Contest 3 Editorial

COMP4128 23T3

20th November 2023

A1. Honey Heist (subtask)

Algorithm

We can model the grid as a graph, with vertices representing the cells and edges representing the key presses. The problem requires us to find the shortest path from the starting cell to (or rather, through) the ending cell. Since the edges are unweighted, this problem is solved using breadth-first search.

Since n is up to $1,000$, we have $V \leq n^2$ and so we can store the entire vertex set as a 2D array. Each vertex has up to four outgoing edges, so $E \leq 4n^2$. We can precompute all edges in $O(n^2)$.

Finally, the BFS also takes $O(V + E) = O(n^2)$ time, so the algorithm runs well within time for $n \leq 1,000$.

Implementation Notes

General

- We strongly recommend keeping the 1-based indexing from the question, as it allows us to deal with the boundary conditions by explicitly placing rocks in rows and columns 0 and $n + 1$.
- At every step of the BFS, we must check whether the bear encounters the honey not only at the endpoint of the move, but at any point during the move.
- Precomputing edges requires us to find the next rock from each cell in each of the four directions. For example, to find the rock left of each cell, we can traverse each row of the grid from left to right, maintaining the column of the most recently encountered rock (initially zero). If the next cell is not a rock, we can record that its left move is blocked by the most recent rock; otherwise we should update the most recent rock.
- We can instead find the edges on the fly. At each step of the BFS, scan cell-by-cell in each of the four directions until a rock is found. This is a bit slower - the worst case is $\Omega(\frac{n^3}{\log n})$ - but still runs within the time limit.
- It's hard to avoid copying and pasting code for each of the four directions. Be particularly careful to edit any similar-looking variable names and offsets of ± 1.

Reference Solutions

```cpp
// Solution by Raveen, precomputing edges

#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
using namespace std;

typedef pair<int,int> ii;
typedef pair<ii,int> state;

const int N = 1010;
bool a[N][N], seen[N][N];
// a[i][j] is false if rock, true if no rock
```

```
int l[N][N], r[N][N], u[N][N], d[N][N];

int main (void) {
        int n, z;
        cin >> n >> z;
        int si, sj, ti, tj;
        cin >> si >> sj >> ti >> tj;

        for (int i = 0; i <= n+1; i++)
                for (int j = 0; j <= n+1; j++)
                        a[i][j] = (i >= 1 && i <= n && j >= 1 && j <= n); // rocks on
                            boundary

        for (int k = 0; k < z; k++) {
                int i, j;
                cin >> i >> j;
                a[i][j] = false; // rocks in interior
        }

        // left arrow
        for (int i = 1; i <= n; i++) { // in each row
                int v = 0;
                for (int j = 1; j <= n; j++) // scan left to right
                        if (a[i][j])
                                l[i][j] = v;
                        else
                                v = j;
        }

        // right arrow
        for (int i = 1; i <= n; i++) { // in each row
                int v = n+1;
                for (int j = n; j >= 1; j--) // scan right to left
                        if (a[i][j])
                                r[i][j] = v;
                        else
                                v = j;
        }

        // up arrow
        for (int j = 1; j <= n; j++) { // in each column
                int v = 0;
                for (int i = 1; i <= n; i++) // scan top to bottom
                        if (a[i][j])
                                u[i][j] = v;
                        else
                                v = i;
        }

        // down arrow
        for (int j = 1; j <= n; j++) { // in each column
                int v = n+1;
                for (int i = n; i >= 1; i--) // scan bottom to top
                        if (a[i][j])
                                d[i][j] = v;
                        else
                                v = i;
        }

        queue<state> q;
        q.emplace(ii(si,sj),0);
        seen[si][sj] = true;
        while(!q.empty()) {
                ii cur = q.front().first; // current cell
                int dis = q.front().second; // moves so far
                q.pop();
                int ci = cur.first, cj = cur.second;

                // left arrow
                int lj = l[ci][cj]+1;
                if (ti == ci && tj >= lj && tj < cj) {
                        cout << dis + 1 << '\n';
                        return 0;
```

```
                }
                if (!seen[ci][lj]) {
                        seen[ci][lj] = true;
                        q.emplace(ii(ci,lj),dis+1);
                }

                // right arrow
                int rj = r[ci][cj]-1;
                if (ti == ci && tj > cj && tj <= rj) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                if (!seen[ci][rj]) {
                        seen[ci][rj] = true;
                        q.emplace(ii(ci,rj),dis+1);
                }

                // up arrow
                int ui = u[ci][cj]+1;
                if (tj == cj && ti >= ui && ti < ci) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                if (!seen[ui][cj]) {
                        seen[ui][cj] = true;
                        q.emplace(ii(ui,cj),dis+1);
                }

                // down arrow
                int di = d[ci][cj]-1;
                if (tj == cj && ti > ci && ti <= di) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                if (!seen[di][cj]) {
                        seen[di][cj] = true;
                        q.emplace(ii(di,cj),dis+1);
                }
        }

        cout << "0\n"; // only get here if not found
}
```

```
// Solution by Raveen, computing edges on the fly

#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
#include <vector>
using namespace std;

typedef pair<int,int> ii;
typedef pair<ii,int> state;

const int N = 1010;
bool a[N][N], seen[N][N];
// a[i][j] is false if rock, true if no rock

int main (void) {
        int n, z;
        cin >> n >> z;
        int si, sj, ti, tj;
        cin >> si >> sj >> ti >> tj;

        for (int i = 0; i <= n+1; i++)
                for (int j = 0; j <= n+1; j++)
                        a[i][j] = (i >= 1 && i <= n && j >= 1 && j <= n); // rocks on
                            boundary

        for (int k = 0; k < z; k++) {
                int i, j;
                cin >> i >> j;
```

```
                a[i][j] = false; // rocks in interior
        }

        queue<state> q;
        q.emplace(ii(si,sj),0);
        seen[si][sj] = true;
        while(!q.empty()) {
                ii cur = q.front().first; // current cell
                int dis = q.front().second; // moves so far
                q.pop();
                int ci = cur.first, cj = cur.second;

                // left arrow
                int lj = cj;
                while (a[ci][lj-1]) { // walk left until rock
                        lj--;
                        if (ti == ci && tj == lj) { // check honey
                                cout << dis + 1 << '\n';
                                return 0;
                        }
                }
                if (!seen[ci][lj]) {
                        seen[ci][lj] = true;
                        q.emplace(ii(ci,lj),dis+1);
                }

                // right arrow
                int rj = cj;
                while (a[ci][rj+1]) {
                        rj++;
                        if (ti == ci && tj == rj) {
                                cout << dis + 1 << '\n';
                                return 0;
                        }
                }
                if (!seen[ci][rj]) {
                        seen[ci][rj] = true;
                        q.emplace(ii(ci,rj),dis+1);
                }

                // up arrow
                int ui = ci;
                while (a[ui-1][cj]) {
                        ui--;
                        if (ti == ui && tj == cj) {
                                cout << dis + 1 << '\n';
                                return 0;
                        }
                }
                if (!seen[ui][cj]) {
                        seen[ui][cj] = true;
                        q.emplace(ii(ui,cj),dis+1);
                }

                // down arrow
                int di = ci;
                while (a[di+1][cj]) {
                        di++;
                        if (ti == di && tj == cj) {
                                cout << dis + 1 << '\n';
                                return 0;
                        }
                }
                if (!seen[di][cj]) {
                        seen[di][cj] = true;
                        q.emplace(ii(di,cj),dis+1);
                }
        }

        cout << "0\n"; // only get here if not found
}
```

A2. Honey Heist (full)

Algorithm

Now that the grid is up to $100,000$ by $100,000$, there are 10^{10} possible vertices. Or are there?

The key observation is that the only relevant cells are the locations of the bear and honey as well as the four cells adjacent to each rock (and the one cell adjacent to each border rock). It is impossible to reach any other cell in the grid. This lowers the number of encountered vertices to $4n + 4r + 2 = 800,002$, making a BFS viable again.

However, it still remains to represent the vertices and edges within the time and memory constraints. For this, we recall the related problem *Tower Power* from Contest 2! The similarity was very much intended by the course staff.

As in that problem, the solution is to maintain data structures representing the rocks in each row and in each column. These data structures will have to support querying the next rock before and after a given value, i.e. the upper bound and its predecessor. We can use a `std::set`, as in Tower Power, but since all rocks are specified before any queries, we could instead use a sorted `std::vector`.

With either data structure, we can query the four edges from a given vertex in $O(\log n)$ time. This gives a time complexity of $O((4n + r) \log n)$, which is sufficient for $n, r \leq 100,000$.

Implementation Notes

General

- Since no two rocks coincide, `upper_bound()` and `lower_bound()` are equivalent.
- `prev()` returns the predecessor of the given iterator.
- Recall that the `upper_bound()` function in `<algorithm>` runs in logarithmic time on sorted ranges, whereas for a `std::set` we should use the class functions of the same name (as detailed on the tips page).
- Usually one would use a seen array to record which states have been encountered in a BFS. Although the number of vertices in the graph is sufficiently small, they cannot be neatly organised into arrays. Therefore a suitable alternative is a `std::set` of seen cells, especially since `set` operations occur a constant number of times per cell encountered in the BFS, so the complexity is unchanged.

Reference Solutions

```
// Solution by Raveen, using sets

#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
using namespace std;

typedef pair<int,int> ii;
typedef pair<ii,int> state;

const int N = 100100;
set<int> row[N], col[N];

int main (void) {
        int n, z;
        cin >> n >> z;
        int si, sj, ti, tj;
        cin >> si >> sj >> ti >> tj;

        // borders
        for (int i = 1; i <= n; i++)
                row[i].insert(0);
        for (int j = 1; j <= n; j++)
                col[j].insert(0);
        for (int i = 1; i <= n; i++)
```

```
                row[i].insert(n+1);
        for (int j = 1; j <= n; j++)
                col[j].insert(n+1);

        // interior rocks
        for (int k = 0; k < z; k++) {
                int i, j;
                cin >> i >> j;
                row[i].insert(j);
                col[j].insert(i);
        }

        queue<state> q;
        q.emplace(ii(si,sj),0);
        set<ii> seen;
        seen.emplace(si,sj);
        while(!q.empty()) {
                ii cur = q.front().first; // current cell
                int dis = q.front().second; // moves so far
                q.pop();
                int ci = cur.first, cj = cur.second;

                // left arrow
                int lj = *prev(row[ci].upper_bound(cj))+1;
                if (ti == ci && tj >= lj && tj < cj) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                ii l(ci,lj);
                if (!seen.count(l)) {
                        seen.insert(l);
                        q.emplace(l,dis+1);
                }

                // right arrow
                int rj = *row[ci].upper_bound(cj)-1;
                if (ti == ci && tj > cj && tj <= rj) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                ii r(ci,rj);
                if (!seen.count(r)) {
                        seen.insert(r);
                        q.emplace(r,dis+1);
                }

                // up arrow
                int ui = *prev(col[cj].upper_bound(ci))+1;
                if (tj == cj && ti >= ui && ti < ci) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                ii u(ui,cj);
                if (!seen.count(u)) {
                        seen.insert(u);
                        q.emplace(u,dis+1);
                }

                // down arrow
                int di = *col[cj].upper_bound(ci)-1;
                if (tj == cj && ti > ci && ti <= di) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                ii d(di,cj);
                if (!seen.count(d)) {
                        seen.insert(d);
                        q.emplace(d,dis+1);
                }
        }

        cout << "0\n"; // only get here if not found
}
```

```cpp
// Solution by Raveen, using sorted vectors

#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
#include <vector>
using namespace std;

typedef pair<int,int> ii;
typedef pair<ii,int> state;

const int N = 100100;
vector<int> row[N], col[N];

int main (void) {
        int n, z;
        cin >> n >> z;
        int si, sj, ti, tj;
        cin >> si >> sj >> ti >> tj;

        // borders
        for (int i = 1; i <= n; i++)
                row[i].push_back(0);
        for (int j = 1; j <= n; j++)
                col[j].push_back(0);
        for (int i = 1; i <= n; i++)
                row[i].push_back(n+1);
        for (int j = 1; j <= n; j++)
                col[j].push_back(n+1);

        // interior rocks
        for (int k = 0; k < z; k++) {
                int i, j;
                cin >> i >> j;
                row[i].push_back(j);
                col[j].push_back(i);
        }

        // sort, to prepare for upper_bound queries
        for (int i = 1; i <= n; i++)
                sort(row[i].begin(),row[i].end());
        for (int j = 1; j <= n; j++)
                sort(col[j].begin(), col[j].end());

        queue<state> q;
        q.emplace(ii(si,sj),0);
        set<ii> seen;
        seen.emplace(si,sj);
        while(!q.empty()) {
                ii cur = q.front().first; // current cell
                int dis = q.front().second; // moves so far
                q.pop();
                int ci = cur.first, cj = cur.second;

                // left arrow
                int lj = *prev(upper_bound(row[ci].begin(),row[ci].end(),cj))+1;
                if (ti == ci && tj >= lj && tj < cj) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                ii l(ci,lj);
                if (!seen.count(l)) {
                        seen.insert(l);
                        q.emplace(l,dis+1);
                }

                // right arrow
                int rj = *upper_bound(row[ci].begin(),row[ci].end(),cj)-1;
                if (ti == ci && tj > cj && tj <= rj) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
```

```
                ii r(ci,rj);
                if (!seen.count(r)) {
                        seen.insert(r);
                        q.emplace(r,dis+1);
                }

                // up arrow
                int ui = *prev(upper_bound(col[cj].begin(),col[cj].end(),ci))+1;
                if (tj == cj && ti >= ui && ti < ci) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                ii u(ui,cj);
                if (!seen.count(u)) {
                        seen.insert(u);
                        q.emplace(u,dis+1);
                }

                // down arrow
                int di = *upper_bound(col[cj].begin(),col[cj].end(),ci)-1;
                if (tj == cj && ti > ci && ti <= di) {
                        cout << dis + 1 << '\n';
                        return 0;
                }
                ii d(di,cj);
                if (!seen.count(d)) {
                        seen.insert(d);
                        q.emplace(d,dis+1);
                }
        }

        cout << "0\n"; // only get here if not found
}
```

B1. Garden (subtask)

Throughout the editorial, we'll refer to "types of special flower" as "colours" for brevity.

Algorithm

First note that the restriction that only n colours are available did not actually need to be given explicitly. You can colour along diagonals in such a way that even an entirely fertile grid can be filled with n colours, without needing any regular flowers.

Let's try to solve for each possible number of colours. For the subtask, this is at most five cases, of which the first and last are trivial. In each case, we proceed to fill in the grid using brute force with backtracking. The running time is improved by immediately pruning any illegal assignments.

As usual, it is hard to concretely estimate the running time of a backtracking algorithm. However, we can try up to four choices for each cell (regular or each of the three colours), and there are sixteen cells. This immediately gives an upper bound of 2^{32}, and it is evident that pruning achieves very large savings: at just the second cell we can discard three sixteenths of these possibilities. In practice, fewer than ten million states are explored in total, so this runs fast enough.

Implementation Notes

General

- Take inspiration from the n queens lecture example.
  - Keep your tentative assignment (or at least the relevant information about it) in global variables.
  - When you make an assignment, edit the global variable, recurse, and then undo that edit.
- Use abstraction! Write helper functions such as "next cell to be filled".

Reference Solution

```
// Solution by Yifan

#include <algorithm>
#include <cstdio>
#include <utility>
using namespace std;

const int N = 5;

int n, r, s;
char garden[N][N];
bool row[N][N], col[N][N]; // row[i][c] = does type c appear in row i yet?
int cnt = 0; // num regular flowers used so far

int ans = 0;

pair<int, int> nxt(int x, int y) {
    if (x == n && y == n) return make_pair(-1, -1); // end
    if (y == n) return make_pair(x + 1, 1); // next row
    return make_pair(x, y + 1); // next cell in this row
}

// fill from (x,y) in lexicographic order, using k colours
void brute(int k, int x, int y) {
    // already done, update answer
    if (x == -1) {
        ans = min(ans, k * s + r * cnt);
        return;
    }

    // already worse than best known answer, early exit
    if (k * s + r * cnt > ans)
        return;
```

```
    // lexicographically next cell
    auto nt = nxt(x, y);
    int nx = nt.first, ny = nt.second;

    // nothing to do if infertile
    if (garden[x][y] == '.') {
        brute(k, nx, ny);
        return;
    }

    // try each type c of special flower here
    for (int c = 1; c <= k; ++c)
        if (!row[x][c] && !col[y][c]) { // if this type doesn't already appear in this
            row and this column
            row[x][c] = col[y][c] = 1; // mark it
            brute(k, nx, ny); // recurse
            row[x][c] = col[y][c] = 0; // unmark it
        }

    // try a regular flower here
    cnt++;
    brute(k, nx, ny); // recurse
    cnt--;
}

int main() {
    scanf("%d%d%d", &n, &r, &s);
    for (int i = 1; i <= n; ++i) {
        scanf("%s", garden[i] + 1); // occupy garden[i][1..n] inclusive
        for (int j = 1; j <= n; ++j) // calculate cost with all regular flowers
            if (garden[i][j] == '*')
                ans += r;
    }

    ans = min(ans, s * n); // always solvable with n types of special flowers only

    for (int k = 1; k < n; ++k) // try solving with k types of special flower for k =
        1..(n-1)
        brute(k, 1, 1);

    printf("%d\n", ans);
}
```

B2. Garden (full)

Algorithm

If we decide to use k colours, we may as well use as many special flowers as possible, as it will only be the regular flowers that we have to pay for individually. Therefore the problem reduces to: what is the greatest number of * cells that can be filled, using a fixed number of colours? Note the small constraints, which make it practical to consider solving this for every number of colours.

Seeing as there is no natural place to start assigning colours, nor is there any order in which to apply dynamic programming, we should consider network flow. A standard construction for grids is to make a bipartite graph with vertices for each row and column. We can add a source vertex to the rows with capacity k and the columns to the sink likewise, and an edge of capacity 1 for each fertile cell, connecting its row to its column. The maximum flow in this graph is the maximum number of fertile cells that can be filled with no more than k in any row or column.

There is unfortunately an extra step of proof required to decompose this into k colours. Consider a graph with vertices corresponding to the rows and columns, and edges corresponding to the flowed row-column edges in the flow network. This graph is clearly bipartite, and no vertex has degree greater than k. It can then be proven that its edges can be partitioned into k matchings. See this Stack Exchange comment for an explanation of the proof provided in *A Textbook of Graph Theory* by Balakrishnan and Ranganathan, page 160.

The graph has $2n + 2$ vertices and up to $n^2 + 2n$ edges, so Dinic's algorithm runs in $O(n^4)$. Repeating this n times gives $O(n^5)$, which is sufficient for $n \leq 50$. A further linear factor can be saved by updating the flow network on each round, rather than starting from scratch; simply increase the capacity of each edge from the source and each edge to the sink and continue finding augmenting paths.

We set the time limit rather generously in this problem. Both Edmonds-Karp and even Ford-Fulkerson were fast enough alternatives to Dinic.

Implementation Notes

General

Nil.

Reference Solution

```
// Solution by Cameron

#include <algorithm>
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

typedef long long ll;

const ll INF = (ll)1<<50;

char garden[51][51];

struct FlowNetwork {
    int n;
    vector<vector<ll>> adjMat, adjList;
    // level[v] stores dist from spc to v
    // uptochild[v] stores first non-useless child.
    vector<int> level, uptochild;

    FlowNetwork (int _n): n(_n) {
        // adjacency matrix is zero-initialised
        adjMat.resize(n);
        for (int i = 0; i < n; i++)
            adjMat[i].resize(n);
        adjList.resize(n);
        level.resize(n);
```

```
        uptochild.resize(n);
    }

    void add_edge (int u, int v, ll c) {
        // add to any existing edge without overwriting
        adjMat[u][v] += c;
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }

    void flow_edge (int u, int v, ll c) {
        adjMat[u][v] -= c;
        adjMat[v][u] += c;
    }

    // constructs the BFS tree and returns whether the sink is still reachable
    bool bfs(int spc, int t) {
        fill(level.begin(), level.end(), -1);
        queue<int> q;
        q.push(spc);
        level[spc] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            uptochild[u] = 0; // reset uptochild
            for (int v : adjList[u])
                if (adjMat[u][v] > 0) {
                    if (level[v] != -1) // already seen
                        continue;
                    level[v] = level[u] + 1;
                    q.push(v);
                }
        }
        return level[t] != -1;
    }

    // finds an augmenting path with up to f flow.
    ll augment(int u, int t, ll f) {
        if (u == t) return f; // base case.
        // note the reference here! we increment uptochild[u], i.e. walk through u'spc
            neighbours
        // until we find a child that we can flow through
        for (int &i = uptochild[u]; i < adjList[u].size(); i++) {
            int v = adjList[u][i];
            if (adjMat[u][v] > 0) {
                // ignore edges not in the BFS tree.
                if (level[v] != level[u] + 1) continue;
                // revised flow is constrained also by this edge
                ll rf = augment(v,t,min(f, adjMat[u][v]));
                // found a child we can flow through!
                if (rf > 0) {
                    flow_edge(u,v,rf);
                    return rf;
                }
            }
        }
        level[u] = -1;
        return 0;
    }

    ll dinic(int spc, int t) {
        ll res = 0;
        while (bfs(spc,t))
            for (ll x = augment(spc,t,INF); x; x = augment(spc,t,INF))
                res += x;
        return res;
    }
};

int main() {
    int n, reg, spc;
    cin >> n >> reg >> spc;

    int fertile = 0;
```

```
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            cin >> garden[i][j];
            if (garden[i][j] == '*')
                fertile++;
        }
    }

    ll ans = INF;

    for (int types = 0; types <= n; types++) {
        struct FlowNetwork flow(2 * n + 2);
        int s = 0, t = 2 * n + 1;

        // vertices 1 to n represent rows
        for (int i = 1; i <= n; i++)
            flow.add_edge(s, i, types);
        // vertices n+1 to 2n represent columns
        for (int j = 1; j <= n; j++)
            flow.add_edge(n + j, t, types);
        // edge for each fertile cell
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                if (garden[i][j] == '*')
                    flow.add_edge(i, n + j, 1);

        // every special flower assigned is a regular flower saved
        ll saved = flow.dinic(s, t);
        ans = min(ans, (fertile - saved) * reg + types * spc);
    }

    cout << ans << '\n';
}
```

C1. Medusa's Snake (subtask)

Algorithm

For $n, q \leq 1,000$, we only need to answer each query in sub-quadratic time. Therefore we can start by thinking about the simplified problem with a single string and no edits, with the task of reporting the venom level only.

The key observation is that any string containing the level k snake also contains the level $k-1$ snake, i.e. we have monotonicity. This allows us to binary search for the answer.

To test whether a level k snake is a subsequence of the given string, we can simply greedily take the first k copies of `S`, then the next k copies of `N`, and so on. To confirm this, see that there can be no advantage in skipping over a letter only to take copies of it even further into the string.

We can now test whether the string has venom at least k in $O(n)$, and hence find the venom level of the string in $O(n \log n)$ time. Answering each edit independently takes $O(nq \log n)$, which is fast enough.

The time complexity can be improved to $O(n \log n + nq)$ with the observation that an edit can only increase or decrease the venom level by one or keep it the same. We still have to run discrete binary search for the initial venom level, but subsequent edits can be answered with at most two runs of the greedy algorithm above.

Implementation Notes

General

- Use binary search code from the Problem Solving Paradigms lecture to avoid bugs.
- $n/5$ is a safe upper bound, as there cannot be more than $\lfloor n/5 \rfloor$ copies of all five letters.
- Make sure you set up a convenient way to convert between the letters `SNAKE` and the corresponding indices. You could use an explicit `if ... else` chain or `switch` statement, or equivalently search a constant string.

Reference Solutions

```
// Solution by Raveen, solving each edit independently

#include <iostream>
#include <string>
using namespace std;

const int N = 100100;
string s;
string snake = "SNAKE";
int n;

void update(int i, char c) {
    s[i] = c;
}

bool canDo(int v) { // greedy
    if (v == 0) return true;
    int cnt = 0;
    for (int i = 0; i < n; i++)
        if (s[i] == snake[cnt/v])
            cnt++;
    return cnt == 5*v;
}

int query(void) { // discrete binary search
    int lo = 0, hi = n/5, ans = -1;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (canDo(mid)) {
            ans = mid;
            lo = mid+1;
```

```
        } else {
            hi = mid-1;
        }
    }
    return ans;
}

int main (void) {
    int q;
    cin >> n >> q;
    cin >> s;
    for (int j = 0; j < q; j++) {
        int i; char c;
        cin >> i >> c;
        update(i-1,c);
        cout << query() << '\n';
    }
}
```

```
// Solution by Raveen, updating the level

#include <iostream>
#include <string>
using namespace std;

const int N = 100100;
string s;
string snake = "SNAKE";
int n, level;

void update(int i, char c) {
    s[i] = c;
}

bool canDo(int v) { // greedy
    if (v == 0) return true;
    int cnt = 0;
    for (int i = 0; i < n; i++)
        if (s[i] == snake[cnt/v])
            cnt++;
    return cnt == 5*v;
}

void update_level(void) { // try +1, try =, default to -1
    if (canDo(level+1))
        level++;
    else if (!canDo(level))
        level--;
}

void find_level(void) { // discrete binary search
    int lo = 0, hi = n/5;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (canDo(mid)) {
            level = mid;
            lo = mid+1;
        } else {
            hi = mid-1;
        }
    }
}

int main (void) {
    int q;
    cin >> n >> q;
    cin >> s;
    find_level();
    for (int j = 0; j < q; j++) {
        int i; char c;
        cin >> i >> c;
        update(i-1,c);
        update_level();
```

```
            cout << level << '\n';
        }
    }
```

C2. Medusa's Snake (full)

Algorithm

As in the subtask, we can safely spend $O(n \log n)$ to answer the first edit, by wrapping a discrete binary search around the greedy algorithm. Each subsequent edit requires us to check whether the level has increased by one, stayed the same, or decreased. However with $n, q \leq 100,000$, we must answer these subsequent edits in sub-linear time, so we have to do better than the earlier greedy.

You might notice some inefficiency in repeating the greedy. We only changed one character in the entire string, so why should we have to walk through the first k copies of `S` one-by-one? Could we instead store where the kth `S` was and jump straight there? Not quite; the position of the kth `S` might change because of the edit we made.

Even so, we should recognise that searching for the kth `S` might take less than linear time. If we could quickly look up how many copies of `S` there are among the first ℓ letters, we could binary search for the position of the kth `S`. Prefix sums seem ideal for this, but there are updates. Instead, we can use a range tree!

Maintain five point-update range-query range trees, one for each letter of `SNAKE`. The leaves are 1 if the string has the corresponding character at that index, or 0 otherwise. Internal nodes then store the sum over their range of responsibility. Updates will increment or decrement the value at a point, and queries will sum a given range.

We can now resolve each edit by:

1. updating the range tree for the letter that used to occupy the position being edited
2. updating the range tree for the new letter in that position
3. testing whether the level has increased by one, then whether it has stayed the same
    - binary search for the first index i where the interval $[0, i]$ contains at least the desired number of letters `S`, which in turn is a range tree query at each step.

The total time complexity is $O(\log^2 n)$ per edit, so $O(n \log n + q \log^2 n)$ overall, which was sufficient.

However, we can do even better by exploiting the structure of the range tree to do the search. As detailed in lecture examples such as First Big Value, we can recurse all the way down to a leaf in $O(\log n)$. This allows us to test a level in $O(\log n)$ and therefore reduces the time complexity to $O((n+q) \log n)$.

In short, the most straightforward range tree solution (as above, but running the discrete binary search for every edit) runs in $O(q \log^3 n)$, which times out, and a solution which saved either (or both) of the two available log factors passed.

Implementation Notes

General

- Be very careful with off-by-ones, especially as you go from searching from one letter to the next.

Reference Solutions

```cpp
// Solution by Raveen, O(n log n + q log^2 n)

#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

const int N = 100100;
int sumrt[5][1<<18];
string s;
string snake = "SNAKE";
int level;
int n;

void updatert(int p, int v, int t, int i = 1, int cL = 0, int cR = n) {
```

```
  if (cR - cL == 1) {
    sumrt[t][i] = v;
    return;
  }
  int mid = (cL + cR) / 2;
  if (p < mid) updatert(p, v, t, i*2, cL, mid);
  else updatert(p, v, t, i*2+1, mid, cR);
  sumrt[t][i] = sumrt[t][i*2] + sumrt[t][i*2+1];
}

void update(int i, char c) {
    updatert(i, 0, snake.find(s[i]));
    s[i] = c;
    updatert(i, 1, snake.find(s[i]));
}

int queryrt(int qL, int qR, int t, int i = 1, int cL = 0, int cR = n) {
  if (cL == qL && cR == qR)
    return sumrt[t][i];
  int mid = (cL + cR) / 2;
  int ans = 0;
  if (qL < mid) ans += queryrt(qL, min(qR, mid), t, i * 2, cL, mid);
  if (qR > mid) ans += queryrt(max(qL, mid), qR, t, i * 2 + 1, mid, cR);
  return ans;
}

bool canDo(int v) {
  if (v == 0) return true;
  int upto = 0;
  for (int t = 0; t < 5; t++) {
    // binary search:
    int lo = upto+1, hi = n, new_upto = 0;
    while (lo <= hi) {
      int mid = (lo+hi)/2;
      // count from when we started looking for this character
      if (queryrt(upto,mid,t) >= v) { // enough
        hi = mid-1;
        new_upto = mid;
      } else // not enough
        lo = mid+1;
    }
    if (new_upto == 0) // didn't find enough
      return false;
    else
      upto = new_upto;
  }
  return true;
}

void update_level(void) {
    if (canDo(level+1))
        level++;
    else if (!canDo(level))
        level--;
}

bool greedy(int v) {
    if (v == 0) return true;
    int cnt = 0;
    for (int i = 0; i < n; i++)
        if (s[i] == snake[cnt/v])
            cnt++;
    return cnt == 5*v;
}

void find_level(void) {
    int lo = 0, hi = n/5;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (greedy(mid)) {
            level = mid;
            lo = mid+1;
        } else {
```

```
            hi = mid-1;
        }
    }
}

int main (void) {
    int q;
    cin >> n >> q;
    cin >> s;
    for (int i = 0; i < n; i++)
        updatert(i, 1, snake.find(s[i]));
    find_level();
    for (int j = 0; j < q; j++) {
        int i; char c;
        cin >> i >> c;
        update(i-1,c);
        update_level();
        cout << level << '\n';
    }
}
```

```
// Solution by Raveen, O((n+q) log n)

#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

const int N = 100100;
int sumrt[5][1<<18];
string s;
string snake = "SNAKE";
int level;
int n;

void updatert(int p, int v, int t, int i = 1, int cL = 0, int cR = n) {
  if (cR - cL == 1) {
    sumrt[t][i] = v;
    return;
  }
  int mid = (cL + cR) / 2;
  if (p < mid) updatert(p, v, t, i*2, cL, mid);
  else updatert(p, v, t, i*2+1, mid, cR);
  sumrt[t][i] = sumrt[t][i*2] + sumrt[t][i*2+1];
}

void update(int i, char c) {
    updatert(i, 0, snake.find(s[i]));
    s[i] = c;
    updatert(i, 1, snake.find(s[i]));
}

bool searchrt(int v, int t, int &upto, int i = 1, int cL = 0, int cR = n) {
  // i's range doesn't have enough 1s
  if (sumrt[t][i] < v) return false;
  if (cR - cL == 1) {
    upto = cL;
    return true;
  }
  int mid = (cL + cR) / 2;
  if (sumrt[t][i*2] >= v)
    return searchrt(v, t, upto, i*2, cL, mid);
  else
    return searchrt(v-sumrt[t][i*2], t, upto, i*2+1, mid, cR);
}

int queryrt(int qL, int qR, int t, int i = 1, int cL = 0, int cR = n) {
  if (cL == qL && cR == qR)
    return sumrt[t][i];
  int mid = (cL + cR) / 2;
  int ans = 0;
  if (qL < mid) ans += queryrt(qL, min(qR, mid), t, i * 2, cL, mid);
  if (qR > mid) ans += queryrt(max(qL, mid), qR, t, i * 2 + 1, mid, cR);
```

```
  return ans;
}

bool canDo(int v) {
  if (v == 0) return true;
  int upto = 0;
  for (int t = 0; t < 5; t++) {
    int ignore = queryrt(0,upto,t);
    if (!searchrt(v+ignore,t,upto))
      return false;
  }
  return true;
}

void update_level(void) {
    if (canDo(level+1))
        level++;
    else if (!canDo(level))
        level--;
}

bool greedy(int v) {
    if (v == 0) return true;
    int cnt = 0;
    for (int i = 0; i < n; i++)
        if (s[i] == snake[cnt/v])
            cnt++;
    return cnt == 5*v;
}

void find_level(void) {
    int lo = 0, hi = n/5;
    while (lo <= hi) {
        int mid = (lo+hi)/2;
        if (greedy(mid)) {
            level = mid;
            lo = mid+1;
        } else {
            hi = mid-1;
        }
    }
}

int main (void) {
    int q;
    cin >> n >> q;
    cin >> s;
    for (int i = 0; i < n; i++)
        updatert(i, 1, snake.find(s[i]));
    find_level();
    for (int j = 0; j < q; j++) {
        int i; char c;
        cin >> i >> c;
        update(i-1,c);
        update_level();
        cout << level << '\n';
    }
}
```