July 21, 2024

New Zealand Informatics Competition 2024

Round Two

Editorial by Anatol C, Zalan V, Jonathon S

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

Origami

Problem authored by Jonathan K
https://train.nzoi.org.nz/problems/1327

Full Solution

We can split the paper in two different ways, either into rectangles of size $A \times B$ or $B \times A$. Consider the $A \times B$ case. How many sheets can we make?

With the vertical cuts, we can split the paper into at most $\lfloor \frac{X}{A} \rfloor$[1] columns of width A. These columns will still have height Y. Then, with vertical cuts, we can split one of these columns into $\lfloor \frac{Y}{B} \rfloor$ rectangles of the desired size. So, the total number of rectangles achieved is $\lfloor \frac{X}{A} \rfloor \times \lfloor \frac{Y}{B} \rfloor$.

Similarly, if we want rectangles of size $B \times A$, we find that we can make $\lfloor \frac{X}{B} \rfloor \times \lfloor \frac{Y}{A} \rfloor$. The answer to the problem is simply the higher of the two values.

Time complexity: $O(1)$

Python Solution

```
X, Y, A, B = map(int, input().split())
way_one = (X // A) * (Y // B)
way_two = (X // B) * (Y // A)
print(max(way_one, way_two))
```

[1] $\lfloor x \rfloor$ represents x rounded down

Beau's Route

Problem authored by Iván G
https://train.nzoi.org.nz/problems/1335

Subtask 2

In this subtask, since there are at most eight mountains, we can simply try every possible order of visiting the mountains (since there are at most N factorial orders). Then, we find the best one.

Alternatively, there are only eight possible cases for this subtask so you can solve all of the cases by hand.

Python Solution

```
from itertools import permutations

N = int(input())

# We track the best result found so far
best_score = 0
best_order = None

# For loop over all permutaions of [1, 2, 3, ..., N]
for order in permutations(range(1, N+1)):
    # Find the minimum enjoyment in this ordering
    enjoyment = 10**9
    for i in range(1, N):
        enjoyment = min(enjoyment, abs(order[i] - order[i-1]))

    if enjoyment > best_score:
        best_order = order
        best_score = enjoyment
    elif enjoyment == best_score:
        #Check if it is lexicographically better
        if order < best_order:
            best_order = order

print(*best_order)
```

Full Solution

Using the program above, or otherwise, we can find solutions to small cases.

N	**Solution**
1	1
2	1, 2
3	1, 2, 3
4	2, 4, 1, 3
5	1, 3, 5, 2, 4
6	3, 6, 2, 5, 1, 4
7	1, 4, 7, 3, 6, 2, 5
8	4, 8, 3, 7, 2, 6, 1, 5

Now, we can try find a pattern in these smaller cases and extend them to larger cases.

The pattern for even numbered cases is slightly simpler so we will start with this. We can discover it by looking at a case (for example $N = 8$) and taking every second element, starting from the first. This gives us the sequence $[4, 3, 2, 1]$. Similarly, if we take every second second element starting from the second, we get $[8, 7, 6, 5]$. These two sequences are just counting backwards from $\frac{N}{2}$ down to 1 and from N down to $\frac{N}{2} + 1$. To generate the solution for some other even N we can just generate these two sequences and re-interleave them.

For the odd-numbered case, it is quite similar, the main difference is that they start with the number one. if we look at the $N = 7$ case and ignore the first element, splitting it into two sequences as before, we get $[4, 3, 2]$ and $[7, 6, 5]$. Once again these sequences follow a very simple pattern. They are counting down from $\lceil \frac{N}{2} \rceil$ to 2 and from N to $\lceil \frac{N}{2} \rceil + 1$.[2]

These patterns give us a full solution.

Python Solution

```
N = int(input())

def interleave(a, b):
  res = []
  for i in range(len(a)):
    res.append(a[i])
    res.append(b[i])
  return res

if N % 2 == 0:
  seq = interleave(
    list(range(N//2, 0, -1)),
    list(range(N, N//2, -1))
  )
else:
  seq = [1] + interleave(
    list(range(N//2 + 1, 1, -1)),
    list(range(N, N//2 + 1, -1))
  )

print(*seq)
```

C++ Solution

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

Extra Challenge

Currently, the solution is based in part on guesswork. Can you prove that this solution is always correct? Hint: Start by showing that you cannot make a solution with a greater minimum enjoyment.

Note that in contests it isn't always necessary to prove that a solution works. In some cases (like this problem), coding up a solution and getting it judged is much faster than proving the solution beforehand. This is not to say you should never prove your solutions. Proving a solution can allow you to find mistakes without spending valuable time implementing said solution.

[2] $\lceil x \rceil$ represents x rounded up

Flower Garden 2

Problem authored by Iván G
https://train.nzoi.org.nz/problems/1392

Subtask 1

In this subtask there are only 2 squares, so the answer is the sum of their areas minus their area of intersection.

If R is a rectangle, let's define the following:

- R_t = y-coordinate of top edge of R
- R_b = y-coordinate of bottom edge of R
- R_l = x-coordinate of left edge of R
- R_r = x-coordinate of right edge of R

If $(0, 0)$ is the top-left corner, then the area of intersection of two rectangles, A and B, is

$$\max(0, \min(A_b, B_b) - \max(A_t, B_t)) \times \max(0, \min(A_r, B_r) - \max(A_l, B_l))$$

When finding the top and left edges for each square, remember that they can't have negative values.

Time complexity: $O(1)$

Python Solution

```
N, A = map(int, input().split())
ay, ax = map(int, input().split())
by, bx = map(int, input().split())

at = max(0, ay - A)
ab = ay + A + 1
al = max(0, ax - A)
ar = ax + A + 1
bt = max(0, by - A)
bb = by + A + 1
bl = max(0, bx - A)
br = bx + A + 1

intersect = max(0, min(ab, bb) - max(at, bt)) * max(0, min(ar, br) - max(al, bl))
a_area = (ab - at) * (ar - al)
b_area = (bb - bt) * (br - bl)

print(a_area + b_area - intersect)
```

Subtask 2

Since A is small enough, we can simulate each watering. Let watered$[y][x] = 1$ if the cell in row y and column x has been watered at least once, otherwise 0. For each i, we can go through all cells in the square being watered, and set their watered value to 1.

We can use a simple 2D list/array for this because r_i and c_i are small enough. If they had the full constraints, we could use a set instead.

Time complexity: $O(A^2 \times N)$

Python Solution

Note: You may need to use PyPy 7.3 for this solution to pass.

```
N, A = map(int, input().split())

watered = [[0 for i in range(1100)] for j in range(1100)]
res = 0

for i in range(N):
    r, c = map(int, input().split())
    for y in range(max(0, r-A), r+A+1):
        for x in range(max(0, c-A), c+A+1):
            if watered[y][x] == 0:
                res += 1
            watered[y][x] = 1
print(res)
```

Subtask 3

In this subtask, the centers of all squares lie on the diagonal line going from top-left to bottom-right. Let's begin by sorting the squares in non-decreasing order of the x (or y) position of their centers, and we'll process them in this order. The area that each square contributes to the answer is its own area minus the area of intersection between itself and the square immediately before it. Therefore, we can use the formula from Subtask 1 and apply it here.

Time complexity: $O(N \log N)$

Python Solution

```
def intersect(a, b):
    return max(0, min(a[3], b[3]) - max(a[2], b[2])) * max(0, min(a[1], b[1]) - max(a[0],
b[0]))

def area(a):
    return (a[3] - a[2]) * (a[1] - a[0])

N, A = map(int, input().split())

squares = []
for i in range(N):
    r, c = map(int, input().split())
    squares.append((max(0,c-A), c+A+1, max(0,r-A), r+A+1))
squares.sort()

res = 0
for i in range(N):
    res += area(squares[i])
    if i > 0:
        res -= intersect(squares[i], squares[i-1])
print(res)
```

Full Solution

If we tried calculating areas of intersections for the full problem, we would need to find the intersections between all subsets of squares, which will be far too slow. Instead, we can process each row separately and use coordinate compression.

Let's focus on a single row for now. Let L_i be the x-coordinate of the leftmost cell of square i and R_i be the x-coordinate of the rightmost cell of square i. For each square i, if it covers part of the current row, we'll add the pairs $(L_i, 1)$ and $(R_i + 1, -1)$ into a list/vector. The first pair signals that all cells

to the right of (and including) L_i will be watered once by this watering. The second pair signals that all cells to the right of R_i will not be watered by this watering (since the -1 cancels out the 1).

After all squares have been processed, we'll sort the list in non-decreasing order of the pair's first value. Now let's define the following:

- count = number of times that the current "interval" has been watered
- x_i = the first value in the i-th pair
- w_i = the second value in the i-th pair

Let's go through the list in the sorted order. For the i-th pair in the list, if count > 0, then all cells in the interval $[x_{i-1}, x_i)$ have been watered at least once, so we add $x_i - x_{i-1}$ to the answer. Then, add w_i to count.

To solve the full problem, we need to do this for every row that has at least one watered cell. Since row numbers can be very large, we need to use a dictionary/map that maps row numbers to the list described above. For every square, we'll go through all the rows that it contains, and add the two pairs into the corresponding list in the dictionary.

Time complexity: $O(A \times N \log N)$

Python Solution

Note: You may need to use PyPy 7.3 for this solution to pass.

```
N, A = map(int, input().split())
rows = {}

def add_pair(y, pair):
    if y in rows:
        rows[y].append(pair)
    else:
        rows[y] = [pair]

tmp = []
for i in range(N):
    r, c = map(int, input().split())
    tmp.append(((max(0,c-A), 1), max(0,r-A), r+A+1))
    tmp.append(((c+A+1, -1), max(0,r-A), r+A+1))
tmp.sort()
for cur in tmp:
    for y in range(cur[1], cur[2]):
        add_pair(y, cur[0])

res = 0
for row in rows.values():
    cnt = 0
    for i in range(len(row)):
        if cnt > 0:
            res += row[i][0] - row[i-1][0]
        cnt += row[i][1]
print(res)
```

C++ Solution

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

Iván's Adventure

Problem authored by Anatol C
https://train.nzoi.org.nz/problems/1415

Subtask 1 & 2

We can solve these subtasks using a brute-force method.

The first thing we want to do is find at every stop, how much money Iván had left. We start at the last stop, where we know he has zero money, then work our way backwards through his stops, each time adding the cost of the flight he takes from the current stop.

Then, for every stop on his adventure, we can simply go through all the cities, calculate how much it costs to reach them, and count the ones he can reach with his available money.

Finding the minimum cost for going from one city to another is facilitated by realising that the fastest route to a city is the direct flight. Even if you don't realise this, you can still calculate the distance from a stop to all cities efficiently using Dijkstra's Algorithm.

For each of the M stops, we can find the distance to the N cities in $O(N)$ time total, giving us an algorithm that runs in $O(N \times M)$ time total. If we use Dijkstra's Algorithm, finding the distance to N cities takes $O(N \log N)$ time and so the algorithm runs in $O(N \times M \log N)$. Both are efficient enough to pass.

Subtask 1 exists to allow less efficient solutions to earn some points.

Python Solution

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

Subtask 3

In this subtask, since the y-coordinates of all cities are zero, all the cities effectively lie on a line. An important side-effect of this is that if Iván is in some city, the set of cities he can reach form a contiguous segment of that line.

Specifically, if he is in some city a, there is some city b and some city c such that all the cities Iván can reach from a lie between (and include) cities b and c.

If we sort the cities by x-coordinate, we can perform a binary search to find these endpoints b and c. Then, if we have the sorted indices of these cities, we can easily find the number of cities between these two cities and so, the number of cities we can reach. The binary searches take $O(\log N)$ time each so for each stop, we can find the number of reachable cities in $O(\log N)$, giving us an algorithm that runs in $O(M \log N)$ time overall.[3]

Python Solution

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

Subtask 4

To solve this subtask we must notice that for each city, once Iván cannot reach a city from some stop, he won't ever be able to reach it again from further stops. This is true the other way as well. If we go backwards through Iván's stops, once we can reach a city, we will be able to reach it from all previous stops.

[3]It is actually $O((N + M) \log N)$ time since we must take into account the time it takes to sort the cities.

Additionally, as we go backwards through the stops the number of cities we cannot reach shrinks very rapidly. The most expensive direct flight possible in this subtask is between two opposite corners of the range. For example, the flight from (0, 0) to (500, 500) would cost $708. This means, when going backwards through the stops, once Iván has $708, which will take at most 708 stops, he will always be able to reach all the cities and so the answer will always be $N - 1$.

We still need to find the answers for when he has less than $708. The idea is to track the set of cities that Iván cannot reach while going backwards through those sets. Because we only have to do this for at most 708 stops and because this set will actually shrink very rapidly, for every stop we can just iterate through this set of cities and check if we can now reach them. Then, the answer at each stop is the just $N - k - 1$ where k is the size of that set - the number of cities Iván cannot reach.

The time complexity of this solution is very high. Around $O(700N)$. Because of this we do need to be careful of constant or logarithmic factors. In C++ we should avoid using an ordered set to store the non-reachable cities and instead opt for an unordered set or a vector. In Python we should avoid using sqrt wherever possible.

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

Original Intended Solution for Subtask 4 (Briefly)
The original intended solution for this subtask involved a similar idea of keeping track of which cities we can reach. Where it differs from the above solution is it makes use of the fact that the reachable area is a shrinking circle and tracks the shape of that circle. This leads to a faster solution.

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

Full Solution

For the full solution, we again use the idea that once a city becomes unreachable, it stays unreachable. Because of this, for each city, we can binary search to find the first point along the route where it becomes unreachable. We can then use all of this to construct the solution.

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

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

Trout

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1423

Subtask 1

In this subtask there are no new trout discoveries and the river is a line which flows from point 0 through $N-1$ and the distance between each point is 1 metre. We can consider the river as an array of points 0 through $N-1$. Then, a trout that is at point L_i could possibly be found at any of the points $L_i - S_i$ through L_i inclusive. Thus, we can keep an array and loop over every possible location the trout could visit in $O(N)$. We can then answer Q type queries in O(1). The complexity of this solution is $O(N \times M + K)$.

Python Solution

```
N, M, K, U = map(int, input().split())
for i in range(N - 1):
    input()

trout_count = [0] * N
for i in range(M):
    L, D = map(int, input().split())
    for j in range(max(L - D, 0), L + 1):
        trout_count[j] += 1

for i in range(K):
    q = int(input().split()[1])
    print(trout_count[q])
```

Subtask 2

For this subtask we can simulate each trout swimming upstream, and increment a counter for each point along the way in $O(N)$. For every point other than point 0 we need to keep track of it's parent (the point upstream) and the distance to the parent. Additionally, there may also be D type updates, which we also process in the same way by simulating the trout swimming upstream. The complexity of this solution is $O((M + K) \times N)$.

Python Solution

```
N, M, K, U = map(int, input().split())
parent = [-1] * N
for i in range(N - 1):
    a, b, w = map(int, input().split())
    parent[b] = (a, w)
trout_count = [0] * N

def add_trout(position, distance):
     while distance >= 0:
        trout_count[position] += 1
        if position == 0:
            break
        distance -= parent[position][1]
        position = parent[position][0]

for i in range(M):
    L, D = map(int, input().split())
```

```
    add_trout(L, D)

for i in range(K):
    query = input().split()
    if query[0] == 'Q':
        print(trout_count[int(query[1])])
    else:
        L, D = map(int, query[1:])
        add_trout(L, D)
```

Subtask 3

In this subtask the river is a line and there are no trout discoveries, so we can reuse the idea of representing the river as an array from Subtask 1. We can construct a prefix sum of the distance to every point from point 0 and can then use a binary search (such as `bisect_left` in Python or `lower_bound` in C++) to determine the range of points which a trout could reach in $O(\log N)$. We can then process all the trout at once in $O(N)$ and answer queries in $O(1)$. Hence, the complexity of this solution is $O(M \log N + K)$.

Python Solution

```
from bisect import bisect_left

N, M, K, U = map(int, input().split())
distance = [0] * N
for i in range(1, N):
    a, b, w = map(int, input().split())
    distance[i] = distance[i - 1] + w

trout_count = [0] * (N + 1)
for i in range(M):
    L, D = map(int, input().split())
    trout_count[bisect_left(distance, distance[L] - D)] += 1
    trout_count[L + 1] -= 1

for i in range(1, N):
    trout_count[i] += trout_count[i - 1]

for i in range(K):
    q = int(input().split()[1])
    print(trout_count[q])
```

Subtask 4

This subtask has the same constraints as Subtask 3, but with the added complexity of new trout discoveries. Once again a trout will swim across some contiguous range of this array, and we can find this range using a binary search over the prefix sums of distances as described in Subtask 3. We can use a Segment Tree to update this range in $O(\log N)$ and can then query the number of trout at a point in the Segment Tree in $O(\log N)$ to answer queries. Thus, the complexity of this solution is $O((M + K) \log N)$ and will also pass Subtasks 1 and 3.

C++ Solution

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

Subtask 5

The solution to this subtask is the extension of the Subtask 3 solution to a general tree rather than just a line. We can perform a depth-first search and keep a stack of the distances from the root to every

ancestor of the current point, which we can binary search to find the most upstream point reachable in $O(\log N)$. We can then process all trout at once in $O(N)$. The complexity of this solution is the same as Subtask 3.

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

Full solution

We can peform an Euler Tour to represent the graph such that the subtree of every point represents a contiguous subarray. To do this, we perform a depth-first search and keep a global counter which we increment when we enter and also when we leave a point. We record the value of this counter for every point, and will name these $\text{in}[x]$ and $\text{out}[x]$ for a point x.

Then, to add a trout that starts at point x and can potentially swim up to point y we add $+1$ to in[x] and -1 to $\text{out}[y]$. Then, number of trout at a given point x is the sum $[\text{in}[x], \text{out}[x])$. We can perform these updates and queries using a point-update, range-sum Segment Tree in $O(\log N)$.

All that is missing from our solution is a way to find the most upstream reachable point for every trout. We will describe two approaches:

Approach 1
We can take advantage of the fact that this problem is not online since we are given all the queries and updates in the input in advance. Thus, we can find the trout initially at every point, and the trout that will later be discovered at every point before answering any queries. Thus, we can use the same technique described in Subtask 5 and binary search the stack of distances of a point's ancestors to precompute the most upstream reachable point for every trout.

Approach 2
We can use a technique called Binary Lifting to find the most upstream reachable point for every trout. We perform a precomputation as follows:

For every point we precalculate 2^n-th ancestor and the total distance to reach it for $0 \leq n \leq \lfloor \log_2(N) \rfloor$. We can calculate this in O(N log N) since for $1 \leq n \leq \lfloor \log_2(N) \rfloor$ the 2^n-th ancestor is equivalent to the 2^{n-1}-th ancestor of the 2^{n-1}-th ancestor. For point 0 we can add an edge to itself with a distance of 0 to simplify the implementation.

To find the lowest point within a given distance we can repeatedly jump to the furthest away 2^n-th ancestor within the swimming distance. By doing this we then only need to make $O(\log N)$ jumps upwards, to find the most upstream reachable point.

The time complexity of both of these approaches works out to be $O((M + K) \log N)$ since we can perform Segment Tree updates and queries and can find the lowest reachable point for a trout in $O(\log N)$.

Full Solution - C++
The code for this solution can be found at the end of this document .

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

Alternative full solution

An alternative solution is to use a technique called heavy-light decomposition. In short: we merge paths in the tree together into so called **heavy-paths** which are then connected by **light-paths** such

that there are at most log N **heavy-paths** between any leaf node and the root. We can then build a prefix sum of distances over each **heavy-path** so that we can find the lowest reachable position for a trout in $O(\log N)$, and can update a **heavy-paths** in $O(\log N)$ using a segment tree. Since we only need to update log N **heavy-paths**, we can perform each update in $O(\log^2 N)$. We can then perform queries in $O(\log N)$ by querying the corresponding point in the segment tree of the **heavy-path**. The complexity of this solution is $O((M + K) \log^2 N)$ which is higher than the previous solution described, but is still well within the bounds to pass the problem with C++. Getting this solution to pass with Python is challenging and requires some clever optimisations.

Additional Example Solutions

Beau's Route Full C++ Solution

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

using namespace std;

int main() {
  int n, a, b;
  cin >> n;
  if (n % 2 == 1) {
    cout << "1 ";
    b = n;
    a = (n + 1) / 2;
    for (int i = 1; i < n; i++) {
      if (i % 2) {
        cout << a << ' ';
        a -= 1;
      }
      else {
        cout << b << ' ';
        b -= 1;
      }
    }
    return 0;
  }
  a = n / 2;
  b = n;
  for (int i = 0; i < n; i++) {
    if (i % 2) {
      cout << b << ' ';
      b -= 1;
    }
    else {
      cout << a << ' ';
      a -= 1;
    }
  }
}
```

Flower Garden 2 Full C++ Solution

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

int N;
ll A;
unordered_map<ll, vector<pair<ll, int>>> rows;

int main() {
    cin >> N >> A;
    for (int i = 0; i < N; i++) {
        ll r, c; cin >> r >> c;
        for (ll y = max(0LL, r-A); y < r+A+1; y++) {
            rows[y].push_back({max(0LL, c-A), 1});
            rows[y].push_back({c+A+1, -1});
```

```
        }
    }

    ll res = 0;
    for (auto row: rows) {
        sort(row.second.begin(), row.second.end());
        int cnt = 0;
        ll prev = 0;
        for (auto cur: row.second) {
            if (cnt > 0) {
                res += cur.first - prev;
            }
            cnt += cur.second;
            prev = cur.first;
        }
    }
    cout << res;
}
```

Iván's Adventure Subtask 1 & 2 Python Solution

```
from math import ceil, sqrt

N, M = map(int, input().split())
cities = [tuple(map(int, input().split())) for _ in range(N)]
path = list(map(int, input().split()))

def cost(a, b):
  return ceil(sqrt((cities[a][0] - cities[b][0]) * (cities[a][0] - cities[b][0]) +
(cities[a][1] - cities[b][1]) * (cities[a][1] - cities[b][1])))

money = [0] * M
for i in range(M-2, -1, -1):
  money[i] = money[i+1] + cost(path[i], path[i+1])

for i in range(M):
  count = 0
  for j in range(N):
    if j == path[i]: continue
    if cost(path[i], j) <= money[i]: count += 1
  print(count, end = " ")
print()
```

Iván's Adventure Subtask 3 Python Solution

```
from math import ceil, sqrt

N, M = map(int, input().split())
cities = [int(input().split()[0]) for _ in range(N)]
path = list(map(int, input().split()))

def cost(a, b):
  return abs(cities[a] - cities[b])

sorted_order = list(range(N))
sorted_order.sort(key = lambda x: cities[x])

position_in_sorted_order = [0] * N
```

```
for i in range(N):
  position_in_sorted_order[sorted_order[i]] = i

money = [0] * M
for i in range(M-2, -1, -1):
  money[i] = money[i+1] + cost(path[i], path[i+1])

for i in range(M):
  sorted_idx = position_in_sorted_order[path[i]]
  #Find left bound (b)
  bl, bu = 0, sorted_idx
  while bl != bu:
    m = (bl + bu) // 2
    if cost(sorted_order[m], path[i]) > money[i]:
      bl = m + 1
    else:
      bu = m
  b = bl

  #Find right bound (c)
  cl, cu = sorted_idx, N-1
  while cl != cu:
    m = (cl + cu) // 2 + 1
    if cost(sorted_order[m], path[i]) > money[i]:
      cu = m - 1
    else:
      cl = m
  c = cl

  print(c - b, end = " ")
print()
```

Iván's Adventure Subtask 4 Python Solution

```
from math import ceil, sqrt

N, M = map(int, input().split())
cities = [tuple(map(int, input().split())) for _ in range(N)]
path = list(map(int, input().split()))

def cost(city_a, city_b):
  return ceil(sqrt((city_a[0] - city_b[0]) * (city_a[0] - city_b[0]) + (city_a[1] -
city_b[1]) * (city_a[1] - city_b[1])))

answers = [0] * M
money = 0

cannot_reach = set(cities)
for i in range(M-2, -1, -1):
  money += cost(cities[path[i]], cities[path[i+1]])
  to_remove = []
  for city in cannot_reach:
    if (city[0] - cities[path[i]][0]) * (city[0] - cities[path[i]][0]) + (city[1] -
cities[path[i]][1]) * (city[1] - cities[path[i]][1]) <= money * money:
      to_remove.append(city)
  for city in to_remove:
    cannot_reach.remove(city)
```

```
  answers[i] = N - len(cannot_reach) - 1
print(*answers)
```

Iván's Adventure Subtask 4 Python Original Intended Solution

```
from math import ceil, sqrt, floor

N, M = map(int, input().split())
cities = [tuple(map(int, input().split())) for _ in range(N)]
path = list(map(int, input().split()))

def cost(city_a, city_b):
  return ceil(sqrt((city_a[0] - city_b[0]) * (city_a[0] - city_b[0]) + (city_a[1] -
city_b[1]) * (city_a[1] - city_b[1])))

def cap_range(r):
  return (max(0, r[0]), min(501, r[1]))

has_city = [[0] * 501 for _ in range(501)]
ranges = [(0, 501) for _ in range(501)]
diameter = (0, 501)

for (x, y) in cities:
  has_city[x][y] += 1

money = [0] * M
for i in range(M-2, -1, -1):
  money[i] = money[i+1] + cost(cities[path[i]], cities[path[i+1]])

reachable_cities = N
for i in range(M):
  if money[i] <= 708:
    mx = cities[path[i]][0]
    my = cities[path[i]][1]

    new_diam = (mx - money[i], mx + money[i] + 1)
    for col in range(*diameter):
      dx = col - mx
      curr_range = ranges[col]

      if abs(dx) > money[i]:
        for row in range(*curr_range):
          reachable_cities -= has_city[col][row]
        continue

      height = floor(sqrt(money[i] * money[i] - dx * dx))
      new_range = (my - height, my + height + 1)

      for row in range(curr_range[0], new_range[0]):
        reachable_cities -= has_city[col][row]
      for row in range(new_range[1], curr_range[1]):
        reachable_cities -= has_city[col][row]
      ranges[col] = cap_range(new_range)
    diameter = cap_range(new_diam)

  print(reachable_cities - 1, end = " ")
print()
```

Iván's Adventure Full Python Solution

```python
from math import ceil, sqrt

N, M = map(int, input().split())
cities = [tuple(map(int, input().split())) for _ in range(N)]
path = list(map(int, input().split()))

def cost(city_a, city_b):
  return ceil(sqrt((city_a[0] - city_b[0]) * (city_a[0] - city_b[0]) + (city_a[1] -
city_b[1]) * (city_a[1] - city_b[1])))

money = [0] * M
for i in range(M-2, -1, -1):
  money[i] = money[i+1] + cost(cities[path[i]], cities[path[i+1]])

num_new_unreachable_cities = [0] * (M+1)

for city in cities:
  lo = 0
  hi = M
  while lo != hi:
    m = (lo + hi) // 2
    if cost(cities[path[m]], city) <= money[m]:
      lo = m + 1
    else:
      hi = m
  num_new_unreachable_cities[lo] += 1

ans = N - 1
for i in range(M):
  ans -= num_new_unreachable_cities[i]
  print(ans, end = " ")
print()
```

Iván's Adventure Full C++ Solution

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

using namespace std;

int N, M;
int x[100000], y[100000];
int path[100000];
int change[100001];
long long money[100000];

long long cost(int a, int b) {
    long long dx = x[a] - x[b];
    long long dy = y[a] - y[b];
    return (long long) ceil(sqrt((double) (dx * dx + dy * dy)));
}

int main() {
    cin >> N >> M;
    for (int i = 0; i < N; i++) cin >> x[i] >> y[i];
    for (int i = 0; i < M; i++) cin >> path[i];
```

```
    for (int i = M - 2; i >= 0; i--) {
        money[i] = money[i+1] + cost(path[i], path[i+1]);
    }

    for (int i = 0; i < N; i++) {
        int lo = 0, hi = M;
        while (lo != hi) {
            int m = (lo + hi) / 2;
            if (cost(i, path[m]) <= money[m]) {
                lo = m + 1;
            } else {
                hi = m;
            }
        }
        change[lo]++;
    }

    int res = N - 1;
    for (int i = 0; i < M; i++) {
        res -= change[i];
        cout << res << " ";
    }
    cout << endl;
}
```

Trout Subtask 4 Solution C++

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



struct segtree {
    vi nodes;

    segtree(int s) : nodes(s * 4) {}

    void update(int n, int tl, int tr, int l, int r) {
        if (l <= tl && tr <= r) {
            nodes[n]++;
            return;
        }
        int tm = (tl + tr) / 2;
        if (l <= tm) update(n * 2, tl, tm, l, r);
        if (tm < r) update(n * 2 + 1, tm + 1, tr, l, r);
    }

    int query(int n, int tl, int tr, int pos) {
        int res = nodes[n];
        if (tl != tr) {
            int tm = (tl + tr) / 2;
            if (pos <= tm) res += query(n * 2, tl, tm, pos);
            else res += query(n * 2 + 1, tm + 1, tr, pos);
        }
        return res;
```

```
    }
};



int main() {
    int N, M, K, U, a, b, c;
    cin.tie(0)->sync_with_stdio(false);
    cin >> N >> M >> K >> U;

    vi dist(N);
    R(N - 1) {
        cin >> a >> b >> c;
        dist[b] = c;
    }
    partial_sum(dist.begin(), dist.end(), dist.begin());

    segtree tree(N);
    R(M) {
        cin >> a >> b;
        int idx = lower_bound(dist.begin(), dist.end(), dist[a] - b) - dist.begin();
        tree.update(1, 0, N - 1, idx, a);
    }

    R(K) {
        char d;
        cin >> d >> a;
        if (d == 'Q') printf("%d\n", tree.query(1, 0, N - 1, a));
        else {
            cin >> b;
            int idx = lower_bound(dist.begin(), dist.end(), dist[a] - b) - dist.begin();
            tree.update(1, 0, N - 1, idx, a);
        }
    }

    return 0;
}
```

Trout Subtask 5 Solution C++

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

constexpr int MAX_N = 100000;

int N, M, K, U;
vi trout[MAX_N];
vector<pair<int, int>> adj[MAX_N];
int troutEnding[MAX_N], troutCount[MAX_N];



int dfs(vi& ancestors, vi& distances, int n, int dist) {
    int res = 0;
    ancestors.push_back(n);
```

```
    distances.push_back(dist);

    for (auto [c, d] : adj[n]) {
        res += dfs(ancestors, distances, c, dist + d);
    }

    for (int a : trout[n]) {
        res++;
        int upstream = lower_bound(distances.begin(), distances.end(), dist - a) -
distances.begin();
        troutEnding[ancestors[upstream]]++;
    }

    troutCount[n] = res;

    distances.pop_back();
    ancestors.pop_back();
    return res - troutEnding[n];
}


int main() {
    int a, b, c;
    cin.tie(0)->sync_with_stdio(false);
    cin >> N >> M >> K >> U;

    R(N - 1) {
        cin >> a >> b >> c;
        adj[a].push_back({b, c});
    }

    R(M) {
        cin >> a >> b;
        trout[a].push_back(b);
    }

    vi ancestors, distances;
    dfs(ancestors, distances, 0, 0);

    R(K) {
        char d;
        cin >> d >> a;
        cout << troutCount[a] << '\n';
    }

    return 0;
}
```

Trout Full Solution C++

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

constexpr int MAX_N = 100000;

int N, M, K, U;
```

```
int nodes[4 * MAX_N], tour_in[MAX_N], tour_out[MAX_N];
vector<int> adj[MAX_N];
pair<int, int> jmp[17][MAX_N];

void update(int pos, int val) {
    for (nodes[pos += 2 * MAX_N] += val; pos > 1; pos >>= 1) nodes[pos >> 1] = nodes[pos]
+ nodes[pos ^ 1];
}

int query(int l, int r) {
    int res = 0;
    for (l += 2 * MAX_N, r += 2 * MAX_N; l < r; l >>= 1, r >>= 1) {
        if (l & 1) res += nodes[l++];
        if (r & 1) res += nodes[--r];
    }
    return res;
}

void tour(int& cnt, int n) {
    tour_in[n] = cnt++;
    for (int c : adj[n]) tour(cnt, c);
    tour_out[n] = cnt++;
}

void addTrout(int n, int d) {
    update(tour_in[n], 1);
    for (int i = 16; i >= 0; --i) {
        auto& [jnode, jdist] = jmp[i][n];
        if (jdist <= d) {
            d -= jdist;
            n = jnode;
        }
    }
    update(tour_out[n], -1);
}

int main() {
    int a, b, c;
    cin.tie(0)->sync_with_stdio(false);
    cin >> N >> M >> K >> U;

    R(N - 1) {
        cin >> a >> b >> c;
        adj[a].push_back(b);
        jmp[0][b] = {a, c};
    }

    R(16) {
        for (int j = 0; j < N; ++j) {
            auto& [jnode, jdist] = jmp[i][j];
            jmp[i + 1][j] = {jmp[i][jnode].first, jdist + jmp[i][jnode].second};
        }
    }

    int cnt = 0;
    tour(cnt, 0);
```

```
    R(M) {
        cin >> a >> b;
        addTrout(a, b);
    }

    R(K) {
        char d;
        cin >> d >> a;
        if (d == 'Q') cout << query(tour_in[a], tour_out[a]) << '\n';
        else {
            cin >> b;
            addTrout(a, b);
        }
    }

    return 0;
}
```

Trout Full Solution Python

```
import sys

N, M, K, U = map(int, sys.stdin.readline().split())
children = [[] for i in range(N)]
parent = [(0, 0)] * N
for i in range(N - 1):
    a, b, w = map(int, sys.stdin.readline().split())
    children[a].append(b)
    parent[b] = (a, w)

point_in  = [0] * N
point_out = [0] * N

stack = [0]
visit = [False] * N
idx = 0
while len(stack) > 0:
    n = stack[-1]
    if not visit[n]:
        visit[n] = True
        point_in[n] = idx
        idx += 1
        for c in children[n]:
            stack.append(c)
    else:
        point_out[n] = idx
        idx += 1
        stack.pop()

jump = [parent]
for i in range(16):
    jump.append([])
    for j in range(N):
        node1, dist1 = jump[-2][j]
        node2, dist2 = jump[-2][node1]
        jump[-1].append((node2, dist1 + dist2))
```

```
tree_size = 2 * N
segtree = [0] * (2 * tree_size)

def update(pos, val):
    pos += tree_size
    segtree[pos] += val
    while pos > 1:
        segtree[pos >> 1] = segtree[pos] + segtree[pos ^ 1]
        pos >>= 1

def query(left, right):
    res = 0
    left += tree_size
    right += tree_size
    while left < right:
        if left & 1:
            res += segtree[left]
            left += 1
        if right & 1:
            right -= 1
            res += segtree[right]
        left  >>= 1
        right >>= 1
    return res

def add_trout(pos, dist):
    update(point_in[pos], 1)
    for i in range(16, -1, -1):
        jump_pos, jump_dist = jump[i][pos]
        if jump_dist <= dist:
            dist -= jump_dist
            pos = jump_pos
    update(point_out[pos], -1)

for i in range(M):
    L, D = map(int, sys.stdin.readline().split())
    add_trout(L, D)

res = []
for i in range(K):
    inp = sys.stdin.readline().split()
    n = int(inp[1])
    if inp[0] == 'Q':
        res.append(query(point_in[n], point_out[n]))
    else:
        add_trout(n, int(inp[2]))

print(*res, sep='\n')
```

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.