New Zealand Informatics Competition 2020
Round 1 Solutions

March 3, 2020

Overview

Questions

1. **Rimuru's Road**
2. **Paddocks**
3. **Lineup**
4. **Lake Waikaremoana**
5. **Blast Off**

The solutions to these questions 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?

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

Read about how the server marking works:

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

See our list of other useful resources here:

`www.nzoi.org.nz/nzic/resources.html`

Tips for next time

Remember, this is a contest. The only thing we care about is that your code runs. It doesn't need to be pretty or have comments. There is also no need to worry about invalid input. Input will always be as described in the problem statement. For example, the code below is not necessary.

```python
# Not needed
def error_handling(prompt):
    while True:
        try:
            N = int(input(prompt))
            if N < 0 or N > 100:
                print('That was not a valid integer!')
            else:
                return N
        except ValueError:
            print('Not a valid integer')
...
```

There are a few other things students can do to improve their performance in contests.

Practice getting input

A number of students tripped up on processing input with multiple integers on a single line. A neat trick for processing this sort of input in Python is to use the `str.split()` method and the `map()` function. The `split()` method will break up a string at space characters, returning a list of the words. The `map()` function can be used to apply `int()` to each string in this list, converting them to integers. For example, suppose we have the following line of input:

```
1 4 2 7
```

We can turn this into a list of integers with the Python statement

```python
my_ints = list(map(int, input().split()))
```

Notice that we used `list()`. This is because `map()` returns us a special generator object, not a list. However, generator objects are easily converted to lists.

We suggest having a go at some of the NZIC Practice Problems.

Move on to the next question

If you are spending too much time on a question, move on. There could be easy subtasks waiting in the next question. Often, you will think of a solution to your problem while working on another question. It also helps to come back to a question with a fresh pair of eyes.

Take time to read the questions

Don't underestimate the value of taking time to read and understand the question. You will waste exponentially more time powering off thinking you have the solution only to discover you missed something obvious.

In the real world, it is very rare to be provided with a problem in a simplistic form. Part of the challenge with these contests is reading and understanding the question, then figuring out what algorithm will be needed. Before submitting, check to make sure you have done everything the question asks you to do.

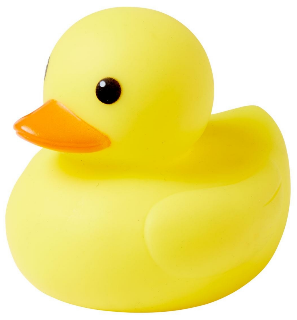
Test your code first!

It is much more time efficient to test your solutions on your own computer first. It can take a while to wait for the server to run your code. It is far more efficient to test it yourself first, submit it, then start reading the next question while you wait for your previous solution to be marked by the server.

While testing, remember to add some edge case tests. For example, if a question specifies "$1 \leq N \leq 10$" then try out an input where $N = 1$. Think of other tricky inputs that might break your code. Look at the feedback the server is giving you.

Use a rubber duck

`https://en.wikipedia.org/wiki/Rubber_duck_debugging`



Rimuru's Road

https://train.nzoi.org.nz/problems/1113

For each type of facility, we only need to consider the two closest facilities, which are the previous and next facilities from the current location. Suppose that we are currently at location D, and we are considering a facility that occurs every K kilometres. We could use integer division to find the location of the previous facility then add K to find the next location. Repeat this step for each facility and record the smallest difference between these facility locations and our current location.

```python
# see the 'Practice getting input' section above to understand 'map'
facilities = list(map(int, input().split()))
location = int(input())
distances = []

for x in facilities:
    prev_facility = location // x * x  # the // is integer divison
    next_facility = prev_facility + x
    distance = min(location - prev_facility, next_facility - location)
    distances.append(distance)

min_distance = min(distances)
print(min_distance)

# determine if there is more than one facility at the same distance
dist_count = 0
for d in distances:
    if d == min_distance:
        dist_count += 1

if dist_count > 1:
    print("can't make up my mind")
```

Alternatively, we can use the modulo operator (%) to calculate the distance to the previous facility as $D \% K$. Then the next facility must be at location $D - (D \% K) + K$, which is at a distance of $K - (D \% K)$ from the current location.

Python

```python
facilities = map(int, input().split())
location = int(input())

dists = sorted(min(location % x, x - location % x) for x in facilities)

print(dists[0])
if dists[0] == dists[1]:
    print("can't make up my mind")
```

C++

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

int main() {
    int location;
    int min_distances[3];
    for (int i = 0; i < 3; ++i) {
        cin >> min_distances[i];
    }
    cin >> location;
    for (int i = 0; i < 3; ++i) {
        int remainder = location % min_distances[i];
        min_distances[i] = min(remainder, min_distances[i] - remainder);
    }
    sort(min_distances, min_distances + 3);

    cout << min_distances[0] << '\n';
    if (min_distances[0] == min_distances[1])
        cout << "can't make up my mind\n";
}
```

Paddocks

https://train.nzoi.org.nz/problems/1120

Subtask 1

There are lots of details in this problem which make it easy to miss something. A good approach is to break it down into steps and tackle them one at a time. For the first subtask, we only need to determine if the farm is healthy or unhealthy. A farm is unhealthy if and only if the average for each of the three years is at least 12%.

1. We calculate the average as the sum of the saturation values divided by the number of saturation values for that year.
2. Do this for all three years.
3. If any year has an average saturation of less than 12%, then output `healthy`, otherwise output `unhealthy`.

A partial solution might look like this...

```python
unhealthy = True
for year in range(3):
    N = int(input())  # The number of saturation values
    total = sum(map(int, input().split()))

    if total < N * 12:
        unhealthy = False

if unhealthy:
    print("unhealthy")
else:
    print("healthy")
```

On line 6 we use `total < N * 12`, which is logically equivalent to `total / N < 12`. We have moved `N` to the right-hand side of the inequality to avoid a floating-point division and comparison. While it isn't strictly necessary in this particular case, it is usually a good idea to avoid using floating-point numbers if possible – not all numbers can be represented exactly in a float, and this can give unexpected results.[1] Note that integer division would also be acceptable here.

Subtask 2

Now that we know if the land is healthy or unhealthy, we can modify the code to check if the farmer should resow or not. For the first year only, we check if at least half of saturation values are at least 25%.

Python Solution

```python
unhealthy = True
resow = False
for year in range(3):
    N = int(input())  # The number of saturation values
    saturations = list(map(int, input().split()))

    # If the average saturation is below 12 for any year,
    # the farm is healthy.
    if sum(saturations) < N * 12:
        unhealthy = False

    if year == 0:  # for the first year only
        # resow if at least half of the saturations are at least 25%
        resow_count = sum(x >= 25 for x in saturations)
        if resow_count * 2 >= N:
            resow = True
        # (resow_count * 2 >= N) is equivalent to (resow_count >= N / 2)

if unhealthy and resow:
    print("resow")
elif unhealthy:
    print("unhealthy")
else:
    print("healthy")
```

[1] https://floating-point-gui.de/errors/comparison/

Note that `resow_count = sum(x >= 25 for x in saturations)` uses a list comprehension. This is equivalent to saying

```
resow_list = []
for x in saturations:
    resow_list.append(x >= 25)
resow_count = sum(resow_list)
```

And yes, it is legal to sum the result of a `True`/`False` condition. If the condition is `True` it will be evaluated as 1 otherwise it will count as 0.

C++ Solution

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

int main() {
    bool unhealthy = true;
    bool resow = false;
    int N, saturation;

    for (int year = 0; year < 3; year++) {
        cin >> N;
        int total = 0;  // total saturation
        int resow_count = 0;  // number of saturation values >= 25

        for (int i = 0; i < N; i++) {
            cin >> saturation;
            total += saturation;
            if (saturation >= 25) resow_count++;
        }

        if (year == 0 && resow_count * 2 >= N) resow = true;
        // (resow_count*2 >= N) is equivalent to (resow_count >= N/2)

        if (total < N * 12) unhealthy = false;
        // (total < N*12) is equivalent to (total/N < 12)
    }

    if (unhealthy && resow) cout << "resow\n";
    else if (unhealthy) cout << "unhealthy\n";
    else cout << "healthy\n";
}
```

Lineup

https://train.nzoi.org.nz/problems/1144

Subtask 1

There are a few different strategies we can use. Here's one: for each player x in the rivalling team, we try to match x against the weakest unmatched player that can beat x from Toddy's team.

Our first implementation of this strategy finds the matching player by iterating over all N players in Toddy's team (for each of the N players in the rivalling team), which runs in $O(N^2)$ time.

```python
n = int(input())
toddys_team = list(map(int, input().split()))
rivalling_team = list(map(int, input().split()))
wins = 0
for x in rivalling_team:  # for each player x in the rivalling team
    # find the weakest player that can beat player x
    best = 1000000
    for y in toddys_team:
        if y > x:
            best = min(best, y)
    if best != -1:
        toddys_team.remove(best)
        wins += 1
print(wins)
```

Subtask 2

Notice that the actual order of the matches is not important – we can sort the players in each team by skill level without affecting the result.

Now consider what would happen if we ran our subtask 1 solution with *sorted* teams. If we iterate over each player from the rivalling team in ascending order of skill level, then the optimal matching player from Toddy's team will also always be in ascending order of skill level. Therefore, we actually only need to iterate over Toddy's team once – for each player y in Toddy's team (in ascending order), we try to match y against the weakest unmatched player from the rivalling team. The time complexity of this solution is $O(N)$.

Python Solution

```python
n = int(input())
toddys_team = sorted(map(int, input().split()))
rivalling_team = sorted(map(int, input().split()))
wins = 0
for y in toddys_team:
    if y > rivalling_team[wins]:
        wins += 1
print(wins)
```

C++ Solution

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

int n, wins;
int a[100010];
int b[100010];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) cin >> a[i];
    for (int i = 0; i < n; i++) cin >> b[i];
    sort(a, a+n);
    sort(b, b+n);
    for (int i = 0; i < n; i++) {
        if(a[i] > b[wins]) wins++;
    }
    cout << wins << '\n';
}
```

Lake Waikaremoana

https://train.nzoi.org.nz/problems/1125

Subtask 1

We need to calculate the overall difficulty when starting from the first hut and going clockwise. Let's switch to 0-based indexing for the trails, and let `ratings[i]` be the difficulty rating of trail i.

- During the first day we'll take trail 0 with N units of food.
  Actual difficulty: `ratings[0]` $\times N$.
- During the second day we'll take trail 1 with $N - 1$ units of food.
  Actual difficulty: `ratings[1]` $\times (N - 1)$.
- During the i-th day (0-based) we'll take trail i with $N - i$ units of food.
  Actual difficulty: `ratings[i]` $\times (N - i)$.

These actual difficulties can be calculated and added up using a `for`-loop to get the overall difficulty.

```python
def calc_score_clockwise_from_first_hut(n, ratings):
    # calculates the overall difficulty when starting from the first
    # hut and going clockwise
    answer = 0
    for i in range(n):
        # i is the current day, starting from 0
        # (n-i) is the amount of food carried
        answer += ratings[i] * (n-i)
    return answer

n = int(input())
ratings = list(map(int, input().split()))
print(calc_score_clockwise_from_first_hut(n, ratings))
```

Subtask 2

We need to calculate the overall difficulty when starting from each hut going clockwise and pick the minimum. Calculating the overall difficulty when starting from hut S (0-based) is similar to subtask 1, except that during the i-th day we take trail $(S+i)\,\%\,N$. The modulo operator is used to wrap around to trail 0 after trail $N-1$.

```python
import math

def calc_score_clockwise(n, ratings, start):
    # calculates the overall difficulty when starting from the given
    # hut and going clockwise
    answer = 0
    for i in range(n):
        # i is the current day, starting from 0
        # (start+i) % n is the trail taken (0-based, with wrap-around)
        # (n-i) is the amount of food carried
        answer += ratings[(start+i)%n] * (n-i)
    return answer

def solve_clockwise(n, ratings):
    # calculates the minimum overall difficulty when starting from any
    # hut and going clockwise
    answer = math.inf
    for start in range(n):
        current = calc_score_clockwise(n, ratings, start)
        answer = min(answer, current)
    return answer
```

Subtask 3

In this subtask we also need to consider walking in the anticlockwise direction.

One approach is to copy the function `solve_clockwise` and change the trail calculation to $(S-i)\,\%\,N$ to walk anticlockwise. (This formula works in languages like Python that use floored division, but in languages like C++ that use truncated division it needs to be changed to $(S-i+N)\,\%\,N$ to avoid negative results.)

Another approach is run `solve_clockwise` once normally, then reverse the difficulty ratings and run it a second time to consider walking anticlockwise.

Subtask 4

The time complexity of the previous algorithm is $O(N^2)$, because there are N starting huts to consider and for each one calculating the overall difficulty is $O(N)$. This is too slow when N is 100 000.

It turns out that the overall difficulty when starting from a particular hut can be computed efficiently based on the overall difficulty when starting from the previous hut.

Suppose $N = 4$ and the difficulty ratings of the trails are a, b, c, d. When starting from the first hut, the overall difficulty is $4a + 3b + 2c + d$. When starting from the second hut, the overall difficulty is $4b+3c+2d+a$, which can be reordered as $a + 4b + 3c + 2d$.

The expressions for the two starting huts are related: the coefficients have all gone up by one (e.g. $3b \Rightarrow 4b$), except for a which changed from $4a$ to a. So the difference is $b+c+d-3a$, which can be rewritten as $(a+b+c+d)-4a$. This can be computed efficiently, since the first term is the sum of all the difficulty ratings and can be precomputed.

So if we know the overall difficulty when starting from hut $i - 1$, we can add `sum_ratings` $- N \times$ `ratings[i-1]` to get the overall difficulty when starting from hut i.

```python
def calc_score_clockwise_from_first_hut(n, ratings):
    answer = 0
    for i in range(n):
        answer += ratings[i] * (n-i)
    return answer

def solve_clockwise(n, ratings):
    sum_ratings = sum(ratings)
    current = calc_score_clockwise_from_first_hut(n, ratings)
    answer = current
    for i in range(1, n):
        current = current + sum_ratings - ratings[i-1]*n
        answer = min(answer, current)
    return answer

def solve(n, ratings):
    answer_clockwise = solve_clockwise(n, ratings);
    answer_anticlockwise = solve_clockwise(n, list(reversed(ratings)))
    return min(answer_clockwise, answer_anticlockwise)
```

Blast Off

https://train.nzoi.org.nz/problems/1133

Subtask 1

This subtask is equivalent to finding the optimal (possibly negative) coefficients a and b, such that $af_1+bf_2 = s_1$. The proof is left as an exercise for the reader.

Since there are only T tiles, we know that the optimal strategy uses at most T rockets (otherwise it would visit some tiles twice, which is never optimal). Therefore, we can try all possible values of a from $-T$ to T. For each possible value of a, we determine if there exists a corresponding b which satisfies the equation. If so, we calculate the cost of this strategy as $|a|c_1 + |b|c_2$.

```python
import math

R, N, T = map(int, input().split())
c1, f1 = map(int, input().split())
c2, f2 = map(int, input().split())
start = int(input())
sol = math.inf
for a in range(-T, T+1):
    if (start + a*f1) % f2 == 0:
        b = (start + a*f1) // f2
        sol = min(sol, abs(a)*c1 + abs(b)*c2)
print(sol)
```

Subtask 2

We can visualise the board as a graph, where each tile is a node, and we have R directed edges for each tile, going from tile x to $|x - f_i|$. For this subtask, $c_i = 1$ for all i, so this is an *unweighted* graph (all edges are of unit length). Therefore, this subtask can be solved using BFS,[2] where the source node is the starting tile (s_i) and the destination is tile 0. Note that we do not have to explicitly construct an adjacency list – we can simply iterate over the R rockets to get the adjacent nodes. The time complexity of BFS is $O(V+E)$ where V is the number of vertices (nodes) and E is the number of edges, so in this case it is $O(T + RT) = O(RT)$. Since there are N different starting positions, the overall time complexity of this solution is $O(NRT)$.

```python
import collections

R, N, T = map(int, input().split())
jumps = []
for i in range(R):
    c, f = map(int, input().split())
    jumps.append((c,f))

for i in range(N):
    x = int(input())
    cost = [-1]*10010
    q = collections.deque()
    q.append(x)
    cost[x] = 0
    while len(q) > 0:
        cur_dist = q.popleft()
        for c,f in jumps:
            next_dist = abs(cur_dist - f)
            if cost[next_dist] == -1:
                cost[next_dist] = cost[cur_dist] + 1
                q.append(next_dist)
    print(cost[0])
```

[2]See the NZIC 2019 Round 3 solutions for an explanation of BFS.

Subtask 3

Performing a BFS from each player's starting tile is too slow. Instead, we can reverse the edges of our graph and perform a single BFS starting from the goal, tile 0. Note that if we do not explicitly construct an adjacency list (and reverse the edges), we will need to check for two possible edges per rocket for each tile – one going "forwards", from x to $x + f_i$, and one going "backwards", from x to $|x - f_i|$. The time complexity of this solution is $O(RT)$.

```python
import collections

R, N, T = map(int, input().split())
jumps = []
for i in range(R):
    c, f = map(int, input().split())
    jumps.append((c,f))

cost = [-1]*10010
q = collections.deque()
q.append(0)
cost[0] = 0
while len(q) > 0:
    cur_dist = q.popleft()
    for c,f in jumps:
        next_dist = cur_dist + f
        if next_dist < T and cost[next_dist] == -1:
            cost[next_dist] = cost[cur_dist] + 1
            q.append(next_dist)
        next_dist = f - cur_dist
        if next_dist > 0 and cost[next_dist] == -1:
            cost[next_dist] = cost[cur_dist] + 1
            q.append(next_dist)
for i in range(N):
    dist = int(input())
    print(cost[dist])
```

Subtask 4

For the full solution, we need to use an algorithm that works on *weighted* graphs. Our solution uses Dijkstra's algorithm. It works very similarly to BFS – the main difference is that instead of a queue, we use a priority queue (heap), so that we always process nodes with the lowest total cost first.

Python Solution

In Python, we recommend using the `heapq` module. `queue.PriorityQueue` can also be used, though it is slightly slower as it's designed for multi-threaded use.

```python
import math
import heapq

R, N, T = map(int, input().split())
jumps = []
for i in range(R):
    c, f = map(int, input().split())
    jumps.append((c,f))

cost = [math.inf]*10010
pq = []
heapq.heappush(pq, (0,0))
cost[0] = 0
while len(pq) > 0:
    cur_cost, cur_dist = heapq.heappop(pq)
    if cost[cur_dist] != cur_cost: continue
    for c,f in jumps:
        next_dist = cur_dist + f
        next_cost = cur_cost + c
        if next_dist < T and cost[next_dist] > next_cost:
            cost[next_dist] = next_cost
            heapq.heappush(pq, (next_cost, next_dist))
        next_dist = f - cur_dist
        if next_dist > 0 and cost[next_dist] > next_cost:
            cost[next_dist] = next_cost
            heapq.heappush(pq, (next_cost, next_dist))
for i in range(N):
    dist = int(input())
    print(cost[dist])
```

C++ Solution

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef pair<int,int> ii;

int r,n,t,x;
int c[50];
int f[50];
int cost[10010];

int main() {
    cin >> r >> n >> t;
    for (int i = 0; i < r; i++) cin >> c[i] >> f[i];
    for (int i = 0; i < 10010; i++) cost[i] = INT_MAX;
    priority_queue<ii, vector<ii>, greater<ii>> pq;  // min-heap
    pq.push({0,0});
    cost[0] = 0;
    while (!pq.empty()) {
        int cur_cost = pq.top().first;
        int cur_dist = pq.top().second;
        pq.pop();
        if (cost[cur_dist] != cur_cost) continue;
        for (int i =  0; i < r; i++) {
            int next_dist = cur_dist + f[i];
            int next_cost = cur_cost + c[i];
            if (next_dist < t && cost[next_dist] > next_cost) {
                cost[next_dist] = next_cost;
                pq.push({next_cost, next_dist});
            }
            next_dist = f[i] - cur_dist;
            if (next_dist > 0 && cost[next_dist] > next_cost) {
                cost[next_dist] = next_cost;
                pq.push({next_cost, next_dist});
            }
        }
    }
    for (int i = 0; i < n; i++) {
        cin >> x;
        cout << cost[x] << '\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 will be too slow 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 it 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. This will take the same amount of computation regardless of N. You can't get much better efficiency than that.

$O(\log N)$ time suggests the program takes a couple of extra operations every time N doubles in size.[3] 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

[3]More formally, it means there exists some constant c for which the program takes at most c extra operations every time N doubles in size.

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 up to $N \leq 10\,000$.