A1. Smuggling (subtask)

Algorithm

It is natural to construct a graph with vertices corresponding to the cities, and edges joining pairs of cities which are distance 1 apart. We need to determine whether there are \( m \) return paths from vertex 1 to vertex \( n \) in this graph, without reusing any intermediate city. Recall from the lecture example “Jumping Frogs” that this is equivalent to finding \( 2m \) vertex-disjoint paths, and can be solved using network flow.

To enforce that no intermediate vertex is used more than once, we apply a vertex capacity of 1 for each vertex other than 1 and \( n \). This is achieved by splitting each of them into an in- and out-vertex, with an edge between them representing the vertex capacity. The source can be connected directly to the out-vertex of city 1, and the sink from the in-vertex of city \( n \).

Finally, we run any maximum flow algorithm and return whether a flow of at least \( 2m \) can be found. Since at most four vertex-disjoint paths are possible, and the number of edges is at most \( 2n \), the time complexity is \( O(n) \).

Implementation Notes

One common source of error was failing to reserve enough space for the graph. Since there are 200 cities, each with an in- and out-vertex, the flow network must have at least 400 vertices.

Remember to construct residual edges, not just ‘forward’ edges.

If using multiplication to test distances, use \texttt{long long} to avoid overflow.

Many students received the TIMELIMIT verdict, indicating an infinite loop. This was often due to errors in the graph construction.

Reference Solution

```c++
// Solution by Raveen
#include <algorithm>
#include <iostream>
#include <queue>
using namespace std;
typedef long long ll;

const int N = 440; // 200 cities, each with an in- and out-vertex
const int M = N*N;
ll n, m;
ll x[N], y[N];

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

// assumes the residual graph is constructed as in the previous section
// n = #nodes, s = source, t = sink
int s, t;
// stores dist from s.
int lvi[N];
// stores first non-useless child.
int nextchild[N];
```
// the index of the first outgoing edge for each vertex, initialised to -1
int start[N];
// if e is an outgoing edge from u, succ[e] is another one, or -1
// cap[e] is the capacity/weight of the e
// to[e] is the destination vertex of e
int succ[M], cap[M], to[M];

int edge_counter = 0;
void add_edge(int u, int v, int c) {
    // set the properties of the new edge
    cap[edge_counter] = c, to[edge_counter] = v;
    // insert this edge at the start of u's linked list of edges
    succ[edge_counter] = start[u];
    start[u] = edge_counter;
    ++edge_counter;
}

// constructs the BFS tree.
// Returns if the sink is still reachable.
bool bfs() {
    fill(lvl, lvl+N, -1);
    queue<int> q;
    q.push(s);
    lvl[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        nextchld[u] = start[u]; // reset nextchld
        for (int e = start[u]; ~e; e = succ[e]) {
            if (cap[e] > 0) {
                int nxt = to[e];
                if (lvl[nxt] != -1) continue; // already seen
                lvl[nxt] = lvl[u] + 1;
                q.push(nxt);
            }
        }
    }
    return lvl[t] != -1;
}

// Finds an augmenting path with up to cflow flow.
int aug(int u, int cflow) {
    if (u == t) return cflow; // base case.
    // Note the reference here! We will keep decreasing nextchld[u]
    // Till we find a child that we can flow through.
    for (int &i = nextchld[u]; i >= 0; i = succ[i]) {
        if (cap[i] > 0) {
            int nxt = to[i];
            // Ignore edges not in the BFS tree.
            if (lvl[nxt] != lvl[u] + 1) continue;
            int rf = aug(nxt, min(cflow, cap[i]));
            // Found a child we can flow through!
            if (rf > 0) {
                cap[i] -= rf;
                cap[i-1] += rf;
                return rf;
            }
        }
        lvl[u]=-1;
        return 0;
    }
}

int mf() {
    int tot = 0;
    while (bfs()) {
        for (int x = aug(s, INF); x = aug(s, INF))
            tot += x;
    }
    return tot;
}

ll canReach(ll x1, ll y1, ll x2, ll y2) {
    return (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) == 1;
}
bool canDo(void) {
    // reset graph
    edge_counter = 0;
    fill(start, start + N, -1);

    // vertex capacities
    for (int i = 1; i < n-1; i++) {
        add_edge(2*i, 2*i+1, 1);
        add_edge(2*i+1, 2*i, 0);
    }

    // flights
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (canReach(x[i], y[i], x[j], y[j])) {
                add_edge(2*i+1, 2*j, 2*m);
                add_edge(2*j, 2*i+1, 0);
            }

    // connect the source to starting city's out-vertex
    add_edge(s, 1, 2*m);
    add_edge(1, s, 0);

    // connect ending city's in-vertex to the sink
    add_edge(2*n-2, t, 2*m);
    add_edge(t, 2*n-2, 0);

    return mf() == 2*m;
}

int main (void) {
    cin >> n >> m;
    s = 2*n;
    t = 2*n+1;

    for (int i = 0; i < n; i++)
        cin >> x[i] >> y[i];

    cout << (canDo() ? "YES\n" : "NO\n");
}
A2. Smuggling (full)

Algorithm

The algorithm for the full problem is a natural extension of that used for the subtask. We need to minimise the longest flight taken, so we can binary search for the answer, much like in “Jumping Frogs”.

At each stage of the binary search, we reconstruct the graph, placing edges between each pair of cities whose distance apart is at most the distance being tested. Then, we test whether a flow of at least $2m$ can be found, and recurse high or low accordingly.

The time complexity is $O(n^3 \log X)$, where $X$ is the upper bound of the binary search (on the order of $10^9$).

Implementation Notes

When overwriting a graph using the edge list representation, we have to take care to reset the necessary variables. In the implementation provided in lectures, these are `edge_counter` and `start`.

Always take care to choose appropriate bounds for a binary search. In this case, two cities at $(1, 1)$ and $(10^9, 10^9)$ are separated by a distance of $(10^9 - 1)\sqrt{2}$. Therefore, students who set the upper bound of the binary search to only $10^9$ received the verdict `WRONG-ANSWER`.

The problem statement specified that the output should be an integer, not a floating-point number. Therefore, the values tested in the binary search should be integers, and integer arithmetic should be used in all the distance calculations to avoid rounding errors. This can be achieved by comparing squared distance rather than distance.

Reference Solution

```cpp
// Solution by Raveen
#include <algorithm>
#include <iostream>
#include <queue>
using namespace std;
typeid long long ll;

const int N = 440; // 200 cities, each with an in- and out-vertex
const int M = N*N;
ll n, m;
ll x[N], y[N];

const ll INF = 1000*1000*1000+7;
// assumes the residual graph is constructed as in the previous section
// n = #nodes, s = source, t = sink
int s, t;
// stores dist from s.
int lvl[N];
// stores first non-useless child.
int nextchild[N];

// the index of the first outgoing edge for each vertex, initialised to -1
int start[N];
// if e is an outgoing edge from u, succ[e] is another one, or -1
// cap[e] is the capacity/weight of the e
// to[e] is the destination vertex of e
int succ[M], cap[M], to[M];

int edge_counter = 0;
void add_edge(int u, int v, int c) {
    // set the properties of the new edge
    cap[edge_counter] = c, to[edge_counter] = v;
    // insert this edge at the start of u's linked list of edges
    succ[edge_counter] = start[u];
    start[u] = edge_counter;
    **edge_counter;}
```
```cpp
bool bfs() {
    fill(lvl, lvl+N, -1);
    queue<int> q;
    q.push(s); lvl[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        nextchld[u] = start[u]; // reset nextchld
        for (int e = start[u]; ~e; e = succ[e]) {
            int nxt = to[e];
            if (lvl[nxt] != -1) continue; // already seen
            lvl[nxt] = lvl[u] + 1;
            q.push(nxt);
        }
    }
    return lvl[t] != -1;
}

// Finds an augmenting path with up to cflow flow.
int aug(int u, int cflow) {
    if (u == t) return cflow; // base case.
    // Note the reference here! We will keep decreasing nextchld[u]
    // Till we find a child that we can flow through.
    for (int &i = nextchld[u]; i >= 0; i = succ[i]) {
        if (cap[i] > 0) {
            int nxt = to[i];
            if (lvl[nxt] != lvl[u] + 1) continue; // ignore edges not in the BFS tree.
            int rf = aug(nxt, min(cflow, cap[i]));
            if (rf > 0) {
                cap[i] -= rf;
                cap[i-'1'] += rf;
                return rf;
            }
        }
    }
    lvl[u]=-1;
    return 0;
}

int mf() {
    int tot = 0;
    while (bfs()) {
        for (int x = aug(s, INF); x = aug(s, INF))
            tot += x;
    }
    return tot;
}

ll sqdist(ll x1, ll y1, ll x2, ll y2) {
    return (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
}

bool canDo(ll maxdist) {
    // reset graph
    edge_counter = 0;
    fill(start, start + N, -1);
    // vertex capacities
    for (int i = 1; i < n-1; i++) {
        add_edge(2*i, 2*i+1, 1);
        add_edge(2*i+1, 2*i, 0);
    }
    // flights
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (sqdist(x[i], y[i], x[j], y[j]) <= maxdist*maxdist)
                add_edge(2*i+1, 2*j, 2*maxdist);
            add_edge(2*j, 2*i+1, 0);
    return false;
}
```
// connect the source to starting city's out-vertex
    add_edge(s, 1, 2*m);
    add_edge(1, s, 0);

// connect ending city's in-vertex to the sink
    add_edge(2*n-2, t, 2*m);
    add_edge(t, 2*n-2, 0);

    return mf() == 2*m;

ll binarysearch(void) {
    ll lo = 1;
    ll hi = INF*3/2; // exceeds max distance between any two cities
    ll bestSoFar = -1;
    // Range [lo, hi];
    while (lo <= hi) {
        ll mid = (lo + hi) / 2;
        if (canDo(mid)) {
            bestSoFar = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return bestSoFar;
}

int main (void) {
    cin >> n >> m;
    s = 2*n;
    t = 2*n+1;
    for (int i = 0; i < n; i++)
        cin >> x[i] >> y[i];
    cout << binarysearch() << '\n';
}
B1. Amenities (subtask)

Algorithm

We have to process range updates and point queries, so the natural data structure to use is a range tree with lazy updates. Each node’s lazy value will be the best upgrade applied to its entire range of responsibility. Then, traversing the path from the root to a leaf will necessarily encounter the best upgrade applied to this station.

The time complexity is \( O((n + q) \log n) \).

Implementation Notes

We are setting lazy values, rather than adding to them as we did in the lecture example. Furthermore, we must be careful not to overwrite a better upgrade with a worse one that appears later.

Be careful in converting between 0-based and 1-based indices.

As discussed in [https://www.cse.unsw.edu.au/cs4128/22t3/range-trees.html](https://www.cse.unsw.edu.au/cs4128/22t3/range-trees.html) make sure to allocate sufficient space for the range tree, namely twice the next power of two.

Reference Solution

```cpp
#include <iostream>
using namespace std;

const int N = 100100;
int lazy[1<<18];

int main(void) {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int a;
        cin >> a;
        update(i, i+1, a);
    }

    int m;
    cin >> m;
    for (int j = 0; j < m; j++) {
        char t;
        cin >> t;
        if (t == 'U') {
            int l, r, x;
            cin >> l >> r >> x;
            update(l, r, x);
        } else {
            int mid = (l + r) / 2;
            if (l < mid) 
                update(l, mid, v, i + 2, mid, mid);
            if (r > mid) 
                update(max(l, mid), r, v, i + 2 + 1, mid, cR);
        }
    }
    for (int i = 0; i < n; i++) {
        int a;
        cin >> a;
        update(i, i+1, a);
    }
}
```
update(l=1,r,x);
} else if (t == 'Q') {
    int l, r;
    cin >> l >> r;
    cout << query(l-1) << '
';
}
Algorithm
We now have range queries as well as range updates, so lazy propagation is required. Recall that:

- the range tree values are dictated by the queries, so we will maintain the maximum in each node’s range of responsibility, and
- the lazy values are dictated by the updates, so we will maintain the best upgrade applied to each node’s entire range of responsibility.

Then the range tree values correctly answer the queries within their own ranges, i.e. assuming that all ancestors have no lazy value, and in order to reach the range tree values we must propagate their ancestors’ lazy updates.

The time complexity is $O((n + q) \log n)$.

Implementation Notes
This problem is very similar to the lecture example “Setting Ranges”, but we must take care to maintain the best upgrade to each range, not the latest. This requires some changes to the update and propagate functions.

Reference Solution
```cpp
#include <iostream>
using namespace std;

const int N = 100100;
const int UNSET = -1;
// Since A is 0 initially, the default values are correct.
int lazyset[1<<18]; // UNSET if no lazy is set
int maxrt[1<<18]; // max in range of responsibility of node i

// Recalculates a node’s values assuming its children are correct.
// do NOT call these on leaves.
void recalculate(int i) {
    if (lazyset[i] != UNSET)
        maxrt[i] = max(maxrt[i], lazyset[i]);
    else
        maxrt[i] = max(maxrt[i], max(maxrt[i*2], maxrt[i*2+1]));
}

void propagate(int i) {
    if (lazyset[i] == UNSET)
        return;
    lazyset[i*2] = max(lazyset[i*2], lazyset[i]);
    lazyset[i*2+1] = max(lazyset[i*2+1], lazyset[i]);
    maxrt[i*2] = max(maxrt[i*2], maxrt[i]);
    maxrt[i*2+1] = max(maxrt[i*2+1], maxrt[i]);
    lazyset[i] = UNSET;
}

int update(int uL, int uR, int v, int i = 1, int cL = 0, int cR = n) {
    if (uL == cL && uR == cR) {
        lazyset[i] = max(lazyset[i], v);
        return;
    }
    propagate(i);
    int mid = (cL + cR) / 2;
    if (uL < mid) update(uL, min(uR, mid), v, i*2, cL, mid);
    if (uR > mid) update(max(uL, mid), uR, v, i*2+1, mid, cR);
    recalculate(i);
    return;
}
```
```cpp
int query(int qL, int qR, int i = 1, int cL = 0, int cR = n) {
    if (qL == cL && qR == cR) {
        return maxrt[i];
    }
    propagate(i);
    int mid = (cL + cR) / 2;
    int ans = -1; // note all values are >= 0 in the question.
    if (qL < mid) ans = max(ans, query(qL, min(qR, mid), i*2, cL, mid));
    if (qR > mid) ans = max(ans, query(max(qL, mid), qR, i*2+1, mid, cR));
    return ans;
}

int main(void) {
    int m;
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        int a;
        cin >> a;
        update(i, i+1, a);
    }
    for (int j = 0; j < m; j++) {
        char t;
        cin >> t;
        if (t == 'U') {
            int l, r, x;
            cin >> l >> r >> x;
            update(l-1, r, x);
        } else if (t == 'Q') {
            int l, r;
            cin >> l >> r;
            cout << query(l-1, r) << '\n';
        }
    }
}
```
C1. Time Travel (subtask)

Algorithm

For the subtask, every city optionally has a start time, but their finish time is unbounded. To solve this problem, we can run a small modification of Dijkstra’s algorithm. Suppose we are at node $u$ with distance $\text{dist}(u)$ and we are considering an outgoing edge $(u, v)$ with weight $w(u, v)$. Ordinarily, we would just push $v$ to the priority queue with a distance of $\text{dist}(u) + w(u, v)$. To account for the start time, we should instead push it with distance

$$\max\{\text{dist}(u) + w(u, v), s_v\}.$$

A detailed proof of correctness is included below. The complexity is $O(m \log n)$.

Proof

Suppose we have a set $S$ of nodes (including the start node) where for each $u \in S$, the shortest distance to $u$ (taking into account start times) has been computed, and denote such distances $\text{dist}(u)$. Now let $v$ be the node outside $S$ which minimises

$$\max\{\text{dist}(u) + w(u, v), s_v\},$$

where $u \in S$ is a neighbour of $v$. Then the claim is that this distance must be the shortest distance to node $v$. If this is true, then we can store this distance as $\text{dist}(v)$, add $v$ to our set, and then repeat the process until we have the shortest distances to every node.

Suppose for a contradiction that there is an alternate path to $v$ with a shorter distance. Consider the last point where this path leaves the set $S$, and let the nodes along the remaining path be $v_1, v_2, \ldots, v_k$ where $v_1 \in S$, $v_2, v_3, \ldots \notin S$ and $v_k = v$. Then all edge weights are non-negative, so

$$\max\{\text{dist}(v_1) + w(v_1, v_2), s_{v_2}\} \leq \text{length of alternate path},$$

and

$$\text{length of alternate path} < \max\{\text{dist}(u) + w(u, v), s_v\}$$

because it is shorter. But we also know that

$$\max\{\text{dist}(u) + w(u, v), s_v\} \leq \max\{\text{dist}(v_1) + w(v_1, v_2), s_{v_1}\}$$

since $v$ was chosen to minimise this quantity. Therefore, we have a contradiction and no such alternate path exists. Hence the distance we have calculated to $v$ is optimal.

Too Slow: SPFA

The Shortest Paths Faster Algorithm is an optimisation of the Bellman-Ford algorithm, which is still $O(VE)$ in the worst case but achieves significant efficiencies in the average case due to early exit strategies. Ordinarily, this should have exceeded the time limit.

For an example of the type of test case which forces this worst case behaviour, suppose:

- you have $n$ nodes in the line
- there are $n - 1$ edge of the form $(i, i + 1)$ with weight 1, and
- there are an additional $n - 1$ edges of the form $(1, i)$ with weight $2i$.

Without loss of generality, assume SPFA processes the outgoing edges of every node from the highest node ID to the smallest. Then the algorithm will relax nodes $n$ to 1, then nodes $n - 1$ to 1, then $n - 1$ to 1 and so on, giving a complexity of $O(n^2)$.

Unfortunately, we neglected to include such test cases in designing the problem, and as a result some SPFA solutions were accepted. Efforts will be taken to strengthen test cases for the final exam.
Implementation Notes

Make sure you are using `long long`. Edge weights can go up to $10^9$, and the longest path could potentially span all $10^5$ edges, giving us a maximum path length of $10^{14}$.

An alternative method of implementation is just pushing distances into the priority queue without considering the start time of a town, and then accounting for it only after pulling off the priority queue. It can be shown via a similar proof that this method also gives correct results.

Reference Solutions

```cpp
// Solution by Kevin
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <vector>
#include <utility>
#include <queue>
#include <climits>

using namespace std;

typedef long long ll;
typedef pair<int, int> pii;
define x first
define y second

define MAXN 100005
#define MAXM 100005

struct Edge {
    ll to, t;
    bool operator<(const Edge &oth) const {
        return t > oth.t;
    }
};

ll s[MAXN], f[MAXN];
vector<Edge> G[MAXN];
ll dist[MAXN];

int main() {
    int N, M;
    cin >> N >> M;
    for (int i = 0; i < N; i++) cin >> s[i] >> f[i];
    for (int i = 0; i < M; i++) {
        ll a, b, t;
        cin >> a >> b >> t;
        G[a-1].push_back({b-1, t});
        G[b-1].push_back({a-1, t});
    }

    // run dijkstra
    fill(dist, dist+MAXN, LLONG_MAX);
    priority_queue<Edge> pq;
    pq.push({0, 0});
    while (!pq.empty()) {
        Edge cur = pq.top();
        pq.pop();
        if (dist[cur.to] != LLONG_MAX) continue;
        dist[cur.to] = cur.t;
        for (Edge nxt : G[cur.to]) {
            if (dist[nxt.to] != LLONG_MAX) continue;
            pq.push({nxt.to, max(cur.t + nxt.t, s[nxt.to])});
        }
    }
    printf("%lld\n", dist[N-1]);
}
```
Algorithm

It may be tempting to adapt Dijkstra’s algorithm in much the same way as we did for the subtask. Suppose we account for the finishing times by adding nodes into our priority queue with distance

\[ \min \{ \max \{ \text{dist}(u) + w(u, v), s_v \}, f_v \} \].

This doesn’t work, because it fails to account for the fact that you can revisit towns with lower times after time travelling.

Consider the following example.

The tuples next to each node represent their start and finish time. Our algorithm will explore the nodes 1, 2, 3 and 4 in that order, and declare their distances to be 0, 100, 1 and 101 respectively. However, we can see that by going on the path 1 → 2 → 3 → 2 → 4, we end up with a final distance of 2.

To handle the finishing times, we notice that the finishing times have the effect of limiting the shortest distance to town \( i \) to no more than \( f_i \). We can cleverly encode this constraint directly into the graph by creating edges! For each town where \( f_i \neq 0 \), construct an edge from 1 to \( i \) with weight \( f_i \). We then run the same modified Dijkstra from the subtask, using the start times of each town. Note that we will never find a distance greater than \( f_i \) to node \( i \), since it would always be faster just to travel along the newly added edge instead. The complexity is \( O((n + m) \log m) \).

Proof

First, add the new edges into the graph. Let \( S \) be the same set as before and once again let \( v \) be the node outside \( S \) minimising

\[ \max \{ \text{dist}(u) + w(u, v), s_v \} \],

where \( u \in S \) is a neighbour of \( v \). As before, if we can show this distance represents the shortest distance to \( v \), then our algorithm works correctly.

Suppose there is an alternate path to \( v \) with a shorter distance and label nodes \( v_1, v_2, \ldots, v_k \) as before. There are two cases:

- **Case 1:** The path from \( v_1, \ldots, v_k \) does not require Tim to travel back in time. Since we are only using starting times, we can use the same proof from before to show that this leads to a contradiction.
- **Case 2:** The path from \( v_1, \ldots, v_k \) takes advantage of time travel to shorten the distance. Then let \( v_i \) be the last node which takes advantage of the finish time on the path \( v_1 \) to \( v_k \) to shorten the path so that

\[ f_{v_i} \leq \text{length of alternate path} \].

Of course, since the alternate path is shorter, we also have that

\[ \text{length of alternate path} < \max \{ \text{dist}(u) + w(u, v), s_v \} \]
and finally, because the edge \((u, v)\) was chosen over any other edge including the new edge from the start to \(v\), we have that
\[
\max \{\text{dist}(u) + w(u, v), s_v\} \leq f_i.
\]

Clearly this forms an impossible cycle of inequalities, and thus we conclude that this alternate path cannot exist.

Too Slow: SPFA

As with the subtask, running SPFA was accepted due to weak test cases, but ordinarily should have timed out for the same reasons.

Implementation Notes

As above.

Reference Solutions

```cpp
// Solution by Kevin
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <vector>
#include <utility>
#include <queue>
#include <climits>
using namespace std;

typedef long long ll;
typedef pair<int, int> pii;
define x first
#define y second

struct Edge {
    ll to, t;
    bool operator<(const Edge &oth) const {
        return t > oth.t;
    }
};

ll s[MAXN], f[MAXN];
vector<Edge> G[MAXN];
ll dist[MAXN];

int main () {
    int N, M;
    cin >> N >> M;
    for (int i = 0; i < N; i++) cin >> s[i] >> f[i];
    for (int i = 0; i < M; i++) {
        ll a, b, t;
        cin >> a >> b >> t;
        G[a-1].push_back({b-1, t});
        G[b-1].push_back({a-1, t});
    }

    // add extra edges
    for (int i = 0; i < N; i++) {
        if (f[i] != -1) {
            G[0].push_back({i, f[i]});
        }
    }

    // run dijkstra
    fill(dist, dist+MAXN, LLONG_MAX);
    priority_queue<Edge> pq;
    pq.push({0, 0});
    while (!pq.empty()) {
        Edge cur = pq.top();
        ...
pq.pop();
if (dist[cur.to] != LLONG_MAX) continue;
dist[cur.to] = cur.t;
for (Edge nxt : G[cur.to]) {
    if (dist[nxt.to] != LLONG_MAX) continue;
    pq.push({nxt.to, max(cur.t + nxt.t, s[nxt.to])});
}

printf("%lld\n", dist[N-1]);