Square Root Decomposition Ideas

Mazen Ghanayem

August 13, 2025

1 Ideas

- Update Query: k v, $a[k] \leftarrow v$
  Range Frequency: l r x, count Frequency of x in range $[l, r]$

  Implementation Details:

  - Preprocess the array into blocks of size $\sqrt{n}$.
  - Maintain a frequency map for each block.
  - For a query, combine results from relevant blocks.

- Update Query: k v, $a[k] \leftarrow v$
  Query: l r c, count elements in range $[l, r]$ greater than or equal to c.

  Implementation Details:

  - Preprocess the array into blocks of Ordered Multiset of size $\sqrt{n}$.

- Update Query: `0 a b`, set power of hole a to value b.
  Jump Query: `1 a`, find the last hole visited and the total number of jumps before the ball leaves the row.

  Implementation Details:

  - Preprocess the array of N holes into blocks of size $\sqrt{N}$.
  - For each hole i, precompute a pair of values: (1) the next hole the ball lands in *outside* its current block, and (2) the number of jumps it takes to get there.
  - A jump query can then be answered in $O(\sqrt{N})$ time by chaining these precomputed block-to-block jumps.
  - An update only requires recomputing the values for the single block that was changed, also in $O(\sqrt{N})$ time.

```cpp
    void process() {
        for (int i = n - 1; i >= 0; i--) {
            int idx = i / SQ, r = min(n - 1, (i / SQ + 1) * SQ - 1LL);
            if (i + arr[i] > r) {
                jumps[i] = {i, 1};
            } else {
                jumps[i] = jumps[i + arr[i]];
                jumps[i].second++;
            }
        }
    }

    void update(int idx, ll val) {
        arr[idx] = val;
        int blk_idx = idx / SQ;
        int l = blk_idx * SQ, r = min(n - 1, (blk_idx + 1) * SQ - 1LL);
        for (int i = r; i >= l; i--) {
            if (i + arr[i] > r) {
                jumps[i] = {i, 1};
            } else {
                jumps[i] = jumps[i + arr[i]];
```

```
                    jumps[i].second++;
                }
            }
        }

        pair<int, int> query(int l) {
            pair<int, int> ans = {0, 0};
            int idx = l;
            while (idx < n) {
                ans.first = jumps[idx].first;
                ans.second += jumps[idx].second;
                idx = jumps[idx].first + arr[jumps[idx].first];
            }
            return ans;
        }
```

- Range Update: `L R X`, add X to heights of hills in range $[L, R]$.
  Jump Query: `i k`, find the final hill after making k jumps starting from hill i. A jump is to the nearest hill on the right that is strictly taller and within a distance of 100.

  Implementation Details:

  – Divide the array into blocks of size $\sqrt{N}$. Use a lazy array (e.g., `lazy_add`) for efficient range updates on full blocks.

  – For each hill i, precompute: the next immediate jump (`next_jump`), the final hill reached if jumping only within the block (`block_exit_node`), and the number of jumps to get there (`jumps_to_exit`).

  – This preprocessing is done efficiently in $O(\sqrt{N})$ per block by iterating backwards with a stack to find the next greater element.

  – A jump query uses the precomputed data to skip across blocks and performs a small linear scan for inter-block jumps, leading to an overall $O(\sqrt{N})$ query time.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

const int MAX_N = 100005, MAX_SQRT_N = 320, MAX_JUMP_DIST = 100;
int n, q, block_size;
ll heights[MAX_N], lazy_add[MAX_SQRT_N];
int next_jump[MAX_N], block_exit_node[MAX_N], jumps_to_exit[MAX_N];

void rebuild_block(int block_idx) {
    int start_idx = block_idx * block_size;
    int end_idx = min((block_idx + 1) * block_size, n);

    if (lazy_add[block_idx] != 0) {
        for (int i = start_idx; i < end_idx; ++i) heights[i] += lazy_add[block_idx];
        lazy_add[block_idx] = 0;
    }

    stack<int> s;
    for (int i = end_idx - 1; i >= start_idx; --i) {
        while (!s.empty() && (s.top() - i > MAX_JUMP_DIST || heights[s.top()] <= heights[i
            ])) {
            s.pop();
        }

        if (!s.empty()) {
            next_jump[i] = s.top();
            block_exit_node[i] = block_exit_node[next_jump[i]];
            jumps_to_exit[i] = jumps_to_exit[next_jump[i]] + 1;
        } else {
            next_jump[i] = i;
            block_exit_node[i] = i;
            jumps_to_exit[i] = 0;
        }
        s.push(i);
    }
}

void update_range(int l, int r, ll val) {
    int start_block = l / block_size;
    int end_block = r / block_size;

    if (start_block == end_block) {
        for (int i = l; i <= r; ++i) heights[i] += val;
        rebuild_block(start_block);
```

```cpp
        return;
    }

    for (int i = l; i < (start_block + 1) * block_size; ++i) heights[i] += val;
    rebuild_block(start_block);

    for (int i = start_block + 1; i < end_block; ++i) lazy_add[i] += val;

    for (int i = end_block * block_size; i <= r; ++i) heights[i] += val;
    rebuild_block(end_block);
}

int query_jumps(int start_idx, int k) {
    int current_hill = start_idx;

    while (k > 0) {
        // Case 1: We don't have enough jumps to take the full precomputed path
        // We must take a single, precomputed step instead.
        if (jumps_to_exit[current_hill] > 0 && k < jumps_to_exit[current_hill]) {
            current_hill = next_jump[current_hill];
            k--;
            continue;
        }

        // Case 2: We have enough jumps to use the precomputed path.
        if (jumps_to_exit[current_hill] > 0) {
            k -= jumps_to_exit[current_hill];
            current_hill = block_exit_node[current_hill];
        }

        // Case 3: We are now at a block's exit point and still have jumps left
        // We must perform one jump by scanning manually.
        if (k > 0) {
            bool jumped = false;
            ll current_true_height = heights[current_hill] + lazy_add[current_hill / block_size];

            for (int j = current_hill + 1; j < min(n, current_hill + 1 + MAX_JUMP_DIST); ++j) {
                ll next_true_height = heights[j] + lazy_add[j / block_size];
                if (next_true_height > current_true_height) {
                    current_hill = j;
                    k--;
                    jumped = true;
                    break;
                }
            }
            if (!jumped) break; // No further jump possible, we are stuck.
        }
    }
    return current_hill + 1;
}

int main() {
    ios_base::sync_with_stdio(false), cin.tie(NULL);

    cin >> n >> q;
    block_size = static_cast<int>(sqrt(n));
    if (block_size == 0) block_size = 1;

    for (int i = 0; i < n; ++i) {
        cin >> heights[i];
    }

    for (int i = (n - 1) / block_size; i >= 0; --i) {
        rebuild_block(i);
    }

    while (q--) {
        int type;
        cin >> type;
        if (type == 1) {
            int i, k;
            cin >> i >> k;
            cout << query_jumps(i - 1, k) << "\n";
        } else {
            int l, r;
            ll x;
            cin >> l >> r >> x;
            update_range(l - 1, r - 1, x);
        }
    }
    return 0;
}
```