** Show how to handle insertions and deletions as well.

4.3 Treap with lazy propagation

It is no surprise that there are also binary tree techniques that can solve this problem faster than sqrt decomposition techniques. However, it's somehow more complicated once you note that the prefix reversal operations make a lot of nontrivial changes to the tree, and these changes might be incompatible with the requirements of the self-balancing binary tree. For example, can an AVL tree handle prefix reversals, while still maintaining efficiency and AVL-tree-ness? Think about it. It's scary, isn't it? Thus, this shows, via intimidation, that the AVL tree cannot handle prefix reversals.

Fortunately, we can still use a treap! The key is in the fact that the priorities used in the treap are random, which makes any sort of rearrangement still random. Sure, the worst case is still a height of $\mathcal{O}(n)$, but it will be hard to trigger the worst case using a sequence of prefix reversals, especially without knowing the details of the random number generator.

So as we have seen before, we can already implement most of the operations using a treap with lazy propagation, except for prefix reversal. So how should we implement prefix reversals?

The key here is utilizing the *split* and *merge* operations of the treap to our advantage. To do so, let's say we want to reverse the first p elements. Then first, we *split* the treap into two trees, the first representing the interval $[0, p-1]$ and the second $[p, n-1]$. Then, we *reverse* the left tree. Finally, we *merge* the two trees again.

Unfortunately, reversing a whole tree is slow. But here, we simply use lazy propagation again. We keep track of a lazy *reverse* flag on each node, and only reverse a tree when we actually visit it. In other words, when we visit a node with the lazy *reverse* flag on, we swap its two subtrees and then *push* the reverse flag to its left and right subtrees. This makes the prefix reversal run in $\mathcal{O}(h)$ time where h is the height of the tree, and due to the randomized property of the treap, h is expected to be $\mathcal{O}(\log n)$!

Here's an implementation: `range_manip_treap.cpp`. Notes:

- Note that `split` and `merge` are the same as usual, only they've been sprinkled with `visit()` and `combine()` calls.
- The `visit()` call also performs lazy reversal in addition to the lazy range update. After doing so, all prefix operations become simple: we simply split the treap into sizes i and $n-i$, operate on the left subtree, and then merge them back again! All operations are done this way.
- From the perspective of the treap implementation, the leaf nodes (corresponding to single elements of the array) are *null* nodes, so they are assigned a priority of $-\infty$.

Here's a slightly different implementation: `range_manip_treap2.cpp`. Note the changes in the implementation of the `RangeTreap` class. This makes it even easier to reuse.

5 Multidimensional operations

We begin by discussing some higher-dimensional variants of data structures that we've already encountered.[18]

Let's consider a simple problem. You are given n 2-dimensional points

$$P = [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)],$$

and you need to preprocess them so that you can answer the following type of operations:

- **Range sum query.** Given a rectangle $[x_L, x_R] \times [y_L, y_R]$, find the number of points in P inside the rectangle.
- **Point insert.** Given a point, insert it into P. We assume that the point is not already in P.
- **Point delete.** Given a point, delete it from P. We assume that the point is already in P.

Actually, let's consider a generalized variant: we let the points have weights as well. Initially, point i has weight w_i, and we have the following operations:

- **Range sum query.** Given a rectangle, find the total weight of all the points in P inside the rectangle.
- **Point insert.** Given a point with a weight, insert it into P. We assume that the point is not already in P.
- **Point delete.** Given a point, delete it from P. We assume that the point is already in P.
- **Point increase update.** Given an existing point and a number d, increase the weight of that point by d.[19]

The previous problem can be seen as special case where each point has weight 1.

It should be noted that most techniques here are easily generalizable to higher dimensions; we will only discuss the 2D case for simplicity, and leave the higher-dimension generalizations to the reader.

As usual, I invite you to come up with solutions yourself first before proceeding. Obviously, the goal is to do better than the brute-force $\mathcal{O}(n)$ per query.

[18] You have already encountered one, namely the 2D Fenwick tree, which allows you to perform 2D range sum queries (or "rectangular" sum queries). It's also quite straightforward to extend to higher dimensions.

[19] We can also consider range increase updates, but that seems to be too hard to solve efficiently, so we will only consider point updates here.

5.1 Static solutions

Let's consider the static version of the problem for now, i.e., there are only rectangular sum queries.

It turns out that, similarly to how we can reduce the range sum $sum(i, j)$ into two prefix sums $prefix\_sum(j) - prefix\_sum(i-1)$, we can also reduce rectangular sum queries into four *quadrant queries*:

$$\begin{aligned} rectangle\_sum([x_L, x_R] \times [y_L, y_R]) = \quad & quadrant\_sum(x_R, y_R) \\ & - quadrant\_sum(x_L - 1, y_R) \\ & - quadrant\_sum(x_R, y_L - 1) \\ & + quadrant\_sum(x_L - 1, y_L - 1) \end{aligned}$$

Thus, we can simply analyze quadrant sums instead of the more general rectangle sums!

It's worth noting that this decomposition only works because addition can be inverted/-canceled using subtraction. If we replace addition with some operations where this isn't true (e.g. min, or certain types of multiplication), then you can't use the decomposition. Fortunately, many techniques described in this section can still be applied, some more easily than others, since many of them only require commutativity and associativity of the query operation.[20]

5.1.1 Static online solutions

What if, instead of a bunch of points in the plane, we consider them as cells in a 2D grid? Then each quadrant sum query (x, y) translates into a sum on this grid up to a certain row and column!

Thus, we can do a preprocessing step that computes all possible quadrant queries, which allows us to answer quadrant queries in $\mathcal{O}(1)$ with lookups.

We can do this preprocessing using dynamic programming. Let's say $Q_{x,y}$ is the sum of all the cells in the first x columns and first y rows. Then we have the following relationship, based on the quadrant decomposition:

$$w_{x,y} = Q_{x+1,y+1} - Q_{x+1,y} - Q_{x,y+1} + Q_{x,y}$$

where $w_{x,y}$ is the weight of the point in (x, y) if any; if the point doesn't exist, we assume $w_{x,y} = 0$. Thus,

$$Q_{x+1,y+1} = w_{x,y} + Q_{x+1,y} + Q_{x,y+1} - Q_{x,y}.$$

A dynamic programming procedure now allows us to compute all the needed $Q_{x,y}$. The base cases are $Q_{x,0} = Q_{0,y} = 0$.[21]

Now, what's the running time? Well, each query is answerable in $\mathcal{O}(1)$, but unfortunately, the grid itself might be large. Its size is $\mathcal{O}(x_{\text{span}} \cdot y_{\text{span}})$ where x_{span} is the difference between the largest and smallest x-coordinate, and y_{span} is defined similarly. Thus, this is not very useful if the coordinates are large. Still, it can be used in cases where the coordinates are not large, and in those cases it's usually desirable because it's simple to implement.

[20] Note that commutativity is important in multidimensional queries because otherwise, it's not clear what order we should combine the operands in.

[21] Here, Q has a more specific name; it's called a summed-area table.

We can improve this slightly by noticing that empty rows and columns are not important. Thus, we can perform a technique called *coordinate compression*; we only consider the x and y coordinates that contain a point. This reduces the size of the grid from $\mathcal{O}(x_{\text{span}} \cdot y_{\text{span}})$ to $\mathcal{O}(x_{\text{distinct}} \cdot y_{\text{distinct}})$ where x_{distinct} is the number of distinct x-coordinates in the input points and y_{distinct} is defined similarly. This is usually an improvement, although in the worst case, all/most coordinates are distinct, so $x_{\text{distinct}} = y_{\text{distinct}} = \Theta(n)$, which gives us a $\Theta(n^2)$ preprocessing step, which sounds slow. Still, it could be used when n is small, or the number of distinct coordinates is small.

5.1.2 Static offline solutions

We can find a faster solution if we can further process the queries offline.

Note that we have reduced our static queries into quadrant queries (x, y). Our goal is to answer all of these queries, but since we're doing offline, we can answer them in any order.

One promising thing to do is to *sort* the queries first, say by x, then we try to answer the queries in this order. So now, we can consider the x-coordinate as the *time* axis, and we assume that we're *sweeping* through time. This way, each quadrant query becomes a normal prefix query on the y-coordinate at the appropriate time, reducing the 2D query to a 1D query.

Obviously, as we sweep through time, we should add each point we encounter as well to our structure. This means that our 1D structure has to be dynamic. However, this is fine since we have plenty of dynamic 1D structures we can use, such as normal Fenwick trees or segment trees!

Now, what's the time complexity? There's the initial sort at the beginning, and since we have to sort all quadrant queries and points together, this takes $\mathcal{O}((n+q)\log(n+q))$ time. The sweeping step uses a segment tree/Fenwick tree and thus takes $\mathcal{O}((n+q)\log n)$, hence the overall algorithm runs in $\mathcal{O}((n+q)\log(n+q))$, which is quite fast!

Exercise 5.1. Can you do it in $\mathcal{O}((n+q)\log n)$? *Hint:* only a few changes need to be done.

To recap, by considering one of the axes as the time axis/sweeping axis, we can usually convert static offline 2D problems to dynamic 1D problems! More generally, we can usually convert static offline k-dimensional range queries into dynamic $(k-1)$-dimensional problems by considering one of the coordinates as the time axis.[22]

5.2 2D Fenwick trees

The coordinate compression + dynamic programming solution has another downside in that it doesn't allow for point updates. Luckily, with a small change, we can accommodate point updates!

We will still consider the points as belonging to a 2D grid and with coordinates compressed, so we are now looking at a $x_{\text{distinct}} \times y_{\text{distinct}}$ grid. However, this time, we will not compute $Q_{x,y}$ since a single update in this grid will require updating too many elements of the table Q. Instead, we would like to build a more dynamic structure that allows for updates. Well, for this, we can just use a *2D Fenwick tree*![23]

[22] You are usually free to use which coordinate you want to use as the time axis, although there are some instances where you have no choice. Just be careful!

[23] The 2D Fenwick tree has been discussed briefly in a previous module. I encourage you to understand how it works.

The preprocessing time then becomes $\mathcal{O}(x_{\text{distinct}} \cdot y_{\text{distinct}} + n \log^2 n)$ (which is $\mathcal{O}(n^2)$ in the worst case), and point updates and range queries then run in $\mathcal{O}(\log^2 n)$ time.

Note that point deletes can also be done. Unfortunately, we can't support range updates since we're using a 2D Fenwick tree, and we can't also support point inserts since the coordinates of the new point might not be among the compressed coordinates we have at the beginning. (You can fix the latter in case you know what the future coordinates will be; you simply include them in your compressed coordinates.)

This solution generalizes well to higher dimensions; for a k dimensions, the preprocessing takes $\mathcal{O}(n^k)$ time and each query is answered $\mathcal{O}(\log^k n)$ time.

5.3 2D sparse tables

The problem with the techniques we've encountered so far is that we're still decomposing rectangular queries into quadrant queries, and we can only do that if the operation has an inverse. Luckily, addition can be "canceled" by subtraction. However, what if our operation cannot be inverted? As mentioned before, there are some useful operations like that, and we would still like to be able to handle them. In this case, we can't use the 2D Fenwick tree solution anymore.

Fortunately, just like in the 1D case, there are other techniques that don't require invertibility for the 2D case. One such technique is using *2D sparse tables*!

The traditional 1D sparse table precomputes an $\mathcal{O}(n \log n)$ table which contains the aggregates of all intervals with power-of-two sizes. Using such a table, we can answer a range query by finding two *overlapping* intervals with power-of-two sizes and combining them. This works for the range minimum query since the min operator is *idempotent*, i.e., including the same element multiple times doesn't affect the answer.

Now, what if the operation is not idempotent anymore? In that case, an exercise in a previous module shows that it can still be done with $\mathcal{O}(n \log n)$ preprocessing and $\mathcal{O}(1)$ query.

Luckily, both types of sparse tables can be extended to the 2D case! For example, in the power-of-two scheme, we can extend it to 2D by computing the aggregates of all rectangles whose area is a power of two! Since each dimension must be a power of two, then it follows that there are only $\lg n \times \lg n$ distinct dimensions, and so the table will be of size $\mathcal{O}(x_{\text{distinct}} \cdot y_{\text{distinct}} \log^2 n) = \mathcal{O}(n^2 \log^2 n)$. Each query can then be answered in $\mathcal{O}(1)$.

The second type of sparse table (the one that can handle non-idempotent operations) can also be extended to 2D; we will leave the details to the reader.

One downside is that 2D sparse tables don't support updates, just like the 1D counterpart.

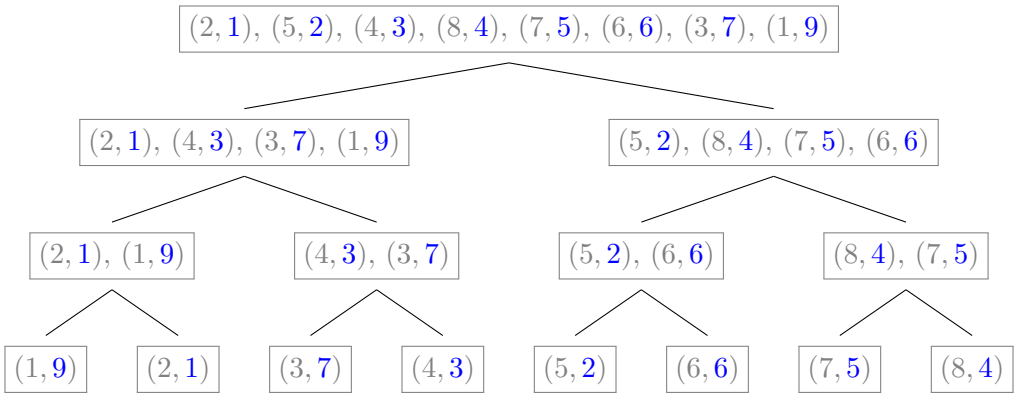
Also, this technique (and most of the previous ones) have a major downside; they require at least quadratic time to preprocess. This follows from reducing 2D space to a 2D grid, which unfortunately creates a lot of space, most of which are empty. What follows are techniques that don't do the reduction to a 2D grid and therefore are much faster.

5.4 Merge sort tree

There's a nice technique that allows us to answer static queries *online* that's much faster than the grid solution.

The idea is to first sort the points on the x-coordinate. Then, we attempt to sort the points again, this time on the y-coordinate, using merge sort. So it could look like this: (Note that I have drawn the diagram with the final array on top, and with the initial array, sorted by

x-coordinate, at the bottom)



However, instead of just taking the final, sorted array, we *keep* all intermediate sorted arrays as well! They naturally form a binary tree structure which actually allows us to answer range queries quickly!

This data structure is called the *merge sort tree*. It can be built in $\mathcal{O}(n \log n)$ time since this is basically just merge sort.

Let's see how we can answer a range sum query, $[x_L, x_R] \times [y_L, y_R]$, using this structure. First, we restrict ourselves to the nodes whose x range is completely contained in $[x_L, x_R]$. Obviously, there are many of those, but we can reduce this by noticing that the merge sort tree is basically a segment tree on the sorted x-coordinates, so just like in a segment tree, we know that we can find $\mathcal{O}(\log n)$ nodes that are disjoint and completely cover the interval $[x_L, x_R]$. Next, we need to count how many points in each node there are whose y coordinate falls into the interval $[y_L, y_R]$. But this is easy to compute for each node since the points are sorted by the y coordinate! So, we can simply use binary search to find the answer in $\mathcal{O}(\log n)$ time per node. Therefore, answering a query takes $\mathcal{O}(\log^2 n)$ time with a merge sort tree.

The following is an implementation. A couple of remarks:

- It's required to sort the input points by x coordinate before passing them to the constructor.
- After constructing the tree, the prefix sums of the weights are prepared so that range sums can be computed easily.

```
struct Pt {
    ll x, y, w;
    bool operator<(const Pt& p) const { // so that points can be sorted
        return x != p.x ? x < p.x : y < p.y;
    }
};

class MergeSortTree {
    ll xi, xj;
    vector<ll> ys, ws;
    MergeSortTree *l, *r;
    MergeSortTree(vector<Pt>& pts, int i, int j): xi(pts[i].x), xj(pts[j].x) {
        ws.push_back(0LL);
        if (i == j) {
            ys.push_back(pts[i].y);
            ws.push_back(pts[i].w);
```

```
            l = r = NULL;
        } else {
            int k = i + j >> 1;
            l = new MergeSortTree(pts, i, k);
            r = new MergeSortTree(pts, k + 1, j);
            // merge the two sorted arrays
            for (int L = 0, R = 0; L < l->ys.size() || R < r->ys.size();) {
                if (L < l->ys.size() && (R >= r->ys.size() ||
                        l->ys[L] < r->ys[R])) {
                    ys.push_back(l->ys[L++]);
                    ws.push_back(l->ws[L]);
                } else {
                    ys.push_back(r->ys[R++]);
                    ws.push_back(r->ws[R]);
                }
            }
        }
    }

    void prepare() { // compute prefix sums
        for (int i = 0; i < ys.size(); i++) ws[i + 1] += ws[i];
        if (l != NULL) { // prepare subtrees as well
            l->prepare();
            r->prepare();
        }
    }

    ll prefix_sum(ll y) {
        int L = -1, R = ys.size();
        while (R - L > 1) { // binary search
            int M = L + R >> 1;
            (ys[M] <= y ? L : R) = M;
        }
        return ws[R];
    }

    ll yrange_sum(ll yI, ll yJ) { // subtract two prefix sums
        return prefix_sum(yJ) - prefix_sum(yI - 1);
    }

public:
    MergeSortTree(vector<Pt>& pts): MergeSortTree(pts, 0, pts.size() - 1) {
        prepare();
    }
    ~MergeSortTree() {
        if (l != NULL) {
            delete l;
            delete r;
        }
    }

    ll sum(ll xI, ll xJ, ll yI, ll yJ) {
        if (xI <= xi && xj <= xJ) { // completely contained
            return yrange_sum(yI, yJ);
        } else if (xJ < xi || xj < xI) { // disjoint
            return 0;
        } else { // partial overlap
            return l->sum(xI, xJ, yI, yJ) + r->sum(xI, xJ, yI, yJ);
        }
    }
};

```

```cpp
int main() {
    int n, q;
    cin >> n >> q;
    vector<Pt> pts(n);
    for (int i = 0; i < n; i++) {
        cin >> pts[i].x >> pts[i].y >> pts[i].w;
    }
    sort(pts.begin(), pts.end()); // important! sort by x first.
    MergeSortTree *tree = new MergeSortTree(pts);
    while (q--) {
        string typ;
        cin >> typ;
        if (typ == "sum") {
            ll xl, xr, yl, yr;
            cin >> xl >> xr >> yl >> yr;
            cout << tree->sum(xl, xr, yl, yr) << '\n';
        } else {
            cerr << "unknown command: " << typ << '\n';
            assert(false);
        }
    }
    delete tree;
}
```

As another aside, there is a way to improve the query time to $\mathcal{O}(\log n)$ by utilizing a technique called *fractional cascading*. The idea of fractional cascading is to use the results of the binary search in one node to immediately answer it for succeeding nodes without doing more binary searches. The idea is not that hard to understand, but implementing it is a pain, and is likely not going to cause significant improvements anyway due to the large constant (hidden in the big-O) involved, so we will not discuss it here.[24]

You may learn more about merge sort trees by watching the YouTube lecture about it by IOI gold medallist Sergey Kulik.[25]

5.5 Range trees

One downside of the merge sort tree is that it's not dynamic, i.e., it doesn't support updates. However, we can easily turn it into a dynamic structure! The only thing we need to change here is to replace the static array in each node with a dynamic one, say, a segment tree. This automatically gives us efficient point updates!

To see why this works, we take a look at the dynamic version of the merge sort tree from a different perspective: instead of looking at it as a concretization of merge sort, we look at it as *a higher-order generalization of the segment tree.*

We can consider a 2D segment tree as a *segment tree of segment trees*, i.e., a normal segment tree but each node has an associated segment tree in it.

The outer segment tree is built upon the x coordinate. Each node represents a particular range of coordinates $[x_l, x_r]$. In each node, an inner segment tree exists, and this inner segment tree is built upon the y coordinate, so each node in it represents points in some rectangle $[x_l, x_r] \times [y_l, y_r]$. In this representation, each point (x, y) is present in $\mathcal{O}(\log n)$ nodes of the tree.

The operations can now be done as usual, by simply doing it for the x-coordinate and then delegating to the inner segment tree for the y-coordinate.

Here's an implementation: `2drange_range_tree.cpp`. Note how similar the outer and inner

[24] Reference: https://en.wikipedia.org/wiki/Fractional_cascading
[25] https://www.youtube.com/watch?v=6x4-lfhHd_Y

segment tree classes are!

Officially, this is called a *range tree*.[26] Note that this data structure doesn't support point insert and delete, but we can easily support those as well by using the techniques mentioned in section 2, e.g., by using balancing binary trees. Unfortunately, there doesn't seem to be a simple way for it to support range updates since the idea of lazy propagation doesn't extend well to higher dimensions.

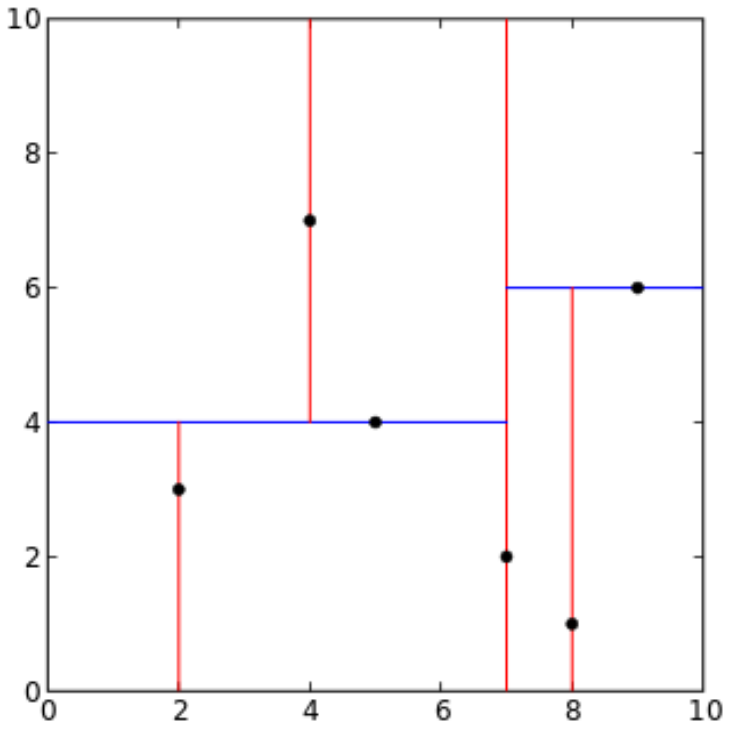
The preprocessing still takes $\mathcal{O}(n \log n)$ time and space, and the query time is still $\mathcal{O}(\log^2 n)$.

Higher-dimensional analogs also exist and are straightforward to construct. However, the time complexity also depends on the dimension of the structure; a k-dimensional segment tree requires $\mathcal{O}(n \log^{k-1} n)$ preprocessing time (and space) and $\mathcal{O}(\log^k n)$ query time.[27]

5.6 k-d tree*

We can also answer range queries by using a different data structure called a k-d tree.

To explain this data structure, let's look at the 2-dimensional variant, the 2-d tree. This data structure is formed by recursively partitioning the space into two parts containing roughly equal points until each partition contains only one point. There are many ways to do this, but the official way is to alternate splitting the space horizontally and vertically. Thus, the end result could look like this (taken from Wikipedia):



The resulting structure is called the 2-d tree. Naturally, this structure is represented as a binary tree. For the general k-d tree, we simply cycle splitting through all k dimensions.

[26] https://en.wikipedia.org/wiki/Range_tree . Note that some people also call this a *merge sort tree*, at least the 2D version, since there's very little difference.

[27] If you only need a *static* tree, then this can slightly be improved to $\mathcal{O}(\log^{k-1} n)$ using fractional cascading which was mentioned before, although again, this is mostly a theoretical improvement.

The range query on a k-d tree can be implemented straightforwardly, like you would with a segment tree, and it can be shown that the final complexity is $\mathcal{O}(n^{1-1/k})$ for $k > 1$. This is clearly worse than the (poly-)logarithmic complexities we get from using range trees, e.g., for $k = 2$, we get a runtime of $\mathcal{O}(\sqrt{n})$, for $k = 3$, we get $\mathcal{O}(n^{2/3})$, etc.. However, this structure can support point manipulations and can also support range queries with other types of shapes, e.g., 2D triangles, 3D tetrahedrons or any convex shape whose edges are not necessarily axis-aligned. These shapes don't have to be oriented in any particular way; the k-d tree can be queried with any such shape!

Additionally, one can also use a k-d tree to answer other kinds of geometric queries, e.g., "nearest neighbor search" (find the nearest point to a given point).

Such flexibility is rarely required, but it's nice to know that it exists when you need it.

For further details on this data structure, please consult the Wikipedia article about k-d trees.[28]

[28] https://en.wikipedia.org/wiki/K-d_tree

6 More data structures

6.1 Sliding range minimum query

Let's consider the following variant of the static range minimum query problem.[29] Suppose that for every two consecutive range queries (i_1, j_1) and (i_2, j_2), we have $i_1 \leq i_2$ and $j_1 \leq j_2$. In other words, if $(i_1, j_1), \ldots, (i_q, j_q)$ are all the queries, then $i_1 \leq i_2 \leq \ldots \leq i_q$ and $j_1 \leq j_2 \leq \ldots \leq j_q$. This variant is called the **sliding range minimum query** because you can imagine a *window* sliding from- left to right and going through the queries in order.

Clearly, the sliding RMQ is a special case of the normal static RMQ and so we can use the trusted techniques like using sparse tables ($\mathcal{O}(n \log n + q)$), segment trees ($\mathcal{O}(n + q \log n)$), converting to LCA ($\mathcal{O}((n + q) \log n)$), etc. These are already pretty fast, but there are cases where that log factor hurts a bit, so you'll sometimes need to remove it.

Fortunately, there's a solution to the sliding RMQ problem that runs in $\mathcal{O}(n+q)$ and that's completely online and has no preprocessing! Furthermore, the procedure can be done on-the-fly, i.e., we can use it even if we don't know what the upcoming elements or queries are. We will describe that solution here.

First, we convert the sliding RMQ to an equivalent *data structure* problem. Suppose we have an array A, initially empty, and we want to support three kinds of operations:

- **Insert on right**. Given an element, insert it to the right of array A.
- **Delete on left**. Delete the leftmost element of array A.
- **Get minimum**. Return the current minimum element of array A.

Note that this can easily be solved using a *binary search tree*. The idea is to also add/remove the elements to a balanced BST, and to get the minimum, we simply have to get the leftmost node of the BST. Array A itself can be represented as a *queue* due to the first-in-first-out (FIFO) nature of the data structure.

Note that this runs in $\mathcal{O}(q \log q)$ time[30] and satisfies our requirement of no preprocessing and not requiring that further queries are known. But amazingly, this can be improved to $\mathcal{O}(q)$ time![31]

To do that, we must take advantage of the FIFO nature of the data structure. The most important observation is the following:

Lemma 6.1. Suppose x is some element of the array. If a new element y is inserted that is strictly smaller than x, then x can never be the minimum again.

Exercise 6.2. Prove Lemma 6.1.

Please make sure you understand this since it's very important.

With this observation, we now find out that it's now perfectly safe to remove x from our structure, since we will not need it again! Our goal, then, is to only keep the elements that could possibly be the minimum in the future, and remove those elements which can never be the minimum anymore.

[29] Most of the techniques here will also apply to the range *maximum* query problem, with trivial modifications.
[30] $\mathcal{O}((n + q) \log n)$ in the original formulation of the problem
[31] $\mathcal{O}(n + q)$ in the original formulation.

Let's consider each update:

- Whenever we insert a new element on the right, we can remove all the elements strictly greater than it.
- Whenever we want to delete the leftmost element, say x, we try to remove x from our structure *if it hasn't already been removed.*

  To do this quickly, we use the following important observation: *x hasn't been removed if and only if x is the current minimum.* Why? Because:

  - If x hasn't been removed, that means no element smaller than it has been inserted after it. Furthermore, since x is the leftmost element, all other remaining elements in the structure has been inserted *after* x, which means x is the current minimum.
  - Conversely, if x has been removed, then that could have only happened if there was a y smaller than it that has been inserted after it. But this means that x is not the current minimum.

To solve the sliding RMQ problem, then, we simply keep track of the elements that could still possibly be minimums in the future in a sorted list, say M, and throw away the rest. Thus,

- Whenever we insert a new element to the right, all strictly-larger elements will be found at the right of M. They can be popped one by one until M becomes empty or the rightmost element is not strictly larger anymore.
- The minimum is always at the leftmost location of M. This allows us to get the minimum quickly. And also, for deletions, this allows us to determine where the leftmost element is in the array since if it exists, it must be the minimum.

By using a *deque* for M, we can implement each operation in amortized $\mathcal{O}(1)$ time, or equivalently, solve the whole problem in $\mathcal{O}(q)$ time![32]

As a side note, it turns out that the full static RMQ problem itself can be solved in $\mathcal{O}(n+q)$ by using an $\mathcal{O}(n)$ preprocessing and $\mathcal{O}(1)$ per query. However, that algorithm is quite complicated and the constant factor is high, so no one really uses it in contests, and furthermore, it can't be used in cases where we don't know all the elements beforehand and we're forced to answer the queries on the fly. So the sliding RMQ still has its uses.[33]

6.2 Merging heaps

By now, you should have already learned how to implement a **priority queue**, an abstract data type that contains a bunch of objects with *priorities* and has the following operations:[34]

- **Push**. Given an element v and a priority p, insert it into the structure.
- **Get**. Return the element with the maximum priority in the structure.
- **Pop**. Remove the element with the maximum priority in the structure.

[32] $\mathcal{O}(n+q)$ in the original formulation.
[33] More reading here: https://wcipeg.com/wiki/Sliding_range_minimum_query
[34] This is called a *max priority queue*; other applications require taking the minimum instead, but a max priority queue can be converted into a minimum priority queue with trivial changes, so we will only consider the max variant for simplicity.

Some applications also require the following operation:

- **Increase key**. Given an element in the data structure and an integer $d \geq 0$, increase its priority by d.

A priority queue is most often implemented with a *heap*, which is a tree satisfying the *heap property*: each node is greater than or equal to its children nodes.

However, suppose we now want to implement a new operation, called **merge**, which takes in two such structures and merges them into one? In this operation, it is assumed that the original heaps may be destroyed in the process, i.e., we don't have to keep a copy of the two input heaps.

6.2.1 Binary heaps

Let's try binary heaps. The simplest way is to simply take the elements of both heaps and form a new heap with them. Unfortunately, this is slow; it takes $\mathcal{O}(n \log n)$ time[35] in the worst case.

Another technique would be to take the elements of one of the heaps and push each of them to the other. Remember that we don't have to keep the input heaps intact so this is allowed. Also, this seems slightly smarter since we're only looping across one of the heaps. However, this still takes $\mathcal{O}(n \log n)$ time in the worst case.

Fortunately, we can improve this significantly with a simple trick: We always take the elements of the *smaller* heap and push each of them to the larger one. It should be clear why this is a little better, since we're iterating through fewer elements. But how much better, exactly?

Well, suppose we start with 0 heaps and there are n operations. For simplicity, let's first assume that no heap will ever be popped. Consider a particular element from the first moment it is pushed to any heap. Along the way, it may have been pushed to different heaps during some merge operations. But how many times will it have been pushed to a different heap? Well, a re-push can only happen if it moves to a heap that's larger than the current heap it's on. But this means that whenever it is repushed to a different heap, the size of the current heap it's on at least doubles! Since the size of the heap is at most n, this means that each element will only be pushed $\mathcal{O}(\log n)$ times, and since each heap push takes $\mathcal{O}(\log n)$ time, this means that n operations will take $\mathcal{O}(n \log^2 n)$ time, or $\mathcal{O}(\log^2 n)$ amortized time per operation. Furthermore, since push and pop already have $\mathcal{O}(\log n)$ worst-case time, the $\mathcal{O}(\log^2 n)$ amortized time only really applies to the merge operation.

Now, what if there are pops? In this case, the statement that a particular element will be pushed $\mathcal{O}(\log n)$ times is not true anymore, since the size of the current heap it's on may decrease. But if you really think about it, each pop can only help *improve* the running time of any future operation since it will never increase the amount of work needed:

- Pushing to and popping from a heap that has been popped always takes less work since it is smaller.
- Merging heaps that have been popped is always easier since the input heaps are smaller. It might happen that heap a is smaller than heap b at without pops but after some pops, b becomes smaller, but the end result is the same either way, and in fact results in a smaller overall heap, so it only improves this and any future operation.

[35] or $\mathcal{O}(n)$ if you use a smarter *heapify* operation.

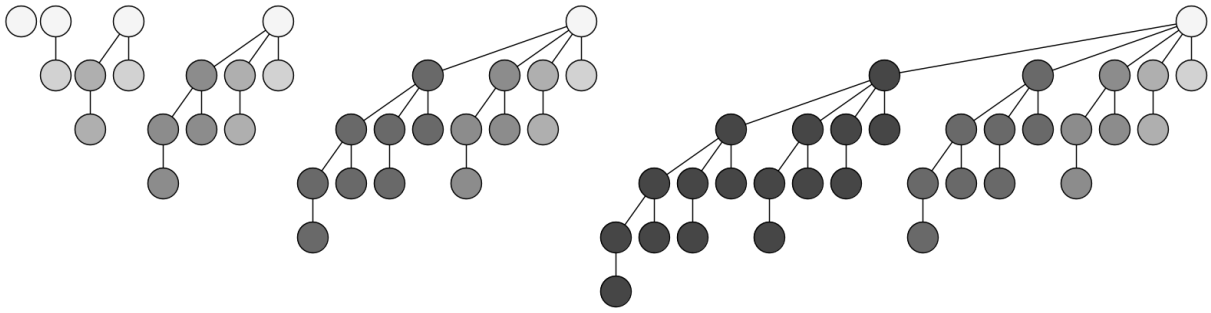
Thus, we can still say that, even in the presence of pops, each merge operation takes $\mathcal{O}(\log^2 n)$ amortized time!

6.2.2 Binomial heaps*

The $\mathcal{O}(\log^2 n)$ amortized time for merge can actually be improved to $\mathcal{O}(\log n)$ *worst-case* time by using a different kind of heap, called the **binomial heap**. This is a really nice data structure!

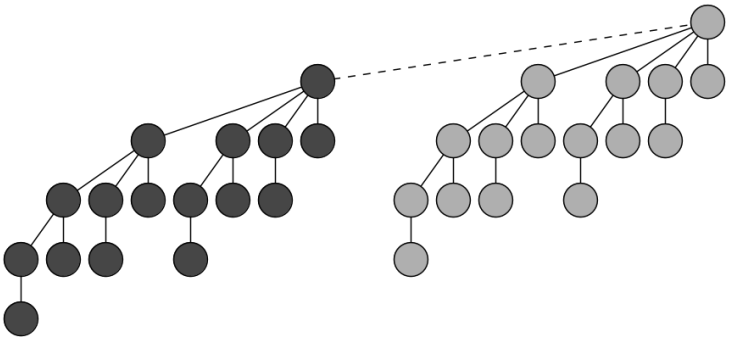
First, we define a binomial tree. A **binomial tree of order** k is a tree that contains exactly k children, where for $0 \leq i < k$, the ith child from the right is a binomial tree of order i.

The following are binomial trees of different orders. The leftmost one has order 0, while the rightmost one has order 5.



Note that a binomial tree is *not* a binary tree. In what follows, we simply use the word "tree" to denote a binomial tree for brevity.

It should be noted that two order-k trees can be combined into an order-$(k+1)$ tree like this:



In other words, you just attach one of the trees as another child of the other. This merging scheme also shows that a binary tree of order k has 2^k nodes.

A **binomial heap** is a collection of binomial trees of *distinct* orders and such that the heap property is satisfied in each tree.

It should be easy to see that a binomial heap with n elements contains at most $\lg(n+1)$ trees. In fact, it satisfies a stronger property. Try thinking about how you can even represent a binomial heap with n elements. Remember that a tree of order k has 2^k nodes, and that the orders have to be distinct and the sizes must add up to n.

For a binomial heap with n elements, there's a unique set of orders $\{k_1, k_2, \ldots, k_t\}$ whose sizes add up to n, and this is precisely the binary representation of n since $n = 2^{k_1} + 2^{k_2} + \ldots + 2^{k_t}$. Thus, the number of trees in a binomial heap with n elements is precisely the number of 1s in the binary representation of n. We're programmers, so we usually like binary, and this makes the binomial heap extra nice!

Let's now implement the operations.

Get maximum. First, getting the maximum is easy; we simply take the largest among all the roots. Since there are $\leq \lg(n+1)$ trees, this runs in $\mathcal{O}(\log n)$ time.

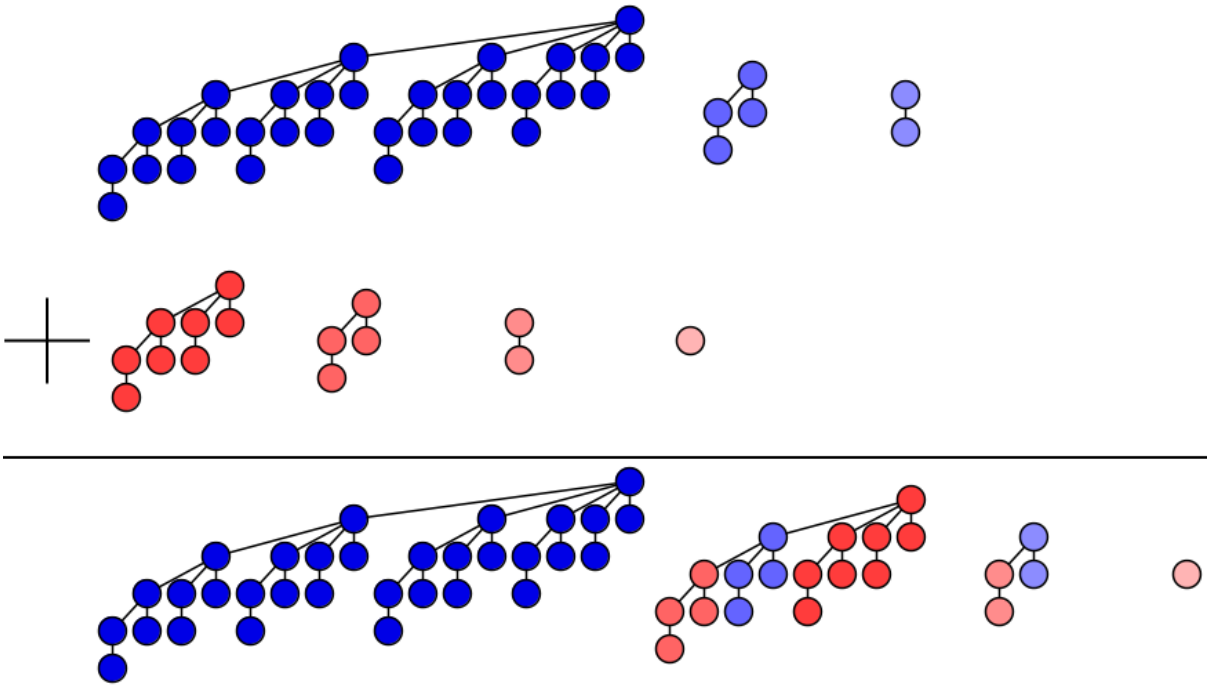
Increase key. Increase key is also implemented by sifting the updated element upwards until the tree containing it satisfies the heap property, just like in a regular binary heap.

Merge. Thus, we're left with the operations push, pop and merge. Here, merge is the most crucial operation, but it turns out to be pretty simple to implement! For this, we remember that we can combine two order-k trees into one order-$(k+1)$ tree. However, note that we also have to ensure that the heap property is satisfied. But this is simple, just make the larger root the overall root!

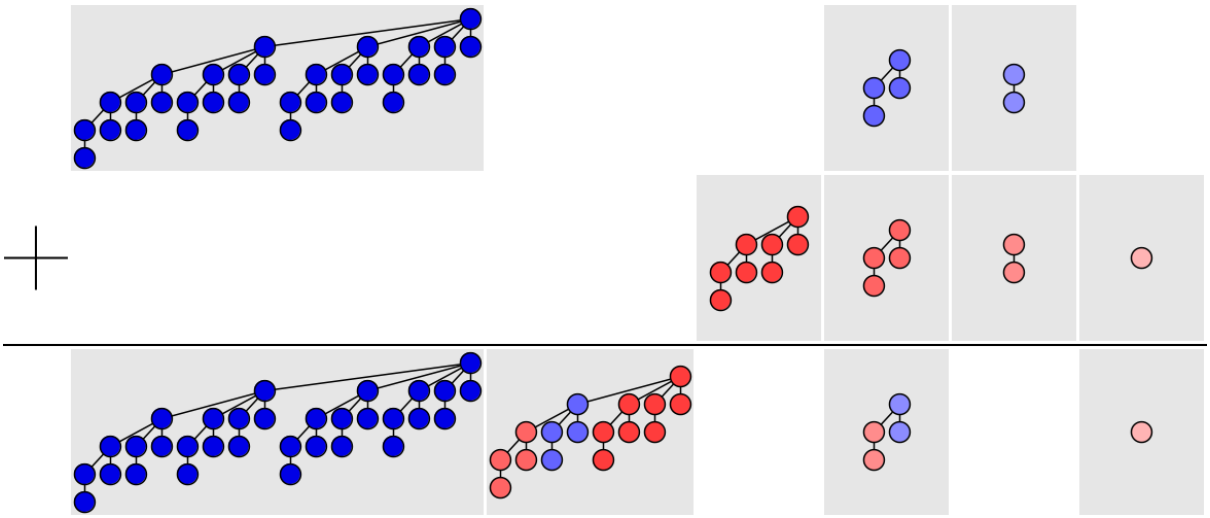
Using this sub-operation, we can now implement merge quickly. Thus, given two collections of binomial trees, we combine trees with the same order starting with the lower orders. We repeat this until there are no more trees with the same order.

The merge operation turns out to be similar to adding two binary integers. Specifically, given two heaps with sizes n_1 and n_2, their merge operation kinda looks like adding the binary representations of n_1 and n_2. Note that whenever we merge two trees, the newly merged tree effectively becomes the "carry".

Here's an illustration of merging two binomial heaps of sizes 38 and 15:



If we align trees with the same order in the same column, like this:



then we can notice the similarity between this and adding 38 and 15 in binary:

$$\begin{array}{cccccccc} & 1 & 0 & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline & 1 & 1 & 0 & 1 & 0 & 1 \end{array}$$

Since combining two same-order trees takes $\mathcal{O}(1)$, this clearly runs in $\mathcal{O}(\log n)$ time.

Push. Now, we need to implement push and pop. It turns out that both are easy after knowing how to merge. To push a new element x, we just form a heap containing only x and *merge* it with the current heap. This clearly runs in $\mathcal{O}(\log n)$ time.

Pop. To pop the maximum value, we first find the tree which contains it. Then, we remove this tree from the heap. We then also remove the root from this tree, leaving us with a bunch of smaller-order trees. These trees form a binomial heap on their own, which can be merged with the rest of the heap!

This clearly runs in $\mathcal{O}(\log n)$ time as well.

6.3 Splay trees*

Splay trees are another kind of self-balancing binary tree. They are usually used as binary search trees, but they can also be turned into segment trees.

The main difference is that a splay tree rebalances during queries as well, instead of just on updates! This is done with the splay operation. The operation *splay*(x) rebalances the tree in a particular way such that the node x becomes the root in the end.

The splay tree's height is $\mathcal{O}(n)$ in the worst case, so a single query could be $\mathcal{O}(n)$ in the worst case, but after every query, the retrieved node is splayed, so it becomes the root. The idea is for subsequent queries on the same node to be fast. Similarly, newly inserted nodes are splayed.

The idea is that even though some particular queries are slow, the splay operation guarantees that such operations can't be repeatedly done to trigger slowness because they become the root after the first operation, and so overall, the average runtime may be faster. In fact, if you do the analysis properly, each operation runs in $\mathcal{O}(\log n)$ amortized time!

To perform the splay, we perform a series of rotations that makes the current node the root. Similar to an AVL tree, there are several kinds of rotations. In splay trees, they're called

zig-zag and **zig-zig**. We won't describe them here in detail, instead, we refer the reader to the Wikipedia article on splay trees.[36]

While splay trees are usually used for BSTs, the unique thing about it is the splay operation, which allows us to turn any node into a root. This operation is quite useful since it allows us to perform range/prefix operations described in section 4. The main idea is that if we want to do an operation on the prefix of size p, then we splay on the $(p+1)$th node so that the left child of the root becomes of size p. Then we can perform the prefix operation easily (possibly with lazy propagation)!

We can also implement *split* and *merge*, just like in treaps! For example,

- To split a splay tree into two sizes p and $n-p$, we simply splay on the $(p+1)$th node. The left subtree will then be of size p, so we can just detach it.
- To merge two splay trees where the keys of the left tree are less than the keys of the right tree, we simply splay the leftmost node of the right splay tree so that the root will not have a left child, and then make the left splay tree the new left child.

The analysis and proof of $\mathcal{O}(\log n)$ amortized running time per operation can be found in the Wikipedia page.

6.4 Scapegoat trees*

Scapegoat trees are yet another kind of self-balancing binary tree. They are usually used as binary search trees, but like many kinds of binary trees, they can also be augmented with aggregate data to become segment trees on their own.

However, what's interesting is the unique way it tries to maintain the balancing of the tree. The general idea with scapegoat trees is to do the insertions normally and not do any rebalancing at all, at least up to a point where we detect that one of the subtrees is "sufficiently imbalanced". In that case, we simply find a sufficiently imbalanced subtree and rebuild it to be perfectly balanced!

The node that will be rebalanced is called the **scapegoat** because we're blaming that node for the imbalance of the whole tree.

To explain scapegoat trees, we first need some definitions. We define a node x to be **α-weight-balanced** if both the following are satisfied:

$$size(left(x)) \leq \alpha \cdot size(x)$$
$$size(right(x)) \leq \alpha \cdot size(x)$$

Also, we say that a binary tree is α-weight-balanced if all the nodes in it are α-weight-balanced.

Thus, $(1/2)$-weight-balanced trees are just almost-complete binary trees, while 1-weight-balanced trees include degenerate trees, i.e., those that aren't even balanced. In what follows, we will assume that $1/2 \leq \alpha < 1$.

6.4.1 Pseudo-scapegoat tree*

Note that an α-weight-balanced binary tree is great since we have a strong guarantee on the height of the tree:

[36] https://en.wikipedia.org/wiki/Splay_tree.

Lemma 6.3. An α-weight-balanced tree T has height $\leq \log_{1/\alpha} size(T)$.

The proof is left as an exercise.

Exercise 6.4. Prove Lemma 6.3. Also, prove that if x is a node whose depth is $> \log_{1/\alpha} size(T)$, then at least one ancestor of x is not α-weight-balanced.

Now, if we fix α at the beginning when we construct our tree, then $\log_{1/\alpha} size(T)$ will be $\mathcal{O}(\log n)$ since the big O notation ignores constants (although, obviously, the closer α is to 1, the larger this constant will be).

Thus, ideally, we would like to maintain that our tree is always α-weight-balanced because of this lemma.

And it turns out that we can do that! In fact, it's quite simple to do. We can store the size of the subtree rooted at every node. Then, for every insert, we first insert it normally. Now, if the tree violates α-weight-balancing after insertion, then we climb up the tree and find the topmost ancestor that is not α-weight-balanced. (We are guaranteed that such a node will exist.) Then we simply rebuild that tree to be $(1/2)$-weight-balanced!

For deletions, we do the same thing; we delete a node, and find the topmost ancestor that is not α-weight-balanced, if any, and then rebuild that subtree.

The node that gets rebuilt this way is called the **scapegoat** for obvious reasons.

Now, what's the complexity of the operations? Since the height of the tree is $\mathcal{O}(\log n)$ (although dependent on α), searching runs in $\mathcal{O}(\log n)$ worst-case time. However, insertions and deletions could run in $\mathcal{O}(n)$ worst-case time since we could potentially rebuild all (or a huge chunk) of the tree. But, it turns out that both of them actually run in $\mathcal{O}(\log n)$ *amortized* time if we assume that $\alpha > 1/2$; the proof is given in Appendix A.

One cool thing about this is that this is easy to implement; it's much simpler than treaps or AVL trees!

Now, what α should we choose? It depends on the number of queries and updates; if there are much more updates than queries, then it makes sense to allow the tree to be slightly more unbalanced (i.e., higher α), but if there are more queries, then we would want the tree to be more balanced (i.e., lower α). Obviously, the only allowed choices for α are those where $1/2 < \alpha < 1$.

6.4.2 The real scapegoat tree*

The "scapegoat tree" presented in the previous part is not actually the scapegoat tree; the real scapegoat tree has a weaker requirement than being α-weight-balanced. In fact, instead of maintaining the tree to be α-weight-balanced, it tries to maintain itself to be *loosely height-balanced*.

Specifically, let's define a tree to be **α-height-balanced** if its height is $\leq \log_{1/\alpha} size(T)$. Now, remember that any α-weight-balanced tree is also α-height-balanced, from Lemma 6.3. However, the real scapegoat tree actually only tries to be α-height-balanced instead of α-weight-balanced!

So now, let's see how we can do insertion with this new requirement. During insertion, we do the following steps:

- First, we insert the new node normally.
- Then, if the height of the newly inserted node exceeds $\log_{1/\alpha} size(T)$, then we try to find

a scapegoat among the node's ancestors. (Such a scapegoat exists among the ancestors; this follows from Lemma 6.3.)

Unfortunately, using this scheme, the height of the scapegoat tree isn't always maintained to be $\leq \log_{1/\alpha} size(T)$. However, it can be shown that the height will always be $\leq \log_{1/\alpha} size(T)+1$, which is of course good enough for our purposes. We call a tree satisfying the latter condition **loosely α-height-balanced**.[37]

Another unique thing about the real scapegoat tree is that we don't even need to store the sizes of the subtrees! We only need to store the size of the whole tree itself, not of the individual nodes.

To see why, note that if we want to check whether the newly-inserted node violates the α-height-balancing property, then we only need to know the size of the whole tree (which we already have) and the depth of the inserted node (which can be computed on the fly as we go down the tree). We will only really need the sizes of individual nodes when we have to find a scapegoat, but in those cases, we can simply compute the size of each subtree we need by actually iterating through the subtree and counting the nodes.

Furthermore, since we're climbing up, we don't have to recompute the size of the subtree we just came from — we only need to compute the size of its sibling — hence, each node in the scapegoat subtree will only be visited once.

Now, you might be thinking, "isn't that slow, since you're iterating through the whole subtree?" Well, that might be true, but remember that we're looking for the scapegoat, and we're going to completely rebuild that scapegoat afterwards anyway, which means that the cost of computing the size of the scapegoat is (asymptotically) no more than the cost of actually rebuilding it. Hence, we're not really incurring any additional costs! Thus, the cost of finding the scapegoat s and rebuilding it is still $\Theta(size(s))$, which, similarly to the pseudo-scapegoat tree, can be amortized away, i.e., we can show that insertion still takes $\mathcal{O}(\log n)$ amortized time.

Now, what about deletions? In that case, we do something slightly different than for insertion, but this turns out to be easier. What we do is we keep track of another number, called *maxsize*, which denotes the largest size that the tree has been since the last time the whole tree has been rebuilt. This number can easily be updated after every insertion and deletion. Then, on deletion, if ever the tree becomes *too small*, more specifically "$size(T) < \alpha \cdot maxsize$", then we simply rebuild the whole tree (and reset $maxsize = size(T)$). It can also be shown that this deletion scheme also runs in $\mathcal{O}(\log n)$ amortized time.[38]

Here's an implementation of a real scapegoat tree: `scapegoat.cpp`. Note that there are no auxiliary data stored in each node, which is one of the unique characteristics of the real scapegoat tree!

As a side note, if you're actually planning on using a scapegoat tree as a segment tree, then it turns out that it's better to implement a pseudo-scapegoat tree, since the segment tree itself will already contain the subtree-size information, by default. Hence, there's no need to get fancy by doing the above.

[37] Note that if there are no deletions performed, then the scapegoat tree is actually α-height-balanced.

[38] We will not be presenting the full analysis here; instead, we refer the reader to the paper by Galperin and Rivest which originally introduced the scapegoat tree (and which is actually easy to read): http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.309.9376&rep=rep1&type=pdf . This includes a proof that the scapegoat tree always has a height $\leq \log_{1/\alpha} size(T) + 1$.

7 Problems

Solve as many as you can! Ask me if anything is unclear.[39] In general, the harder problems will be worth more points.

7.1 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

Exercises mentioned in the main text are worth [5★].

First timers, solve as many as you can. Veterans, solve at least [80★].

N1 [10★] Show that $\Theta((f(n)+g(n))^{11}) = \Theta(\max(f(n),g(n))^{11})$.

N2 [10★] Show that $\Theta((n+q)\log(n+q)) = \Theta(n\log n + q\log q)$.

N3 [20★] Reduce range surgery into as few prefix reversals (in the worst case) as possible. Prove that this is the fewest possible. Non-optimal decompositions will get partial points.

N4 [25★] Explain why the following solution to the 2D range sum problem with point increases, while correct, is slow: `2drange_range_tree_slow.cpp`.

N5 [5★] Show that, using the solution to the sliding RMQ using a deque as described in subsection 6.1, each operation runs in $\mathcal{O}(1)$ amortized time.

N6 [25★] Suppose we implement a splay tree with lazy propagation to support the following operations: range sum, range increase, range reversals, point deletions and range deletions. Suppose further that range deletions are implemented by deleting each element one by one, hence can be $\mathcal{O}(n\log n)$ in the worst case. Without knowing further, this seems to result in a bad $\mathcal{O}(qn\log n)$ worst case time complexity.

Show that q consecutive operations actually run in $\mathcal{O}((n+q)\log n)$ time. You may assume that the splay operation takes $\mathcal{O}(\log n)$ amortized time.

N7 [10★] Show that for any $\alpha \geq 1$, all binary trees are α-weight-balanced, and for any $\alpha < 1/2$, there are only a finite number of distinct binary trees that are α-weight-balanced.

[39] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

7.2 Coding problems

Class-based implementations are strongly recommended; the idea is to make the implementation easily reusable.[40] Making the implementation self-contained in a class makes it very easy for you to reuse, which is handy for your future contests!

In the following, n will usually refer to the number of elements in the array, while q will usually refer to the number of queries.

First timers, solve as many as you can. Veterans, solve at least [222★].

C1 [15★] Extend `range_manip_sqrt.cpp` to include indexing. Your goal is to add a method in both classes `SqrtDecomp` and `Block` called `ll get_element(int i)` which returns the ith element. This should run in $\mathcal{O}(s)$ time. You may also opt to overload the indexing operator.[41]

C2 [15★] Implement an offline range sum, range increase, point insert and point delete using a technique similar to the one described in subsection 2.1. Then explain why an offline solution to this problem is no better/simpler than the online solution.

C3 [30★] Suppose you want to trigger the worst case for the treap-based solution to the prefix-reversal problem, `range_manip_treap2.cpp`. Suppose we get access to the seed of the random number generator used. Write a program that takes this random number seed as input and prints a sequence of operations that makes the runtime of the program very slow, say, 8 seconds or more. The resulting input must have $n \leq 10^5$, $q \leq 2 \cdot 10^5$, $A_i \leq 10^9$. The last query must be a prefix sum query.

You may use the program `range_manip_treap_seed.cpp`, which is a variant of the program that takes the seed as a command line argument. The only change is in the first few lines of the `main` implementation:

```
int main(int argc, char **argv) {
    int seed = atoi(argv[1]); // take the seed from argument
    srand(seed);
```

Thus, you can now compile and run it like this:

```
# compile the program
g++ -O2 -std=c++14 range_manip_treap_seed.cpp -o range_manip_treap_seed

# run it with seed 1100 and input from input.txt,
# and print the running time
time ./range_manip_treap_seed 1100 < input.txt
```

Your generator program itself must run quickly.

C4 [20★] Use lazy sqrt decomposition to solve the *online* range sum, range increase, point insert and point delete in $\mathcal{O}(n + q\sqrt{n+q})$ time.

C5 [20★] Use lazy sqrt decomposition to solve the *online* range minimum, range increase, point insert and point delete in $\mathcal{O}(n+q\sqrt{(n+q)\log(n+q)})$ time. Partial points if it runs in $\mathcal{O}(n + q\sqrt{n+q}\log(n+q))$ time.

[40] See also: https://en.wikipedia.org/wiki/Modular_programming.

[41] Simply implement the method `operator[]` instead of `get_element`, and then you can use `x[i]` instead of `x.get_element(i)`.

C6 [20★] Solve range minimum query + range increase updates on enormous arrays (i.e., $n \approx 10^{12}$) using lazy sqrt decomposition and segment trees. It should run in $\mathcal{O}(q\sqrt{q \log q})$ time. Partial points if it runs in $\mathcal{O}(q\sqrt{q} \log q)$ time.

C7 [20★] Implement a 3D range tree. The preprocessing should take $\mathcal{O}(n \log^2 n)$ time and it should support 3D range sum queries and point updates in $\mathcal{O}(\log^3 n)$ time. *Bonus:* $\mathcal{O}(\log^2 n)$ query and update time using fractional cascading.

C8 [20★] Implement sliding range minimum query as described in subsection 6.1 (using the "data structure" interpretation). It must have no preprocessing, be purely online, and must answer all queries in $\mathcal{O}(q)$ time.

C9 [20★] Implement a 2D sparse table: Given a 2D array of dimensions $n \times n$, you must be able to compute static rectangular range minimums in $\mathcal{O}(1)$ time, with $\mathcal{O}(n^2 \log^2 n)$ preprocessing.

C10 [30★] Implement a 2D sparse table: Given a 2D array of dimensions $n \times n$, you must be able to compute static rectangular range sums in $\mathcal{O}(1)$ time, with $\mathcal{O}(n^2 \log^2 n)$ preprocessing. No subtraction is allowed.

C11 [20★] Implement a binomial heap with worst-case $\mathcal{O}(\log n)$ push, pop-maximum, merge and get-maximum. *Bonus:* $\mathcal{O}(1)$ worst-case time for get-maximum. *Bonus:* Implement increase-key as well.

C12 [20★] Implement a pseudo-scapegoat tree with lazy propagation. Support the following operations: range sum, range minimum, range increase, point insert and point delete. The operations must run in $\mathcal{O}(n + q \log(n + q))$ time.

C13 [30★] Implement a splay tree with lazy propagation. Support the following operations: range sum, range minimum, range increase, point insert, point delete, range reversal and range surgery. Each operation must run in $\mathcal{O}(\log n)$ amortized time.

Please also attach your submissions in the Google classroom.

S1 [30★] **Testing the Game:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/testing-the-game

S2 [30★] **Jabahw:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/jabahw

S3 [30★] **Agadoo:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/agadoo

S4 [20★] **Chooga Choo Choo [Redux]:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/chooga-choo-choo-x

S5 [30★] **Max and Maximization:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/tslop-1

S6 [25★] **Confidential Message:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/confidentialegassem-8 (Use sqrt decomposition.)

S7 [25★] **Confidential Message:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/confidentialegassem-8 (Use a treap.)

S8 [40★] **Challenge of the Bignums:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/challenge-of-the-bignums

S9 [40★] **The Untended Antiquity:** https://codeforces.com/problemset/problem/869/E

S10 [40★] **Dynamic len(set(a[L:R])):** UVa 12345: ($\mathcal{O}((n+m)\log^2 n)$ required.)

S11 [30★] **Unique Art:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/unique-art

S12 [40★] **Card Shuffling:** https://www.codechef.com/problems/SHUFFLE2

A Amortized analysis of the pseudo-scapegoat tree

Here, we show that the amortized running time of insertion and deletion in a pseudo-scapegoat tree with $\alpha > 1/2$ is $\mathcal{O}(\log n)$.

We first note that each insertion/deletion goes through three steps:

- First, we perform a normal insertion/deletion procedure as we would in a normal binary search tree. This runs in $\mathcal{O}(\log n)$ worst-case time (because of Lemma 6.3).
- Next, along the path from the root to the inserted/deleted node, we find the scapegoat, i.e., the topmost node that is not α-weight-balanced, if any. This also runs in $\mathcal{O}(\log n)$ worst-case time.
- Finally, if the scapegoat s exists, then we rebuild that whole tree. This runs in $\mathcal{O}(\mathit{size}(s))$ time.

Obviously, the last step is potentially expensive. So, we will now use the accounting method to prove that the last step can be amortized away, i.e., that we can accumulate enough money to pay for that expensive step.

The key idea that allows this to work is that it takes an $\Theta(\mathit{size}(s))$ amount of work to turn a $(1/2)$-weight-balanced tree to a α-weight-*un*balanced tree. If we "pay" enough money alongside that work in advance, then by the time we will rebuild the scapegoat, we will have accumulated enough money to pay for the expensive rebuilding.

To simplify things a bit, suppose that rebuilding a subtree at node s requires $k \cdot \mathit{size}(s)$ pesos for some $k > 0$. We also assume that there is money at each node, all initially 0. To accumulate money, we use the following rule: *we pay x pesos for every node we encounter,* for some positive x that is as yet unspecified.[42]

Now, consider the scapegoat s. Note that every node that starts with 0 pesos in it is always going to be $(1/2)$-weight-balanced, since there are only two ways for a node to have 0 pesos:

- When it is a newly-inserted node, which means it has no children, and it is definitely $(1/2)$-weight-balanced.
- When it was part of a recent rebuild operation and has not been touched at any point since. This means that its subtree is $(1/2)$-weight-balanced due to the rebuild operation.

Now, how much work does it take to turn a $(1/2)$-weight-balanced tree to an α-weight-unbalanced tree? Well, it can only happen if enough insertions and deletions are done to the subtree to make things unbalanced. To be more specific, if we use the notation $\mathit{size}'(t)$ to denote the size of node t at the time when s has 0 pesos, then:

- at the start, we have $\mathit{size}'(\mathit{left}(s)) \leq (1/2) \cdot \mathit{size}'(s)$ and $\mathit{size}'(\mathit{right}(s)) \leq (1/2) \cdot \mathit{size}'(s)$,
- but right before rebuilding, we have $\mathit{size}(\mathit{left}(s)) > \alpha \cdot \mathit{size}(s)$ or $\mathit{size}(\mathit{right}(s)) > \alpha \cdot \mathit{size}(s)$.

Now, let's ask ourselves: what is the minimum number of insertions/deletions needed to make this happen?

[42] Note that we don't actually store this money in the implementation of the structure; this is a virtual currency that's only used to prove the efficiency of the operations.

It should not be hard to see that the shortest sequence of operations to do it will involve a bunch of deletions from one subtree and then a bunch of insertions to the other.[43] By symmetry, let's assume without loss of generality that we'll perform d deletions from the left subtree and i insertions on the right subtree. Then we will have the equalities:

$$\begin{aligned} \mathit{size}(\mathit{left}(s)) &= \mathit{size}'(\mathit{left}(s)) - d \\ \mathit{size}(\mathit{right}(s)) &= \mathit{size}'(\mathit{right}(s)) + i \\ \mathit{size}(s) &= \mathit{size}'(s) - d + i \end{aligned}$$

And it should also be clear that $\mathit{size}(\mathit{left}(s)) \leq \mathit{size}(\mathit{right}(s))$ and $\mathit{size}(\mathit{right}(s)) > \alpha \cdot \mathit{size}(s)$. But then, from this, we have

$$\begin{aligned} \alpha \cdot \mathit{size}(s) &< \mathit{size}(\mathit{right}(s)) \\ &= \mathit{size}'(\mathit{right}(s)) + i \\ &\leq (1/2) \cdot \mathit{size}'(s) + i \\ &= (1/2)(\mathit{size}(s) + d - i) + i \end{aligned}$$

which implies that

$$\begin{aligned} (1/2)(\mathit{size}(s) + d - i) + i &> \alpha \cdot \mathit{size}(s) \\ \mathit{size}(s) + d - i + 2i &> 2\alpha \cdot \mathit{size}(s) \\ d + i &> (2\alpha - 1) \cdot \mathit{size}(s) \end{aligned}$$

Since $\alpha > 1/2$, the coefficient $(2\alpha - 1)$ is positive, and informally, it basically says that the amount of work needed to unbalance the tree ($\geq d+i$) is at least some proportion of its size. To be more precise, by performing at least $d+i$ operations, we accumulate at least $x \cdot (2\alpha - 1) \cdot \mathit{size}(s)$ pesos in the scapegoat. If this quantity is greater than the rebuild cost, $k \cdot \mathit{size}(s)$, then we have enough money to pay for the rebuild. But we can simply make x large enough for that to be true! For example, we can simply select $x = \frac{k}{2\alpha - 1}$, which is a positive finite value.

So we can now conclude the amortized analysis. Since there's always enough money to pay for the rebuild, we can now assume that it can be accounted for. This means that the amortized running time of delete and insert is proportional to the height of the tree, and since we pay x pesos for every node we pass through, then each insertion/deletion costs $\mathcal{O}(\log n + \log n + x \cdot \log n)$, but since x is a fixed constant (dependent on α), this is basically $\mathcal{O}(\log n)$.

Furthermore, the value we get for x also sheds light on why a more balanced tree, i.e., a smaller α, yields a worse insertion/deletion cost: Notice that as α decreases, $x = \frac{k}{2\alpha - 1}$ increases, signifying a greater insertion cost. In fact, if we also look at the dependence of the time complexity on α, we get

- A worst-case search time of $\mathcal{O}(\log_{1/\alpha} n) = \mathcal{O}(\frac{1}{\lg(1/\alpha)} \lg n)$. The constant $\frac{1}{\lg(1/\alpha)}$ increases as α increases to 1.
- An amortized insertion/deletion time of $\mathcal{O}(\frac{1}{2\alpha - 1} \log_{1/\alpha} n) = \mathcal{O}(\frac{1}{(2\alpha - 1)\lg(1/\alpha)} \lg n)$. The constant $\frac{1}{(2\alpha - 1)\lg(1/\alpha)}$ decreases as α increases to 1 or decreases to 1/2.

It also sheds light on why we can only choose $1/2 < \alpha < 1$; if we let $\alpha = 1/2$, then $2\alpha - 1$ becomes zero and $x = \frac{k}{2\alpha - 1}$ becomes "infinity", which informally says that we have to pay an infinite amount of money to pay for the rebuilding cost. On the other hand, if $\alpha = 1$, then $\frac{1}{\lg(1/\alpha)}$ is "infinity", which informally signifies that the height can grow asymptotically larger than $\lg n$.

[43] This should be intuitive to see because any other sequence of operations would be "wasteful" in some way, i.e., we can find a shorter sequence that still unbalances the tree. We will leave the formal proof to the reader.