Contest 2 Editorial

COMP4128 25T2

July 25, 2025

A Talent Show

Subtask Solutions

Since all students have the same skill, we simply want to select as many students as possible.

Forward solution

We reason that since every student has the same value, we try to place the most constrained students (earliest departure time) first, and fit as many others around that.

We sort the students by increasing departure time, and one by one we add each student to the schedule only if their departure time is greater than the number of students before them on the schedule.

Reverse solution

For the sake of fitting in as many students as possible, there is a pressure to make later-departing students perform as late as possible.

We sort the students by decreasing departure time and one by one assign the student to the latest time slot that they could perform at. We keep track of where we placed the last student so that we know that the next one must go before them. Eventually we will place down all the students, or we will place a student in the first time slot and be unable to place any more.

This process leaves gaps in the schedule, but nobody was able to perform in those gaps so we compress the schedule down to remove them.

Full Solutions

Latest first

The end of the schedule is much easier to determine because there are fewer students around that are valid options. So starting at the end of the schedule, we place the highest skill valid student into each slot.

We keep a priority queue of the students who haven't yet departed (currently empty), sorted such that the highest skill student is on top

We start at time slot n and decrement until we reach 1. At each time slot i we:

- Add the students who depart after performance i to the priority queue
- Select the highest skill student (if any) from the priority queue and put them in the schedule at this time.

We can prove this is optimal by exchange argument. Let's name the greedy schedule $G = (g_1, g_2, ..., g_n)$ and we will consider an alternate schedule $A = (a_1, a_2, ..., a_n)$. We will show that all such alternatives can be transformed to A' such that $\text{total}(A') \geq \text{total}(A)$ and repeated transformations will result in G.

Find the latest student that differs: ($a_k \neq g_k$ but $\forall i \in (k, n] : a_i = g_i$).

- If g_k doesn't appear anywhere in A then set $a'_k = g_k$. Since g_k is the most skilful student who is available at that time that isn't used later, we know $g_k \geq a_k$ and therefore $\text{total}(A') \geq \text{total}(A)$
- If $\exists j$ such that $a_j = g_k$, then we swap $(a'_j, a'_k) = (a_k, a_j)$. We know a_j can be slotted later because the greedy does, and a_k is always available earlier, so this is a valid assignment. This keeps the total the same so $\text{total}(A') \geq \text{total}(A)$

Each iteration of this transform makes one more student match the greedy schedule's suffix, so after n iterations, we can demonstrate that all alternative schedules have a total less than or equal to the greedy. Thus the greedy is optimal

Most Skilful First

We definitely want to take the most skilled student. We want to take the second most skilled as long as there's still space. In order to leave as much space each time, we slot each student in the last remaining time slot before they leave.

We keep a `set<int>` of which slots in the schedule are remaining, initially $[1, \ldots, n]$. The set is reverse ordered (using `greater<int>` rather than the default `less`) so that when we use `set.lower_bound(d)`, instead of returning the first number $\geq d$, it returns the first number $\leq d$.

We sort the students by decreasing skill. For each student:

- We use `set.lower_bound(d)` to find the last slot up to d that is still available, and we remove that slot. If we don't find any, then we do not put that student in.

The proof of correctness is left as an exercise to the reader. Use an exchange argument.

Alternatively, instead of a set, we could use the union-find data structure with path-compression only.

- We want `root(d)` to give us the last available slot $\leq d$.
- Initially, `root(x) = x` for all `x` as required.
- Each time we use the slot `root(d)`, we want to enforce that next time we look for something that leads to `root(d)`, we should instead look for the next slot to the left - this is exactly defined by `root(root(d)-1)`. So we set `par[root(d)] = root(root(d)-1)`.
- path compression ensures that the total runtime won't exceed $\mathcal{O}(n \log n)$.

Earliest First

We try to fit in the most constrained students (earliest departure time) first, but if we come across a student that can't fit in the schedule, we find the lowest skill student in the current schedule and replace them.

We keep the students on our schedule (currently empty) in a priority queue, arranged such that the lowest skill student is on top.

We sort the students by increasing departure time, and one by one:

- add the student to the priority queue, and add their skill to the total
- if the size of the priority queue is larger than the student's departure time we remove the lowest skill student (top) and subtract their skill from the total.

  This is equivalent to swapping the student with the previously least-skilled student – unless the most recent student was the least-skilled, in which case he is summarily removed.

This guarantees that each student is added to the schedule at a time before their departure, and students are only left out if there is insufficient space and nobody worse to cut.

How to check your greedy

A common error is to come up with a greedy solution, only to find out that it's incorrect after writing and submitting your code. Here are some strategies you can use to debug this before you start coding:

Reason about what information is necessary

To decide if a student should be put into the schedule, we want to know if there's anyone better to put in that slot. There are two ways that could happen:

- a more skilled student is competing for that slot, or
- a less skilled student is about to leave but they could perform if the current student performs later.

If your greedy doesn't address these in some way, then it's not going to be correct. This framing also prompts us towards two greedy approaches:

- start with the latest slot and put in the best student you can, or
- start with the best students, and put them in the latest slot you can

Look for breaking cases

Knowing what your algorithm does, try to guide it into making what you know is the wrong decision. Let's try the greedy approach "Put the most skilled student in the first slot".

We start by adding a student Alicia that we know the algorithm wants to take (skill $= 999$, departure $=$ 10), and then we try to create a scenario where that student should not be taken first. We can do this by only adding a student Bradley who needs to go first instead (skill $= 1$, departure $= 1$). Now we see that the best schedule is (Bradley, Alicia), but our greedy will put Alicia first and leave Bradley out, demonstrating that it's suboptimal.

Implementation Notes

- any priority queue can be replaced with a `multiset<int>` or a `set<pair<int, int>>` where the second int is a unique identifier to prevent duplicates from being ignored.

Reference Solution(s)

```cpp
// Subtask solution by Raveen, reverse order
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 200200;
int dep[N];

int main (void) {
    int n;
    cin >> n;
    int end = 0;
    for (int i = 0; i < n; i++) {
        int s, d;
        cin >> s >> d;
        dep[d]++;
        end = max(end, d);
    }
    long long ans = 0;
    int cnt = 0;
    for (int i = end; i >= 1; i--) {
        cnt += dep[i];
        if (cnt != 0) {
            ans++;
            cnt--;
        }
    }
    cout << ans << '\n';
}
```

```cpp
// Solution by Raveen, latest first
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
```

```cpp
using namespace std;

const int N = 200200;
vector<int> dep[N];
priority_queue<int> pq;

int main (void) {
    int n;
    cin >> n;
    int end = 0;
    for (int i = 0; i < n; i++) {
        int s, d;
        cin >> s >> d;
        dep[d].push_back(s);
        end = max(end, d);
    }
    long long ans = 0;
    for (int i = end; i >= 1; i--) {
        for (int s : dep[i]) { pq.push(s); }
        if (!pq.empty()) {
            ans += pq.top();
            pq.pop();
        }
    }
    cout << ans << '\n';
}
```

```cpp
// Solution by Gordon, most skilled first using set
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

int main() {
    int n;
    cin >> n;
    vector<pair<int,int>> s;
    set<int, greater<int>> taken; // reverse order
    for (int i = 1; i <= n; i++) {
        taken.insert(-i);
    }
    for (int i = 0; i < n; i++) {
        int x,y;
        cin >> x >> y;
        s.push_back({x,y});
    }
    sort(s.begin(), s.end());
    reverse(s.begin(), s.end());
    ll ans = 0;
    for (auto p : s) {
        auto it = taken.lower_bound(p.second);
        if (it != taken.end()) {
            ans += p.first;
            taken.erase(it);
        }
    }
    cout << ans << "\n";
}
```

```cpp
// Solution by Yiheng, most skilled first using union-find
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

#define f first
#define s second

const int N = 200200;

struct UnionFind {
        int par[N];
        UnionFind() {
                for (int i = 0; i < N; i++) {
```

```
                        par[i] = i;
                }
        }
        int find(int x) {
                return par[x] == x ? x : par[x] = find(par[x]);
        }
        void merge(int x, int y) {
                x = find(x);
                y = find(y);
                par[y] = x;
        }
};

int main() {
        cin.tie(0)->sync_with_stdio(0);
        cin.exceptions(cin.failbit);
        int n; cin >> n;
        vector<pair<int, int>> v(n);
        for (int i = 0; i < n; i++) { cin >> v[i].f >> v[i].s; }
        sort(begin(v), end(v), [&](const auto &x, const auto &y) {
                return x.f == y.f ? x.s < y.s : x.f > y.f;
        });
        UnionFind uf;
        ll ans = 0;
        for (auto [s, d] : v) {
                if (uf.find(d) == 0) continue;
                int x = uf.find(d);
                uf.merge(x - 1, x);
                ans += s;
        }
        cout << ans << "\n";
}
```

```
// Solution by Laeeque, earliest first
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
typedef long long ll;

const ll N = 200200;
vector<pair<ll, ll>> students;
priority_queue<ll, vector<ll>, greater<ll>> schedule;

signed main (void) {
    ll n;
    cin >> n;
    ll end = 0;
    for (ll i = 0; i < n; i++) {
        ll skill, departure;
        cin >> skill >> departure;
                students.push_back({departure, skill});
    }
        sort(students.begin(), students.end());
    ll total = 0;
        for (auto st : students) {
                ll departure, skill; tie(departure, skill) = st;
                total += skill;
                schedule.push(skill);
                if (schedule.size() > departure) {
                        total -= schedule.top();
                        schedule.pop();
                }
        }
    cout << total << '\n';
}
```

B Enclosure

Subtask

When all the values are 1, a rectangular field (and therefore the minimum length of walls to divide it) is entirely determined by its dimensions (width and height). For each rectangle, try every possible way to split it into two rectangles using a single horizontal or vertical wall, solving the subrectangles recursively, and select the decomposition with the cheapest total cost. The base case is a rectangle of area at most T, for which the answer is 0. With memoisation, this takes $O(n)$ time for each of n^2 different rectangle shapes, for a total of $O(n^3)$ runtime.

Full

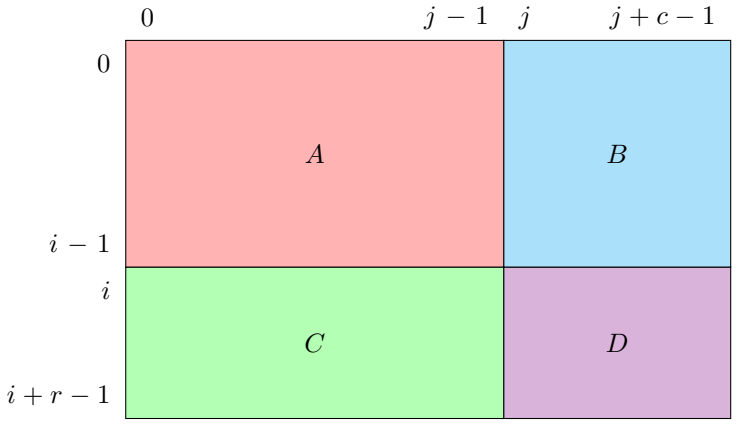
Each rectangle in the grid can be identified by its dimensions (width and height) and location (say, row and column index of its top left corner). Each of these four parameters are up to n. As before, there are up to $2n$ different ways to slice a rectangle into two smaller rectangles, and we can try each one to see which is cheapest.

Subproblem

Let best(i, j, r, c) be the cheapest way to cut the rectangle from (i, j) with height r and width c into smaller rectangles of at most weight T, using only slices that cut rectangles into two smaller ones.

Base Cases

If the elements of the rectangle (i, j, r, c) sum to less than T, then best(i, j, r, c) should be set to 0. Otherwise, initialize it to ∞. The sum of elements, say rectsum(i, j, r, c) can be calculated in $O(1)$ time by creating a 2D prefix sum for the grid, and using inclusion-exclusion to calculate the area of a specific rectangle.



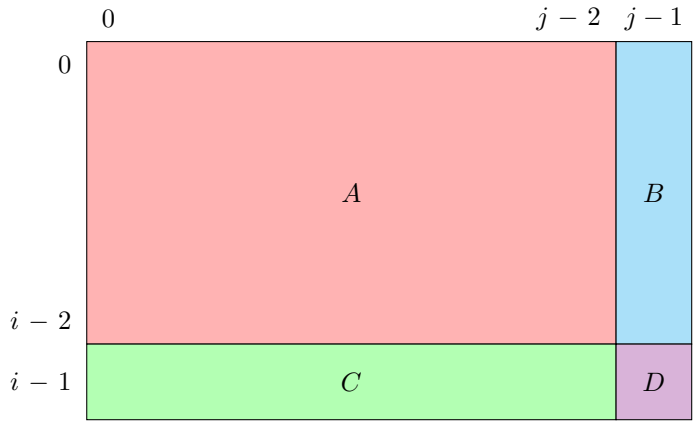
Rectangle D can be calculated as

- $A+B+C+D$ ($i+r$ rows and $j+c$ columns),
- less $A+B$ (i rows and $j+c$ columns) and $A+C$ ($i+r$ rows and j columns),
- plus back A (i rows and j columns), as it was included once and excluded twice.

```
// sum[r][c] contains the sum of all values in the first r rows and c columns
// sum[r][c] = sum(values[0..r)[0..c))
// so sum[0][c] and sum[r][0] are 0 and sum[R][C] is the sum of the entire grid.
int values[MAXR][MAXC], sum[MAXR+1][MAXC+1];

// The sum of the values in the rectangle from (i, j) of height r and width c
// rectsum(i, j, r, c) = sum(values[i..i+r)[j..j+c))
inline int rectsum(int i, int j, int r, int c) {
    return sum[i+r][j+c] - sum[i][j+c] - sum[i+r][j] + sum[i][j];
}
```

This relies on precomputation of the 2D prefix sum. In this problem, that could be conducted naïvely in $O(n^4)$ as n is only up to 60, but it can be done in $O(1)$ per entry (i.e. $O(n^2)$ overall) by noticing that `values[i-1][j-1]` should equal rectsum$(i-1, j-1, 1, 1)$ and rearranging the formula.



Rectangle A+B+C+D can be calculated as

- A+B ($i-1$ rows and j columns) plus A+C (i rows and $j-1$ columns),
- less A ($i-1$ rows and $j-1$ columns), which has been double-counted,
- plus the single entry in rectangle D.

```
for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++) {
        cin >> values[i][j];
    }
}

for (int i = 1; i <= R; i++) {
    for (int j = 1; j <= C; j++) {
        sum[i][j] = sum[i-1][j] + sum[i][j-1] - sum[i-1][j-1] + values[i-1][j-1];
    }
}
```

Recurrence

Try every horizontal and vertical slice and add the cost of the slice to the cost of splitting up the remaining 2 subproblems. Find the one that is cheapest.

```
int tmp = INF;
for (int split = 1; split < dr; split++) { // horizontal walls
    tmp = min(tmp, c + dp[i][j][split][c] + dp[i+split][j][r-split][c]);
}
for (int split = 1; split < dc; split++) {// vertical walls
    tmp = min(tmp, r + dp[i][j][r][split] + dp[i][j+split][r][c-split]);
}
dp[i][j][r][c] = tmp;
```

Order of Calculation, Final Answer and Time Complexity

As long as all smaller rectangles are already calculated, larger rectangles have all the information needed to calculate the optimal split. So we can proceed bottom-up by increasing r and c, followed by any order of i and j. Alternatively, top-down implementation is equally viable.

Final answer is best$(0, 0, R, C)$: the cheapest way to split the entire grid.

Each state takes $O(n)$ time to calculate and there are n^4 states, for a time complexity of $O(n^5)$. With input size of 60, this gives us 8×10^8, which is borderline too large. Fortunately, the number of states we actually use is $\frac{1}{4}n^4$ and the average dp state relies on $\frac{2}{3}n$ previous states: these constant factor reductions make it fast enough to run in time.

Implementation Notes

If implementing bottom-up, take care to ensure that $i + r$ and $j + c$ do not extend beyond the edges of the grid.

Reference Solution(s)

```cpp
// Solution by Raveen for Subtask 1 (top-down)
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 62;
int a[N][N];
int dp[N][N][N][N];

const int INF = 1000*1000*1000+7;
int n, m, T;

int solve (int i, int j, int r, int c) {
    if (dp[i][j][r][c] != INF) { return dp[i][j][r][c]; }
    if (r * c <= T) { return dp[i][j][r][c] = 0; }
    int ret = INF;
    for (int k = 1; k < r; k++) {
        ret = min(ret, c + solve(i, j, k, c) + solve(i+k, j, r-k, c));
    }
    for (int k = 1; k < c; k++) {
        ret = min(ret, r + solve(i, j, r, k) + solve(i, j+k, r, c-k));
    }
    return dp[i][j][r][c] = ret;
}

int main (void) {
    cin >> n >> m >> T;

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

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                for (int l = 0; l < N; l++) {
                    dp[i][j][k][l] = INF;
                }

    cout << solve(0, 0, n, m) << '\n';
}
```

```cpp
// Solution by Raveen (bottom-up)
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 62;
int a[N][N];
int b[N][N];
int dp[N][N][N][N];

const int INF = 1000*1000*1000+7;

int main (void) {
    int n, m, T;
    cin >> n >> m >> T;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> a[i][j];
        }
    }
```

```
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            b[i][j] = b[i-1][j] + b[i][j-1] - b[i-1][j-1] + a[i-1][j-1];
        }
    }

    for (int r = 1; r <= n; r++)
        for (int c = 1; c <= m; c++)
            for (int i = 0; i <= n-r; i++)
                for (int j = 0; j <= m-c; j++) {
                    if (b[i+r][j+c] - b[i+r][j] - b[i][j+c] + b[i][j] <= T) {
                        dp[i][j][r][c] = 0;
                    } else {
                        dp[i][j][r][c] = INF;
                        for (int k = 1; k < r; k++) {
                            dp[i][j][r][c] = min(dp[i][j][r][c], c + dp[i][j][k][c] + dp
                                [i+k][j][r-k][c]);
                        }
                        for (int k = 1; k < c; k++) {
                            dp[i][j][r][c] = min(dp[i][j][r][c], r + dp[i][j][r][k] + dp
                                [i][j+k][r][c-k]);
                        }
                    }
                }

    cout << dp[0][0][n][m] << '\n';
}
```

C LinkedList

This problem was hard — well done to the 20 students who received a non-zero score, of which only 9 students solved this problem fully.

Algorithm

This problem was motivated by the workshop problem `Restructuring Company`. This problem can be solved **without** the use of any graph algorithms at all.

General ideas

As hinted at by `Update 1`, to facilitate group-chat merges, we utilize the Union-find data structure. Recall that if N groups are present, Union-find (with path-compression and union-by-size optimisation) allows for `find/merge` operations to be performed in $O(\alpha(N))$ time each.

If Union-find is applied naively to this problem:

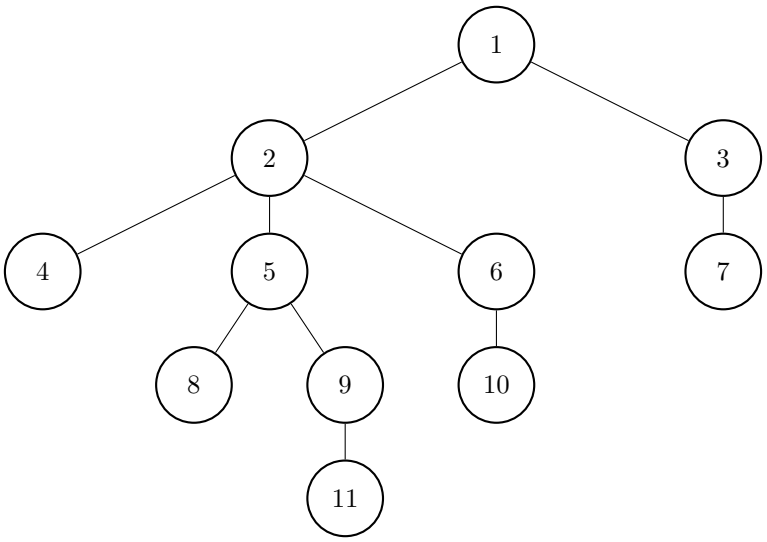
- `Update 1` and `Query` can be handled in $O(\alpha(N))$ time.
- `Update 2` can be handled at the time complexity of $O(\text{length of path} \cdot \alpha(N))$ per update. However, length of path can be at most N — consider the case where the tree is a line and `Update 2` is applied to the endpoints — resulting in a worst-case time complexity of $O(N\alpha(N))$ per `Update 2`.

When Q operations are present, the total worst-case time complexity is $O(NQ\alpha(N))$ which will receive `Time Limit Exceeded` for both Subtask 1 and Subtask 2 due to $N, Q \leq 500,000$.

To score points on this problem, a solution which involves more thought and ingenuity is required.

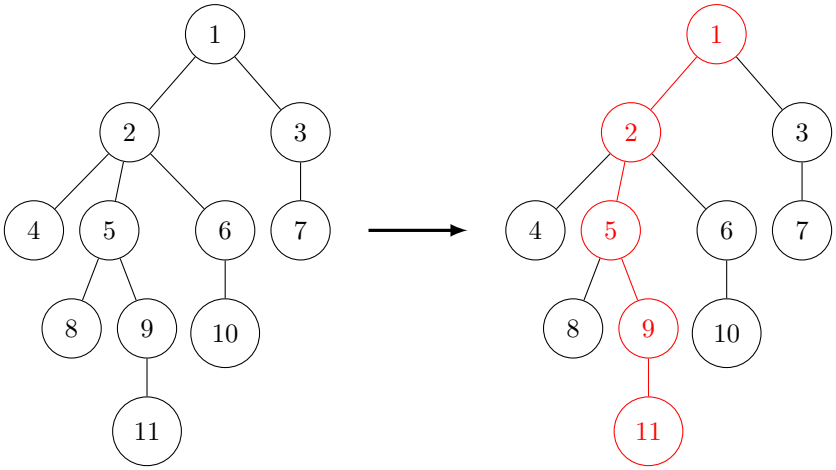
Subtask 1

Motivation for the solution of this Subtask can be gained by visualising `Update 2` on some small hand-cases. Consider the following tree.

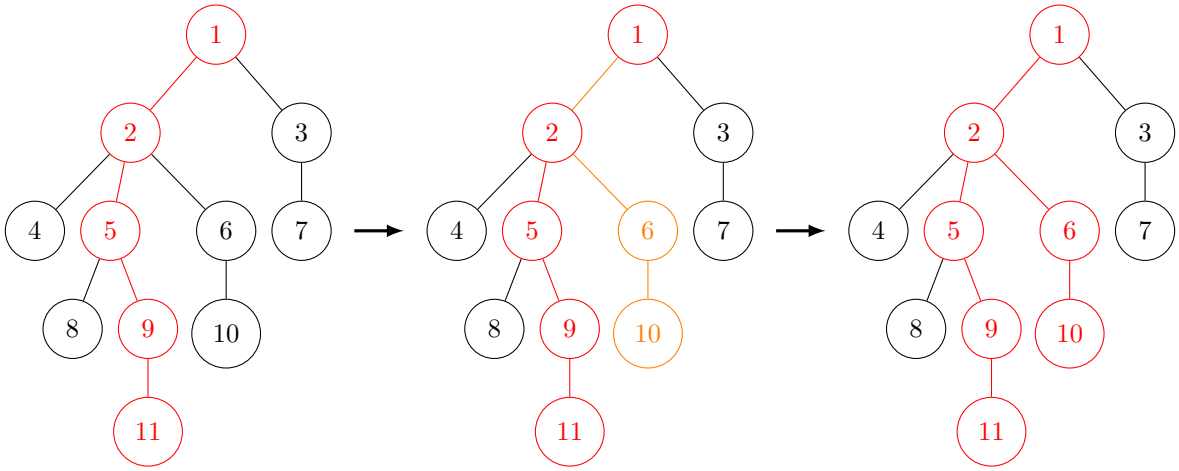


Let us consider `Update 2` only for now, maintaining group-chat information with a Union-find. We will mark nodes merged into node 1's group-chat as red.

Consider `Update 2` on nodes 11 and 1.



Now consider a subsequent `Update 2` on nodes 10 and 1.



Notice that the path from node 10 to node 1 can be split into the path from node 10 to 2, then the path from node 2 to 1. Because node 2 had already been merged into 1 via a previous `Update 2`, there is no point merging that path again. Thus, we can simply stop merging upwards after we reach node 2.

This holds in general for this subtask. Suppose a node is reached that had already been merged with node 1 previously through an `Update 2`. All parents of that node must already have merged with node 1, so we can simply stop merging upward. We can maintain this information using a boolean array `merged`, where `merged[i]` is true if node `i` had already been merged to node 1 via an `Update 2`, and false otherwise. This ensures that each node is merged via an `Update 2` operation at most once, allowing for amortized $O(\alpha(N))$ per operation.

`Update 1` and `Query` is handled on top of this with the same Union-find at $O(\alpha(N))$ per operation.

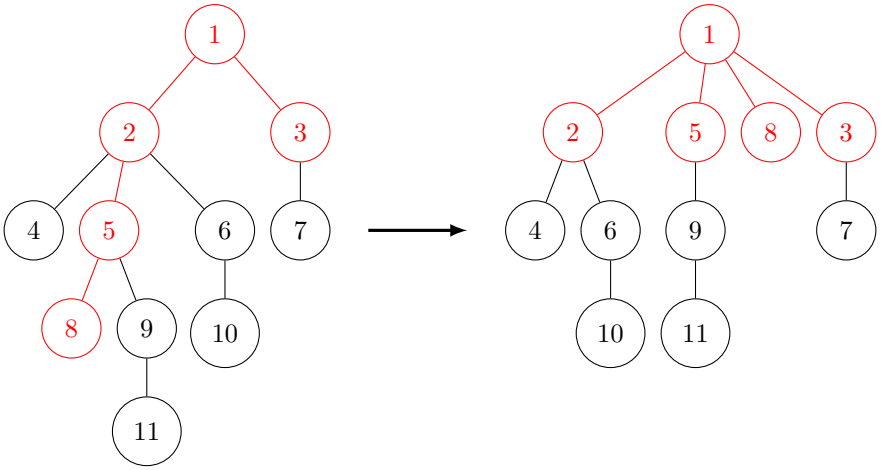
The total complexity of this solution is $O((N + Q)\alpha(N))$, which is sufficient to pass within the given constraints.

Subtask 2

Not all `Update 2` operations are to node 1, so the Subtask 1 solution will no longer work. However, the idea that 'we shouldn't do more work than we need to' still remains present.

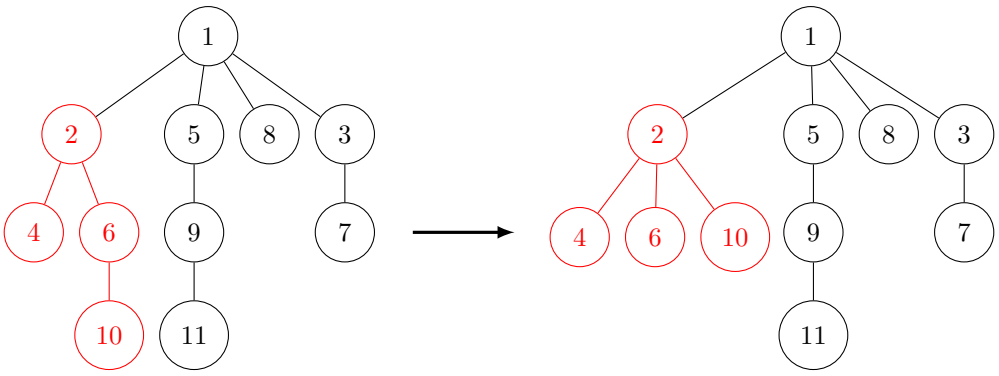
Recall that Union-find uses the path compression optimization, which alone allows for amortized $O(\log N)$ `find/merge` operations. We explore the same path compression idea in the context of this problem. Again, we maintain group-chat information with a Union-find.

Consider the same tree as in Subtask 1, performing `Update 2` on nodes 8 and 3.



After traversing the path from node 8 to node 3 and merging those group chats, we rearrange the parents of all nodes on this path to the *lowest common ancestor* of the endpoints. Note the algorithm to obtain the LCA in $O(\log N)$ is **not** required, as we must still walk up to it to perform the merges — so naively finding the LCA, by walking up the tree, is sufficient.

Consider another `Update 2` on nodes 4 and 10.



We apply the same path compression idea to all nodes on this `Update 2` path, moving the parent of node 10 to be node 2.

If we consider the given tree as another Union-find structure, the whole idea of this algorithm is that each edge is only 'inserted' once by an `Update 2`, where after inserting previously un-inserted edges into the tree, a `find` operation is called on the endpoints. This is virtually identical to a Union-find data structure with only path compression applied, which allows for amortized $O(\log N)$ operations. A merge in the external Union-find is also required to ensure accurate group-chat information, resulting in amortized $O(\alpha(N) \log N)$ per `Update 2`.

`Update 1` and `Query` are still handled on top of this with the same external Union-find that handles `Update 2` group-chat merges, at $O(\alpha(N))$ per operation.

The total complexity of this solution is $O(Q\alpha(N) \log N)$, which will pass comfortably under the time limit with the given constraints.

Implementation Notes

General

- The tree is given as a parent array, and so **no extra** `DFS/BFS` is required to preprocess the tree for later path compression.
- The LCA used in an `Update 2` can be found naively as aforementioned. Suppose `Update 2` was applied on nodes u and v. This can be done by walking u and v up the tree simultaneously, until one walks to a node walked to by the other.

- Some solutions with the idea of path compression received a `Memory Limit Exceeded` verdict due to inefficient handling.
  - The lowered 128MB memory limit was to catch unintended $O(N^2)$ solutions that may slip under the time limit, however was more than enough to solve the problem.
  - The reference solution provided uses maximum 6MB.

Reference Solution(s)

```cpp
// solution by Yiheng for subtask 1

#include <bits/stdc++.h>

using namespace std;

// standard union-find
struct UnionFind {
    vector<int> par, sz;
    UnionFind() {}
    UnionFind(int _n) {
        par.resize(_n);
        iota(par.begin(), par.end(), 0);
        sz.resize(_n);
        fill(sz.begin(), sz.end(), 1);
    }
    int find(int x) {
        return par[x] == x ? x : par[x] = find(par[x]);
    }
    void merge(int x, int y) {
        x = find(x);
        y = find(y);
        if (x != y) {
            if (sz[x] < sz[y]) swap(x, y);
            par[y] = x;
            sz[x] += sz[y];
        }
    }
};

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q; cin >> n >> q;
    vector<int> par(n + 1, 0);
    for (int i = 2; i <= n; i++) {
        cin >> par[i];
    }
    UnionFind uf(n + 1);

    // array to check whether a node has already merged to root via an update 2
    vector<bool> done(n + 1, false);
    while (q--) {
        int op, u, v; cin >> op >> u >> v;
        if (op == 1) {
            // if update 1, merge in union-find naively
            uf.merge(u, v);
        } else if (op == 2) {
            // if update 2, keep merging upwards until reaching a node previously merged
                 via an update 2
            if (u != 1) swap(u, v);
            while (v != 1 && !done[v]) {
                done[v] = true;
                uf.merge(v, 1);
                v = par[v];
            }
        } else if (op == 3) {
            // if query, simply output whether u, v are in the same component
            cout << (uf.find(u) == uf.find(v) ? "YES" : "NO") << "\n";
        }
    }
}
```

```cpp
// solution by Yiheng

#include <bits/stdc++.h>

using namespace std;

// standard union-find
struct UnionFind {
    vector<int> par, sz;
    UnionFind() {}
    UnionFind(int _n) {
        par.resize(_n);
        iota(par.begin(), par.end(), 0);
        sz.resize(_n);
        fill(sz.begin(), sz.end(), 1);
    }
    int find(int x) {
        return par[x] == x ? x : par[x] = find(par[x]);
    }
    void merge(int x, int y) {
        x = find(x);
        y = find(y);
        if (x != y) {
            if (sz[x] < sz[y]) swap(x, y);
            par[y] = x;
            sz[x] += sz[y];
        }
    }
};

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    int n, q; cin >> n >> q;
    vector<int> par(n + 1, 0);
    for (int i = 2; i <= n; i++) cin >> par[i];
    UnionFind uf(n + 1);

    // arrays to assist with finding the LCA
    vector<int> seen_u(n + 1, -1), seen_v(n + 1, -1);
    int idx = 0;
    while (q--) {
        int op, u, v; cin >> op >> u >> v;
        if (op == 1) {
            // if update 1, merge in union-find naively
            uf.merge(u, v);
        } else if (op == 2) {
            // if update 2, find the lca by walking naively up the tree until paths meet
            int tmp_u = u, tmp_v = v;
            int lca = -1;
            while (true) {
                // to prevent needing to clear the seen arrays, store a timer for the
                // current update instead of true/false
                seen_u[tmp_u] = idx;
                seen_v[tmp_v] = idx;
                if (seen_u[tmp_v] == idx) {
                    lca = tmp_v;
                    break;
                }
                if (seen_v[tmp_u] == idx) {
                    lca = tmp_u;
                    break;
                }
                if (tmp_u) tmp_u = par[tmp_u];
                if (tmp_v) tmp_v = par[tmp_v];
            }
            idx++;
            tmp_u = u, tmp_v = v;

            // merge nodes to the lca
            while (tmp_u != lca) {
                uf.merge(tmp_u, lca);
                int nxt = par[tmp_u];
```

```
                // redirect parent pointers of all nodes on path to lca
                par[tmp_u] = lca;
                tmp_u = nxt;
            }

            // merge nodes to the lca
            while (tmp_v != lca) {
                uf.merge(tmp_v, lca);
                int nxt = par[tmp_v];

                // redirect parent pointers of all nodes on path to lca
                par[tmp_v] = lca;
                tmp_v = nxt;
            }
        } else if (op == 3) {
            // if query simply output whether u, v are in the same component
            cout << (uf.find(u) == uf.find(v) ? "YES" : "NO") << "\n";
        }
    }
}
```