New Zealand Informatics Competition 2022
Round 2 Solutions

August 7, 2022

Overview

Questions

1. **Shopping List**
2. **Doubles**
3. **Skill Issue**
4. **Contest Supervision III**

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`

Ever wondered why your submission scored zero?

`Why did I score zero? - some common mistakes`

See our list of other useful resources here:

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

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.

```
# 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

```
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.

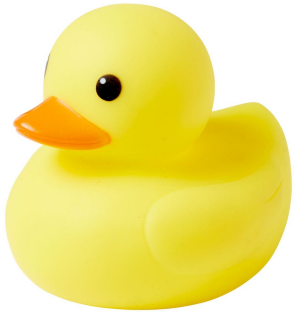
Test your code first!

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



Shopping List

https://train.nzoi.org.nz/problems/1241

For each item, we should buy as much stock as we can (up to as many as we want) from the cheaper supermarket. But if there isn't enough in stock, we need to buy any remaining items from the more expensive supermarket.

```
N = int(input())
cost = 0
for _ in range(N):
    w,a,x,b,y = map(int, input().split())
    if x > y:
        a,x,b,y = b,y,a,x
    cost += min(w, a) * x
    w -= min(w, a)
    cost += min(w, b) * y
print(cost)
```

Doubles

https://train.nzoi.org.nz/problems/1270

Let a, b, c, d be the strengths of any four contestants, such that $a \leq b \leq c \leq d$. There are four possible ways to pair up these contestants:

- (a, b) and (c, d)
- (a, c) and (b, d)
- (a, d) and (b, c)

Notice that the third option is always optimal, since $a + b \leq a + c \leq \boldsymbol{a + d} \leq b + d \leq c + d$, and $a + b \leq a + c \leq \boldsymbol{b + c} \leq b + d \leq c + d$ – (a, d) and (b, c) are both at least as strong as than the weakest teams and at most as strong as the strongest teams from the other two options. If a and d are the weakest and strongest contestants in the tournament, then this proves that it is always optimal to pair up players a and d. We can then repeat this process and pair up the second weakest and second strongest players, and so on until all players are paired up.

```
N = int(input())
s = list(map(int,input().split()))
s.sort()
pairs = []
for i in range(N//2):
    pairs.append(s[i] + s[N-1-i])
print(max(pairs) - min(pairs))
```

Skill Issue

https://train.nzoi.org.nz/problems/1269

Subtask 1

Since all skill points have the same value, we can try all possible combinations in $O(N)$ time.

```
N = int(input())
v = list(map(int,input().split()))
best = 0
accuracy = 0
attack = sum(v)
for p in v:
    accuracy = min(100, accuracy + p)
    attack -= p
    best = max(best, accuracy * attack)
print(best)
```

Subtask 2

Let k be the sum of all skill points, $1+2+\cdots+N$. If we ignore the limit on accuracy, it shouldn't be too hard to see that to maximise damage, we want accuracy and attack to be as close as possible to each other – ideally, accuracy and attack both have a value of $k/2$, assuming k is even. But it turns out that it is always possible to select some subset of numbers from 1 to N that sum to $k/2$. In fact, we can always select some subset of numbers from 1 to N that sum to any number from 1 to k. The proof is left as an exercise to the reader.

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

```
total = sum(v)
accuracy = min(100, total // 2)
attack = total - accuracy
print(accuracy * attack)
```

Subtask 3

In this subtask, N is low enough to brute force across all possible combinations. The time complexity of this solution is $O(N \times 2^N)$.

```
import itertools

N = int(input())
v = list(map(int,input().split()))
total = sum(v)

best = 0
for length in range(N+1):
    for comb in itertools.combinations(v, length):
        accuracy = sum(comb)
        attack = total - attack
        best = max(best, min(100, accuracy) * attack)
print(best)
```

Subtask 4

The full solution requires use of dynamic programming. Firstly, notice that it is never optimal to have an accuracy value of 200 or greater, since the maximum value of a skill point is 100, and increasing accuracy above 100 has no effect on damage. Now, suppose we know all the possible accuracy values that could be obtained using the first k skill points. Then if we add the $(k+1)$th point, we can update our list of possible accuracy values in a single loop over the 200 possible values. We can repeat this process for each skill point to determine all possible accuracy values in $O(N)$ time.

```
N = int(input())
v = list(map(int, input().split()))
possible = [False] * 200
possible[0] = True  # It is always possible to have 0 accuracy

# For each skill point, update our list of possible accuracy values
for p in v:
```

```
    for i in range(199, p-1, -1):
        possible[i] |= possible[i - p]

total = sum(v)
best = 0
for accuracy in range(200):
    if possible[accuracy]:
        attack = total - accuracy
        best = max(best, min(100, accuracy) * attack)
print(best)
```

Contest Supervision III

https://train.nzoi.org.nz/problems/1267

Subtask 1

In this subtask there is only ever a single contestant. Therefore, you should always sit in the same column as the contestant, as sitting in any other column will result you being further away from that contestant.

Python Subtask 1 Solution

```python
_dontcare = input()
_dontcare = input()
row, col = input().split()
print(col)
```

C++ Subtask 1 Solution

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

using namespace std;

long long N, R, C, row, col;

int main() {
    cin>>R>>C>>N;

    cin>>row>>col;

    cout<<col<<endl;
}
```

Subtask 2

In this subtask the number of contestants and the size of the room are relatively small. We can just try to sit at every possible column and choose which one is the best. For each column, we'll calculate the eye strain by looping through each contestant and choosing the furthest one from us. The distance to each contestant can be calculated using Pythagoras' Theorem. Let's say that we're currently at row 0, column x. Let's say that a contestant is at row r, column c. Then the distance from us to that contestant will be $\sqrt{(c-x)^2+(r-0)^2} = \sqrt{(c-x)^2+r^2}$.

So to find the furthest contestant, can we just calculate the distance using the formula above and pick the one with the greatest distance? Yes - but there's a small catch. The locations of the contestants are all integers, which a computer can store without any issue. However, to find the distance, we have to use the square root operation, which will may not produce an integer. This result would have to be stored in a **floating-point number**, which may result in a rounding error (see this post for an explanation).

For this subtask, the rounding error may not be an issue, because the distances are relatively small and therefore the errors would be quite small as well. However, for future subtasks where the distances could be more than a billion meters, this becomes a problem. So how do we work around this? Note that for this problem we don't actually care what the exact distance to each contestant is - we only care which contestant has the furthest distance. This means that we can just calculate $(c-x)^2+r^2$ - i.e. the square of the distance, as we've removed the square root - and compare those instead. If a contestant's squared distance is greater than another contestant's, then they must be further away.

Python Subtask 2 Solution

```
R, C = map(int, input().split())
N = int(input())

students = []
for i in range(N):
    r, c = map(int, input().split())
    students.append((r, c))

min_dist = 10000000
best_col = 0
for col in range(C):
    furthest_student = 0
```

```
    for r, c in students:
        dist = (col - c) ** 2 + r ** 2
        furthest_student = max(dist, furthest_student)

    if furthest_student < min_dist:
        min_dist = furthest_student
        best_col = col

print(best_col)
```

C++ Subtask 2 Solution

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

using namespace std;

long long N, R, C, ri, ci;

int main() {
    cin>>R>>C>>N;

    vector<pair<long long, long long>> points;

    for (int i=0; i<N; i++) {
        cin>>ri>>ci;
        points.emplace_back(ri, ci);
    }

    long long min_dist = LLONG_MAX;
    long long best_col = 0;
    for (int col=0; col<C; col++) {
        long long furthest_dist = 0;
        for (auto point: points) {
            long long dist = (col - point.second) * (col - point.second)
            ↪  + point.first * point.first;
            furthest_dist = max(furthest_dist, dist);
        }
        if (furthest_dist < min_dist) {
            min_dist = furthest_dist;
            best_col = col;
        }
    }

```

```
    cout<<best_col<<endl;
}
```

Subtask 3

In this subtask, we can't use the same strategy of trying out all the possible columns, as the number of columns is very large. However, the number of rows is still quite small. Can this help us narrow down the number of columns we have to try out?

Imagine a situation where there was a contestant at column 0 and a contestant at column 500,000,000. Should we bother checking if we should sit at column 499,999,999? At that column we are 499,999,999 meters away from the contestant at column 0, so there must be some other contestant that's at least that far away from us on the other side (otherwise we could just move closer to column 0 and reduce our distance). However, this is impossible as the number of rows is limited to 1,000 - so the furthest contestant from us on the right side would be at row 1,000, which is closer than 499,999,999 meters away. In general, because of this property, we only need to search for a small interval around the midpoint of the lowest and highest numbered columns that are occupied by contestants.

C++ Subtask 3 Solution

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

using namespace std;

long long N, R, C, F, ri, ci;

int main() {
    cin>>R>>C>>N;

    vector<pair<long long, long long>> points;

    for (int i=0; i<N; i++) {
        cin>>ri>>ci;
        points.emplace_back(ri, ci);
    }

    pair<long long, long long> min_col_point =
    ↪  *min_element(points.begin(), points.end(), [](const auto &a,
    ↪  const auto &b) {return a.second < b.second; });
```

```
    pair<long long, long long> max_col_point =
    ↪  *max_element(points.begin(), points.end(), [](const auto &a,
    ↪  const auto &b) {return a.second < b.second; });

    auto min_col = min_col_point.second;
    auto max_col = max_col_point.second;
    auto midpoint = (min_col + max_col) / 2;

    long long min_dist = LLONG_MAX;
    long long best_col = 0;
    for (int col=midpoint - 2000; col<midpoint + 2000; col++) {
        long long max_dist = 0;
        for (auto point: points) {
            long long dist = (col - point.second) * (col - point.second)
            ↪  + point.first * point.first;
            max_dist = max(max_dist, dist);
        }
        if (max_dist < min_dist) {
            min_dist = max_dist;
            best_col = col;
        }
    }

    cout<<best_col<<endl;
}
```

Subtask 4

We can extend the idea from Subtask 3 of only looking at columns that could be worth trying out. Let's pick an arbitrary column c. We find the furthest contestant F from that position, which is d meters away, and observe that C_f is to our left (i.e. it's in a lower numbered column than c). What does that tell us? Well, consider a column c_r, which is to the right of c (i.e. $c_r > c$). The distance from c_r to F must be greater than d. If F is also the furthest contestant from c_r, then the eye strain is greater than d. Otherwise, there must be some other contestant that is even further away from c_r than F is, in which case the eye strain is even greater than d! In both cases, then, the eye strain for column c_r must be greater than the eye strain at c.

Therefore, the eye strain for every column to the right of c must be greater than the eye strain at c. That means that there's no point even trying out any of the columns to the right of c. Similarly, if the furthest column from c is to the right of

c, there's no point trying out any of the columns to the left of c. What if there's a tie, and there's two 'furthest' contestants that are the same distance on either side of c? Well in that case, every column to our left results in more eye strain, every column to our right results in more eye strain, so that means that c must be the best column!

We can use the above observations to *binary search* for the best column. We keep an upper and lower bound for the possible best columns, try out a column between the bounds, and then use that knowledge to narrow the bounds down. Both the upper and lower bound could potentially be the best column, so we keep narrowing down the bounds until they're adjacent and then try out both to see which one's better.

Python Subtask 4 Solution

```
R, C = map(int, input().split())
N = int(input())

points = []

for i in range(N):
    r, c = map(int, input().split())
    points.append((r, c))

upper = 10000000000
lower = 0

def get_distance(col, point):
    return (point[1]-col)*(point[1]-col) + point[0]*point[0]

def get_furthest(col):
    return max(points, key=lambda point: get_distance(col, point))

while upper - lower > 1:
    col = (upper + lower) // 2

    furthest_r, furthest_c = get_furthest(col)
    if furthest_c < col:
        upper = col
    else:
        lower = col

```

```python
if get_distance(lower, get_furthest(lower)) <= get_distance(upper, get_furthest(upper)):
    print(lower)
else:
    print(upper)
```

C++ Subtask 4 Solution

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

using namespace std;

int N, R, C, ri, ci;
vector<pair<long long, long long>> points;

long long get_distance(int col, pair<long long, long long> point) {
    return (col - point.second) * (col - point.second) + point.first * point.first;
}

pair<long long, long long> get_furthest(int col) {
    long long furthest_dist = 0;
    pair<int, int> furthest_point;
    for (auto point: points) {
        long long dist = get_distance(col, point);
        if (dist > furthest_dist) {
            furthest_dist = dist;
            furthest_point = point;
        }
    }
    return furthest_point;
}


int main() {
    cin>>R>>C>>N;

    for (int i=0; i<N; i++) {
        cin>>ri>>ci;
        points.emplace_back(ri, ci);
    }

    int upper = 1000000000;
```

```
35      int lower = 0;
36      while (upper - lower > 1) {
37          int col = (upper + lower) / 2;
38          long long furthest_col = get_furthest(col).second;
39          if (furthest_col < col) {
40              upper = col;
41          } else {
42              lower = col;
43          }
44      }
45
46      if (get_distance(lower, get_furthest(lower)) <= get_distance(upper,
        ↪  get_furthest(upper))) {
47          cout<<lower<<endl;
48      } else {
49          cout<<upper<<endl;
50      }
51  }
```

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 couple of extra operations every time

N doubles in size.[1] 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 $O(N)$.

$O\left(N^2\right)$ 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.

[1] 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.