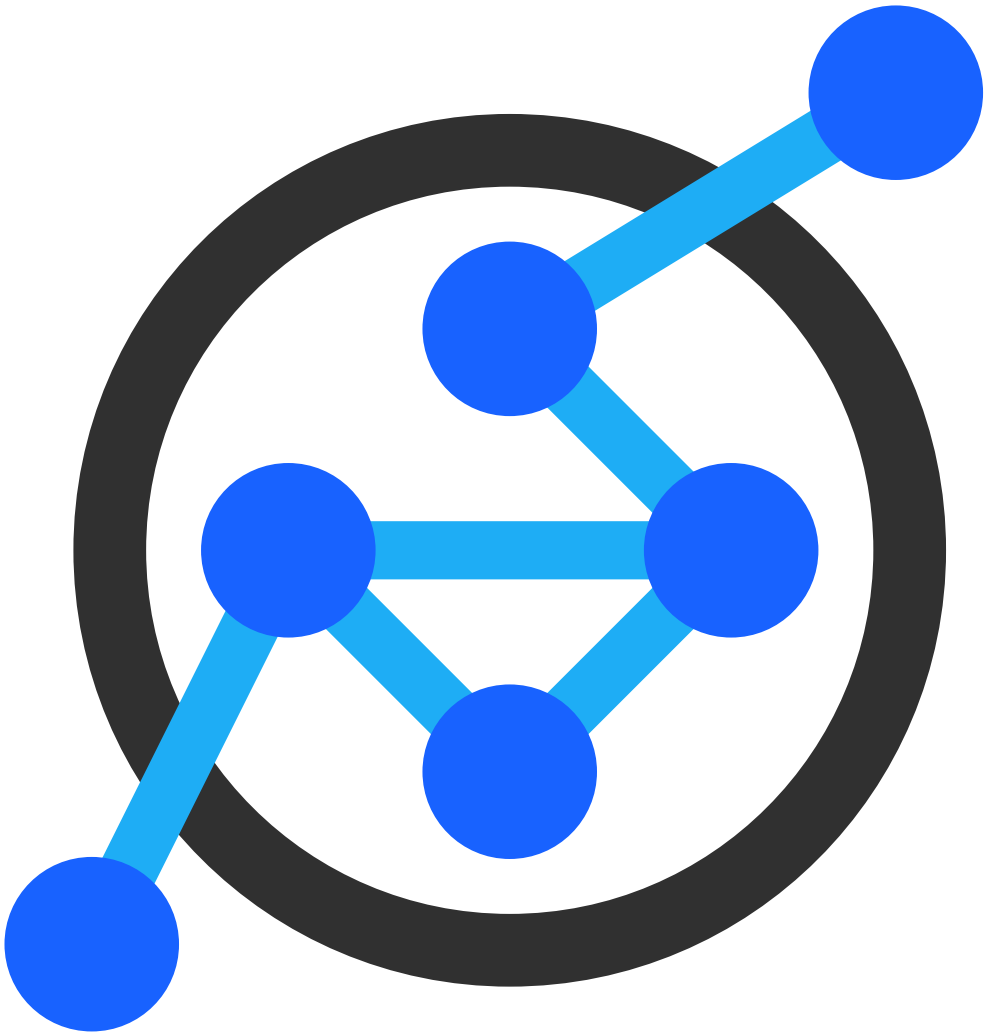
SOI 2P 2024

Solution Booklet



Swiss Olympiad in Informatics

9–10 March 2024

Hidden Keys

Task Idea	Anna Khanova
Task Preparation	Anna Khanova
Description English	Anna Khanova
Description German	Bibin Muttappillil
Description French	Benjamin Faltin
Solution	Anna Khanova

We are given a grid of $n \times m$ numbers where the number in line i and column j is denoted by $c_{i,j}$

Our task is to find i, j, k, l such that $i \neq k$ and $j \neq l$ and $c_{i,j} \neq c_{k,l}$

Subtask 1: $n \cdot m \leq 100$

In this case we can, for each $c_{i,j}$, check all possible k, l to try and find $i \neq k$ and $j \neq l$ and $c_{i,j} \neq c_{k,l}$. If such k, l is found for any i, j we are done and we can output i, j, k, l otherwise no i, j, k, l meet the criteria and we should output -1. We would have a time complexity of $O(n^2m^2)$ which is sufficient.

```cpp
#include <algorithm> // score: 32
#include <vector>
#include <iostream>
using namespace std;

const int MAXN = 100100;

int n, m;
vector<int> a[MAXN];

void print_ans(int x1, int y1, int x2, int y2)
{
        cout << x1 << ' ' << y1 << ' ' << x2 << ' ' << y2 << endl;
        exit(0);
}

void try_at(int x, int y)
{
        for (int i = 0; i < n; ++i)
                for (int j = 0; j < m; ++j)
                        if (i != x && j != y && a[i][j] != a[x][y])
                                print_ans(x, y, i, j);
}

int main()
{
        cin >> n >> m;
        for (int i = 0; i < n; ++i)
        {
                a[i] = vector<int>(m);
                for (int j = 0; j < m; ++j)
                        cin >> a[i][j];
        }
        for (int i = 0; i < n; ++i)
                for (int j = 0; j < m; ++j)
                        try_at(i, j);
        cout << -1 << endl;
        return 0;
}
```

Subtask 2: $n, m \leq 100$

Here, $O(n^2m^2)$ is too slow however, we can realize that if $c_{i,j}$ and $c_{k,l}$ is a solution, then we know that one of the 8 following cases is also a valid solution:

$c_{i,0}$ and $c_{k,l}$, $c_{0,j}$ and $c_{k,l}$, $c_{i,j}$ and $c_{k,0}$, $c_{i,j}$ and $c_{0,l}$, $c_{i,0}$ and $c_{0,j}$, $c_{i,0}$ and $c_{0,l}$, $c_{k,0}$ and $c_{0,j}$, $c_{k,0}$ and $c_{0,l}$

This is because if $c_{i,j} = c_{i,0}$ then $c_{k,l}$ and $c_{i,0}$ is a solution. The same logic applies to $c_{0,j}, c_{k,0}, c_{0,l}$. If none of these equalities hold, then either all $c_{i,0}, c_{0,j}, c_{k,0}, c_{0,l}$ are the same and at least one of the 8 inequalities hold since $c_{i,j} \neq c_{k,l}$ or there is a difference between a number on the line 0 and another on the column 0.

Therefore we only need to check if there is a $c_{i,j}$ which doesn't match with a number on the line 0 or the column 0. We have a complexity of $O((n+m)nm)$

```cpp
#include <algorithm> // score: 51
#include <vector>
#include <iostream>
using namespace std;

const int MAXN = 100100;

int n, m;
vector<int> a[MAXN];

void print_ans(int x1, int y1, int x2, int y2)
{
        cout << x1 << ' ' << y1 << ' ' << x2 << ' ' << y2 << endl;
        exit(0);
}

void try_at(int x, int y)
{
        for (int i = 0; i < n; ++i)
                for (int j = 0; j < m; ++j)
                        if (i != x && j != y && a[i][j] != a[x][y])
                                print_ans(x, y, i, j);
}

int main()
{
        cin >> n >> m;
        for (int i = 0; i < n; ++i)
        {
                a[i] = vector<int>(m);
                for (int j = 0; j < m; ++j)
                        cin >> a[i][j];
        }
        try_at(0, 0);
        for (int i = 1; i < n; ++i)
                try_at(i, 0);
        for (int j = 1; j < m; ++j)
                try_at(0, j);
        cout << -1 << endl;
        return 0;
}
```

Subtask 3: $n \cdot m \leq 100\,000$

If there is a mismatch between $c_{i,j}$ and $c_{k,l}$ and $i, j, k, l > 0$, then at least one of them is different than $c_{0,0}$ If there is a mismatch between $c_{i,j}$ and $c_{k,l}$ and i, j, k or l is 0, then either one of them is different than $c_{0,0}$, $c_{1,0}$ or $c_{0,1}$ and the coordinates don't have any common part, or, if they are equal, we have $c_{0,0} \neq c_{0,1}$, $c_{0,0} \neq c_{1,0}$ or $c_{0,1} \neq c_{1,0}$ which has the consequence of having at least one difference either between $c_{0,1}$ and $c_{1,0}$ or between one of the three points and one other point

somewhere in the grid.

To solve the problem we only need to check all $c_{i,j}$ against $c_{0,0}$, $c_{0,1}$ and $c_{1,0}$

```cpp
#include <algorithm> // score: 100
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 500500;

int n, m;
vector<int> a[MAXN];

void print_ans(int x1, int y1, int x2, int y2)
{
        cout << x1 << ' ' << y1 << ' ' << x2 << ' ' << y2 << endl;
        exit(0);
}

void try_at(int x, int y)
{
        for (int i = 0; i < n; ++i)
                for (int j = 0; j < m; ++j)
                        if (i != x && j != y && a[i][j] != a[x][y])
                                print_ans(x, y, i, j);
}

int main()
{
        cin >> n >> m;
        for (int i = 0; i < n; ++i)
        {
                a[i] = vector<int>(m);
                for (int j = 0; j < m; ++j)
                        cin >> a[i][j];
        }
        try_at(0, 0);
        if (m != 1)
                try_at(0, 1);
        if (n != 1)
                try_at(1, 0);
        cout << -1 << endl;
        return 0;
}
```

Moving Faster

Task Idea	Luca Versari
Task Preparation	Théo von Düring
Description English	Théo von Düring
Description German	Bibin Muttappillil
Description French	Théo von Düring
Solution	Théo von Düring

We are given a value k, a weighted connected graph where each edge has a weight t and a decreasing value d, and two nodes s and e. We want to compute the minimum time needed to go back and forth between s and e a total of k times.

One first observation is that the direction of the path doesn't matter, going from s to e or from e to s on the same set of edges decreases each edge by the same amount and the sum of all edge distances is also the same. Therefore the problem is the same as going k times from s to e and decreasing the edge weights each time. We will use this simplified problem in the solutions.

Subtask 1: A tree

If the graph is a tree we know that the path between s and e is unique and therefore it will always be used to travel between the two nodes. To solve this task we can just run a DFS between the two nodes and save the path taken. Then compute the time needed to use k times each edge.

```cpp
#include <bits/stdc++.h> // score: 17

using namespace std;

#define int int64_t

vector<vector<array<int,2>>> g;
stack<array<int,2>> st;

bool dfs(int u, int target, int i, int p) {
    st.push({u,i});

    if (u == target)
        return true;

    for (auto [v,j] : g[u]) {
        if (v == p)
            continue;
        if (dfs(v, target, j, u))
            return true;
    }
    st.pop();
    return false;
}

signed main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int n,m,k,s,e;
    cin >> n >> m >> k >> s >> e;

    g.assign(n, {});
    vector<array<int,2>> edge(m);

```

```
    for (int i = 0; i < m; i++) {
        int a,b,t,d;

        cin >> a >> b >> t >> d;
        edge[i] = {t,d};
        g[a].push_back({b,i});
        g[b].push_back({a,i});
    }

    dfs(s, e, -1, s);

    int ans = 0;
    while (!st.empty()) {
        auto [u, i] = st.top();
        auto [t, d] = edge[i];
        st.pop();
        if (i == -1)
            continue;
        ans += k*t - d*(k*(k-1)/2);
    }
    cout << ans;
}
```

Subtask 2: $k = 1$

Since $k = 1$, we only need to travel once from s to e and since we never traverse an edge twice, d doesn't have any effect. Thus, to solve this case you need to go from s to e using the shortest path without taking into account d. This problem can be solved using a standard dijkstra implementation on the weighted graph

```
#include <bits/stdc++.h> // score: 15

using namespace std;

#define int int64_t

signed main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int n,m,k,s,e;
    cin >> n >> m >> k >> s >> e;

    vector<vector<array<int,2>>> g(n);
    for (int i = 0; i < m; i++) {
        int a,b,t,d;
        cin >> a >> b >> t >> d;

        g[a].push_back({b, t});
        g[b].push_back({a, t});
    }

    priority_queue<array<int,2>> pq;
    pq.push({0, s});
    vector<int> dist(n, -1);
    while (!pq.empty()) {
        auto [d, a] = pq.top();
        pq.pop();

        if (dist[a] != -1)
            continue;
        dist[a] = -d;

        for (auto [b, t] : g[a]) {
            if (dist[b] != -1)
                continue;
```

```
            pq.push({d-t, b});
        }
    }
    cout << dist[e] << "\n";
}
```

General problem

For the general case it can also be shown that using the same path k times is optimal. The idea of the proof is that if you suppose that the optimal solution uses $n > 1$ paths which we will call $p_1, p_2, \ldots, p_n$ each used $c_1, c_2, \ldots, c_n$ times respectively, then there exist i, j, such that using p_i $c_i - 1$ times and p_j $c_j + 1$ times is faster which contradicts the hypothesis and shows that we only use one path. To solve the problem we will, for each edge, compute the time it will take to use it k times and use dijkstra on the graph with these new precomputed times as weights. This gives us the shortest weighted path and the time needed to use it k times

```cpp
#include <bits/stdc++.h> // score: 100


using namespace std;

#define int int64_t

signed main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int n,m,k,s,e;
    cin >> n >> m >> k >> s >> e;

    vector<vector<array<int,2>>> g(n);

    for (int i = 0; i < m; i++) {
        int a,b,t,d;

        cin >> a >> b >> t >> d;

        assert(t > 0);
        assert(d > 0);
        assert(t*k > 0);
        assert(d*(k*(k-1)/2) >= 0);
        assert(t - d*k > 0);

        t = t*k - d*(k*(k-1)/2);
        assert(t > 0);

        g[a].push_back({b,t});
        g[b].push_back({a,t});
    }

    vector<int> dist(n, -1);

    priority_queue<array<int,2>> pq;
    pq.push({0, s});
    while(!pq.empty()) {
        auto [d1, u] = pq.top();
        pq.pop();

        if (dist[u] != -1)
            continue;

        dist[u] = -d1;
        assert(dist[u] >= 0);

        for (auto [v,d2] : g[u]) {
```

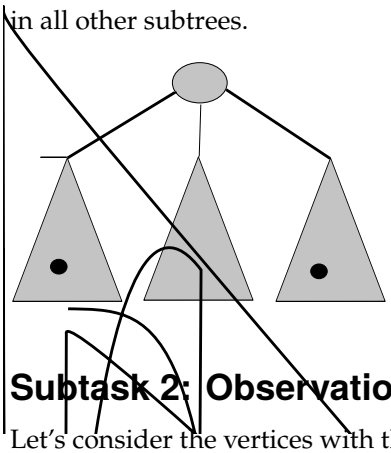
```
50              if (dist[v] != -1)
51                  continue;
52              pq.push({d1-d2, v});
53          }
54      }
55
56      assert(dist[e] > 0);
57
58      cout << dist[e] << "\n";
59  }
```

Ruined Chocolates

Task Idea	Andrei Feodorov, Johannes Kapfhammer
Task Preparation	Andrei Feodorov
Description English	Andrei Feodorov
Description German	Charlotte Knierim
Description French	Théo von Düring
Solution	Johannes Kapfhammer

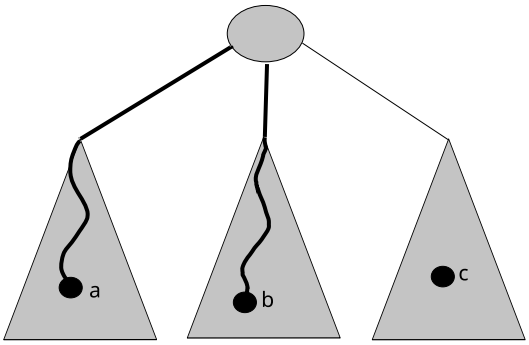
Subtask 1: Observation: All queries go through the root

Root the tree in the vertex that has the maximal value. Then: if a query does not go through the root, the answer is just the value of the root. Else we just have to consider answering the question for two subtrees where the query starts at the root of the subtree. The final answer is the maximum answer of the two subqueries inside the two subtrees, and of the maximum value in all other subtrees.



Subtask 2: Observation: Two paths and an extra vertex

Let's consider the vertices with the k largest values. Choose the smallest k such that there is no path that contains them all.

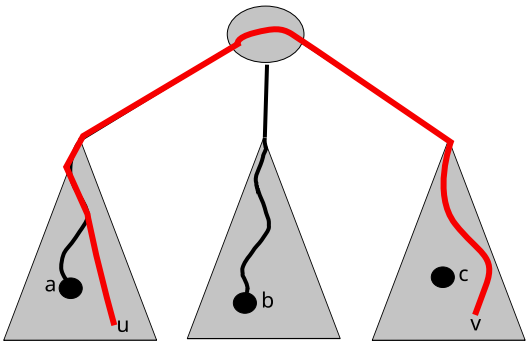


Since the largest value is the root, we can describe this set by giving the two endpoints of the path containing the first $k-1$ elements, call them a and b, and the extra vertex c.

Note that:

- a and b lie in different subtrees because the root must be part of the path (recall we are considering the largest k values). There is one exception if the root only has a single child.
- c can be in the same subtree as a or b or in a different one.
- Any possible query can never cover all vertices path$(a, b) \cup \{c\}$ – because queries are paths and by definition there is no path through those vertices.

- Therefore, the answer to any query is at least value(c).
- As a corollary, the values of vertices outside of path(a, b) $\cup \{c\}$ don't matter.



So to answer a query (v, w), we only have check:

- If v and a are in the same subtree, let $q = LCA(a, v)$ be the vertex where they come together. The answer of the query is the maximum of all values on path$(a, q) \setminus \{q\}$. Similarly with v and b, w and a, and w and b.
- If vertex c does not lie on the path $v \to w$, then also consider the value of c a possible answer.
- By the observations above, this is enough – we will have considered at least one value because the set path(a, b) $\cup \{c\}$ is a strict superset of the query. And all other values are smaller.

Subtask 3: Implementation

We need to do the following precomputation which can be implemented in $O(n)$:

1. Root the tree at the vertex with the maximum value.
2. Iterate over the vertices in decreasing value, and find the two endpoints of the path a and b, and our extra vertex c.
3. For every other vertex, precompute the nearest ancestor that lies on the path $a \to b$. For each vertex on the path compute the maximum of all vertices strictly below.

And then for each query (v, w) we can answer in $O(1)$:

- Find the two points $p = LCA(a, v)$ (or w) and $q = LCA(b, w)$ (or v) where we meet up with the path. Then the maximum value below p and q are possible answers. All of this is already precomputed.
- Check whether c lies on the path $v \to w$. If not, then the value of c is also a possible answer.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);

  int n, q; cin >> n >> q;
  vector<vector<int>> g(n);
  for (int i=1; i<n; ++i) {
    int a, b; cin >> a >> b;
    g[a].push_back(b);
    g[b].push_back(a);
  }
```

```
  vector<int> value; copy_n(istream_iterator<int>(cin), n, back_inserter(value));
  vector<int> order(n); iota(order.begin(), order.end(), 0);
  sort(order.begin(), order.end(), [&](int a, int b) { return value[a]>value[b]; });
  int root = order[0];

  vector<int> pre(n), post(n);
  vector<int> subtreerootof(n), parent(n);
  {
    int time = 0;
    auto dfs = [&](auto&& self, int v, int p, int subroot) -> void {
      pre[v] = time++;
      subtreerootof[v] = subroot;
      parent[v] = p;
      for (auto w : g[v])
        if (w != p)
          self(self, w, v, subroot);
      post[v] = time;
    };
    for (auto w : g[root])
      dfs(dfs, w, root, w);
  }
  auto is_ancestor_of = [&](int a, int b) { return pre[a] <= pre[b] && post[b] <= post[a]; };
  array<int, 2> subtree_root{-1, -1};
  array<int, 2> subtree_pathend{};
  array<int, 2> subtree_maxvalue{};
  int extra_vertex = -1;

  for (auto v : order) {
    if (v == root) continue;
    int i=0;
    for (; i<2; ++i) {
      if (subtree_root[i] == -1) {
        subtree_root[i] = subtreerootof[v];
        subtree_pathend[i] = v;
        subtree_maxvalue[i] = value[v];
        break;
      } else if (subtree_root[i] == subtreerootof[v]) {
        if (is_ancestor_of(subtree_pathend[i], v))
          subtree_pathend[i] = v;
        else if (!is_ancestor_of(v, subtree_pathend[i]))
          extra_vertex = v;
        break;
      }
    }
    if (i == 2 && extra_vertex == -1)
      extra_vertex = v;
    if (extra_vertex != -1)
      break;
  }
  vector<int> subtree_answer(n, -1);
  for (int i=0; i<2 && subtree_root[i] != -1; ++i) {
    int pos = subtree_pathend[i];
    int submax = 0;
    auto fill = [&](auto&& self, int v, int p) -> void {
      subtree_answer[v] = submax;
      for (int w : g[v])
        if (w != p && subtree_answer[w] == -1)
          self(self, w, v);
    };
    while (pos != root) {
      fill(fill, pos, parent[pos]);
      submax = max(submax, value[pos]);
      pos = parent[pos];
    }
  }

  auto subtree_index = [&](int v) {
    if (v == root) return -1;
```

```
82      int subroot = subtreerootof[v];
83      auto it = find(subtree_root.begin(), subtree_root.end(), subroot);
84      return it == subtree_root.end() ? -1 : static_cast<int>(it - subtree_root.begin());
85    };
86    while (q--) {
87      int a, b; cin >> a >> b;
88      int i = subtree_index(a), j = subtree_index(b);
89      int ans = 0;
90      if (i != -1 && j != -1 && i == j) {
91        ans = value[root];
92      } else {
93        for (int k=0; k<2; ++k) {
94          if      (k == i) ans = max(ans, subtree_answer[a]);
95          else if (k == j) ans = max(ans, subtree_answer[b]);
96          else ans = max(ans, subtree_maxvalue[k]);
97        }
98        if (extra_vertex != -1 &&
99            !is_ancestor_of(extra_vertex, a) &&
100           !is_ancestor_of(extra_vertex, b))
101         ans = max(ans, value[extra_vertex]);
102     }
103     cout << ans << '\n';
104   }
105 }
```

Tram Tickets

Task Idea	Bibin Muttappillil
Task Preparation	Charlotte Knierim
Description English	Charlotte Knierim
Description German	Charlotte Knierim
Description French	Benjamin Faltin
Solution	Johannes Kapfhammer

Step 1: Without Binna (51 Points)

We sort by t_i then we do the following DP approach:

$$\begin{aligned}\texttt{DP}[i] &= \text{cheapest assignment for} t_0, \ldots, t_{i-1} \\ &= \min(\text{buy single ticket at } t_i, \text{ buy multiride ticket at time } t_i - d) \\ &= \min(\texttt{DP}[i-1] + a, \texttt{DP}[\text{lower\_bound}(t_i - d)] + b)\end{aligned}$$

where lower_bound$(t_i - d)$ is the largest index j such that $t_j < t_i - d$. This index can be found via binary search for a $O(n \log n)$ solution, or via two pointer which runs in $O(n)$.

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int int64_t

signed main(){
  int n, _, singleticket, multiride, d;
  cin >> n >> _ >> singleticket >> multiride >> d;

  vector<int> time; copy_n(istream_iterator<int>(cin), n, back_inserter(time));
  sort(time.begin(), time.end());

  vector<int> dp(n+1, 0);
  for (int i=0, j=0; i < n; ++i) {
    while(j+1 < n && time[j] <= time[i] - d)
      ++j;
    dp[i+1] = min(dp[i] + singleticket,
                  dp[j] + multiride);
  }
  cout << dp[n] << '\n';
}
```

Step 2: With Binna (75 Points)

$$\begin{aligned}\texttt{DP}[i][j] =& \text{cheapest assignment for } t_0, \ldots, t_{i-1} \\ & \text{with Binna having multiride ticket remaining for } j \text{ more time units}\end{aligned}$$

Let $t_0, \ldots, t_{n-1}$ be the unique times in which we have tickets. Define asum(l, r) as the number of distinct times t_i with $l \le t_i < r$ multiplied by a (the cost for single tickets).

We call tickets which are "doubled" if there exactly two tickets at the same time. Let's define bsum(l, r) as number of distinct times t_i with $l \le t_i < r$ where we have doubled tickets, again multiplied by a (so this will be the cost for doubled single tickets). Let's define bsum to be 0 if $r \le l$. With prefix sums precomputation this can be evaluated in constant time.

$$\texttt{DP}[i][j] = \min \begin{cases} \min_{0 \le k \le d} (\texttt{DP}[i-d][k] + \text{bsum}(i-d+k, i) + b) & \text{Stofl: buy M at } i-b\text{, Binna: only S} \\ \min_{0 \le k \le d} (\texttt{DP}[i-d][k] + \text{bsum}(i-d+k, i-d+j) + 2 \cdot b) & \text{Stofl: M at } i-b\text{, Binna: M at } i-(d-j) \\ \texttt{DP}[i-1][j+1] + \text{bsum}(i-1, i) \quad \text{if } j+1 \le d & \text{Use Binna's M, Stofl buys doubled S} \\ \texttt{DP}[i-1][0] + \text{asum}(i-1, i) + \text{bsum}(i-1, i) \quad \text{if } j = 0 & \text{Both buy S} \\ \texttt{DP}[i][0] + b \quad \text{if } j = d & \text{Binna: buy M at } i \end{cases}$$

gives a DP that runs in $O(n \cdot d^2)$.

This can be improved by noticing that the first case does not depend on j so it can be cached, and the last three cases can be computed in constant time. So we only need to speed up the second case:

$$\begin{aligned} A[j][k] &= \texttt{DP}[i-d][k] + \text{bsum}(i-d+k, i-j) + 2 \cdot b \\ &= \underbrace{\texttt{DP}[i-d][k] + 2 \cdot b}_{\text{independent of } j} + \underbrace{\text{bsum}(i-d+k, i-j)}_{\text{only a prefix changes}} \end{aligned}$$

What happens when we go from $A[j+1]$ to $A[j]$?

$$A[j][k] = A[j+1][k] + \begin{cases} \text{bsum}(i-j-1, i-j) & \text{if } i-j-1 > i-d+k \\ 0 & \text{otherwise} \end{cases}$$

This means we can store array $A[j]$ as segment tree, which supports the two operations we need: increment a prefix, get the global minimum.

This runs in $O(n \cdot d \log d)$ if we only look at times i where there is a ticket. Instead of segment trees we can also use sets: We have $A[j][k] \le A[j][k+1]$ (having Binna's ticket last longer is always more expensive) so the array $A[k]$ is non-decreasing. Let's just keep the differences in a map with entries (k,A[j][k]-A[j][k-1]). When we increment a prefix we increment the delta of the first entry and decrement the delta of some entry in the middle. Let's say the new element is (k_i, d_i) and the entry before is (k_{i-1}, d_{i-1}). Then we replace merge both entries into a new one $(k_i, d_i + d_{i-1})$ or remove it if $d_i + d_{i-1} = 0$. This gets rid of the irrelevant entries and we only keep the prefix minima. The minimum is always at the beginning of the set.

And instead of segtree/sets we can also just use vectors: since the prefixes are always getting smaller, and the array is non-decreasing initially, we pop all values that are no longer getting changed from the end of the vector and keep one minimum of the values that have been removed. We also store a global `offset` that we want to add to all elements of the vector. Then incrementing a prefix just means popping the last value of the vector, add `offset` to it and update the global minimum. Then increment the `offset`. The overall minimum is the minimum between the first entry of the vector and the popped minimum.

This runs in $O(n \cdot d)$.

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

#define int int64_t

const int INF = 1e9;

void xmin(int& x, int v) {
  if (v < x)
    x = v;
}

```

```cpp
signed main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin.exceptions(ifstream::badbit | ifstream::failbit | ifstream::eofbit);

  int num_stofl_tickets, num_binna_tickets, singleticket, multiride, d;
  cin >> num_stofl_tickets >> num_binna_tickets >> singleticket >> multiride >> d;

  vector<int> time;
  vector<bool> both;
  {
    array<vector<int>, 2> xs;
    xs[0].resize(num_stofl_tickets);
    xs[1].resize(num_binna_tickets);
    for (auto& v : xs) {
      for (auto& x : v) cin >> x;
      sort(v.begin(), v.end());
    }
    time.push_back(-d-2);
    set_union(xs[0].begin(), xs[0].end(), xs[1].begin(), xs[1].end(), back_inserter(time));
    vector<int> bs;
    set_intersection(xs[0].begin(), xs[0].end(), xs[1].begin(), xs[1].end(), back_inserter(bs));
    for (auto t : time)
      both.push_back(binary_search(bs.begin(), bs.end(), t));
  }
  const int n = time.size();

  vector<vector<int>> dp(n+1, vector<int>(d+1, INF));
  dp[0][0] = 0;
  vector<int> dp_i_d(d+1);
  for (int i=1, i_d=0; i<n; ++i) {
    int time_i_d = time[i] - d;
    while (time[i_d+1] <= time_i_d)
      ++i_d;
    { // dp_i_d is the interpolated DP state at time_i_d.
      int padding = time_i_d - time[i_d];
      assert(padding >= 0);
      if (padding == 0) {
        dp_i_d = dp[i_d];
      } else {
        fill(dp_i_d.begin(), dp_i_d.end(), INF);
        for (int k=d-padding; k>=0; --k)
          dp_i_d[k] = min(dp_i_d[k+1], dp[i_d][k+padding]);
        dp_i_d[0] = min(dp_i_d[0], dp[i_d][0]);
      }
    }

    int inbetween = 0;
    for (int i_k=i_d + (time[i_d] < time[i]-d); i_k <= i; ++i_k)
      inbetween += both[i_k];

    // buy Stofl a double ticket at time t
    vector<int> aj(d+1, INF);
    int aj_offset=0, aj_popmin=INF;
    for (int k=0, i_k=i_d + (time[i_d] < time[i]-d); k<=d; ++k) {
      assert(time[i_k] >= time[i] - d + k);

      // cover [i-d,i] with a multiride but don't buy a second one
      xmin(dp[i][0], dp_i_d[k] + singleticket*inbetween + multiride);

      // prepare array a[j] for the next case:
      // cover [i-d,i] with a multiride, buy new multiride at time i-d+j
      xmin(aj[k], dp_i_d[k] + singleticket*inbetween + 2*multiride);

      // Needed to make a[j] decreasing
      if (k) xmin(aj[k], aj[k-1]);

      if (i_k+1 < n && time[i_k] < time[i] && time[i_k] == time[i]-d+k)
        inbetween -= both[i_k++];
```

```
    }
    for (int j=d, i_k=i; j>=0; --j) {
      assert(time[i_k] <= time[i]-d+j);
      if (time[i_k] == time[i]-d+j) {
        if (both[i_k])
          aj_offset -= singleticket;
        --i_k;
      }
      assert((int)aj.size() == j+1);
      xmin(aj_popmin, aj.back() + aj_offset);
      aj.pop_back();
      xmin(dp[i][j], min(aj_popmin, aj.empty() ? INF : aj.back()+aj_offset));
    }

    // use up existing multiride and buy singles
    int td = time[i] - time[i-1];
    for (int j=1; j + td <= d; ++j)
      xmin(dp[i][j], dp[i-1][j + td] + (both[i] ? singleticket : 0));

    // just buy singletickets
    xmin(dp[i][0], dp[i-1][0] + (both[i] ? 2 : 1)*singleticket);

    // buy a new multiride
    xmin(dp[i][d], dp[i-1][0] + multiride + (both[i] ? singleticket : 0));

    // make monotone (not needed)
    for (int j=d-1; j>=0; --j)
      xmin(dp[i][j], dp[i][j+1]);
  }
  cout << dp[n-1][0] << '\n';
}
```

Step 3: Happy subtask merging

Submit both solutions to step 1 and step 2 and subtask merging will do its job giving you well-deserved 100 points.

Baguette Magique

Task Idea	Anna Khanova, Johannes Kapfhammer
Task Preparation	Anna Khanova
Description English	Johannes Kapfhammer
Description German	Bibin Muttappillil, Johannes Kapfhammer
Description French	Benjamin Faltin
Solution	Jan Schär

Subtask 1: Cut single centimeters

In this subtask, we can only cut out parts of length 1 cm from the baguette. We need to pick some flavor which should remain at the end, and cut out all centimeters which have a different flavor. We can just try this for all possible flavors and pick the best option.

```cpp
#include<bits/stdc++.h> // score: 12
using namespace std;

signed main () {
  int n, k;
  string baguette;
  cin >> n >> k >> baguette;

  int best_cuts = n - 1;
  for (char keep = 'a'; keep <= 'z'; keep++) {
    int count = 0;
    for (char c : baguette) if (c != keep) count++;
    best_cuts = min(best_cuts, count);
  }
  cout << best_cuts << "\n";
}
```

This solution runs in $O(n)$ time (when treating the size of the alphabet as a constant).

Subtask 4: 10 meter baguette

The baguette can be at most 1'000 cm in length.

Once we have decided which flavor we want to keep, the problem becomes simpler: We need to remove all the centimeters which are not this flavor.

There are still two tricky things: It can be that it is more efficient to remove a larger part which also includes centimeters of the flavor we want to keep. For example, let's say we have a baguette which contains the pattern `aba`, and we can remove parts of length at most 3 cm, and want to end up with only flavor `b`. Then it's better to cut out the entire pattern at once, rather than cutting out the two centimeters of flavor `a` individually.

But then we need to be careful that we don't end up with an empty baguette.

One possible solution is to brute-force a centimeter of the baguette which we definitely keep. Once we know which centimeter to keep, we can then scan the baguette left to right, and when we see a centimeter of the wrong flavor, we remove a part of the baguette which is as long as possible, stopping before the centimeter which we definitely keep.

It's optimal to remove a part which is as long as possible, because we only need to optimize the number of cuts, not their length, and this way we maximize the chance of removing as many centimeters of the wrong flavor as possible.

```cpp
#include<bits/stdc++.h> // score: 67
using namespace std;

signed main () {
  int n, k;
  string baguette;
  cin >> n >> k >> baguette;

  int best_cuts = n;
  for (int keep = 0; keep < n; keep++) {
    int cuts = 0;
    int cut_until = 0;
    for (int i = 0; i < n; i++) {
      if (i == keep) {
        cut_until = 0;
      } else if (cut_until <= i && baguette[i] != baguette[keep]) {
        cuts++;
        cut_until = i + k;
      }
    }
    best_cuts = min(best_cuts, cuts);
  }
  cout << best_cuts << "\n";
}
```

This solution runs in $O(n^2)$ time, which is enough for this subtask.

Subtask 5: 100 km baguette

The baguette can be at most 10^7 cm in length. The previous solution is too slow for this.

We can improve it by making the observation that we can actually ignore the constraint that the baguette must be non-empty. We can only violate this constraint if we remove the entire baguette. In this case, we can just add back the last centimeter of the baguette, it doesn't matter what flavor it has.

The solution is to try out all flavors, and for each, scan the baguette from left to right and remove a maximal part whenever we see a centimeter of the wrong flavor.

```cpp
#include<bits/stdc++.h> // score: 100
using namespace std;

signed main () {
  int n, k; cin >> n >> k;
  string s; cin >> s;

  int ans = n;

  for (char keep = 'a'; keep <= 'z'; ++keep) {
    int cuts = 0;
    for (int i = 0; i < s.size();) {
      if (s[i] != keep) {
        i += k;
        cuts += 1;
      } else {
        i += 1;
      }
    }
    ans = min(ans, cuts);
  }
  cout << ans << "\n";
}
```

This solution runs in $O(n)$ time.

Catering

Task Idea	Johannes Kapfhammer
Task Preparation	Amirkeivan Mohtashami
Description English	Amirkeivan Mohtashami
Description German	Bibin Muttappillil, Johannes Kapfhammer
Description French	Benjamin Faltin
Solution	Amirkeivan Mohtashami

We are given a weighted connected graph with n nodes and m edges representing a network of roads and the price needed to cover each road with cheese. Our goal is to cover enough roads with cheese such that between any two nodes there exists a path between them which is completely covered in cheese. A sponsor also accepts to cover k of the roads we chose for free. We want to compute the minimum cost needed to cover the remaining roads we chose with cheese.

Observation

The roads we want to cover with cheese must form a spanning tree. If the solution wasn't a tree then there would exist a cycle and the number of roads we must pay for ourselves could be reduced by one by removing one edge in the cycle and the solution wouldn't be optimal. Additionally, it must be a spanning tree since all two nodes must be connected by a path.

Solution

Since the roads must form a tree with $n-1$ edges we must pay for $max(0, n-k-1)$ of these edges. The claim is that these $n-k-1$ edges are the $n-k-1$ smallest edges contained in a minimum spanning tree of the weighted graph. Let's prove this. The lemma 1 isn't needed to prove that the algorithm is correct but is needed to prove that the $n-k-1$ smallest edges of any spanning trees form a solution.

Lemma 1.

All minimum spanning trees have edges of same weight, alternatively, the i-th smallest edge of two different spanning trees are of same weight.

Proof. We will proceed by contradiction. Let's suppose we have two minimum spanning trees T_1, T_2 with different edge weights.
Then T_1 and T_2 must have a different number of edges with weight w for some weight w in the possible edge weights. Lets call $n_{w,1}$ the number of edges of weight w in T_1 and $n_{w,2}$ the number in T_2.
Without loss of generality, lets suppose $n_{w,1} > n_{w,2}$
Now, for each edge e_w of weight w in T_1 we can add it to T_2 and it will create a cycle containing the edge e_w provided e_w isn't already in T_2.
Since T_2 is a MST, all edges on that created cycle must have weight smaller or equal to w otherwise we could remove one of the longer edges and have a smaller MST than T_2 which is absurd
Since T_1 is a MST, there must be an other edge on the created cycle in T_2 which doesn't belong to T_1 with weight w. If that wasn't the case, we could remove e_w from T_1 and add one of the smaller edge in that created cycle to connect the two newly disconnected subtrees in T_1 which would imply that we can have a smaller MST than T_1 which is absurd.
We will now, for each edge e_w not already in T_2, remove one edge of weight w not in T_1 from T_2 and add e_w to T_2.

We conclude that we can remove all edges with weight w from T_2 and insert all the ones from T_1 and know that T_2 is still a MST after the changes. But then T_2 has more edges than T_1 since $n_{w,1} > n_{w,2}$ hence it is no longer a MST which is absurd
Therefore, T_1 and T_2 must have edges with the same weights. □

Lemma 1. shows us that all MST have the same set of edge weights and so taking the $n-k-1$ smallest edges isn't impacted by the found MST. In this solution Kruskal's algorithm was used to compute the edges but because of Lemma 1., any other method to get a valid MST gives a correct answer.
Let's now show that the $n-k-1$ smallest edges of an MST are a set without cycles with minimum weight sum.
(We will use the $n-k-1$ first edges obtained using Kruskal's algorithm.)

Proof. Let S_1 be a set of $n-k-1$ edges which is solution to the problem.
We will proceed using induction.

(I.H.) The set S_2 of l edges obtained using Kruskal on the first m edges is optimal. i.e. the l edges in S_2 are the l first edges of some solution S_1 and all edges between the l-th edge of S_2 and the m-th edge of the graph are not in S_1.

($m = 0$) The set containing edges present in the 0 first smallest edges of the graph is the empty set which is contained in S_1. Therefore the algorithm is correct for a $m = 0$.

($m \implies m+1$) Suppose the algorithm works for m edges. Let's show it works for $m+1$ edges.
If at any point the subset S_2 of edges obtained with Kruskal applied on the m first edges is of size $n-k+1$, we are done and the proof is finished.
Consider e_{m+1} the $m+1$-th smallest edge of the graph, we have two cases

- if e_{m+1} is in S_1:
  We are done and Kruskal is optimal on $m+1$ edges.
- if e_{m+1} isn't in S_1:
  If when we insert e_{m+1} in S_2 (optimal by (I.H)) it create a cycle, not adding it to S_2 is correct as adding it would not create a valid solution.

  If it doesn't create a cycle, then adding it to S_1 must create a cycle with no edges greater than e_{m+1} as otherwise we could remove a longer edge f from S_1 and $S_1 \cup \{e\} \setminus \{f\}$ would be a better solution than S_1 which is absurd.
  Additionally there must exist an edge with weight greater or equal to e_{m+1} on the cycle since otherwise it would create a cycle in S_2 which is not the case by hypothesis.

  Therefore there exists an edge f with weight equal to e_{m+1} such that $S_1' = S_1 \cup \{e_{m+1}\} \setminus \{f\}$ is a valid solution. We can now replace S_1 with S_1' while keeping all our induction steps true since S_1' works like S_1 for all previous induction steps.

Since Kruskal gives us a tree with size $n-1 \geq n-k-1$, the set S_2 obtained is always a solution by induction. □

Again, by Lemma 1., this shows us that any MST algorithm works since Kruskal's algorithm works for this problem.

```cpp
#include <bits/stdc++.h> // score: 100

using namespace std;

#define int int64_t
#define all(x) x.begin(), x.end()
#define sz(x) (int)x.size()

vector<int> dsu;

```

```cpp
int find(int u) {
    if (dsu[u] == u)
        return u;
    return dsu[u] = find(dsu[u]);
}

signed main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int n,m,k;
    cin >> n >> m >> k;

    dsu.resize(n);
    for (int i = 0; i < n; i++)
        dsu[i] = i;

    vector<array<int,3>> e(m);
    for (int i = 0; i < m; i++) {
        int u,v,c;
        cin >> u >> v >> c;

        e[i] = {c, u, v};
    }

    sort(all(e));

    int cost = 0;
    int cnt = k;
    for (int i = 0; cnt < n-1; i++) {
        auto [c,u,v] = e[i];

        u = find(u);
        v = find(v);
        if (u == v)
            continue;
        cost += c;
        cnt++;
        dsu[u] = v;
    }
    cout << cost << "\n";
}
```