Competitive Programming Notes

Pakin Olanraktham

December 3, 2025

Contents

1   Range Queries

1.1   Square Root Decomposition

1.1.1   Blocking

1.1.2   Combining

1.1.3   Batching

1.1.4   Mo's Algorithm

2   Data Structures

2.1   Wavelet Tree

Wavelet Tree is a Data Structure mainly for finding the k'th minimum value in some range $[l, r]$. The tree is built dynamically meaning that we will use pointers.

```cpp
struct Wavelet {
    vector <int> p;
    Wavelet *l, *r;
};
```

When we build the tree, we check if the b'th bits is 0 or 1. If it's 0, we put that element into the left side. Otherwise, we put in the right side. The vector p is a prefix sum array counting number of elements in the left side.

```cpp
void build(const vector <int> &A, int b) {
    if (b == 0 || A.empty()) return;
    l = new Wavelet();
    r = new Wavelet();
    int c = 0;
    vector <int> L, R;
    for (int x : A) {
        if (x & (1 << (b-1))) R.emplace_back(x);
        else L.emplace_back(x), c++;
        p.emplace_back(c);
    }
    l->build(L, b-1);
    r->build(R, b-1);
}
```

When we query, we count the number of elements in the left side in range l to r. If the number of elements in the left side is more than or equal to k, we recursive to the right side. Otherwise we go to the right side. We have to change the value of l, r, k too

when moving down to the right side.

```cpp
int qr(int l, int r, int k, int b) { // [l, r), k, 0-index
    if (b == 0 || p.empty()) return 0;
    int pr = (r > 0 ? p[r-1] : 0);
    int pl = (l > 0 ? p[l-1] : 0);
    int cnt = pr - pl;
    if (k < cnt) return this->l->qr(pl, pr, k, b-1);
    else return (1 << (b-1)) + this->r->qr(l-pl, r-pr, k-cnt, b-1);
}
```

Note that these functions are inside the `Wavelet` struct.

3 Dynamic Programming Optimization

3.1 Convex Hull Trick

Convex Hull Trick is an algorithm to find $\min_{j<i}(f_j(x))$ or $\max_{j<i}(f_j(x))$ efficiently.

First, we convert the function $f(x)$ to $mx + c$, which x depends only on i, m and c depends only on j. For example

```
dp[i] = max(dp[j] + x[i]*y[i] - x[j]*y[i] - a[i])
      = max((-x[j])*y[i] + dp[j]) + x[i]*y[i] - a[i]
```

As we can see, m = -x[j], x = y[i], c = dp[j]. Note that if we want to do CHT, the slope **must** be monotonic. Otherwise, we have to use Li Chao Tree.

Now, how do we compute the maximum or minimum? We know that the recurrence is a line with monotonic slope.

3.1.1 Inserting

We will use deque <int> to store the lines indices.

For computing maximum, we want to maintain the lines as lower hull shaped. For minimum, we want the lines as upper hull shaped.

Before inserting, we have to check whether the old line is useless or not. We can do that by checking the intersection point between the [new line, top] and [top, top-1]. (You should try drawing on paper by yourself.)
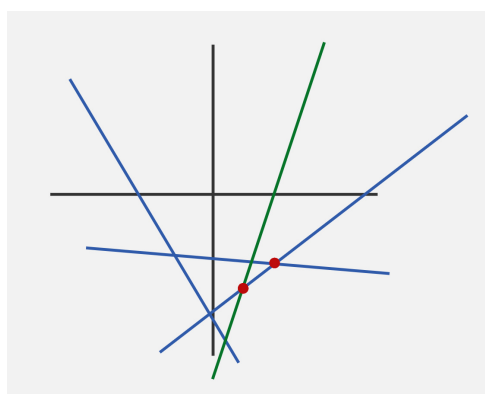
Lower Hull (Maximum)
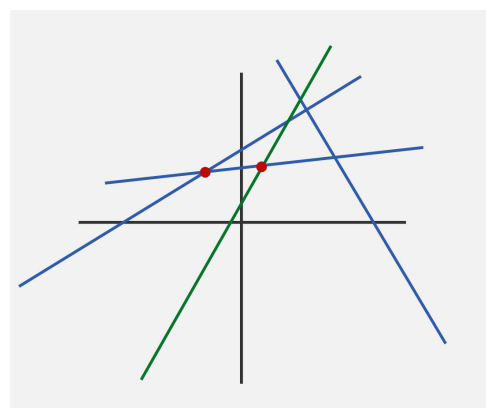
Upper Hull (Minimum)

For finding the intersection point we can do

```
double m(int x, int y) {
    return (double) (c[x] - c[y]) / (m[y] - m[x]);
}
```

3.1.2 Querying

Monotone Query

If the hull is maintained correctly for each type of query, we will always query from front for non-decreasing queries, and always from back for non-increasing queries.
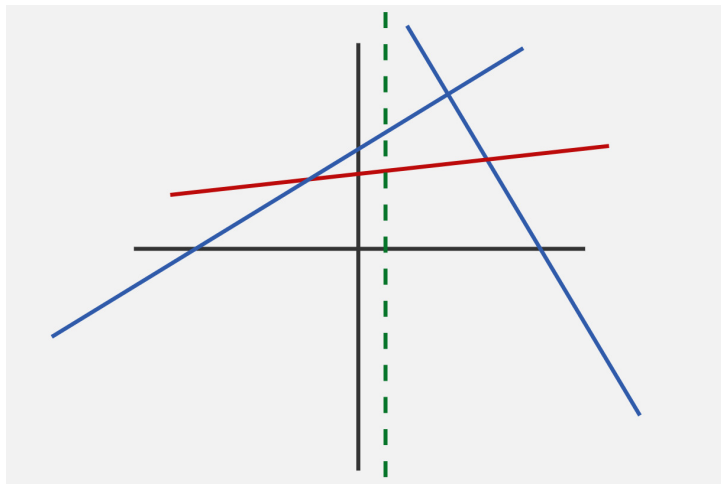
We will check the top and top-1. For maximum, if $f_{\text{top}}(x) \leq f_{\text{top-1}}(x)$, we pop the top. For minimum, if $f_{\text{top}}(x) \geq f_{\text{top-1}}(x)$, we pop the top. For example

```
while (sz > 1 && calc(cht[0], p[i]) <= calc(cht[1], p[i])) {
    cht.pop_front(), sz--;
}
```

Then the answer will be $f_{\text{top}}(x) + c$ where c is a constant that depends only on x.

Arbitrary Query

We perform a binary search for the line that contain x.


```
int l = 0, r = sz-2, idx = sz-1;
while (l <= r) {
    int mid = l + (r-l)/2;
    if (m(cht[mid], cht[mid+1]) <= x) l = mid+1;
    else r = mid-1, idx = mid;
}
```

4 Tree Algorithms

4.1 Centroid Decomposition

4.1.1 Centroid

A centroid of a tree is defined as a node such that if removed, no connected component has more than $\frac{N}{2}$ nodes.

We can find a centroid in a tree by selecting an arbitrary node in the tree. Then loop through its children's subtree. If the size of a child's subtree is more than $\frac{N}{2}$, traverse down that node. Otherwise, if no child's subtree has size more than $\frac{N}{2}$, then that node is the centroid.

```cpp
void fsz(int u, int p) {
    sz[u] = 1;
    for (auto v : adj[u]) if (v != p) fsz(v, u), sz[u] += sz[v];
}

int ct(int u, int p) {
    for (auto v : adj[u]) if (sz[v] * 2 > n) return ct(v, u);
    return u;
}
```

Properties

1. If the centroid of a tree isn't unique, there are exactly two centroids which are adjacent. Removing the edge between them divides the tree into two connected components of the equal sizes.
2. When a leaf is added to or removed from a tree, the centroid changes by at most one node.
3. When two trees are connected, the centroid of the resulting tree lies on the path between the centroids of the original trees.

4.1.2 Centroid Decomposition

Centroid Decomposition is a divide-and-conquer technique for trees, which recursively splits the tree by removing its centroid. The resulting decomposition tree has a height of at most $\mathcal{O}(\log N)$.

Properties

1. The height of the decomposed tree is at most $\log N$
2. Every path between u and v can be decomposed into two parts: u to $\text{lca}_{ctd}(u, v)$ and $\text{lca}_{ctd}(u, v)$ to v. We can use this property for querying and updating. In the **Xania and Tree** problem, we update and query by traversing up to the $\log N$ parents for each node.

```cpp
int fsz(int u, int p) {
    sz[u] = 1;
    for (auto v : adj[u]) if (v != p && !vis[v]) sz[u] += fsz(v, u);
    return sz[u];
}

int ct(int u, int p, int tsz) {
    for (auto v : adj[u]) if (v != p && !vis[v]) if (sz[v] * 2 > tsz
        ) return ct(v, u, tsz);
    return u;
}

void ctd(int u, int p) {
    int C = ct(u, u, fsz(u, u));
    vis[C] = 1;
    if (p == -1) p = C;
    par[C] = p;
    for (auto v : adj[C]) if (!vis[v]) ctd(v, C);
}
```

5 Miscellaneous

5.1 Bitset

Bitset or `std::bitset` is a boolean array datastructure, which can do operations like normal bits such as AND, OR, XOR, etc. It is faster than normal array about x32-64.

idk, not finnished...