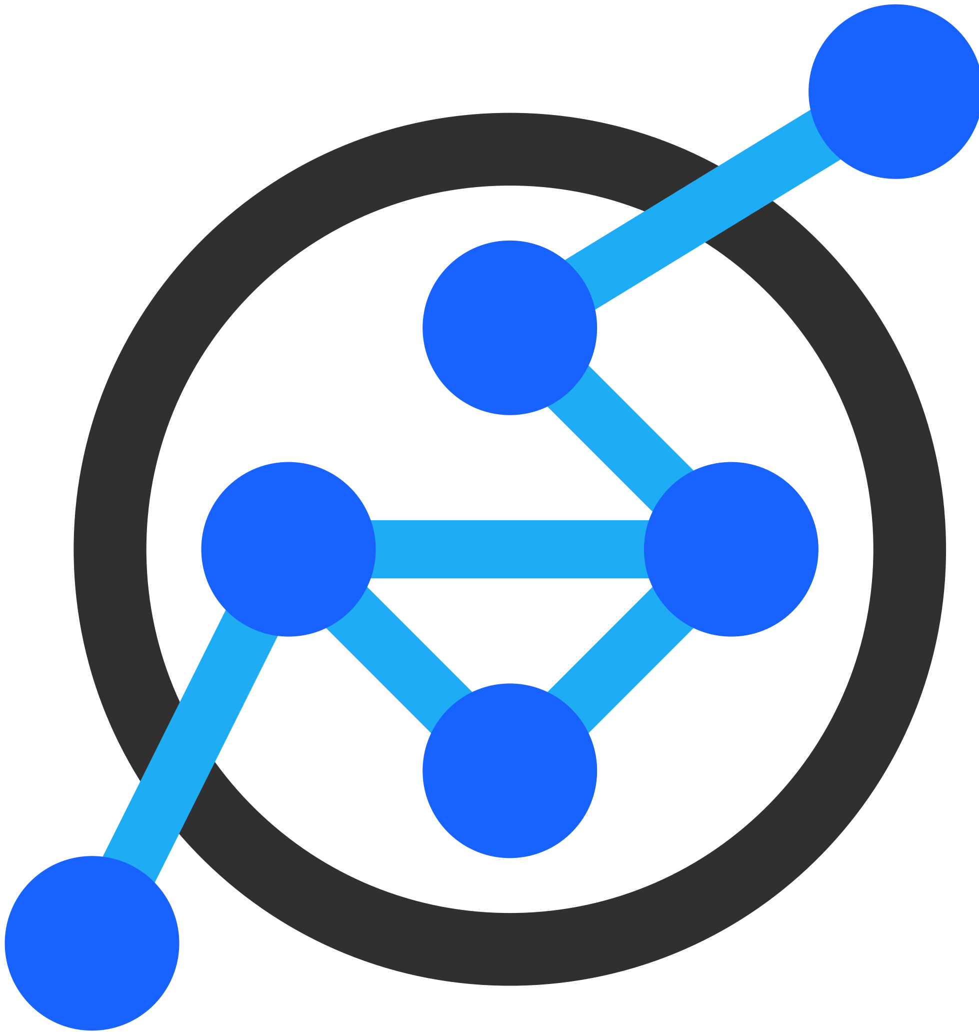
Qualification Round SOI 2024/2025

Solution Booklet



Swiss Olympiad in Informatics

15 September – 30 November 2024

Cheese Machine

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

In this task we are given N integers and we want to know if the N integers form an arithmetic progression, and if they do, what would the next number in the sequence be.

An arithmetic progression is defined as a sequence of integers for which the difference between any integer and its preceding integer is always the same.
Formally if the sequence is $a_0, a_1, \ldots, a_{N-1}$, then $a_k = a_0 + k \cdot d$ where $d = a_{i+1} - a_i$ for any $k = 0, \ldots, N-1$ and $i = 0, \ldots, N-2$.

Subtask 1: A small test (10 points)

In this first subtask, the sequence consists of only 2 integers. Therefore, we don't need to verify if the difference d is the same for all pairs of consecutive integers, since there are only 2 elements and 1 difference. Thus, d is the difference between the first and second integers and when applying our formula we obtain the third element.

```
1  def get_input():
2      input()
3      return map(int, input().split())
4
5  def solve():
6      a,b = get_input()
7      d = b-a
8      return a + 2*d
9
10 t = int(input())
11 for i in range(t):
12     print(f"Case #{i}: {solve()}")
```

The time and space complexity of this algorithm is $O(1)$ since we execute a constant amount of operations for all inputs and store a constant amount of numbers.

Subtask 2: Small test, no problems (13 points)

In this subtask, we have 3 numbers and have the guarantee that the sequence is an arithmetic progression. We only now need to compute the fourth element of the sequence. By applying our formula with d being the difference between the first two integers, we can solve the problem in a similar way as before.

```
1 def solve():
2     a,b,c = get_input()
3     d = b-a
4     return a + 3*d
```

The time and space complexity are again $O(1)$ for the same reasons.

Subtask 3: Imprecise machine (17 points)

The third subtask is identical to the second one, but unlike the last subtask, we need to check if the sequence is an arithmetic progression before predicting the next element. We thus only need to add a condition checking if the differences between the first and second number is the same as the one between the second and third.

```
def solve():
    a,b,c = get_input()
    d1 = b-a
    d2 = c-b
    if (d1 != d2):
        return "NO"
    return a + 3*d1
```

The time and space complexity are again $O(1)$ for the same reasons.

Subtask 4: Finally a perfect machine (26 points)

This subtask is identical to the second one, we are guaranteed that the sequence forms an arithmetic progression, however, now the sequence can be much longer. Since we know it's an arithmetic progression we can apply our formula using the difference between the first and second number.

```
def solve():
    l = list(get_input())
    d = l[1]-l[0]
    N = len(l)
    return l[0] + N*d
```

The time and space complexity are $O(N)$ since we need to read the N numbers and store them in a list.

Subtask 5: The machine needs fixing (34 points)

Compared to the fourth subtask, in the fifth subtask we need to check that the sequence is an arithmetic progression before outputing the next element of the sequence. We just iterate through all consecutive 2 elements and compare their difference to the difference between the first 2 elements. If at some point the differences don't match, then we should output "NO", otherwise we use the same formula as before to calculate the next element.

```
def solve():
    l = list(get_input())
    d = l[1]-l[0]
    N = len(l)
    for i in range(1,N):
        if l[i]-l[i-1] != d:
            return "NO"
    return l[0] + N*d
```

The time and space complexity are $O(N)$ since we need to read the N numbers, compare the differences of $N-1$ pairs of integers and store these elements in a list.

Rotation

Task Idea	Charlotte Knierem
Task Preparation	Elias Bauer
Description English	Elias Bauer
Description German	Benjamin Schmid
Description French	Théo von Düring
Solution	Elias Bauer

In this task one was given a string S of lowercase letters of size $1 \leq N \leq 10^7$ as well as $1 \leq Q \leq 10^6$ queries. The queries were one of the following:

- Output the character at index i.
- Swap the characters at indices i and j.
- Repeat the following K times: Remove the last character of the array and insert it at the front. (Rotate the string K characters to the right)

Subtask 1: No modifications (10 Points)

In this subtask, there were only queries of type 1. This means that we only needed to output the character at a given index of the given string.

```
void solve(int t) {
    int N, Q;
    cin >> N >> Q;

    vector<char> s(N);
    for (auto& x : s)
        cin >> x;

    cout << "Case #" << t << ": ";

    while (Q--) {
        int type;
        cin >> type;

        int i;
        cin >> i;
        cout << s[i];
    }

    cout << "\n";
}
```

Subtask 2: Small Wheel (16 Points)

In this subtask, the limits on N and Q were very small ($1 \leq N \leq 100$ and $1 \leq Q \leq 100$), so we can just simulate the queries one by one.

```
void solve(int t) {
    int N, Q;
    cin >> N >> Q;

    vector<char> s(N);
    for (auto& x : s)
        cin >> x;

    cout << "Case #" << t << ": ";
```

```
10
11      while (Q--) {
12          int type;
13          cin >> type;
14
15          if (type == 1) {
16              int i;
17              cin >> i;
18              cout << s[i];
19          }
20          else if (type == 2) {
21              int i, j;
22              cin >> i >> j;
23              swap(s[i], s[j]);
24          }
25          else if (type == 3) {
26              int x;
27              cin >> x;
28
29              for (int i = 0; i < x; i++) {
30                  char last = s.back();
31                  s.pop_back();
32                  s.insert(s.begin(), last);
33              }
34          }
35      }
36
37      cout << "\n";
38  }
```

Subtask 3: No rotations (11 Points)

In this subtask, there were no queries of type 3, so we only needed to swap two elements of the array and output the i-th element. Therefore, even though the limits were large again, it was still possible to simulate the queries one by one.

```
1   void solve(int t) {
2       int N, Q;
3       cin >> N >> Q;
4
5       vector<char> s(N);
6       for (auto& x : s)
7           cin >> x;
8
9       cout << "Case #" << t << ": ";
10
11      while (Q--) {
12          int type;
13          cin >> type;
14
15          if (type == 1) {
16              int i;
17              cin >> i;
18              cout << s[i];
19          }
20          else if (type == 2) {
21              int i, j;
22              cin >> i >> j;
23              swap(s[i], s[j]);
24          }
25      }
26
27      cout << "\n";
28  }
```

Subtask 4: No swaps (27 Points)

The solution to this subtask is the same as subtask 5, just without the swap operations, therefore it is omitted.

Subtask 5: Escape (36 Points)

The main observation required to fully solve this task is the fact that the indices can be shifted when performing operations of types 1 and 2 instead of shifting the entire array for each query of type 3.

Let's call the total number of shift operations performed so far x.

When performing operations 1 and 2, indices are transformed as following: $i \to (i - x) \% N$, where % is the modulo operator.

(Implementation detail: Due to the fact that the `%` operator in C++ is actually the remainder operator and does not produce the desired results for negative numbers, we need to first add N to make sure the sum of which the remainder is taken is always positive.)

```
1  void solve(int t) {
2      int N, Q;
3      cin >> N >> Q;
4
5      vector<char> s(N);
6      for (auto& x : s)
7          cin >> x;
8
9      cout << "Case #" << t << ": ";
10
11     int total_shifts = 0;
12
13     while (Q--) {
14         int type;
15         cin >> type;
16
17         if (type == 1) {
18             int i;
19             cin >> i;
20             i = (i - total_shifts + N) % N;
21             cout << s[i];
22         }
23         else if (type == 2) {
24             int i, j;
25             cin >> i >> j;
26             i = (i - total_shifts + N) % N;
27             j = (j - total_shifts + N) % N;
28
29             swap(s[i], s[j]);
30         }
31         else if (type == 3) {
32             int x;
33             cin >> x;
34             total_shifts += x;
35             total_shifts %= N;
36         }
37     }
38
39     cout << "\n";
40 }
```

Landscaping

Task Idea	Johannes Kapfhammer
Task Preparation	Ferdinand Ornskov
Description English	Ferdinand Ornskov
Description German	Ferdinand Ornskov
Description French	Théo von Düring
Solution	Ferdinand Ornskov

Subtask 1: Finding pyramids (13 points)

In this subtask, you are only suppossed to check whether the given array already forms a valid pyramid sequence. Here, one can first realize that a pyramid sequence is only possible if the length of the array is odd, as there can only be one peak. Therefore, we can immediately exclude this case. Next, we can split the pyramid sequence into two parts, an increasing and a decreasing part. In the increasing case, we can realize that the height of the pyramid at position i is given by the formula $h[i] = i + 1$. Similarly, the height for the decreasing part can be computed by the formula $h[i] = N - i$. Putting it all together, we get that height at position i should be $h[i] = \min(i + 1, N - i)$. This condition can now be checked with the help of a for-loop.

```
1 def solve(N, h):
2     if N%2 == 0:
3         return "NO"
4     for i in range(N):
5         if h[i] != min(i+1, N-i):
6             return "NO"
7     return "YES"
```

Subtask 2: Explosive operations (19 points)

In this subtask, you don't only need to check whether a given array already forms a valid pyramid sequence but also try to use operations of type 1 to remove some numbers at the start (i.e. a prefix) and some numbers at the end (i.e. a suffix) of the array. However, since we are not allowed to modify any values in the array, any valid pyramid sequence must already exist as a subsequence of the array. Therefore, it only remains to check every possible subsequence to see if it forms a pyramid sequence. One initial idea to solve this problem might be to iterate over all possible start- and endpoints and then use the same code from subtask 1 to check if the subarray forms a pyramid sequence. Unfortunately, however, this takes up to $O(N^3)$ operations which is too slow. One way to circumvent this would be to start from the peak, then go left and right to check whether we get a valid pyramid sequence. This can be done quite easily as we know the height of the terrain j positions away from the peak (at position i). Namely, the height is given by $h[i - j] = h[i + j] = h[i] - j$. Again, this condition can be checked by a for-loop starting from every possible peak position i.

```
1 def solve(N, h):
2     max_length = 0
3     for i in range(N):
4         found_pyramid_sequence = False
5         for j in range(h[i]):
6             if i-j < 0 or i+j >= N:
7                 break
8             if h[i-j] != h[i]-j or h[i+j] != h[i]-j:
9                 break
10            if h[i-j] == 1:
11                found_pyramid_sequence = True
12
```

```
13        if found_pyramid_sequence:
14            max_length = max(max_length, 2*h[i]-1)
15
16    return max_length
```

Some things to watch out for when implementing this idea are, first of all, that you don't get any out-of-bounds errors when checking for a pyramid sequence that isn't fully contained in the array. Furthermore, a pyramid sequence is also only valid if it actually reaches 1 at both ends, so an extra condition checking for that is also necessary.

Subtask 3: Only chiseling (17 points)

In this subtask, you are now only allowed to perform operations of type 2 to landscape the terrain (i.e. decrease some values in the array) in order to form a new pyramid sequence. Here, one can first observe that the length of the array never changes. Hence, the only valid non-zero pyramid sequence has length N. This means that we already know what the array should look like in the end. And since the only operation we can do is to decrease elements in the array, we just need to check whether the terrain at each index is at least as high as the required pyramid sequence. We can implement this by slightly modifying the code from subtask 1.

```
1 def solve(N, h):
2     if N%2 == 0:
3         return 0
4     for i in range(N):
5         if h[i] < min(i+1, N-i):
6             return 0
7     return N
```

Subtask 4: Both operations (22 points)

For this subtask, we are now allowed to make use of both operations. From this, one might get the idea that we somehow have to combine the ideas from the previous two subtasks. And in fact, there is a way to do exactly this. When looking back at the changes we had to make from subtask 1 to subtask 3, we see that instead of checking for equality we only had to check if the terrain was higher than some certain desired height, as we could always chisel it down with operation 2. So, in some sense, we were only limited by the lowest points (relative to the desired height) of our terrain. This might lead us to now try a similar adaptation of our code for subtask 2. Again, we will try starting from every possible peak position and then going outwards from there, but now, instead of checking if all terrain heights match some desired height, we will check for the part of the terrain with the lowest relative height (i.e. the most restrictive point), as this, in the end, is what limits our sequence length. More formally, we know that if the height of the terrain at some position i is $h[i]$, then a peak a distance j away can have a height at most $h[i] + j$ and all the terrain height larger than that have to be chiseled down. So now, with these new insights, we can modify our code from subtask 2. But now, instead of keeping track of whether we found a valid pyramid sequence, we keep track of the maximum possible peak height if the peak would be located at position i. This height is given by taking the minimum over all maximum possible heights of the surrounding terrain. Such a maximum possible height from the terrain j positions away from the peak is given by max_possible_height = $\min(h[i-j], h[i+j]) + j$. And then we have to take the minimum over all distances j.

```
1 def solve(N, h):
2     max_length = 0
3     for i in range(N):
4         max_peak_height = h[i]
5         for j in range(h[i]):
6             if i-j < 0 or i+j >= N:
7                 break
8             max_peak_height = min(max_peak_height, h[i-j]+j, h[i+j]+j)
9
```

```
10          max_length = max(max_length, max_peak_height)
11
12      return max_length
```

Similar to subtask 2, we have to watch out for possible out-of-bounds errors here as well.

Subtask 5: Larger terrain (29 points)

Compared to the previous subtask, we are now only allowed to make up to $O(N)$ operations. Therefore, we might try to find some optimization which allows us to get rid of one of the for-loops in the subtask 4 code. As an initial step, we could try and have another look at our computation of max_possible_height. First, we can see that the computation can be split up into two parts: the computation of the longest increasing part and one for the longest decreasing part, and then taking the minimum of the two in the end. For this, we define two new variables max_height_increasing and max_height_decreasing. Since both of these cases can be treated analogously, we will only discuss the increasing case from here on out. When looking at a certain index j, we can see that our previous solution iterates over it potentially many times for each peak to the right of it, seemingly performing very similar calculations every time. After all, the maximum height of all the terrain to the left remains the same, regardless of what happens on the right. With this insight in hand, one might try to find a way to pass the calculations for one peak i onto the next peak $i+1$ on the right. Now, let's say we've found the maximum possible height for the increasing part at position i. Since the sequence has to be a staircase, we know the maximum possible height for the increasing part at position $i+1$ can be at most larger by one. It can, however, be lower if the terrain at index $i+1$ is lower than that. Furthermore, we can also decrease the other terrain to the left appropriately using operations of type 2. Therefore, the maximum height can be computed as follows: max_height_increasing$[i+1] = \min(h[i+1],$ max_height_increasing$[i]+1)$. Similarly, the values of max_height_decreasing$[i]$ can be calculated as well, but now going in reverse order instead. Taking the minimum of these two values then gives us the answer at each position i. Finally, taking the maximum over all these values gives us the desired result.

```
1  def solve(N, h):
2      max_height_increasing = [1]*N
3      max_height_decreasing = [1]*N
4
5      for i in range(0, N-1):
6          max_height_increasing[i+1] = min(h[i+1], max_height_increasing[i]+1)
7
8      for i in range(N-1, 1, -1):
9          max_height_decreasing[i-1] = min(h[i-1], max_height_decreasing[i]+1)
10
11     max_length = 0
12     for i in range(N):
13         max_length = max(max_length, min(max_height_increasing[i], max_height_decreasing[i])*2-1)
14
15     return max_length
```

Trampoline

Task Idea	Charlotte Knierim, Johannes Kapfhammer
Task Preparation	Anna Khanova
Description English	Anna Khanova
Description German	Benjamin Schmid
Description French	Cheng Zhong
Solution	Anna Khanova, Karolina Alexiou, Leo Chen

Observation 1. *The problem describes a directed acyclic graph (DAG) structure: each trampoline i (for $0 \leq i < n-1$) has exactly one outgoing edge to $i + j_i$. The last trampoline $(n-1)$ has no outgoing edges. Every edge goes from a trampoline with a lower index to a trampoline with a higher index which ensures that there are no cycles. We want to maximize the length of a path in this DAG, with variants allowing changes in starting points or even modifying one trampoline's jumpiness.*

Subtask 1: From the beginning (8 points)

If we must start from the first trampoline (index 0), we follow the unique path determined by j_i until we reach the last trampoline. The number of visited trampolines is simply the length of that path. Since there is no choice involved, the solution is straightforward:

- Start at 0, count trampolines along the path defined by $i \to i + j_i$ until reaching $n-1$.

Running time $O(n)$ is sufficient, as you just simulate the jumps until reaching the end.

Solution for Subtask 1:

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

int solve() {
    int n; cin >> n;
    vector<int> j(n);
    for (int i = 0; i < n; i++) cin >> j[i];

    int res = 1;
    int i = 0;
    while (i != n-1) {
        i =  i + j[i];
        res++;
    }

    return res;
}

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

    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        cout << "Case #" << t << ": " << solve() << "\n";
    }
}
```

Subtask 2: From the optimal trampoline (14 points)

Now we can start from any trampoline. We want to find the longest path in the DAG. Since each node has exactly one outgoing edge (except the last node), each trampoline defines a chain leading to the end. To find the longest path:

- Consider that for each node i, the path length to the end is $1 + \text{length}(\text{node } i + j_i)$.
- Compute these lengths. The maximum length over all nodes is the answer.

Running time The number of trampolines is small ($n \leq 100$), a simpler solution ($O(n^2)$) checking every start is feasible.

Subtask 3: More trampolines (29 points)

The logic from subtask 2 scales up to $n \leq 200\,000$. We must ensure an $O(n)$ solution. By storing $DP[i]$ = length of path starting at i, we have:

$$DP[n-1] = 1$$
$$DP[i] = 1 + DP[i + j_i]$$

Compute these in reverse order (from right to left). After computing all DP[i], the answer is $\max_i$ DP[i].

Running time This approach is $O(n)$: each trampoline is processed once.

Solution for Subtasks 2&3:

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

int solve() {
    int n; cin >> n;
    vector<int> j(n);
    for (int i = 0; i < n; i++) cin >> j[i];

    vector<int> dp(n, 0);
    dp[n-1] = 1;

    for (int i = n-2; i >= 0; i--) {
        dp[i] = dp[i + j[i]] + 1;
    }

    return *max_element(dp.begin(), dp.end());
}

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

    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        cout << "Case #" << t << ": " << solve() << "\n";
    }
}
```

Subtask 4: Adjusting one trampoline (14 points)

We now have the option to change the jumpiness of exactly one trampoline to any value we want (while still respecting $i + j_i < n$). We also choose the starting trampoline optimally.

One approach is to try all trampolines to adjust, and for each possible adjustment, compute the longest path again. Since n is small, this brute-force approach is feasible.

Running time $O(n^3)$: n trampolines, n possible jumpiness values, $O(n)$ to compute the longest path.

However, the following considerations can improve the complexity (and be useful for the next subtask).

When we introduce the possibility of changing the jumpiness of one trampoline, we have three cases when considering a starting trampoline i:

- We do not use any modification along the path starting at i. In this case, the result is simply DP[i].
- We modify a trampoline that we land on during the sequence (not the last one) to potentially reach a more advantageous path, thus extending the total count of visited trampolines. If we choose to use the modification at some trampoline, we can imagine that from this point onwards, the best we can do is take the best sequence achievable if we were able to choose any next step. This leads to a recurrence similar to the one described above, but now allowing a one-time "jump" improvement.
- We may also consider modifying the starting trampoline itself before making the first jump.

We don't need to consider all possible adjustments, but only the one which leads us to the best trampoline. By going from right to left we can maintain the value of the best trampoline (if there are many best, we can choose any).

Running time $O(n^2)$: n trampolines, $O(n)$ to compute the longest path, only one possible adjustment to consider.

Subtask 5: No limits (35 points)

For large n ($n \leq 200\,000$), we need an $O(n)$ approach. The key idea from subtask 4 still holds, but must be implemented efficiently:

By carefully combining three scenarios described above, we compute a second DP array, let's call it better, which considers at each trampoline the best possible outcome if we may adjust this trampoline or one of the next trampolines. To recompute this array we move from right to left and choose the maximum of:

- Using no adjustment at all (simply DP[i]),
- Using an adjustment at a future trampoline after this jump better[$i + j_i$] + 1,
- Using an adjustment on this trampoline to achieve a better jump (maximum over DP[k], where $i < k$, can be maintained in an extra variable through the iterations).

This DP allows us to find, for every starting position, the maximum achievable chain length when one trampoline can be adjusted. We then take the global maximum over all potential starting positions.

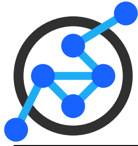
Running time $O(n)$ can be achieved by the careful implementation.

Solution for Subtasks 4&5:

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

int solve() {
```

```
6      int n; cin >> n;
7      vector<int> j(n);
8      for (int i = 0; i < n; i++) cin >> j[i];
9
10     // dp as in subtask 3
11     vector<int> dp(n, 0);
12     dp[n-1] = 1;
13
14     for (int i = n-2; i >= 0; i--) {
15         dp[i] = dp[i + j[i]] + 1;
16     }
17
18     // new dp with one improvement
19     vector<int> better(n, 0);
20     int best = 1;
21     for (int i = n-2; i >= 0; i--) {
22         // dp[i]: none | better[i + j[i]] + 1: use improved field | best + 1: improve this field
23         better[i] = max({dp[i], better[i + j[i]] + 1, best + 1});
24         best = max(best, dp[i]);
25     }
26
27     return *max_element(better.begin(), better.end());
28 }
29
30 signed main()
31 {
32     ios_base::sync_with_stdio(false);
33     cin.tie(0);
34
35     int T; cin >> T;
36     for (int t = 0; t < T; t++) {
37         cout << "Case #" << t << ": " << solve() << "\n";
38     }
39 }
```



Train

Task Idea	Cheng Zhong
Task Preparation	Cheng Zhong
Description English	Cheng Zhong
Description German	Benjamin Schmid
Description French	Cheng Zhong
Solution	Cheng Zhong, Johannes Kapfhammer, Yaël Arn

In this task, you're given M trains with their stop stations in range $0, 1, 2, ..., N-1$. There's a rule that says, if a train stops at one station, it'll also stop at all stations whose level is no less than it. You're asked to find the largest K, so that all the first K trains comply with the rule.

Subtask 1: Mini Railway Line (23 points)

We can work on the relationship of the stations: regard each station as a vertex and construct a graph; then check whether the graph complies with the rule. We use binary search to find K. For a given K, for each train, we can use $O(N^2)$ time to construct directed edges (low level station -> high level station). For all K trains, we need $O(N^2 \cdot K) \leq O(N^2 \cdot M)$ time. With a directed graph, we use $O(N^2)$ time to see if there's a circle (contradiction). Therefore, in total the time is $O(\max(N^2 \cdot M, N^2) \cdot \log(M)) = O(N^2 \cdot M \cdot \log(M))$.

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

const unsigned int RANDOM_PRIME_NUMBER = 1001029;

struct edge_hash {
    size_t operator() (const pair<int, int>& p) const {
        auto h1 = hash<int>{}(p.first);
        auto h2 = hash<int>{}(p.second);
        return
            static_cast<unsigned int>(h1) * RANDOM_PRIME_NUMBER +
            static_cast<unsigned int>(h2);
    }
};

int t, n, m;
int last_train;
unordered_set<pair<int, int>, edge_hash> edges;
vector<unordered_set<int>> train;
vector<int> starting;
vector<int> terminal;
unordered_map<int, vector<int>> graph;
unordered_map<int, int> visited;

void init() {
    cin >> n >> m;
    train.resize(m);
    starting.resize(m);
    terminal.resize(m);

    int stops_number;
    int cur_stop;
    for (int i = 0; i < m; i++) {
        cin >> stops_number;
        train[i].clear();
        train[i].reserve(stops_number);
        for (int j = 0; j < stops_number; j++) {
            cin >> cur_stop;
```

```
39              cur_stop--;
40              train[i].insert(cur_stop);
41              if (j == 0) {
42                  starting[i] = cur_stop;
43              } else if (j == stops_number - 1) {
44                  terminal[i] = cur_stop;
45              }
46          }
47      }
48  }
49
50  void process_input() {
51      edges.clear();
52      for (int i = 0; i <= last_train; i++) {
53          for (int j = starting[i]; j <= terminal[i]; j++) {
54              if (train[i].find(j) == train[i].end()) {
55                  for (int k : train[i]) {
56                      edges.emplace(j, k);
57                  }
58              }
59          }
60      }
61
62      graph.clear();
63      visited.clear();
64      for (const auto& edge : edges) {
65          graph[edge.first].push_back(edge.second);
66          visited[edge.first] = 0;
67          visited[edge.second] = 0;
68      }
69  }
70
71  bool dfs(int node) {
72      visited[node] = 1;
73      for (int neighbor : graph[node]) {
74          if (visited[neighbor] == 0) {
75              if (dfs(neighbor)) {
76                  return true;
77              }
78          } else if (visited[neighbor] == 1) {
79              return true;
80          }
81      }
82      visited[node] = 2;
83      return false;
84  }
85
86  bool can_toposort() {
87      process_input();
88      for (const auto& node : visited) {
89          if (node.second == 0) {
90              if (dfs(node.first)) {
91                  return false;
92              }
93          }
94      }
95      return true;
96  }
97
98  int max_last_train() {
99      int left = 0;
100     int right = m - 1;
101     while (left < right) {
102         last_train = (left + right + 1) / 2;
103         if (can_toposort()) {
104             left = last_train;
105         } else {
106             right = last_train - 1;
```

```
107         }
108     }
109     return left;
110 }
111 
112 int main() {
113     ios_base::sync_with_stdio(false);
114     cin.tie(nullptr);
115     cin >> t;
116     for (int i = 0; i < t; i++) {
117         init();
118         cout << "Case #" << i << ": " << max_last_train() + 1 << endl;
119     }
120     return 0;
121 }
```

Alternatively, we can also work on the relationship of the trains, to check if the rule been complied with. We add the trains one by one. While adding a new train, we compare it with all previous trains. In $O(N)$ time, we can tell if there's a contradiction between the new train and one of the previous trains: simply scan the common area of two trains and see if one is a subset of the other. If neither is a subset of the other, there's a contradiction. We can prove that if train A and B are compatible, B and C are compatible, C and A are compatible, then A, B, C together must be compatible. So, comparing all pairs is sufficient. This takes $O(M^2 \cdot N)$ time in total.

Subtask 2: Fixed Length Trains (15 points)

In this subtask, we work on the relationship of the trains. We use binary search to find K. For a given K, we first assume all trains comply with the rule, and then check if there's any contradiction. As all trains have the same starting station and terminal station, for any two trains A and B that are compplient with the rule we need to have that A's stops is either a subset or a superset of B's stops. In particular, if train A's number of stops is greater than train B's number of stops, A's stops must be a superset of B's stops.

Therefore, if we sort the trains based on the number of their stops in ascending order, the trains are actually ordered from the most "subset" to the most "superset". Then we use the following procedure to verify whether, in the sorted train list, the $(i+1)^{\text{th}}$ train is indeed a superset of the ith train (for all $0 \le i \le M-2$):

- At the beginning, there are N slots, representing the N stations. Initially they are all marked "empty".
- When we process a train, we update the slots from its starting station to its terminal station. If the train stops at a station, we change the corresponding slot to "stop", otherwise change it to "empty".
- During the whole process, if there's ever a slot being changed from "stop" to "empty", there must be a contradiction. Otherwise, everything is fine.

```
1 #include "bits/stdc++.h"
2 using namespace std;
3 
4 int t, n, m;
5 int last_train;
6 vector<vector<int>> train;
7 vector<unordered_set<int>> train_set;
8 
9 vector<pair<int, int>> index_and_size;
10 vector<int> slot;
11 
12 void init() {
13     cin >> n >> m;
14     int stops_number;
```

```
15      int cur_stop;
16      train.resize(m);
17      train_set.resize(m);
18      for (int i = 0; i < m; i++) {
19          cin >> stops_number;
20          train[i].clear();
21          train[i].reserve(stops_number);
22          train_set[i].clear();
23          train_set[i].reserve(stops_number);
24          for (int j = 0; j < stops_number; j++) {
25              cin >> cur_stop;
26              train[i].push_back(cur_stop);
27              train_set[i].insert(cur_stop);
28          }
29      }
30  }
31
32  void toposort() {
33      index_and_size.clear();
34      for (int i = 0; i < m; ++i) {
35          index_and_size.push_back({i, train[i].size()});
36      }
37      sort(index_and_size.begin(), index_and_size.end(),
38           [](const std::pair<int, int>& a, const std::pair<int, int>& b) {
39              return a.second < b.second;
40           }
41      );
42  }
43
44  bool comply() {
45      slot.resize(n);
46      for (int i = 0; i < n; i++) {
47          slot[i] = false;
48      }
49      for (int i = 0; i < m; i++) {
50          int train_index = index_and_size[i].first;
51          if (train_index > last_train) {
52              continue;
53          }
54          for (int j = train[train_index].front(); j <= train[train_index].back(); j++) {
55              if (train_set[train_index].find(j) != train_set[train_index].end()) {
56                  slot[j] = true;
57              } else {
58                  if (slot[j] == true) {
59                      return false;
60                  }
61              }
62          }
63      }
64      return true;
65  }
66
67  int max_last_train() {
68      int left = 0;
69      int right = m - 1;
70      while (left < right) {
71          last_train = (left + right + 1) / 2;
72          if (comply()) {
73              left = last_train;
74          } else {
75              right = last_train - 1;
76          }
77      }
78      return left;
79  }
80
81  int main() {
82      ios_base::sync_with_stdio(false);
```

```
83     cin.tie(nullptr);
84     cin >> t;
85     for (int i = 0; i < t; i++) {
86         init();
87         toposort();
88         cout << "Case #" << i << ": " << max_last_train() + 1 << endl;
89     }
90     return 0;
91 }
```

Subtask 3: Medium Railway Line (20 points)

Similar to last subtask, we use binary search to find K. For a given K, we assume all trains comply with the rule, and sort the trains from the most "subset" to the most "superset" similarly. Unlike the last subtask, our customized comparator of two trains is no longer comparing their total stops, but comparing the number of stops in their common area. If we pre-calculate a prefix array, we can get the number of stations in a given interval in $O(1)$, thus finish each comparison in $O(1)$.

Then we can use the same method as last subtask, to process the trains one by one from the most "subset" to the most "superset", and see if there's any contradiction. This takes $O((M+N)^2 \cdot \log(M))$ time in total.

We will share the base code of the last subtask and only replace the following two functions:

```
1  void init() {
2      cin >> n >> m;
3      int stops_number;
4      int cur_stop;
5      train.clear();
6      train.resize(m);
7      train_set.clear();
8      train_set.resize(m);
9      prefix.clear();
10     prefix.resize(m, vector<int>(n));
11     for (int i = 0; i < m; i++) {
12         cin >> stops_number;
13         train[i].clear();
14         train[i].reserve(stops_number);
15         train_set[i].clear();
16         train_set[i].reserve(stops_number);
17         for (int j = 0; j < stops_number; j++) {
18             cin >> cur_stop;
19             train[i].push_back(cur_stop);
20             train_set[i].insert(cur_stop);
21         }
22
23         if (train_set[i].find(0) != train_set[i].end()) {
24             prefix[i][0] = 1;
25         } else {
26             prefix[i][0] = 0;
27         }
28         for (int j = 1; j < n; j++) {
29             prefix[i][j] = prefix[i][j - 1];
30             if (train_set[i].find(j) != train_set[i].end()) {
31                 prefix[i][j]++;
32             }
33         }
34     }
35
36     edges.clear();
37     for (int i = 0; i < m - 1; i++) {
38         for (int j = i + 1; j < m; j++) {
39             int common_left = max(train[i].front(), train[j].front());
40             int common_right = min(train[i].back(), train[j].back());
```

```
41              if (common_left <= common_right) {
42                  int stops_i = prefix[i][common_right] -
43                                (common_left == 0 ? 0 : prefix[i][common_left - 1]);
44                  int stops_j = prefix[j][common_right] -
45                                (common_left == 0 ? 0 : prefix[j][common_left - 1]);
46                  if (stops_i < stops_j) {
47                      edges.emplace(i, j);
48                  } else if (stops_i > stops_j) {
49                      edges.emplace(j, i);
50                  }
51              }
52          }
53      }
54  }
55
56  bool toposort() {
57      topo_order.clear();
58      topo_order.reserve(m);
59      in_degree.clear();
60      in_degree.resize(m);
61      adj_list.clear();
62      adj_list.resize(m);
63      for (int i = 0; i <= last_train; i++) {
64          in_degree[i] = 0;
65          adj_list[i].clear();
66      }
67      for (const auto& edge : edges) {
68          int u = edge.first, v = edge.second;
69          if ((u <= last_train) && (v <= last_train)) {
70              adj_list[u].push_back(v);
71              in_degree[v]++;
72          }
73      }
74      queue<int> q;
75      for (int node = 0; node <= last_train; node++) {
76          if (in_degree[node] == 0) {
77              q.push(node);
78          }
79      }
80      while (!q.empty()) {
81          int node = q.front();
82          q.pop();
83          topo_order.push_back(node);
84          for (int neighbor : adj_list[node]) {
85              in_degree[neighbor]--;
86              if (in_degree[neighbor] == 0) {
87                  q.push(neighbor);
88              }
89          }
90      }
91      if (topo_order.size() != (last_train + 1)) {
92          return false;
93      }
94      return true;
95  }
```

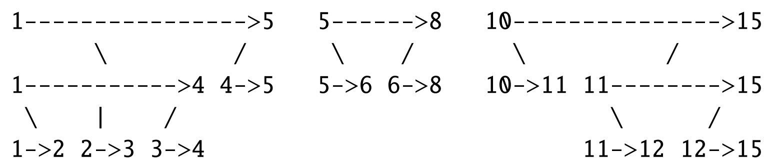
Subtask 4: Direct Trains (16 points)

See the first solution of the next subtask. Here the trains don't have any intermediate stop, one train corresponds to one hop. In this case, a valid forest must be toposortable. Therefore, we only need to implement Step 0 and Step 1.

Subtask 5: Long Railway Line (26 points)

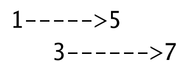
Similar to previous subtasks, we use binary search to find K. For a given K, we want to check if the first K trains together are valid.

- **Step 0: Preparing the hops**
  - For a train that has stops `1,2,4,9,15`, we say it has 4 hops: `1->2`, `2->4`, `4->9`, and `9->15`.
  - As `5->6` is fully contained in `4->9`, we say `4->9` is the parent of `5->6`.
  - We put the hops of all K trains into a set (not multiset).
- **Step 1: Checking if the hops (in the set) form a forest**
  - For example, a valid forest may look like:



```
1---------------->5    5------>8    10--------------->15
      \          /      \     /      \            /
1----------->4 4->5    5->6 6->8    10->11 11-------->15
 \    |    /                               \       /
1->2 2->3 3->4                             11->12 12->15
```

  - The purpose is to prevent two hops like:



```
1----->5
   3------>7
```

  - There're various ways to do the check. One way is: we sort all the hops by the starting point. For the hops with the smallest starting point (e.g. `1->5`, `1->4`, `1->2`), we know the longest (`1->5`) must be one of the roots and the second longest (`1->4`) must be a child of that root. With some recursive calculation, we can finish the check of this tree. And then we continue with the next hops.

    While checking the above scenario that we want to prevent (`1->5`, `3->7`), after we determine that `1->5` is the root node, as $3 \leq 5$, we would conclude that `3->7` is in the subtree rooted at `1->5`. However, as $7 > 5$, which means `3->7` is not fully contained in the root `1->5`, we find a contradiction.
- **Step 2: Checking if the graph of trains is toposortable**
  - We first define the topological order. For example:

```
Train A: 1->2, 2->3, 3------->5
Train B: 1------->3, 3------->5
```

    If train A has a hop a (e.g. `1->2`), and B has a hop b (e.g. `1->3`), such that a is a child of b (in the hops forest), we will construct a directed edge from train A to B.
  - The purpose is to prevent two trains like:

```
Train A: 1->2, 2->3, 3------->5
Train B: 1------->3, 3->4, 4->5
```

    which is not toposortable because we have both edges from A to B and from B to A.
  - There're various ways to do the check. One way is:

```
while (there're still remaining trains) {
    delete all the trains that only contain leaf hops
    (as defined in the hops forest);
    delete all the leaf hops (from the hops forest)
    no longer in any remaining train;
}
```

If in the end, all the trains are deleted, then they are valid. Otherwise if it is an infinite loop, they are not valid.

Overall, it can be proved that the time complexity is no more than $O(input \cdot \log(input))$. It can be further improved but it's not necessary for this subtask. To help understand the above algorithm, here are the example executions for a given K:

- **Example 1**

```
Train A: 1------->5
Train B: 1->3, 3---->7
```

  As hop `1->5` and `3->7` cannot form a forest, they are invalid.

- **Example 2**

```
Train A: 1->2, 2->3, 3------->5
Train B: 1------->3, 3->4, 4->5
```

  We have the hops forest:

```
1------>3    3------>5
  \     /      \     /
1->2 2->3    3->4 4->5
```

  No train can be deleted because none of the trains only contain leaf hops. So they are not valid.

- **Example 3**

```
Train A: 1->2, 2->3, 3------->5
Train B: 1------->3, 3------->5
```

  We have the hops forest:

```
1------>3    3------>5
  \     /
1->2 2->3
```

  As train A only contains leaf hops, it will be deleted. Then as `1->2` and `2->3` no longer exist in the remaining train, they're deleted from the hops forest. We will see the following remainder:

```
Train B: 1------->3, 3------->5
```

  with the updated hops forest:

```
1------>3    3------->5
```

  Then `1->3` becomes a leaf hop. Therefore, in the next iteration, train B will be deleted. In the end, all trains are deleted so they're valid.

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

const unsigned int RANDOM_PRIME_NUMBER = 1001029;
const int NONEXISTENT = -1;

struct hops_set_compare {
    bool operator()(const pair<int, int>& a, const pair<int, int>& b) const {
        if (a.first != b.first) {
            return a.first < b.first;
        } else {
            return a.second > b.second;
        }
    }
};
```

```
16
17 struct hops_map_hash {
18     std::size_t operator() (const std::pair<int, int>& p) const {
19         auto h1 = std::hash<int>{}(p.first);
20         auto h2 = std::hash<int>{}(p.second);
21         return static_cast<unsigned int>(h1) * RANDOM_PRIME_NUMBER +
22         static_cast<unsigned int>(h2);
23     }
24 };
25
26 struct train_type {
27     // index in the train array
28     int index;
29     // number of non-leaf hops
30     int count;
31
32     train_type(int index, int count) : index(index), count(count) {}
33
34     bool operator<(const train_type& other) const {
35         if (count == other.count) {
36             return index < other.index;
37         }
38         return count < other.count;
39     }
40 };
41
42 struct forest_node_type {
43     // index in the hops array
44     int index;
45     // number of children
46     int count;
47
48     forest_node_type(int index, int count) : index(index), count(count) {}
49
50     bool operator<(const forest_node_type& other) const {
51         if (count == other.count) {
52             return index < other.index;
53         }
54         return count < other.count;
55     }
56 };
57
58 int t, n, m;
59 int last_train;
60 set<pair<int, int>, hops_set_compare> hops_set;
61 vector<vector<int>> train;
62 set<train_type> train_set;
63 vector<int> train_hop_count;
64
65 int hops_number;
66 vector<pair<int, int>> hops;
67 vector<int> parent;
68 vector<vector<int>> children;
69 vector<int> children_count;
70 // hop : index in hops array
71 unordered_map<pair<int, int>, int, hops_map_hash> hops_map;
72 // index in the hops array -> index in the train array
73 vector<unordered_set<int>> hop_train;
74 // for each leaf, we wait for all relevant trains being removed, and then remove it
75 set<forest_node_type> forest_nodes_one;
76 // for each leaf, we immediately remove such hop in all relevant trains, and then remove it
77 set<forest_node_type> forest_nodes_two;
78
79 void init() {
80     cin>>n>>m;
81     int stops_number;
82     int cur_stop;
83     train.clear();
```

```
84      train.resize(m);
85      for (int i = 0; i < m; i++) {
86          cin>>stops_number;
87          train[i].reserve(stops_number);
88          for (int j = 0; j < stops_number; j++) {
89              cin>>cur_stop;
90              train[i].push_back(cur_stop);
91          }
92      }
93  }
94
95  void process_input() {
96      hops_set.clear();
97      for (int i = 0; i <= last_train; i++) {
98          for (int j = 0; j < train[i].size() - 1; j++) {
99              hops_set.emplace(train[i][j], train[i][j + 1]);
100         }
101     }
102
103     hops_number = hops_set.size();
104     hops_map.clear();
105     hops_map.reserve(hops_number);
106     int i = 0;
107     hops.clear();
108     hops.resize(hops_number);
109     for (const auto& p : hops_set) {
110         hops[i] = p;
111         assert(hops_map.insert({p, i}).second);
112         i++;
113     }
114     assert(i == hops_number);
115
116     parent.assign(hops_number, NONEXISTENT);
117     children.assign(hops_number, vector<int>());
118 }
119
120 // Return the index of the first hop after the current subtree, it should also
121 // be the root of the next tree.
122 // Return NONEXISTENT if we see any contradiction.
123 int construct_tree(int hop_index) {
124     int current_index = hop_index + 1;
125     int current_stop = hops[hop_index].first;
126     while ((current_index < hops_number) &&
127             (hops[current_index].first < hops[hop_index].second)) {
128         assert(hops[current_index].first >= current_stop);
129         if (hops[current_index].second > hops[hop_index].second) {
130             return NONEXISTENT;
131         }
132         parent[current_index] = hop_index;
133         children[hop_index].push_back(current_index);
134         current_stop = hops[current_index].second;
135         current_index = construct_tree(current_index);
136         if (current_index == NONEXISTENT) {
137             return NONEXISTENT;
138         }
139     }
140     return current_index;
141 }
142
143 bool construct_forest() {
144     int current_index = 0;
145     while (current_index < hops_number) {
146         current_index = construct_tree(current_index);
147         if (current_index == NONEXISTENT) {
148             return false;
149         }
150     }
151     return true;
```

```
152 }
153
154 void prepare_data() {
155     children_count.clear();
156     children_count.resize(hops_number);
157     for (int i = 0; i < hops_number; i++) {
158         children_count[i] = children[i].size();
159     }
160
161     train_set.clear();
162     train_hop_count.clear();
163     train_hop_count.resize(m);
164     for (int i = 0; i <= last_train; i++) {
165         train_set.insert(train_type(i, train[i].size() - 1));
166         train_hop_count[i] = train[i].size() - 1;
167     }
168
169     hop_train.assign(hops_number, unordered_set<int>());
170     for (int i = 0; i <= last_train; i++) {
171         for (int j = 0; j < train[i].size() - 1; j++) {
172             pair<int, int> current_hop = {train[i][j], train[i][j + 1]};
173             assert(hops_map.find(current_hop) != hops_map.end());
174             int hop_index = hops_map[current_hop];
175             hop_train[hop_index].insert(i);
176         }
177     }
178
179     forest_nodes_one.clear();
180     forest_nodes_two.clear();
181     for (int i = 0; i < hops_number; i++) {
182         forest_nodes_one.insert(forest_node_type(i, children[i].size()));
183         forest_nodes_two.insert(forest_node_type(i, children[i].size()));
184     }
185 }
186
187 void remove_leaf_from_train() {
188     while ((!forest_nodes_two.empty()) && (forest_nodes_two.begin()->count == 0)) {
189         for (int train_index : hop_train[forest_nodes_two.begin()->index]) {
190             train_set.erase(train_type(train_index, train_hop_count[train_index]));
191             assert(train_hop_count[train_index] > 0);
192             train_hop_count[train_index]--;
193             train_set.insert(train_type(train_index, train_hop_count[train_index]));
194         }
195         forest_nodes_two.erase(forest_nodes_two.begin());
196     }
197 }
198
199 void remove_leaf_from_forest(int hop_index) {
200     assert(children_count[hop_index] == 0);
201     assert(forest_nodes_one.erase(forest_node_type(hop_index, 0)) == 1);
202     if (parent[hop_index] != NONEXISTENT) {
203         int parent_index = parent[hop_index];
204         forest_node_type parent_node =
205                 forest_node_type(parent_index, children_count[parent_index]);
206         assert(forest_nodes_one.erase(parent_node) == 1);
207         assert(forest_nodes_two.erase(parent_node) == 1);
208         children_count[parent_index]--;
209         parent_node.count--;
210         forest_nodes_one.insert(parent_node);
211         forest_nodes_two.insert(parent_node);
212     }
213 }
214
215 void remove_zero_hop_train() {
216     while ((!train_set.empty()) && (train_set.begin()->count == 0)) {
217         int train_index = train_set.begin()->index;
218         for (int i = 0; i < train[train_index].size() - 1; i++) {
219             pair<int, int> current_hop = {train[train_index][i], train[train_index][i + 1]};
```

```
220             assert(hops_map.find(current_hop) != hops_map.end());
221             int hop_index = hops_map[current_hop];
222             assert(hop_train[hop_index].erase(train_index) == 1);
223             if (hop_train[hop_index].size() == 0) {
224                 remove_leaf_from_forest(hop_index);
225             }
226         }
227         train_set.erase(train_set.begin());
228     }
229 }
230
231 bool comply_rule() {
232     process_input();
233     if (!construct_forest()) {
234         return false;
235     }
236     prepare_data();
237     int last_forest_node_number = forest_nodes_one.size();
238     int last_train_number = train_set.size();
239     while ((last_forest_node_number > 0) || (last_train_number > 0)) {
240         remove_leaf_from_train();
241         remove_zero_hop_train();
242         if ((last_forest_node_number == forest_nodes_one.size()) &&
243             (last_train_number == train_set.size())) {
244             return false;
245         }
246         last_forest_node_number = forest_nodes_one.size();
247         last_train_number = train_set.size();
248     }
249     return true;
250 }
251
252 int max_last_train() {
253     int left = 0;
254     int right = m - 1;
255     while (left < right) {
256         last_train = (left + right + 1) / 2;
257         if (comply_rule()) {
258             left = last_train;
259         } else {
260             right = last_train - 1;
261         }
262     }
263     return left;
264 }
265
266 int main() {
267     ios_base::sync_with_stdio(false);
268     cin.tie(nullptr);
269     cin>>t;
270     for (int i = 0; i < t; i++) {
271         init();
272         cout << "Case #" << i << ": " << max_last_train() + 1 << "\n";
273     }
274     return 0;
275 }
```

Alternatively, we may regard a station as a vertex. In addition to the algorithm that constructs a graph of stations in the first subtask, we will also add each train as an intermediary "virtual vertex". To be specific: for a train, add edges from all the stop stations towards the "virtual vertex", and add edges from the "virtual vertex" towards all non-stop stations. With this strategy, we can avoid direct edges from stop stations to non-stop stations, and largely reduced the number of edges. However at this stage, the graph still has $O(N \cdot M)$ edges in the worst case, and cannot handle large amount of trains and stations within the time limit of this subtask.

To further reduce the time complexity, we can use a binary tree (e.g. segment tree, sparse table) over the stations. While constructing edges, instead of adding an edge from "virtual

vertex" towards each single non-stop station, we add an edge towards a bunch of consecutive stations.

For example, $N = 8$ and we have stations $0, 1, ..., 7$. We prepare the following binary tree. Each leaf node represents a station, each non-leaf node represents some consecutive stations.

```
[          0..7          ]
[    0..3   ] [    4..7   ]
[0..1] [2..3] [4..5] [6..7]
[0][1] [2][3] [4][5] [6][7]
```

If a train stops at stations $1, 4, 7$, then we will construct edges `[1]->v`, `[4]->v`, `[7]->v` and `v->[2..3]`, `v->[5]`, `v->[6]`. With this approach, we will have at most $O(input \cdot \log(N))$ edges, and thus reduce the time complexity to $O(input \cdot \log(N) \cdot \log(M))$. It can be further improved but this would be enough to solve this subtask.

Hiking Signs

Task Idea	Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer, Yaël Arn
Description English	Johannes Kapfhammer
Description German	Charlotte Knierim
Solution	Charlotte Knierim, Yaël Arn, Johannes Kapfhammer

Observation 1. *For a given tree with at least 3 vertices, the set of hiking signs is unique if and only if every vertex has at most one neighbor whose subtree does not have a landmark.*

Proof. To see that it is necessary to have at most one neighbor whose subtree does not have a landmark, assume there is a vertex v that has two neighbors u and w that do not have a landmark in their subtree. Then from both u and w, all landmarks are reached by going through v. As both u and w have distance 1 to v, this means their hiking signs are identical.

For the other direction, assume the hiking signs are not unique. Assume there is at least one landmark as else the condition is satisfied trivially. Then, we have two distinct vertices u and w that have identical hiking signs. Consider the unique path between u and w in the tree. If there was a landmark in the subtree of u or w that does not intersect the path, then the hiking signs cannot be identical as the distance from the landmark in the subtree below u or w cannot have the same distance to both.

Thus all landmarks need to be on the path or in subtrees branching of from the path. As u and w need to have equal distance to all landmarks, the only vertex on the path that can contain landmarks in its subtree is the one which is in the middle of the path at equal distance of u and w. If the path has an even number of vertices then such a vertex does not exists (and the existence of u and w implies that there are no landmarks in the tree). In case the path has an odd number of vertices, let v be the middle vertex and let s and t be its neighbors on the path (note that we can have $s = u$ and $t = w$). Observe that, from the perspective of v, neither the subtree of s nor the subtree of t can have a landmark, concluding the proof of the fact. □

Subtask 1: 13 Points

In this subtask we need to check if a given tree satisfies the condition mentioned above. Note that in here the answer will always be 0 or `impossible`. We can do this by starting a DFS in a vertex that contains a landmark (if none exists the answer is `impossible` as soon as the tree has 2 or more vertices) and count the number of subtrees that do not have a landmark.

```
#include<bits/stdc++.h>

using namespace std;

int dfs(int v, int p, vector<bool> &landmarks, vector<vector<int>> &g) {
  bool has_landmarks = landmarks[v];
  int no_landmark_count = 0;

  for (int w: g[v]) {
    if (w == p) continue;
    int is_marked = dfs(w, v, landmarks, g);
    if (is_marked == -1) return -1;
    if (is_marked == 0) no_landmark_count++;
    else has_landmarks = true;
  }

  if (no_landmark_count > 1) return -1; // too many unmarked subtrees
  else return has_landmarks;
```

```
}

signed main() {
  int t;
  cin >> t;

  for (int c = 0; c < t; ++c) {
    int n, m, k;
    cin >> n >> m >> k;

    vector<bool> landmarks(n);
    for (int i = 0; i < k; ++i) {
      int l;
      cin >> l;
      landmarks[l] = true;
    }
    vector<vector<int>> g(n);
    for (int i = 0; i < m; ++i) {
      int a, b;
      cin >> a >> b;
      g[a].push_back(b);
      g[b].push_back(a);
    }

    int res = -1;
    for (int i = 0; i < n; ++i) {
      if (landmarks[i]) {
        res = dfs(i, -1, landmarks, g);
        break;
      }
    }
    if (res == -1 && n >= 2) cout << "Case #" << c << ": Impossible" << '\n';
    else cout << "Case #" << c << ": 0" << '\n';
  }
}
```

This code runs in $O(n)$ and uses $O(n)$ memory.

Subtask 2: 15 Points

In this subtask, we have a tree on $M + 1$ vertices and $N - M - 1$ isolated vertices that are all landmarks.

If the tree has a landmark, we can then start a DFS from a landmark in the tree and identify all the vertices that violate the condition. Then we can fix this by adding an edge from a vertex not yet in the tree (that is guaranteed to be a landmark) to one of the subtrees causing the violation. Note that we have to be careful about the order in which we process the vertices to assure that we do not add too many edges. We do this by adding an edge to a landmark as soon as we violate the condition for the current vertex in the dfs. We will show in subtask 4 that it is always optimal to do it this way.

Otherwise, if $M = 0$, we are done, as one isolated vertex is not a problem. If $M > 0$, we can go through all vertices and see what happens if we connected it to a landmark. After the connection, the solution for a tree with landmark can be used to determine how many extra edges this would require. By taking the minimum over all vertices, we get the desired solution.

```
#include <bits/stdc++.h>

using namespace std;

bool dfs(int n, int p, int &free_landmark, vector<vector<int>> &g, vector<bool> &is_landmark,
         vector<pair<int, int>> &extra_edges) {
  int num_marked = 0, num_unmarked = 0;
  for (auto w: g[n]) {
```

```
9      if (w == p) continue;
10     bool is_marked = dfs(w, n, free_landmark, g, is_landmark, extra_edges);
11     if (is_marked) {
12       num_marked += 1;
13     } else if (num_unmarked == 1) {
14       // we already have an unmarked subtree, this one must have a sign
15       extra_edges.emplace_back(w, free_landmark++);
16       num_marked += 1;
17     } else {
18       num_unmarked += 1;
19     }
20   }
21   return num_marked > 0 || is_landmark[n];
22 }
23
24 int main() {
25   int t;
26   cin >> t;
27
28   for (int c = 0; c < t; ++c) {
29     int n, m, k;
30     cin >> n >> m >> k;
31
32     vector<bool> is_landmark(n, false);
33     for (int i = 0; i < k; ++i) {
34       int l;
35       cin >> l;
36       is_landmark[l] = true;
37     }
38     vector<vector<int>> g(m + 1);
39     for (int i = 0; i < m; ++i) {
40       int a, b;
41       cin >> a >> b;
42       g[a].push_back(b);
43       g[b].push_back(a);
44     }
45     cout << "Case #" << c << ": ";
46     vector<pair<int, int>> extra_edges;
47     bool has_landmark = false;
48     bool is_path = true;
49     for (int i = 0; i <= m; ++i) {
50       has_landmark = has_landmark || is_landmark[i];
51       is_path = is_path && g[i].size() < 3;
52     }
53     if (has_landmark) {
54       for (int i = 0; i <= m; ++i) {
55         if (is_landmark[i]) {
56           int extra_landmarks = m + 1;
57           dfs(i, i, extra_landmarks, g, is_landmark, extra_edges);
58           break;
59         }
60       }
61     }
62     else {
63       for (int i = 0; i <= m; ++i) {
64         if (g[i].size() > 0) { // if M = 0, we don't need a landmark
65           int extra_landmarks = m + 2;
66           vector<pair<int,int>> potential_extra_edges;
67           potential_extra_edges.push_back({m + 1, i});
68           dfs(i, i, extra_landmarks, g, is_landmark, potential_extra_edges);
69           if (potential_extra_edges.size() < extra_edges.size() || extra_edges.empty())
70             extra_edges = potential_extra_edges;
71         }
72       }
73     }
74
75     if (extra_edges.size() + m < n) {
76       cout << extra_edges.size() << '\n';
```

```
77        for (auto [a, b]: extra_edges)
78          cout << a << ' ' << b << '\n';
79      } else {
80        cout << "Impossible\n";
81      }
82    }
83  }
```

This code runs in $O(n^2)$ and uses $O(n)$ memory.

Actually, due to one wrong number in the judge, there was always at least one landmark in the tree, so the $O(n^2)$ part was never executed.

Subtask 3: 16 Points

In this subtask, we are given a collection of paths. For every of these paths we can check if its hiking signs are unique. Note that for paths, this is the case exactly when we have at least two landmarks or one landmark at the end of the chain. We pair up the paths violating the condition that have a landmark, connecting them by adding an edge between two of their endpoints. Note that we can have exactly one isolated vertex that is not a landmark and all paths that are longer need at least one landmark. We connect each path without a landmark (except for potentially the isolated vertex we can keep) to a path with at least one landmark.

```
1  #include "bits/stdc++.h"
2
3  using namespace std;
4
5  struct Path {
6    pair<int, int> endpoints;
7
8    bool has_landmark;
9    bool valid;
10 };
11
12 Path dfs(int n, vector<vector<int>> &g, vector<bool> &is_landmark, vector<bool> &vis) {
13   vis[n] = true;
14   Path res = {{-1, -1}, is_landmark[n], false};
15
16   for (auto w: g[n]) {
17     if (vis[w]) continue;
18     auto sub_path = dfs(w, g, is_landmark, vis);
19     res.endpoints = sub_path.endpoints;
20     if (res.has_landmark && sub_path.has_landmark) res.valid = true;
21     res.has_landmark = res.has_landmark || sub_path.has_landmark;
22     res.valid = res.valid || sub_path.valid;
23   }
24   if (g[n].size() < 2) {
25     if (res.endpoints.first == -1) res.endpoints.first = n;
26     else res.endpoints.second = n;
27     if (g[n].empty()) res.endpoints.second = n; // single vertex
28     if (is_landmark[n]) res.valid = true;
29   }
30   return res;
31 }
32
33
34 signed main() {
35   int t;
36   cin >> t;
37
38   for (int c = 0; c < t; ++c) {
39     int n, m, k;
40     cin >> n >> m >> k;
41
42     vector<bool> landmarks(n);
```

```
43     for (int i = 0; i < k; ++i) {
44       int l;
45       cin >> l;
46       landmarks[l] = true;
47     }
48     vector<vector<int>> g(n);
49     for (int i = 0; i < m; ++i) {
50       int a, b;
51       cin >> a >> b;
52       g[a].push_back(b);
53       g[b].push_back(a);
54     }
55     vector<bool> vis(n);
56     vector<Path> valid_paths;
57     vector<Path> empty_paths;
58     vector<Path> invalid_paths;
59     bool has_isolated_vertex = false;
60     for (int i = 0; i < n; ++i) {
61       if (g[i].size() < 2 && !vis[i]) {
62         auto path = dfs(i, g, landmarks, vis);
63         if (!path.has_landmark) {
64           if (path.endpoints.second == path.endpoints.first && !has_isolated_vertex) {
65             has_isolated_vertex = true;
66             continue;
67           }
68           empty_paths.push_back(path);
69         } else if (path.valid) {
70           // if we have a single-landmark valid path, don't connect wrong endpoints.
71           if (landmarks[path.endpoints.second]) swap(path.endpoints.first,path.endpoints.second);
72           valid_paths.push_back(path);
73         } else invalid_paths.push_back(path);
74       }
75     }
76     cout << "Case #" << c << ": ";
77     vector<pair<int, int>> extra_edges;
78     while (!invalid_paths.empty()) {
79       if (invalid_paths.size() >= 2) {
80         Path a = invalid_paths.back();
81         invalid_paths.pop_back();
82         Path b = invalid_paths.back();
83         invalid_paths.pop_back();
84         valid_paths.push_back({{b.endpoints.first, a.endpoints.first}, true, true});
85         extra_edges.push_back({a.endpoints.second, b.endpoints.second});
86       } else if (invalid_paths.size() == 1) {
87         if (valid_paths.empty()) {
88           cout << "Impossible\n";
89           break;
90         }
91         Path a = invalid_paths.back();
92         invalid_paths.pop_back();
93         Path b = valid_paths.back();
94         valid_paths.pop_back();
95         valid_paths.push_back({{a.endpoints.first, b.endpoints.first}, true, true});
96         extra_edges.push_back({a.endpoints.second, b.endpoints.second});
97       }
98     }
99     if (!invalid_paths.empty()) continue;
100    while (!empty_paths.empty()) {
101      if (valid_paths.empty()) {
102        cout << "Impossible\n";
103        break;
104      }
105      Path a = valid_paths.back();
106      valid_paths.pop_back();
107      Path b = empty_paths.back();
108      empty_paths.pop_back();
109      // be careful with single-sign paths
110      valid_paths.push_back({{a.endpoints.first, b.endpoints.first}, true, true});
```

```
111       extra_edges.push_back({a.endpoints.second, b.endpoints.second});
112     }
113     if (!empty_paths.empty()) continue;
114
115     cout << extra_edges.size() << '\n';
116     for (auto [a, b]: extra_edges)
117       cout << a << ' ' << b << '\n';
118
119   }
120 }
```

Subtask 4: 27 Points

In this subtask, we don't have any components without landmark. For components with a landmark, we define connection points:

Definition 1 (Connection Point)**.** We define a connection point to be a vertex which needs to be connected to a component with a landmark in order for the current component to be valid.

Observation 2. *The procedure from subtask 2 finds a minimal subset of connection points in a component with a landmark.*

Proof. Assume the contrary, that a set A of connection points exists such that it is smaller than the set generated in subtask 2. Root the tree at some landmark. Then, consider running the procedure as in subtask 2. Consider some vertex such that we add less connection points according to A in the subtrees which have no connection point or landmark yet. This vertex must exist, as in total we add less connection points, and we never add a connection point to a subtree which already contains a landmark or connection point in subtask 2. This vertex then violates the condition for a valid component which we discussed in observation 1 because we have at least two subtrees which do not contain a landmark or connection point. So A can not exist, and our procedure is optimal. □

To make the input forest valid, we thus can for each component find out all the connection points in the same way as in subtask 2. A component is valid exactly when it contains no connection points. So we see that when we add an edge between two connection points, we decrease the total number of connection points by two. If we connect a valid component and a connection point, we decrease the number of connection points by one. So we want to maximize the number of times we connect two connection points in order to minimize the number of edges. This can be done by always connecting the two components with the maximum amount of connection points into one new component. This way, we always avoid merging components into a valid component which can not be used any more for merging connection points by two. When only one invalid components remains, we repeatedly connect it to the other components. If at some point we run out of components to connect, we output impossible.

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  struct Component {
6    vector<int> connection_points;
7
8    int landmark;
9  };
10
11 bool dfs(int n, vector<vector<int>> &g, vector<bool> &is_landmark, vector<bool> &vis,
12      Component &comp) {
13   vis[n] = true;
14   int num_marked = 0, num_unmarked = 0;
15   for (auto w: g[n]) {
16     if (vis[w]) continue;
17     bool is_marked = dfs(w, g, is_landmark, vis, comp);
```

```
18      if (is_marked) {
19        num_marked += 1;
20      } else if (num_unmarked == 1) {
21        // we already have an unmarked subtree, this one must have a sign
22        comp.connection_points.push_back(w);
23        num_marked += 1;
24      } else {
25        num_unmarked += 1;
26      }
27    }
28    return num_marked > 0 || is_landmark[n];
29  }
30
31  int main() {
32    int t;
33    cin >> t;
34
35    for (int c = 0; c < t; ++c) {
36      int n, m, k;
37      cin >> n >> m >> k;
38
39      vector<bool> is_landmark(n, false);
40      for (int i = 0; i < k; ++i) {
41        int l;
42        cin >> l;
43        is_landmark[l] = true;
44      }
45      vector<vector<int>> g(n);
46      for (int i = 0; i < m; ++i) {
47        int a, b;
48        cin >> a >> b;
49        g[a].push_back(b);
50        g[b].push_back(a);
51      }
52      cout << "Case #" << c << ": ";
53      vector<bool> vis(n, false);
54      vector<Component> valid_components;
55      vector<Component> invalid_components;
56      for (int i = 0; i < n; ++i) {
57        if (is_landmark[i] && !vis[i]) {
58          Component comp;
59          comp.landmark = i;
60          dfs(i, g, is_landmark, vis, comp);
61          if (comp.connection_points.empty()) valid_components.push_back(comp);
62          else invalid_components.push_back(comp);
63        }
64      }
65      vector<pair<int, int>> extra_edges;
66      std::sort(invalid_components.begin(), invalid_components.end(),
67                [](Component &l, Component &r)
68            { return l.connection_points.size() < r.connection_points.size(); });
69      while (!invalid_components.empty()) {
70        if (invalid_components.size() == 1 && valid_components.empty()) break;
71        if (invalid_components.size() >= 2) {
72          Component a = invalid_components.back();
73          invalid_components.pop_back();
74          Component b = invalid_components.back();
75          invalid_components.pop_back();
76          extra_edges.push_back({a.connection_points.back(), b.connection_points.back()});
77          a.connection_points.pop_back();
78          b.connection_points.pop_back();
79          a.connection_points.insert(a.connection_points.end(), b.connection_points.begin(),
80                                     b.connection_points.end());
81          if (a.connection_points.empty()) valid_components.push_back(a);
82          else invalid_components.push_back(a);
83        } else if (invalid_components.size() == 1) {
84          Component a = invalid_components.back();
85          invalid_components.pop_back();
```

```
86          Component b = valid_components.back();
87          valid_components.pop_back();
88          extra_edges.push_back({a.connection_points.back(), b.landmark});
89          a.connection_points.pop_back();
90          if (a.connection_points.empty()) valid_components.push_back(a);
91          else invalid_components.push_back(a);
92        }
93      }
94      if (invalid_components.empty()) {
95        cout << extra_edges.size() << '\n';
96        for (auto [a, b]: extra_edges)
97          cout << a << ' ' << b << '\n';
98      } else {
99        cout << "Impossible\n";
100     }
101   }
102 }
```

This code runs in $O(n^2)$ because of the (inefficient) merging of the components and uses $O(n)$ memory.

It can be optimized to run in $O(n \log n)$ by using the smaller to larger technique, but this was not required in this task.

Subtask 5: 29 Points

In this subtask, we have to deal with the general version. We can combine observations from subtask 3 and subtask 4 to get the general solution. First, we see that one lone isolated vertex is still irrelevant, but we have to special-case it. Then, we can do the distinction between components without landmark, with landmarks and connection points, and with landmark but valid. For components with landmark, we can find out the connection points as in subtask 4. For components without landmark, we consider all possibilities of adding one connection point. Then, we continue as if it were a landmark and find the set of connection point this generates. Over all these possibilities, we take the minimal one. This takes $O(n^2)$ time, but this passes as $n \leq 10000$. Then, we must be careful how we connect these components. The general plan is to

1. connect empty-component connection points with each other
2. connect empty-component connection points to connection points of components with a landmark
3. connect empty-component connection points to valid components
4. carry on as in subtask 4

In the first three steps, we must be extremely careful in dealing with paths and lone vertices (which are a special case of paths). Considering them as having two connection points for the first two steps solves the problem. Each step 1. and 2. decreases the number of connection points by two, which is optimal. Further, we never create valid components as each empty component has at least two connection points, so we don't need to do the sorting first. Again, if at any point we run into problems, we output `Impossible`.

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  struct Component {
6    vector<int> connection_points;
7
8    int landmark;
9    bool path;
10 };
```

```
11
12 bool dfs(int n, vector<vector<int>> &g, vector<bool> &is_landmark, vector<bool> &vis,
13      Component &comp) {
14   vis[n] = true;
15   int num_marked = 0, num_unmarked = 0;
16   for (auto w: g[n]) {
17     if (vis[w]) continue;
18     bool is_marked = dfs(w, g, is_landmark, vis, comp);
19     if (is_marked) {
20       num_marked += 1;
21     } else if (num_unmarked == 1) {
22       // we already have an unmarked subtree, this one must have a sign
23       comp.connection_points.push_back(w);
24       num_marked += 1;
25     } else {
26       num_unmarked += 1;
27     }
28   }
29   return num_marked > 0 || is_landmark[n];
30 }
31
32 void componentDfs(int n, vector<vector<int>> &g, vector<bool> &vis, vector<int> &vertices) {
33   // find all vertices in a component
34   vertices.push_back(n);
35   vis[n] = true;
36   for (auto w: g[n]) {
37     if (vis[w]) continue;
38     componentDfs(w, g, vis, vertices);
39   }
40 }
41
42 int main() {
43   int t;
44   cin >> t;
45
46   for (int c = 0; c < t; ++c) {
47     int n, m, k;
48     cin >> n >> m >> k;
49
50     vector<bool> is_landmark(n, false);
51     for (int i = 0; i < k; ++i) {
52       int l;
53       cin >> l;
54       is_landmark[l] = true;
55     }
56     vector<vector<int>> g(n);
57     for (int i = 0; i < m; ++i) {
58       int a, b;
59       cin >> a >> b;
60       g[a].push_back(b);
61       g[b].push_back(a);
62     }
63     cout << "Case #" << c << ": ";
64     vector<bool> vis(n, false);
65     vector<Component> valid_components;
66     vector<Component> invalid_components;
67     vector<Component> empty_components;
68     bool has_isolated_vertex = false;
69     for (int i = 0; i < n; ++i) {
70       if (!vis[i]) {
71         vector<int> componentVertices;
72         componentDfs(i, g, vis, componentVertices);
73         Component comp{{}, -1, true};
74         for (auto e: componentVertices) {
75           if (is_landmark[e]) comp.landmark = e;
76           if (g[e].size() > 2) comp.path = false;
77         }
78         if (comp.landmark == -1) {
```

```
        // empty component
        if (componentVertices.size() == 1 && !has_isolated_vertex) {
          // special case one lone vertex
          has_isolated_vertex = true;
          continue;
        }
        // find best set of connection points
        for (auto e: componentVertices) {
          Component tempComp{{e}};
          for (auto f: componentVertices) vis[f] = false;
          dfs(e, g, is_landmark, vis, tempComp);
          if (comp.connection_points.empty() ||
              tempComp.connection_points.size() < comp.connection_points.size()) {
            comp.connection_points = tempComp.connection_points;
          }
        }
        if (comp.path) {
          // for paths, add both endpoints
          comp.connection_points.clear();
          for (auto e: componentVertices) {
            if (g[e].size() == 1) comp.connection_points.push_back(e);
            if (g[e].empty()) {
              comp.connection_points.push_back(e);
              comp.connection_points.push_back(e);
            }
          }
        }
        empty_components.push_back(comp);
      } else {
        for (auto f: componentVertices) vis[f] = false;
        dfs(comp.landmark, g, is_landmark, vis, comp);
        if (comp.connection_points.empty()) valid_components.push_back(comp);
        else invalid_components.push_back(comp);
      }
    }
  }
  vector<pair<int, int>> extra_edges;
  std::sort(invalid_components.begin(), invalid_components.end(),
            [](Component &l, Component &r)
        { return l.connection_points.size() < r.connection_points.size(); });
  while (empty_components.size() > 1) {
    // fist step, merge empty components
    Component a = empty_components.back();
    empty_components.pop_back();
    Component b = empty_components.back();
    empty_components.pop_back();
    extra_edges.push_back({a.connection_points.back(), b.connection_points.back()});
    a.connection_points.pop_back();
    b.connection_points.pop_back();
    a.connection_points.insert(a.connection_points.end(), b.connection_points.begin(),
                               b.connection_points.end());
    empty_components.push_back(a);
  }
  if (!empty_components.empty()) {
    // second and third step, merge empty component to non-empty component
    Component a = empty_components.back();
    empty_components.pop_back();
    if (!invalid_components.empty()) {
      Component b = invalid_components.back();
      invalid_components.pop_back();
      extra_edges.push_back({a.connection_points.back(), b.connection_points.back()});
      a.landmark = b.landmark;
    } else if (!valid_components.empty()) {
      Component b = valid_components.back();
      valid_components.pop_back();
      vector<bool> vis2(n, false);
      vector<int> vertices;
      componentDfs(b.landmark, g, vis2, vertices);
```

```
147          vector<int> endpoints;
148          for (auto e: vertices) {
149            if (g[e].size() <= 1) endpoints.push_back(e);
150          }
151          // preferably connect to a leaf without landmark
152          if (endpoints.size() >= 2 && is_landmark[endpoints[0]]) b.landmark = endpoints[1];
153          else b.landmark = endpoints[0];
154          extra_edges.push_back({a.connection_points.back(), b.landmark});
155          a.landmark = b.landmark;
156        } else {
157          empty_components.push_back(a);
158        }
159        // recompute connection points (because of paths)
160        for (auto [s, e]: extra_edges) {
161          g[s].push_back(e);
162          g[e].push_back(s);
163        }
164        if (a.landmark != -1) {
165          a.connection_points.clear();
166          vector<bool> vis2(n, false);
167          dfs(a.landmark, g, is_landmark, vis2, a);
168        }
169        if (a.connection_points.empty()) valid_components.push_back(a);
170        else invalid_components.push_back(a);
171      }
172      // continue as in subtask 4
173      while (!invalid_components.empty()) {
174        if (invalid_components.size() == 1 && valid_components.empty()) break;
175        if (invalid_components.size() >= 2) {
176          Component a = invalid_components.back();
177          invalid_components.pop_back();
178          Component b = invalid_components.back();
179          invalid_components.pop_back();
180          extra_edges.push_back({a.connection_points.back(), b.connection_points.back()});
181          a.connection_points.pop_back();
182          b.connection_points.pop_back();
183          a.connection_points.insert(a.connection_points.end(), b.connection_points.begin(),
184                                     b.connection_points.end());
185          if (a.connection_points.empty()) valid_components.push_back(a);
186          else invalid_components.push_back(a);
187        } else if (invalid_components.size() == 1) {
188          Component a = invalid_components.back();
189          invalid_components.pop_back();
190          Component b = valid_components.back();
191          valid_components.pop_back();
192          extra_edges.push_back({a.connection_points.back(), b.landmark});
193          a.connection_points.pop_back();
194          if (a.connection_points.empty()) valid_components.push_back(a);
195          else invalid_components.push_back(a);
196        }
197      }
198      if (invalid_components.empty() && empty_components.empty()) {
199        cout << extra_edges.size() << '\n';
200        for (auto [a, b]: extra_edges)
201          cout << a << ' ' << b << '\n';
202      } else {
203        cout << "Impossible\n";
204      }
205    }
206  }
```

This code again in $O(n^2)$ and uses $O(n)$ memory. It is also quite slow (~1 min), but this is enough to pass the subtask. There exist $O(n)$ solutions again using the smaller to larger technique for merging[1] and the rerooting technique for the optimal connection points in components without landmarks.

[1]Not $O(n \log n)$ because we merge almost everything into the largest component

Circuit

Task Idea	Johannes Kapfhammer, Mikhail Pyaderkin
Task Preparation	Mikhail Pyaderkin
Description English	Mikhail Pyaderkin
Description German	Charlotte Knierim
Description French	Cheng Zhong
Solution	Mikhail Pyaderkin

The tutorial below describes basic ideas how to approach the task. To get a higher score, you need to experiment with various approaches and optimizations on top of the basic idea and check which yield better results. It is not guaranteed that there exists a solution that can score 100 points.

Subtask 1: 20 points

Since $n = 1$ and all integers are positive, it essentially means that the best cross that we can get is just to take everything. Now our task is to design a circuit that computes the sum of input elements. Generally, if a task requires you to use a custom language (in this task you need to express your solution as a circuit), it is a good idea to first solve it using a standard language. In case you need to sum up the integers, the most basic approach is to do it with a for loop:

```
1 sum = 0;
2 for (i = 0; i < n; i++) {
3     sum = sum + in[0][i];
4 }
5 out = sum;
```

Let's now turn this into a circuit. We can not reuse the variable `sum` multiple times, since every variable might appear on the left side at most once. Instead, we need to create a new variable every time. Let's illustrate what needs to be done for $n = 4$. A program in a standard language would look like this:

```
1 sum = 0;
2 sum = sum + in[0][0];
3 sum = sum + in[0][1];
4 sum = sum + in[0][2];
5 sum = sum + in[0][3];
6 out = sum;
```

The corresponding circuit looks very similar:

```
1 sum_0 = 0
2 sum_1 = sum_0 + in[0][0]
3 sum_2 = sum_1 + in[0][1]
4 sum_3 = sum_2 + in[0][2]
5 sum_4 = sum_3 + in[0][3]
6 out = sum_4
```

For a given n, the resulting circuit contains n nodes, and the longest path between an input node and the output node is also n, since every node depends on the previous one. Luckily we don't have to sum the integers in order: we can instead design a circuit with a lower depth by rearranging the items. Indeed, to sum up 4 integers we can instead compute the sum of the first two, the sum of the last two, and then the result:

```
1 sum_01 = in[0][0] + in[0][1]
2 sum_23 = in[0][2] + in[0][3]
3 out = sum_01 + sum_23
```

To generalize this approach to higher n, we can use recursion: to compute the sum of n elements, split them into 2 halves, compute the sum of the first half, of the second half, and the result. This

corresponds to building a binary tree with the input elements as leaves. The corresponding circuit also has n nodes, but its depth is not greater than $\log_2 n + 1$.

Subtask 2: 20 points

The solution to the second subtask is not simpler than the one that can also solve the third one, so please see the solution for the third subtask.

Subtask 3: 20 points

In the second and third subtasks it might no longer be optimal to take everything, since there might be negative numbers, which means that computing the sum is not enough. Luckily, we can compute a bit more information to still be able to solve the task. Let's design a recursive function that for a given range of elements $[l, r)$ computes the following values:

1. $\texttt{sum}_{l,r}$ — the sum of all elements in this range,
2. $\texttt{pref}_{l,r}$ — the maximum sum among all prefixes of this range,
3. $\texttt{suf}_{l,r}$ — the maximum sum among all suffixes of this range,
4. $\texttt{ans}_{l,r}$ — the maximum sum among all subranges of this range.

If we compute all those values, than the answer to the problem is just $\texttt{ans}_{0,n}$. We are going to compute all these values for a given range $[l, r)$ the same way we computed the sum: recursively. If the range $[l, r)$ contains a single element, it means its value defines all 4 quantities `sum`, `pref`, `suf` and `ans`. Otherwise, let m be $\frac{l+r}{2}$. Let's compute these values for the range $[l, m)$ and for the range $[m, r)$. Then, we can compute the desired values as follows:

1. For $\texttt{sum}_{l,r}$ it's trivial: we just need to sum up the already computed values $\texttt{sum}_{l,m} + \texttt{sum}_{m,r}$.
2. For $\texttt{pref}_{l,r}$ it's a bit more involved. Among all prefixes of the range $[l, r)$, there are two groups:
   - the ones that are also the prefixes of $[l, m)$,
   - the ones that contain $[l, m)$ as a whole, and additionally a prefix of the range $[m, r)$.

   To pick the maximum prefix among the first group, we just need to look at $\texttt{pref}_{l,m}$. To pick the maximum prefix among the second group note that, they all contain the whole range $[l, m)$, so the only difference between them is what we pick as a prefix of $[m, r)$ and luckily we already know the best one there — it's $\texttt{pref}_{m,r}$. The final formula is $\texttt{pref}_{l,r} = \max(\texttt{pref}_{l,m}, \texttt{sum}_{l,m} + \texttt{pref}_{m,r})$,
3. Following similar logic, $\texttt{suf}_{l,r} = \max(\texttt{suf}_{l,m} + \texttt{sum}_{m,r}, \texttt{suf}_{m,r})$.
4. The most interesting fact is how to compute $\texttt{ans}_{l,r}$. Among all the subsegments of $[l, r)$ there are three groups:
   - the ones that are also subsegments of $[l, m)$,
   - the ones that are also subsegments of $[m, r)$,
   - the ones that do not belong to the first two groups.

   To pick the maximum subsegment among the first group, it's enough to look at $\texttt{ans}_{l,m}$. Similarly for the second group we already know the answer — it is $\texttt{ans}_{m,r}$. Note that for the third group, every subsegment that is not contained as a whole in either of the halves intersects the middle point m, meaning it consists of a suffix of the range $[l, m)$ and a prefix of the range $[m, r)$. Since we already know the maximum suffix and the maximum prefix of the corresponding ranges, we can easily compute the answer: $\texttt{ans}_{l,r} = \max(\texttt{ans}_{l,m}, \texttt{ans}_{m,r}, \texttt{suf}_{l,m} + \texttt{pref}_{m,r})$.

This recursive approach is very similar to the first subtask, and the resulting circuit also has $O(n)$ nodes and $O(\log_2 n)$ depth.

Subtask 4: 20 points

If all integers on the grid are non-negative, then it must consists of an entire row and an entire column, because any other cross can be extended without decreasing the sum. To compute the sum in each row we can use the same recursive algorithm as we used in the first subtasks, let $\texttt{row}_r$ be the sum of the elements in the row r. Similarly we can compute $\texttt{col}_c$ — the sum for every column. Given a cell (r, c), the optimal cross having (r, c) as the center cell has the sum $\texttt{best}_{r,c} = \texttt{row}_r + \texttt{col}_c - \texttt{in}_{r,c}$. Now we have $n \cdot m$ integers $\texttt{best}_{r,c}$ and we need to find the maximum among them. The easiest way is again to use recursion: similarly to the sum, we can compute the maximum among k elements using a circuit of size $O(k)$ with depth $O(\log_2 k)$. Given we have $O(n \cdot m)$ elements stored in `best`, we need a circuit of size $O(n \cdot m)$ with depth $\log_2 (n \cdot m)$. Given that we also need to compute the sum in each row and in each column, we end up with a circuit of size $O(n \cdot m)$ with depth $O(\log_2 n + \log_2 m)$.

Subtask 5: 20 points

Following the solution for the previous subtask one can notice that for a given cell (r, c) we just need to find the optimal horizontal segment containing this cell, find the optimal vertical segment containing this cell, and sum them up to compute $\texttt{best}_{r,c}$. Then we can use the same approach as before to compute the maximum among all values efficiently. The vertical and horizontal segments can be found independently, so we reduced the task to the following: given an array (that represents either a column or a row), for each index i compute the subsegment with the maximum sum that contains the element at index i.

There are multiple approaches to achieve that, and while the one described below does not give you the smallest depth and as such not the highest number of points, its depth is still asymptotically $O(\log_2 n + \log_2 m)$, and its size is $O(n \cdot m)$, and has a very short implementation.

Note that any segment containing an index i contains of a suffix of the range $[0, i)$, an element i and a prefix of the range $[i, n)$. Since computing the optimal suffix is the same as computing the optimal prefix but for a reversed array, it is enough to know how to compute the optimal prefix for each i and reuse the code.

To compute the optimal prefix for each i we can use the same idea as for the subtasks 2 and 3. Essentially with the recursive approach we build a segment tree, where for each node we store the sum, the maximum prefix, the maximum suffix and the maximum subsegment. Such nodes can be easily merged, the same way we recursively compute them, so indeed we can build a segment tree. Then for every prefix we can just query the segment tree.

This approach is a bit wasteful, since if we need to compute the answer for all prefixes, we combine the same segment tree nodes into the answer multiple times. Let's iterate over all the prefixes from left to right and to compute the answer for the prefix ending with position i, we need to know the longest segment tree node that ends in position i (that corresponds to the last node that will used in a normal segment tree node), let this node correspond to the range $[l, i)$. Since we already computed the answer for prefix ending at position l, we can easily combine it with the sum on the range $[l, i)$ to compute the answer for our prefix ending at position i.

Additional optimizations can include a smarter way of reusing information between prefixes and suffixes, or by simply caching the results: if for some reason we need to create a node that computes the sum or maximum of two values, let's first check if there isn't already a node that does exactly that.

Cake icing

Task Idea	Linus Lüchinger, Johannes Kapfhammer
Task Preparation	Linus Lüchinger, Benjamin Schmid
Description English	Linus Lüchinger, Hannah Oss
Description German	Hannah Oss, Linus Lüchinger
Solution	Linus Lüchinger

By the nature of creativity tasks, a big variety of solutions can work well on this task. In order to limit the scope of this analysis, we will focus on strategies which define some scoring function to rate pours and then search for pours maximizing that scoring function. Note that many strategies, even if not originally intended this way, can be formalized like this. For example, a greedy strategy which simply takes the largest piece on every move can be viewed as the described strategy with the score of a pour being the importance of the most important covered piece.

These strategies must perform two main steps:

- Define a scoring function $S(C, P)$ which given the state of a cake C and a pour P, gives a score of how good playing that pour on the cake is.
- Find a way to come up with pours that give high scores given a specific cake.

Scoring function

A good scoring function needs to consider at least the following facts:

1. It is better to ice pieces with higher importance.
2. It is better to ice pieces iced by the opponent rather than un-iced ones.
3. It is useless to ice pieces which are currently iced by ourselves.
4. A piece should not receive too little icing as the opponent would likely ice that piece again, yielding our pour useless.
5. A piece should not receive too much icing as that leaves less icing to claim other pieces.
6. Pieces should be iced as early as possible in order to not give a chance to the opponent to ice them, which would make it more expensive to re-ice them.

One can notice that the exact combination of pieces involved in a pour isn't of great importance. Rather, we can say that the score of a pour is the sum of scores of the pieces iced in the pour. For each piece we can give a score of how good it is to claim the piece with a given thickness. We denote this scoring function for pieces as $s_C(p, t)$ where C is the state of the cake, p is the piece and t is the thickness with which the piece is to be iced. The score of a pour P can then be calculated as $S(C, P) = \sum_{p \in P.\text{pieces}} s_C(p, P.\text{thickness})$ where we denote the set of pieces in a pour as P.pieces and the thickness of a pour as P.thickness. This is a restriction in how the scoring function is allowed to work and thus does not allow for some scoring functions which one might want. However, we argue that good scoring functions can quite well be approximated with such a sum scoring function (as well as a penalty for using much icing which will be discussed later). The reason we want such as sum scoring function is that it greatly simplifies pour search.

We can implement the requirements $1-3$ in our piece-scoring function s_C as follows: We define a value v for each piece: 0 if the piece is currently iced by us, importance if it is uniced and $2 \cdot$ importance if it is iced by the opponent. The factor of 2 arises because icing that piece adds importance to our score and subtracts importance from the opponent's score, resulting in an effective gain of $2 \cdot$ importance. We then make $s_C(p, t)$ proportional to the pieces v. For

requirement 4, we calculate an ideal icing amount for each piece (for example by distributing the remaining icing on all pieces according to their value) and let the score be very small (or 0) if the icing spent on a piece is significantly below the ideal icing amount. The exact method for calculating this should be determined through bot-specific testing, as theoretical considerations alone are insufficient. For requirement 5, we can either also add a penalty in case the icing spent on a piece is significantly above the ideal amount of icing or rely on the fact that if a pour significantly overinvests on many pieces, there will be another pour which has a better score as overinvesting does not result in additional score. An example of a piecewise scoring function following the discussed concepts is:

$$s_C(p,t) = \begin{cases} p.v \text{ if } 3^{t-1} > p.\text{importance} \cdot \text{ipi} \\ p.v \cdot \frac{p.\text{importance} \cdot \text{ipi} - 3^{t-1}}{3^t - 3^{t-1}} \text{ if } 3^{t-1} < p.\text{importance} \cdot \text{ipi} < 3^t \\ 0 \text{ otherwise} \end{cases}$$

where ipi (icing per importance) is the remaining icing of our bot plus the remaining icing of the opponent divided by the sum of importances of pieces which are not claimed with a sufficient thickness (so basically the icing that should roughly be spent per importance), $p.v$ is the pieces value as defined above and $p.$importance is the pieces' importance.

Note that with this kind of scoring function, the optimal pour would likely be one spending nearly the entire remaining icing. To prevent this, one can either limit the amount of icing on a specific turn or calculate an adjusted score of a pour based on the sum score and the amount of icing used in the pour (by dividing the sum score by the amount of icing used to the power of some constant between 0 and 1 or something similar). Both of these options are efficiently implementable with the move search we will discuss. Note that in the first case, one needs to decide how much icing should be available in each turn. This is where we can implement requirement 6 by making more icing available for early turns.

Pour search

As discussed, we only consider score functions where the score of a pour is the sum of the scores of the pieces in the pour. Thus, pour search consists of finding a sequence of pieces where consecutive pieces are adjacent such that the sum of scores of the pieces is maximized. Note that as the score for a piece depends on the thickness of the pour, this process needs to be repeated for multiple thicknesses. Since the amount of possible thicknesses is logarithmic in the icing amount this is acceptable.

There are many standard optimization techniques that work for this problem such as greedy, evolutionary algorithms, ant colony optimization, simulated annealing... One can also use many different heuristics such as using a sequence of moves from high scoring pieces to other high scoring pieces while maximizing the sum of scores on the path between the two. In this analysis, a dynamic programming approach will be discussed in more detail:

The core idea is to restrict the problem such that it can easily be solved with dynamic programming: What is the best pour if one is only allowed to move down, to the left and to the right. This can be solved with the following dp: $l[k][x][y]$ = maximum score of a pour of length k starting from the piece (x,y) while only going to the left on the yth row and $r[k][x][y]$ = the same but only going to the right on the yth row. The observation to be made is that on each row, one can go either only to the right or only to the left but never both. Thus, when going down to a new row, one can choose whether to go to the left or to the right on that row but after that, one can only either follow the chosen direction or go down one row and choose a direction again. Thus $l[k][x][y] = \text{score}[x][y] + \max(l[k-1][x-1][y], \max(l[k-1][x][y-1], r[k-1][x][y-1]))$ and $r[k][x][y] = \text{score}[x][y] + \max(r[k-1][x+1][y], \max(l[k-1][x][y-1], r[k-1][x][y-1]))$. The score of the best pour under the additional constraints will then be in some $r[k][x][y]$ or $l[k][x][y]$. In order to also be able to reconstruct the pour resulting in that score, one can save

the first move direction in the optimal pour along with its score in the dp table. To get the second move direction, one can look at $r/l[k-1][x_n][y_n]$ where x_n and y_n are the coordinates of the piece reached after executing that first move. The rest of the pour can be reconstructed analogously. Searching for pours under these constraints in most cases still results in decent pours (i.e there is often a good pour which does not require moving up from a piece).

However, since the maps are small in this task, the bot usually has time for additional calculations. This can be taken advantage of by considering multiple different constrained version which can all be solved with a similar dp and then taking the best pour found under any constraints. To get alternative constraints which can still be solved with a similar dp solution, we generalize the constraint of not being able to move up: Instead of dividing the cake into rows and only allowing to move from a higher row to a lower one, we divide the cake into trees and assign each tree a 'height'. An example of how to divide the cake into trees can be found in Figure 1. From a piece, one is then only allowed to move along a tree edge or to a neighboring tree with a lower height. Under these constraints, one only has to pay attention to not go back to the piece that one came from, just as before, we needed to pay attention to not go left immediately after going right or vice versa. If this is obeyed, it is impossible to visit a piece twice in a pour and thus impossible to double count it in the dp. Our dp state then again consists of the x and y of the starting piece and the length of the pour k, but now additionally also stores the direction d in which we are not allowed to move in as our first move (instead of having two separate dps each with a direction which is not allowed as the first move as before). As before, we can calculate $\text{dp}[k][x][y][d]$ from $\text{dp}[k-1]$ by going through all the neighbors we are allowed to move to (remember that we also have the tree edge and tree "height" constraints) and which are not in the direction d and taking the maximum of $\text{dp}[k-1][x_{\text{neighbour}}][y_{\text{neighbour}}][d_{\text{from neighbour to current piece}}]$. Reconstructing the pour works the same as before.

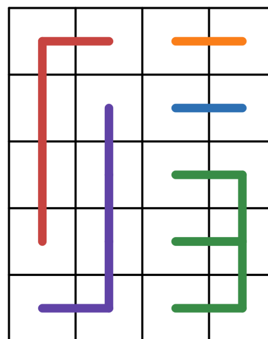


Figure 1: An example of how to divide the cake into trees.

The division of the cake into trees should happen in a randomized manner and be repeated multiple times, as long as the bot still has time. The trees should not be too small (as the "height" constraint is too constraining in that case) as well as not too large (as the tree edge constraint is too constraining in that case). The optimal size for each tree is probably around $\min(W, H)$. The height assignment to the trees should not happen completely randomly as that would mean there are lots of "sinks" (trees with no lower neighboring tree) and "sources" (trees with no higher neighboring tree). Sinks are undesirable as pours starting in that tree do not have an option to leave the tree and consequently not much freedom and thus probably no very good option. Sources are undesirable as there is a lot of freedom for pours starting in that tree, more than probably neccesary and it would be better to give some freedom to a neighboring tree by making it higher than the source. One way to achieve a good height assignment is to run bfs (on the graph of trees, where neighboring trees have an edge) from a tree on the edge of the cake and assign heights according to the order in which all trees are visited.