May 15, 2025

New Zealand Informatics Competition 2025

Round One

Editorial by Anatol C, Zalan V

Contents

Introduction

The solutions to this round of NZIC problems are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

Resources

Ever wondered what the error messages mean?

https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Ever wondered why your submission scored zero?

https://www.nzoi.org.nz/nzic/resources/why-did-i-score-zero.pdf

See our list of other useful resources here:

https://www.nzoi.org.nz/nzic/resources

Midnight Snack

Problem authored by Zalan V & Anatol C
https://train.nzoi.org.nz/problems/1531

Subtask 1

In this subtask, Andrew can eat all cookies with tastiness of at least 1. Since all cookies have tastiness of at least 1, he simply eats all cookies. Therefore, the total tastiness of the cookies he eats is the sum of the tastiness of all cookies.

Python Solution

```
N = int(input())
K = int(input())

total_tastiness = 0
for i in range(N):
  tastiness = int(input())
  total_tastiness += tastiness

print(total_tastiness)
```

Full Solution

In this case, we need to only count cookies with tastiness of at least K. We can achieve this with almost the same code as before. The only difference is that before adding a cookie's tastiness to the total, we check that its tastiness is at least K.

Python Solution

```
N = int(input())
K = int(input())

total_tastiness = 0
for i in range(N):
  tastiness = int(input())
  if tastiness >= K:
    total_tastiness += tastiness

print(total_tastiness)
```

C++ Solution

```
#include <iostream>
using namespace std;
int main() {
    int n, k;
    cin >> n >> k;
    int total = 0;
    for (int i = 0; i < n; i++) {
        int t;
        cin >> t;
        if (t >= k) total += t;
    }
    cout << total << "\n";
}
```

Carpentry

Problem authored by Iván G
https://train.nzoi.org.nz/problems/1449

Subtask 1

In this subtask there are always exactly 3 logs. This means that the staircase can have size at most 3. Note that we can always make a staircase of size 1. So, we only need to check if we can make a staircase of size 2 or 3. So, we can use the following method to find the answer:

- You can make a staircase of size 3 if there is one log of size at least 3, and another log of size at least 2).
- Otherwise, you can make a staircase of size 2 if there is one log of size at least 2.
- Otherwise, you can make a staircase of size 1.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 2

There is no specific intended solution to this subtask, but extending a solution to Subtask 1 can solve this subtask.

Subtask 3

In this subtask $N \leq 1,000$. To solve this subtask, we can repeatedly take the smallest log and try to make our staircase taller if we can. We can simply find and remove the smallest log in $O(N)$. The overall time complexity of this solution is $O(N^2)$

Python Solution

```
N = int(input())
logs = list(map(int, input().split()))
height = 0
for i in range(N):
    smallest = min(logs)
    logs.pop(logs.index(smallest))
    # We can only use this log if it is taller than the current height
    if height < smallest: height += 1
print(height)
```

Full Solution

For the full solution we can improve the time complexity of the Subtask 3 solution by sorting the logs before we process them. The time complexity of this solution is $O(N)$.

Python Solution

```
N = int(input())
logs = sorted(map(int, input().split()))
height = 0
for log in logs:
    if log <= height: continue
    height += 1
print(height)
```

Repeat Repeat

Problem authored by Anatol C
https://train.nzoi.org.nz/problems/1510

Subtask 1

In this subtask $N \times M \leq 100$. There is no specific intended solution for this subtask, but a number of approaches with a time complexity of $O(N^2)$ or $O(N^3)$ will pass this subtask.

Subtask 2

In this subtask we are given that $M = 2$. Thus, a plausible guess can be formed by repeating the first half of T. Then, the value of X will be the number of i such that the i-th and $(i + N)$-th characters differ.

Python Solution

```
N, M = map(int, input().split())
T = input()

X = 0
for i in range(N):
    if T[i] != T[i + N]: X += 1
print(X)
print(T[:N] * 2)
```

Subtask 3

In this subtask $N = 1$. Thus, we can form a plausible guess by finding the most common character in T and repeating it M times. We can count the number of occurrences of each letter `A` through `Z` and find the most common one in $O(N)$. The overall time complexity of this solution is $O(N)$.

Python Solution

```
N, M = map(int, input().split())
T = input()

LETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

counts = [T.count(letter) for letter in LETTERS]
most_frequent = 0
for i in range(26):
    if counts[i] > counts[most_frequent]: most_frequent = i
print(M - counts[most_frequent])
print(M * LETTERS[most_frequent])
```

Subtask 4

In this subtask T consists of only `A` or `B` characters. Thus, for each $0 \leq i < N$ we can find whether `A` or `B` is the most common character in the substring $T[i], T[i + N], T[i + 2N], \ldots$ and a plausible guess can be formed by repeating the most common character in each position.

Python Solution

```
N, M = map(int, input().split())
T = input()

X = 0
S = ""
```

```
for i in range(N):
    a_count = T[i::N].count('A')
    if a_count >= M / 2:
        S += 'A'
        X += M - a_count
    else:
        S += 'B'
        X += a_count
print(X)
print(S * M)
```

Full Solution

Similarly to Subtask 4, for the full solution we need to find the most common character in each of the substrings $T[i], T[i+N], T[i+2N], \ldots$ then we can form a plausible guess by repeating the most common character in each position M times. The overall time complexity of this solution is $O(N)$.

Note: Appending to the end of a string in Python is $O(1)$ when submitting with CPython and $O(N)$ when submitting with PyPy. Thus, this solution when submitted using PyPy will run in $O(N^2)$, but should still be fast enough to pass.

Python Solution

```
from collections import Counter

N, M = map(int, input().split())
T = input()

S = ""
X = 0
for i in range(N):
    letter, count = Counter(T[i::N]).most_common(1)[0]
    S += letter
    X += M - count
print(X)
print(S * M)
```

Safe Crossings

Problem authored by Joseph G
https://train.nzoi.org.nz/problems/1513

Subtask 1

In this subtask $M = 1$ so there is only one lane. Thus, we can simply find the maximum safety score of all gaps. One edge case that we need to be wary of is that because we do not start in lane 1 but in front of it, and crossing into a lane takes one second, the earliest that we can arrive into the first lane is at time 1. The time complexity of this solution is $O(N)$.

Python Solution

```
N, M = map(int, input().split())
gaps = [tuple(map(int, input().split())) for _ in range(N)]
max_safety = 0
for l, s, e in gaps:
    max_safety = max(max_safety, e - max(s, 1))
print(max_safety)
```

Subtask 2

In this subtask there is exactly one gap per lane. Thus, the optimal solution will be to always cross into the next lane as early as possible. Whenever we cross into the next lane, we can calculate the safety score achieved based on when we cross into the next lane. Our answer will then be the minimum of the safety scores in each lane. The time complexity of this solution is $O(N)$.

Python Solution

```
N, M = map(int, input().split())
gaps = sorted(tuple(map(int, input().split())) for _ in range(N))
curr_gap_start = 0
curr_gap_end = 10**9
min_safety = 10**9
for l, s, e in gaps:
    curr_gap_start = max(curr_gap_start + 1, s)
    safety = curr_gap_end - curr_gap_start + 1
    if safety <= 0:
        print(0)
        break
    min_safety = min(min_safety, safety)
    curr_gap_end = e
else:
    min_safety = min(min_safety, curr_gap_end - curr_gap_start)
    print(min_safety)
```

Subtask 3

In this subtask the maximum size of any gap is 100. This means that the maximum possible safety score that can be achieved is 100.

For a given safety score n we can consider adjusting all gaps to be of the form $(s, e - n)$ and ignore gaps where the end is before the start. Then, we can cross with this safety score only if it is possible to cross using the safety adjusted gaps.

An idea that can help simplify the implementation of this solution is to shift the gaps in lane l to be $(s - l, e - l)$, then we only need to look for strictly intersecting gaps between lanes.

We can go through the possible safety scores in increasing order, and check that we can still cross, and stop when it is no longer possible to cross. The time complexity of this solution is $O(N^2 \times S)$ where S is the maximum possible safety score.

Python Solution
The code for this solution can be found at the end of this document.

Subtask 4

We can make a simple improvement to the previous subtask by using binary search to find the maximum possible safety score. This reduces the time complexity to $O(N^2 \log(S))$ and will score 70 points.

Python Solution
The code for this solution can be found at the end of this document.

Full Solution

We can further optimise the solution to Subtask 4 by using a two pointers approach to determine the reachable adjusted gaps in each lane. To do this we need to sort the gaps in each lane, and keep a pointer to the gaps in the current lane which we increase when it no longer intersects with the gap we are considering in the next lane.

The overall time complexity of this solution is then $O(N \log(S))$.

Python Solution
The code for this solution can be found at the end of this document.

Trout II

Problem authored by Zalan V

https://train.nzoi.org.nz/problems/1443

Subtask 1

In this subtask the lakes are arranged in a line and $N \leq 100$. We can observe that the number of shipment routes disrupted by sabotaging any given channel will be the number of pairs of lakes with the same type of lure where one lake is to the left of the channel and the other is to the right. In this subtask we can try every possible channel to sabotage, and count the number of pairs in $O(N^2)$. Since there are $O(N)$ channels, the overall complexity of this solution is $O(N^3)$ and it will score 20 points.

Python Solution

```
N, M = map(int, input().split())
lure_types = list(map(int, input().split()))
max_disrupted = 0
for i in range(1, N):
    disrupted = 0
    for j in range(i):
        for k in range(i, N):
            if lure_types[j] == lure_types[k]:
                disrupted += 1
    max_disrupted = max(max_disrupted, disrupted)
print(max_disrupted)
```

Subtask 2

We can improve the time complexity of the previous solution by keeping track of the number of lakes with each lure type to the left. This allows us to calculate the number of disrupted shipping routes in $O(N)$. The overall complexity of this solution is $O(N^2)$ and it scores 30 points.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 3

Let L_i and R_i denote the number of lakes with lures of type i to the left and right of the current channel respectively. Then, the total disruption is the sum of $L_i \times R_i$ for all i. We observe that when moving to consider the next channel, only the number of pairs of the current lure type can change. Thus, we can keep track of the number of disrupted routes and update this number as we iterate over the possible channels to remove. The complexity of this solution is $O(N)$ and it scores 40 points.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 4

In this subtask the lakes are arranged in a tree. Thus, removing any channel will cause some routes to be disrupted. Let us root the tree at lake 0.

Let T_i denote the total number of lures of type i. Let $S_j(i)$ denote the number of lures of type i in the subtree of lake j. Similarly to the previous subtasks, the disruption for a channel between a lake j and its parent is given by the sum of $\big(T_i - S_j(i)\big) \times S_j(i)$ over all lures types.

We can perform a depth-first search where from each lake we return the disruption caused by removing the (parent -> lake) channel, and a hash map (dictionary in Python) of (lure_type, count). For each lake we can merge together the map of lure counts from each child and update the disruption using some math similarly to Subtask 3.

The issue with this approach is that merging the sets is $O(N)$, which makes the overall complexity of the solution $O(N^2)$, which is too slow to solve this subtask. However, when merging two sets, if we take care to always move the elements of the smaller set into the larger set, then we can prove that any given element will only be moved between sets $O(\log(N))$ times. Since moving each element is $O(1)$ the overall complexity of merging all the sets is $O(N \log(N))$. Implementing this optimisation makes this approach fast enough to solve this subtask.

The overall complexity of this solution is $O(N \log(N))$ and it scores 60 points.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 5

An edge is referred to as a **bridge** if removing it will divide the graph into two pieces, between which there is no longer any path. We observe that any edge that is not a bridge will not cause any disruption (as by definition there will be some other path between any two nodes). The following Python code demonstrates how to detect bridges in a graph (you can read more about finding bridges here):

```
low = [None] * N
def dfs(node, parent, depth):
  low[node] = depth
  for child in adjacent[n]:
    # Ignore the parent
    if child == parent: continue
    if low[child] is None:
      dfs(child, node, depth + 1)
      low[node] = min(low[node], low[child])
    # This is an ancestor of the current node
    else: low[node] = min(low[node], low[child])
  if low[node] == depth and parent is not None:
    # This indicates that the (parent, node) edge is a bridge

dfs(0, None, 1)
```

Let S_j denote the number of lakes in the subtree of lake j. Then, since every lake has the same type of lure in this subtask, we can easily calculate the disruption of removing a given bridge as $(N - S_j) \times S_j$.

This complexity of this solution is $O(N)$ and it scores 25 points. This solution can easily be combined with the solution to Subtask 4 by checking if $M = N - 1$ to score 85 points.

Python Solution

The code for this solution can be found at the end of this document.

Full Solution

We will extend the Subtask 4 solution to solve the full problem. We consider that in our depth first search, if the edge from the parent to the current node is a bridge, then the Subtask 4 solution will give the correct number of disrupted routes. Thus, we can combine the Subtask 4 and Subtask 5 solutions so

that we only consider the current (parent $\longrightarrow$ node) channel if it is a bridge. The overall time complexity of this solution is $O(N \log(N))$.

Note: A recursive solution using Python will likely be too slow even submitting with PyPy, so to fully solve this problem in Python we need to perform the DFS iteratively rather than recursively.

C++ Solution

The code for this solution can be found at the end of this document .

Big O Complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million (100,000,000) operations per second. For example, if the problem specifies a time limit of 1 second and an input of N as large as 100,000, then you know that an $O(N^2)$ algorithm might be too slow for large N since $100{,}000^2 = 10{,}000{,}000{,}000$, or 10 billion operations.

Time Complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a + b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry i corresponds to, but these will take the same amount of computation regardless of N. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a constant number of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O(N^2)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb in contests, $O(N^2)$ algorithms are only useful for input sizes of $N \lesssim 10{,}000$. Usually, if you have a nested loop in your program (loop inside a loop) then your solution is $O(N^2)$ if both these loops run about N times.

Additional Example Solutions

Carpenty Subtask 1 Python Solution

```
N = int(input())
h = list(map(int, input().split()))

for i in range(3):
  if h[i] >= 3:
    # Check if there is another log of size at least 2
    for j in range(3):
      if h[j] >= 2 and i != j:
        print(3)
        break
else:
  if max(h) >= 2:
    print(2)
  else:
    print(1)
```

Safe Crossings Subtask 3 Python Solution

```
N, M = map(int, input().split())
lanes = [[] for _ in range(M)]
for _ in range(N):
    l, s, e = map(int, input().split())
    s = max(s - l + 1, 1)
    e = e - l + 1
    if s < e: lanes[l - 1].append((s, e))

def can_cross(safety):
    adjusted_lanes = [[(s, e - safety) for s, e in lane if s <= e - safety] for lane
in lanes]
    # We can arrive as early as possible to every gap in the first lane
    arrival_times = [gap[0] for gap in adjusted_lanes[0]]
    for lane in range(M - 1):
        next_arrival_times = [10**9] * len(adjusted_lanes[lane + 1])
        for j in range(len(adjusted_lanes[lane])):
            start = arrival_times[j]
            end = adjusted_lanes[lane][j][1]
            # Ignore invalid gaps
            if end < start: continue
            for k in range(len(adjusted_lanes[lane + 1])):
                next_start, next_end = adjusted_lanes[lane + 1][k]
                # Check if the gaps overlap
                if start <= next_end and next_start <= end:
                    next_arrival_times[k] = min(next_arrival_times[k], max(start,
next_start))
        arrival_times = next_arrival_times
    for x in arrival_times:
        if x < 10**9: return True
    return False

for i in range(1, 100_000_001):
    if not can_cross(i): break
print(i - 1)
```

Safe Crossings Subtask 4 Python Solution

```
N, M = map(int, input().split())
lanes = [[] for _ in range(M)]
for _ in range(N):
    l, s, e = map(int, input().split())
    s = max(s - l + 1, 1)
    e = e - l + 1
    if s < e: lanes[l - 1].append((s, e))

def can_cross(safety):
    adjusted_lanes = [[(s, e - safety) for s, e in lane if s <= e - safety] for lane
in lanes]
    # We can arrive as early as possible to every gap in the first lane
    arrival_times = [gap[0] for gap in adjusted_lanes[0]]
    for lane in range(M - 1):
        next_arrival_times = [10**9] * len(adjusted_lanes[lane + 1])
        for j in range(len(adjusted_lanes[lane])):
            start = arrival_times[j]
            end = adjusted_lanes[lane][j][1]
            # Ignore invalid gaps
            if end < start: continue
            for k in range(len(adjusted_lanes[lane + 1])):
                next_start, next_end = adjusted_lanes[lane + 1][k]
                # Check if the gaps overlap
                if start <= next_end and next_start <= end:
                    next_arrival_times[k] = min(next_arrival_times[k], max(start,
next_start))
        arrival_times = next_arrival_times
    for x in arrival_times:
        if x < 10**9: return True
    return False

low = 0
upp = 100_000_000
while low < upp:
    mid = (low + upp + 1) // 2
    if can_cross(mid): low = mid
    else: upp = mid - 1
print(low)
```

Safe Crossings Full Python Solution

```
N, M = map(int, input().split())
lanes = [[] for _ in range(M)]
for _ in range(N):
    l, s, e = map(int, input().split())
    s = max(s - l + 1, 1)
    e = e - l + 1
    if s < e: lanes[l - 1].append((s, e))

for i in range(M):
    lanes[i].sort()

def can_cross(safety):
    adjusted_lanes = [[(s, e - safety) for s, e in lane if s <= e - safety] for lane
in lanes]
    possible_gaps = adjusted_lanes[0]
```

```
    for i in range(1, M):
        next_possible_gaps = []
        gap_idx = 0
        for start, end in adjusted_lanes[i]:
            # Skip over gaps that end before the start of the current gap
            while gap_idx < len(possible_gaps) and possible_gaps[gap_idx][1] < start:
                gap_idx += 1
            if gap_idx >= len(possible_gaps): break
            if possible_gaps[gap_idx][0] <= end and start <= possible_gaps[gap_idx]
[1]:
                next_possible_gaps.append((max(start, possible_gaps[gap_idx][0]),
end))
        possible_gaps = next_possible_gaps
    return len(possible_gaps) > 0

low = 0
upp = 100_000_000
while low < upp:
    mid = (low + upp + 1) // 2
    if can_cross(mid): low = mid
    else: upp = mid - 1
print(low)
```

Trout II Subtask 2 Python Solution

```
N, M = map(int, input().split())
lure_types = list(map(int, input().split()))
left_counts = [0] * 100_000
max_disrupted = 0
for i in range(N - 1):
    left_counts[lure_types[i]] += 1
    disrupted = 0
    for j in range(i + 1, N):
        disrupted += left_counts[lure_types[j]]
    max_disrupted = max(max_disrupted, disrupted)
print(max_disrupted)
```

Trout II Subtask 3 Python Solution

```
N, M = map(int, input().split())
lure_types = list(map(int, input().split()))

left_counts = [0] * 100_000
right_counts = [0] * 100_000
for lure in lure_types:
    right_counts[lure] += 1

max_disrupted = 0
disrupted = 0
for lure in lure_types:
    disrupted -= left_counts[lure] * right_counts[lure]
    left_counts[lure] += 1
    right_counts[lure] -= 1
    disrupted += left_counts[lure] * right_counts[lure]
    max_disrupted = max(max_disrupted, disrupted)
print(max_disrupted)
```

Trout II Subtask 4 Python Solution

```python
from collections import defaultdict
import sys
sys.setrecursionlimit(200000)

N, M = map(int, input().split())
lure_types = list(map(int, input().split()))

total_lure_counts = [0] * 100_000
for lure in lure_types:
    total_lure_counts[lure] += 1

adj = [[] for _ in range(N)]
for _ in range(M):
    a, b = map(int, input().split())
    adj[a].append(b)
    adj[b].append(a)

def solve(node, parent, depth):
    lowest_reachable[node] = depth
    disrupted = 0
    lure_counts = defaultdict(int)

    for child in adj[node]:
        # Ignore this edge if it leads back to the parent
        if child == parent: continue
        # Only traverse into the child node if we have not yet visited it
        if lowest_reachable[child] == 0:
            child_disrupted, child_lure_counts = solve(child, node, depth + 1)
            disrupted += child_disrupted

            # Swap the two dictionaries so that we merge the smaller one into the
larger one
            if (len(child_lure_counts) > len(lure_counts)):
                child_lure_counts, lure_counts = lure_counts, child_lure_counts

            for a, b in child_lure_counts.items():
                disrupted -= 2 * lure_counts[a] * b
                lure_counts[a] += b
        lowest_reachable[node] = min(lowest_reachable[node], lowest_reachable[child])

    disrupted += total_lure_counts[lure_types[node]] - 2 *
lure_counts[lure_types[node]] - 1
    lure_counts[lure_types[node]] += 1

    # We only consider the disruption if there is no other edge that leads into this
subtree than (parent <-> node)
    if (lowest_reachable[node] == depth):
        global max_disrupted
        max_disrupted = max(max_disrupted, disrupted)

    return (disrupted, lure_counts)

lowest_reachable = [0] * N
max_disrupted = 0
```

```
solve(0, -1, 1)
print(max_disrupted)
```

Trout II Subtask 5 Python Solution

```
import sys
sys.setrecursionlimit(200000)

N, M = map(int, input().split())
input()
adj = [[] for _ in range(N)]
for _ in range(M):
    a, b = map(int, input().split())
    adj[a].append(b)
    adj[b].append(a)

def dfs(node, parent, depth):
    lowest_reachable[node] = depth
    low = depth
    subgraph_size = 1

    for child in adj[node]:
        if child == parent: continue
        if not lowest_reachable[child]: subgraph_size += dfs(child, node, depth + 1)
        low = min(low, lowest_reachable[child])

    if low == depth:
        global ans
        ans = max(ans, (N - subgraph_size) * subgraph_size)

    lowest_reachable[node] = low
    return subgraph_size

ans = 0
lowest_reachable = [0] * N
dfs(0, -1, 1)
print(ans)
```

Trout II Full C++ Solution

```
#include <bits/stdc++.h>
#define  R(a) for (int i = 0; i < a; ++i)
using namespace std;
typedef long long ll;
typedef vector<int> vi;


ll max_disruption;
vi adj[100'000];
int total_lure_counts[100'000];
int lure_types[100'000];
int lowest_reachable[100'000];


pair<ll, unordered_map<int, int>> solve(int node, int parent, int depth) {
    lowest_reachable[node] = depth;
    pair<ll, unordered_map<int, int>> res;
    for (int child : adj[node]) {
```

```
        if (child == parent) continue;
        if (!lowest_reachable[child]) {
            auto child_res = solve(child, node, depth + 1);
            res.first += child_res.first;
            if (child_res.second.size() > res.second.size()) swap(res.second,
child_res.second);
            for (auto [type, count] : child_res.second) {
                res.first -= 2ll * res.second[type] * count;
                res.second[type] += count;
            }
        }
        lowest_reachable[node] = min(lowest_reachable[node],
lowest_reachable[child]);
    }

    res.first += total_lure_counts[lure_types[node]] - 2 *
res.second[lure_types[node]] - 1;
    res.second[lure_types[node]]++;

    if (lowest_reachable[node] == depth) max_disruption = max(max_disruption,
res.first);

    return res;
}



int main() {
    int N, M;
    cin >> N >> M;

    R(N) cin >> lure_types[i];
    R(N) total_lure_counts[lure_types[i]]++;

    R(M) {
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    solve(0, -1, 1);
    cout << max_disruption << '\n';

    return 0;
}
```