New Zealand Informatics Competition 2021
Round 3 Solutions

July 19, 2021

Overview

Questions

1. **Dorothy's Red Shoes**
2. **Bulk Buying**
3. **KB Lo-Fi**
4. **Unofficial Contestants**
5. **Tim Jhomas Returns**

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.

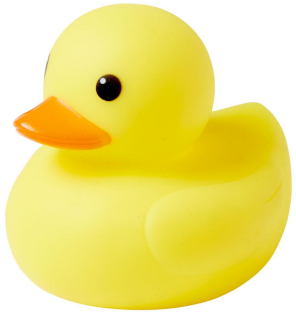
Test your code first!

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



Dorothy's Red Shoes

To figure out if Dorothy is in the room we need to count how many red shoes there are. In other words, the number of 'R's in the input string need to be counted. You are assured that no one else would wear Red shoes, so the only possible counts are 0 (she's missing), 1 (needs to find the other shoe) or 2 (she's in the classroom).

Most languages has a built-in method to count the number of occurrences of a character in a string. For example, counting can be achieved by using a string's `.count()` method in Python, or the `count()` function in C++.
Otherwise, keep a running total of "R"s, initially set to 0. Then, iterate over the string, adding 1 to the total if "R" is found.

All we then need to do is check if that number is 0, 1, or 2 and output the associated message.

https://train.nzoi.org.nz/problems/1228

Python Solution

```
shoes_in = input()
num_red = shoes_in.count('R')

'''  If not using the count() method
num_red = 0
for shoe in range(shoes_in):
    if shoe == 'R':
        num_red += 1
'''

if num_red == 2:
    print("Dorothy is in the classroom.")
elif num_red == 1:
    print("Hop along Dorothy and find that other shoe.")
```

```
else:
    print("Maybe Dorothy is in Kansas.")
```

C++ Solution

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

using namespace std;

int main() {
    string shoes_string;
    cin>>shoes_string;

    int num_red = count(shoes_string.begin(), shoes_string.end(), 'R');

    if (num_red == 2) {
        cout<<"Dorothy is in the classroom."<<endl;
    } else if (num_red == 1) {
        cout<<"Hop along Dorothy and find that other shoe."<<endl;
    } else {
        cout<<"Maybe Dorothy is in Kansas."<<endl;
    }
}
```

Bulk Buying

https://train.nzoi.org.nz/problems/1232

Note: all solutions for this problem are in Python.

Subtask 1

If it costs \$2 to buy 2 ducks and \$3 to buy 3 ducks, then each duck always costs \$1. Therefore, the total cost of buying N ducks is always N dollars.

```
print(input())
```

Subtask 2

In this subtask, you can spend \$3 to buy 2 ducks or \$5 to buy 3 ducks. The first method costs $\$3/2 = \$1.50$ per duck, and the second costs $\$5/3 \approx \$1.66$ per duck, so it is clearly better to use the first method as much as possible. The only cases where we cannot buy all of our ducks in lots of 2 is when N is odd – then we simply need to buy a single lot of 3 ducks to make the remaining N even.

```
N = int(input())
A = int(input())
B = int(input())

cost = 0
if N % 2 == 1:  # If N is odd, buy a single lot of 3 ducks
    N -= 3
    cost += B
cost += N // 2 * A

print(cost)
```

Subtask 3

If $A/2 < B/3$, then it is more optimal to buy ducks in lots of 2, so we can use our subtask 2 solution. Otherwise, if $A/2 > B/3$, it is better to buy ducks in lots of 3. But what should we do if N is not a multiple of 3?

Let r be the remainder of dividing N by 3.

- If r is 2, then we can get exactly N ducks by buying a single lot of 2 ducks, and the rest in lots of 3.
- If r is 1, then it is optimal to buy two lots of 2 ducks (as $2 \times 2 = 4$, which has a remainder of 1 when divided by 3), and the rest in lots of 3.

```
N = int(input())
A = int(input())
B = int(input())

cost = 0
if A / 2 <= B / 3:      # Better to buy lots of 2
    if N % 2 == 1:      # If N is odd, buy one lot of 3 ducks
        N -= 3
        cost += B
    cost += N // 2 * A
else:                   # Better to buy lots of 3
    if N % 3 == 1:      # If remainder is 1, buy two lots of 2 ducks
        N -= 4
        cost += 2 * A
    elif N % 3 == 2:    # If remainder is 2, buy one lot of 2 ducks
        N -= 2
        cost += A
    cost += N // 3 * B

print(cost)
```

Alternative solutions

We could instead take a brute-force approach, iterating over every possible value for the number of lots of 2 we should buy, and determining how many lots of 3 we would need to make up the rest.

```
N = int(input())
A = int(input())
B = int(input())

cost = 10**10
for x in range(N // 2 + 1):
    required = N - x * 2  # Ducks needed if we buy x lots of 2
    if required % 3 > 0:
        continue
    y = required // 3     # Buy y lots of 3 ducks to make N
    cost = min(cost, x * A + y * B)

print(cost)
```

Or we could use dynamic programming to calculate the cost of buying every number of ducks up to N in a single for-loop.

```
N = int(input())
A = int(input())
B = int(input())

cost = [0] * (N + 1)
cost[1] = 10**10  # Impossible to buy just 1 duck
cost[2] = A       # Always costs A to buy 2 ducks

for x in range(3, N + 1):
    # The cost of buying x ducks is the minimum of:
    #   the cost of buying x - 2 ducks, plus A
    #   the cost of buying x - 3 ducks, plus B
    cost[x] = min(cost[x - 2] + A, cost[x - 3] + B)

print(cost[N])
```

KB Lo-Fi

https://train.nzoi.org.nz/problems/1225

Subtask 1

When there's only one customer, we might as well sell them the most expensive phone they could possibly buy. We will keep track of the current 'best' phone in a variable, and loop through all available phones. If we see a phone that is more expensive (but that the customer can still buy), then we update that variable.

Python Subtask 1 Solution

```python
n = int(input())

customer = int(input())

m = int(input())

best_phone = 0

for i in range(m):
    phone = int(input())
    if phone <= customer and phone > best_phone:
        best_phone = phone

print(best_phone)
```

C++ Subtask 1 Solution

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

```

```cpp
using namespace std;

int main() {
    int n, m, customer, phone;

    cin>>n>>customer>>m;

    int best_phone = 0;
    for (int i=0; i<m; i++) {
        cin>>phone;
        if (phone <= customer && phone > best_phone) {
            best_phone = phone;
        }
    }

    cout<<best_phone<<endl;
}
```

Subtask 2

Note: *This subtask turned out to be a bit harder than we intended, so don't feel bad if you don't fully understand the solution. Recursion is a hard idea to grasp your head around! The solutions to later subtasks don't rely on this subtask.*

Because there's only a smaller number of customers and phones, we can try every possible assignment of customers to phones. For the first customer, we will try every possible phone that can be assigned to that customer. For each of those assignments, we will then try every remaining phone that can be assigned to the second customer, and so on. This gives us a *recursive* solution to the problem, where the solution to the problem (best assignment of M phones to N customers) relies on the solutions to smaller versions of the problem (best assignment of $M-1$ phones to $N-1$ customers). This solution is similar in spirit to the recursive solution to finding all permutations of a list of items.

Python Subtask 2 Solution

```python
def solve(customers, phones):
    maximum = 0
    if len(customers) == 0 or len(phones) == 0:
        return 0
    for i in range(len(phones)):
        if phones[i] <= customers[0]:
```

```
            maximum = max(maximum, phones[i] + solve(customers[1:],
            ↪  phones[:i] + phones[i+1:]))
    maximum = max(maximum, solve(customers[1:], phones))
    return maximum

n = int(input())
customers = []
for i in range(n):
    customers.append(int(input()))

m = int(input())

phones = []
for i in range(m):
    phones.append(int(input()))

print(solve(customers, phones))
```

C++ Subtask 2 Solution

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

using namespace std;

long solve(vector<int> &customers, vector<int> phones, int
↪  current_customer = 0) {
    long maximum = 0;
    if (current_customer == customers.size() || phones.size() == 0) {
        return 0;
    }

    for (int i = 0; i < phones.size(); i++) {
        if (phones[i] <= customers[current_customer]) {
            vector<int> phones_remaining = vector<int>(phones);
            phones_remaining.erase(phones_remaining.begin() + i);   //
            ↪  Get all unsold phones by removing the current phone

            maximum = max(maximum, phones[i] + solve(customers,
            ↪  phones_remaining, current_customer + 1));
        }
    }
```

```
19      maximum = max(maximum, solve(customers, phones, current_customer +
        ↪  1)); // Include the case where we don't sell any phones to this
        ↪  customer
20
21      return maximum;
22  }
23
24  int main() {
25      int n, m, customer, phone;
26
27      vector<int> customers, phones;
28
29      cin>>n;
30      for (int i=0; i<n; i++) {
31          cin>>customer;
32          customers.push_back(customer);
33      }
34
35      cin>>m;
36      for (int i=0; i<m; i++) {
37          cin>>phone;
38          phones.push_back(phone);
39      }
40
41      cout<<solve(customers, phones)<<endl;
42  }
```

Subtask 3

Instead of trying to think about assigning the phones to all the customers, let's think back to Subtask 1, where we only had to worry about one customer. If we wanted to sell a phone to the customer with the highest bid (who we will call c_1), which phone should it be? Intuitively, it should be the most expensive phone they can buy (which we shall call p_1)! But if we sell p_1 to c_1, does it put us at a disadvantage down the line?
No! Firstly, p_1 is the most expensive phone that c_1 can buy, so by definition, we can't possibly extract more money from c_1. Could we make more money overall by selling that phone to another customer? Also no!

Assume that the optimal solution involves selling p_1 to another customer, c_2. Then, there are two cases:

- Case 1: The optimal solution does not sell a phone to c_1
- Case 2: The optimal solution involves selling another phone, p_2, to c_1

If Case 1 is true, then we can simply sell p_1 to c_1 instead of c_2 and make the same amount of money. If Case 2 is true, then we know that c_2 must be able to buy p_2, as c_2 can buy p_1 and $p_2 \leq p_1$ (remember, p_1 is the most expensive phone we can sell). Therefore, if we instead sold p_1 to c_1 and p_2 to c_2 we would make the same amount of money. **Therefore, it is always in our best interests to sell the most expensive possible phone to the customer with the highest bid.**

We can continue to apply this logic to successive customers. In other words, we will first want to sell the most expensive possible phone to the customer with the highest bid. Then, we want to sell the most expensive possible *remaining* phone to the customer with the next highest (or equal) bid. We can continue this until we either run out of phones that can be sold, or have sold a phone to every customer.

To implement this, we will sort in descending order the list of customer bids. Then, for each bid we iterate through, we loop through the list of phone prices and find the most expensive one we can sell to the current customer. We also need to remember to remove each phone we sell from the list. With N total bids and M total phones to loop through per bid, this algorithm has a time complexity of $O(N * M)$.

Python Subtask 3 Solution

```
n = int(input())

customers = []
for i in range(n):
    customers.append(int(input()))
customers.sort(key=lambda x: -x)

m = int(input())

phones = []
for i in range(m):
    phones.append(int(input()))

profit = 0

```

```python
for customer in customers:
    best_phone = -1

    for phone in phones:
        if phone <= customer and phone > best_phone:
            best_phone = phone

    if best_phone != -1:
        profit += best_phone
        phones.remove(best_phone)

print(profit)
```

C++ Subtask 3 Solution

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

using namespace std;

int main() {
    int n, m, customer, phone;
    vector<int> customers, phones;

    cin>>n;
    for (int i=0; i<n; i++) {
        cin>>customer;
        customers.push_back(customer);
    }
    sort(customers.begin(), customers.end(), greater <>());

    cin>>m;
    for (int i=0; i<m; i++) {
        cin>>phone;
        phones.push_back(phone);
    }

    long long profit = 0;
    for (auto customer: customers) {
        int best_phone = -1;

        for (auto phone: phones) {
            if (phone <= customer && phone > best_phone) {
                best_phone = phone;
```

```
            }
        }

        if (best_phone != -1) {
            profit += best_phone;
            phones.erase(find(phones.begin(), phones.end(),
            ↪  best_phone)); // Delete the phone we just sold
        }
    }
    cout<<profit<<endl;
}
```

Subtask 4

Our Subtask 3 solution's time complexity of $O(N * M)$ is far too slow for the full solution. Is there a way we can avoid having to iterate through all phones to find the best for each customer?

Notice that we will always pick phones in descending order of price. This is because we go through customers in descending order of bid, and will always pick the most expensive phone for each customer. Therefore, to find the best phone for the current customer, we don't need to iterate through all phones - we only need to iterate through all phones cheaper than the phone we picked for the previous customer. Let's sort the phones in descending order of price, and keep track of the index of the previous phone. When we need to find the next best phone, we just need to increment the index until we reach the first phone that our current customer can buy.

This may not seem like that much of an improvement. In the worst case, for a specific customer we could still end up iterating through the whole list of phones. However, in total, for all customers, we will only iterate through each phone at most once - we only ever increment the index, so there are only so many times we can increment before we hit the end of the list! Therefore, the total complexity is $O(N + M)$.

Python Subtask 4 Solution

```
n = int(input())

customers = []
for i in range(n):
    customers.append(int(input()))
```

```
customers.sort(key=lambda x: -x)

m = int(input())

phones = []
for i in range(m):
    phones.append(int(input()))
phones.sort(key=lambda x: -x)

profit = 0

best_phone_index = 0

for customer in customers:
    while best_phone_index < len(phones) and phones[best_phone_index] >
    ↪  customer:
        best_phone_index += 1

    if best_phone_index >= len(phones):
        break

    profit += phones[best_phone_index]
    best_phone_index += 1

print(profit)
```

C++ Subtask 4 Solution

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

using namespace std;

int main() {
    int n, m, customer, phone;
    vector<int> customers, phones;

    cin>>n;
    for (int i=0; i<n; i++) {
        cin>>customer;
        customers.push_back(customer);
    }
    sort(customers.begin(), customers.end(), greater <>());

```

```
    cin>>m;
    for (int i=0; i<m; i++) {
        cin>>phone;
        phones.push_back(phone);
    }
    sort(phones.begin(), phones.end(), greater <>());

    long long profit = 0;
    int best_phone_index = 0;
    for (auto customer: customers) {
        while (best_phone_index < phones.size() &&
        ↪  phones[best_phone_index] > customer) {
            best_phone_index += 1;
        }

        if (best_phone_index >= phones.size()) {
            break;
        }

        profit += phones[best_phone_index];
        best_phone_index += 1;
    }
    cout<<profit<<endl;
}
```

Unofficial Contestants

https://train.nzoi.org.nz/problems/1222

Subtask 1

We can start out by trying to keep track of whether each contestant is official or not. We create an array/list called `isOfficial`, where `isOfficial[i]` is `true` if contestant number i is currently and official contestant (and `false` if unofficial). When we get a toggle query for some contestant j, we just invert the value of `isOfficial[j]`.
Now that we're constantly keeping track of who's official and not, how do we figure out a contestant's rank at any given point? Observe that the official rank of a contestant is simply the number of official contestants ranked *above* it, plus one (itself). Therefore, when we get an output query for contestant j, we count the number of `true` values between indexes 0 and $j-1$ of $isOfficial$. That (plus one) will be the official rank of the contestant.
We have to be careful though - it could be possible that contestant j is unofficial! Therefore, for each output query we must first check whether or not `isOfficial[j]` is `false` - if it is, then we output `"UNOFFICIAL"` instead.

Python Subtask 1 Solution

```
n = int(input())
m = int(input())

official = [True] * 100001
official[0] = False  # No contestant 0 exists

for i in range(m):
    query = input().split()
    query_type = query[0]
```

```
    competitor = int(query[1])

    if query_type == 'o':
        if official[competitor]:
            print(official[:competitor].count(True) + 1)
        else:
            print("UNOFFICIAL")
    else:
        official[competitor] = not official[competitor]
```

C++ Subtask 1 Solution

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

using namespace std;

int main() {
    int n, m, competitior;
    char queryType;

    cin>>n>>m;
    vector<bool> isOfficial(n + 1, true);

    for (int i=0; i<m; i++) {
        cin>>queryType>>competitior;

        if (queryType == 'o') {
            if (!isOfficial[competitior]) {
                cout<<"UNOFFICIAL"<<endl;
            } else {
                cout<<accumulate(isOfficial.begin(), isOfficial.begin()
                ↪  + competitior, 0)<<endl;    // false and true can be
                ↪  coerced into 0 and 1, so we can just add them up
            }
        } else {
            isOfficial[competitior] = !isOfficial[competitior];
        }
    }
}
```

Subtask 2

Our previous approach works well if there are a 'small' number of contestants and queries. ('small' relative to the full solution - in reality we would love to have over 1,000 NZIC competitors!). In this subtask we have $100,000$ contestants and $50,000$ queries. Using the previous strategy, we would have to count around $50,000$ values for each of the $50,000$ queries, resulting in a total of $50,000 \times 50{,}000 = 2,500,000,000$ calculations! Clearly this is a problem.

Luckily for us, the only competitor we output for is competitor number $50,000$. We can build on our observation from the last subtask - the official rank of a contestant is simply the number of official contestants ranked *above* it, plus one (itself). Therefore, we only ever really care about the number of official contestants ranked higher than competitor $50,000$.

Observe that, if an official contestant with a number smaller than $50,000$ becomes unofficial, then the rank of competitor $50,000$ will decrease by one, as $50,000$ has one less contestant ranked higher that it. Conversely, if an unofficial contestant smaller than $50,000$ becomes official then the rank of $50,000$ increases by one. Therefore, we can simply keep the rank of $50,000$ in one variable, and update it using that strategy whenever a toggle query come in. When an output query comes in we just output the value of the stored rank variable.

Python Subtask 2 Solution

```
n = int(input())
m = int(input())

official = [True] * 100001
rank = 50000

for i in range(m):
    query = input().split()
    query_type = query[0]
    competitor = int(query[1])
    if query_type == 'o':
        if official[competitor]:
            print(rank)
        else:
            print("UNOFFICIAL")
    else:
        official[competitor] = not official[competitor]
        if competitor < 50000:
```

```
            if official[competitor]:
                rank += 1
            else:
                rank -= 1
```

C++ Subtask 2 Solution

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

using namespace std;

int main() {
    int n, m, competitor;
    char queryType;

    cin>>n>>m;
    vector<bool> isOfficial(n + 1, true);
    int rank = 50000;

    for (int i=0; i<m; i++) {
        cin>>queryType>>competitor;

        if (queryType == 'o') {
            if (!isOfficial[competitor]) {
                cout<<"UNOFFICIAL"<<endl;
            } else {
                cout<<rank<<endl;
            }
        } else {
            isOfficial[competitor] = !isOfficial[competitor];
            if (competitor < 50000) {
                if (isOfficial[competitor]) {
                    rank += 1;
                } else {
                    rank -= 1;
                }
            }
        }
    }
}
```

Subtask 3

In Subtask 2 we updated the ranks of one competitor whenever the status of others were toggled. In this Subtask we take the idea further.
We only care about the ranks of competitors number $50,000$ to $50,050$. Therefore, when an official contestant becomes unofficial, we decrease the ranks of every competitor between $50,000$ and $50,050$ numbered higher than that contestant. Similarly, when an unofficial contestant becomes official, we increase the ranks of every competitor between $50,000$ and $50,050$ numbered higher than that contestant.

Python Subtask 3 Solution

```
from collections import defaultdict

n = int(input())
m = int(input())

official = [True] * 100001

ranks = list(range(50000, 50051))

for i in range(m):
    query = input().split()
    query_type = query[0]
    competitor = int(query[1])
    if query_type == 'o':
        if official[competitor]:
            print(ranks[competitor - 50000])
        else:
            print("UNOFFICIAL")
    else:
        official[competitor] = not official[competitor]
        for i in range(len(ranks)):
            if 50000 + i > competitor:
                if official[competitor]: # Turned from unofficial to
                ↪ official
                    ranks[i] += 1
                else:   # Turned from official to unofficial
                    ranks[i] -= 1

```

C++ Subtask 3 Solution

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

using namespace std;

int main() {
    int n, m, competitor;
    char queryType;

    cin>>n>>m;
    vector<bool> isOfficial(n + 1, true);
    vector<int> ranks(51);
    iota(ranks.begin(), ranks.end(), 50000); // Fill ranks with values 50000, 50001, 50002 etc.

    for (int i=0; i<m; i++) {
        cin>>queryType>>competitor;

        if (queryType == 'o') {
            if (!isOfficial[competitor]) {
                cout<<"UNOFFICIAL"<<endl;
            } else {
                cout<<ranks[competitor - 50000]<<endl;
            }
        } else {
            isOfficial[competitor] = !isOfficial[competitor];
            for (int i = 0; i <= 50; i++) {
                if (competitor < 50000 + i) {
                    if (isOfficial[competitor]) {  // Turned from unofficial to official
                        ranks[i] += 1;
                    } else {  // Turned from official to unofficial
                        ranks[i] -= 1;
                    }
                }
            }
        }
    }
}
```

Subtask 4

In this case, there are a limited number of queries. Remember that our approaches to the previous subtasks took advantage of the fact that there were only a limit number of distinct competitors we had to output - only competitors number $50,000$ for Subtask 2 and competitors $50,000$ to $50,050$ for Subtask 3. Does the same apply in this subtask?

Note that for this problem, you don't have to process a query immediately after receiving it. In other words, we could take in all the queries, process them, and then print the output for all the queries in one go. This is what's called an *offline* problem (as opposed to an *online* problem, where you have to output after each query).

What does that mean for us? Since there are at most 500 total queries, there are at most 500 total competitors we need to output for. That means there are only 500 total competitors we 'care about'. We could first read in all the input and store all the competitors we will have to output for. We then take a similar approach to the previous subtask - for every toggle query, we update the rank of every competitor we care about that is affected by the toggle.

There's an additional change to the previous subtasks - N can now be up to $1,000,000,000$. If we continued to use our `isOfficial[i]` list, it would likely break the memory limit. Even if we used only one bit per competitor, we will still need at least $1,000,000,000$ bits, or around 125MB. To get around this, we can use a map/dictionary to only store the status of competitors that appear in queries. Thus, the map will contain at most 500 total values.

Python Subtask 4 Solution

```
from collections import defaultdict

n = int(input())
m = int(input())

official = defaultdict(lambda: True)
ranks = {}

queries = []
out_competitors = set()

for i in range(m):
    query = input().split()
```

```
    query_type = query[0]
    competitor = int(query[1])

    queries.append((query_type, competitor))
    if query_type == 'o':
        out_competitors.add(competitor)
        ranks[competitor] = competitor

for query_type, competitor in queries:
    if query_type == 'o':
        if official[competitor]:
            print(ranks[competitor])
        else:
            print("UNOFFICIAL")
    else:
        official[competitor] = not official[competitor]
        for out_competitor in out_competitors:
            if competitor < out_competitor:
                if official[competitor]: # Turned from unofficial to
                ↪  official
                    ranks[out_competitor] += 1
                else:   # Turned from official to unofficial
                    ranks[out_competitor] -= 1

```

C++ Subtask 4 Solution

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

using namespace std;

int main() {
    int n, m, competitor;
    char queryType;

    cin>>n>>m;

    vector<pair<char, int>> queries;
    unordered_set<int> outCompetitors;
    unordered_map<int, bool> isUnOfficial;  // Unofficial instead of
    ↪  official as bools default to false
    unordered_map<int, int> ranks;

```

```
    for (int i=0; i<m; i++) {
        cin>>queryType>>competitor;
        assert(competitor >= 1 && competitor <= n && (queryType == 'o'
        ↪  || queryType == 't'));

        queries.emplace_back(queryType, competitor);
        outCompetitors.insert(competitor);
        ranks[competitor] = competitor;
    }


    for (auto query: queries) {
        queryType = query.first;
        competitor = query.second;

        if (queryType == 'o') {
            if (isUnOfficial[competitor]) {
                cout<<"UNOFFICIAL"<<endl;
            } else {
                cout<<ranks[competitor]<<endl;
            }
        } else {
            isUnOfficial[competitor] = !isUnOfficial[competitor];
            for (auto out_competitor : outCompetitors) {
                if (competitor < out_competitor) {
                    if (isUnOfficial[competitor]) {
                        ranks[out_competitor] -= 1;
                    } else {
                        ranks[out_competitor] += 1;
                    }
                }
            }
        }
    }
}
```

Subtask 5 (extra for experts)

We will return to the approach we used in Subtask 1. There exists data structures that allow modifying an arbitrary range of elements in $O(log(N))$ time, and finding sums of arbitrary ranges in $O(log(N))$. One such example is a Fenwick Tree. If we store an official status as a 1 and unofficial as a 0, then the rank of any competitor

is the sum between the start of the array to the competitor.
There's still one issue - N could be up to $1,000,000,000$. We can't have a Fenwick Tree with $1,000,000,000$ without breaking the memory limit. However, we can combine this with our tricks from Subtask 4. There are up to $50,000$ queries so there can only be at most $50,000$ competitors we care about. Therefore, instead of having one array value for every competitor, what if we had only one for every competitor we receive in the input? Instead of being a 1 or a 0, each value stores the sum of official contestants between that competitor and the previous competitor in the input. To toggle the state of a competitor, we add/subtract 1 to every value after that competitor, using the Fenwick Tree. To output the rank of a competitor, we output the sum between the start of the array and that competitor, also using the Fenwick Tree.

Python Subtask 5 Solution

```
from collections import defaultdict

def get_sum(tree, i):
    s = 0
    i += 1

    while i > 0:
        s += tree[i]
        i -= i & (-i)
    return s

def update(tree, i, v):
    i += 1

    while i < len(tree):
        tree[i] += v
        i += i & (-i)

n = int(input())
m = int(input())

assert (n <= 1000000000)
assert (m <= 50000)

queries = []
competitors = []
isUnOfficial = defaultdict(lambda: False)
```

```python

for i in range(m):
    query = input().split()
    query_type = query[0]
    competitor = int(query[1])

    queries.append((query_type, competitor))
    competitors.append(competitor)

competitors.sort()
tree = [0] * (len(competitors) + 1)
compress = {}

prev = 0
for i in range(len(competitors)):
    update(tree, i, competitors[i] - prev)
    prev = competitors[i]
    compress[competitors[i]] = i


for query_type, competitor in queries:
    competitor = compress[competitor]
    if query_type == 'o':
        if isUnOfficial[competitor]:
            print("UNOFFICIAL")
        else:
            print(get_sum(tree, competitor))
    else:
        isUnOfficial[competitor] = not isUnOfficial[competitor]
        if isUnOfficial[competitor]:
            update(tree, competitor, -1)
        else:
            update(tree, competitor, 1)
```

C++ Subtask 5 Solution

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

using namespace std;

int getSum(vector<int> &tree, int index) {
    int sum = 0;
    index = index + 1;
```

```
8
9       while (index > 0){
10          sum += tree[index];
11          index -= index & (-index);
12      }
13      return sum;
14  }
15
16  void update(vector<int> &tree, int index, int val) {
17      index = index + 1;
18
19      while (index < tree.size()) {
20          tree[index] += val;
21          index += index & (-index);
22      }
23  }
24
25  int main() {
26      int n, m, competitor, q=0;
27      char queryType;
28
29      cin>>n>>m;
30      assert(n <= 1000000000 && n >= 1);
31      assert(m <= 50000 && m >= 1);
32      vector<pair<char, int>> queries;
33      set<int> competitorSet;
34      unordered_map<int, bool> isUnOfficial;
35
36      for (int i=0; i<m; i++) {
37          cin>>queryType>>competitor;
38          assert(competitor >= 1 && competitor <= n && (queryType == 'o'
            ↪  || queryType == 't'));
39
40          queries.emplace_back(queryType, competitor);
41          competitorSet.insert(competitor);
42      }
43
44      vector<int> tree(competitorSet.size() + 1, 0);
45      unordered_map<int, int> compress;
46
47      int i = 0, prev = 0;
48      for (auto c: competitorSet) {
49          update(tree, i, c - prev);
```

```
50          prev = c;
51          compress[c] = i++;
52      }
53
54      for (auto query: queries) {
55          competitor = compress[query.second];
56          if (query.first == 'o') {
57              q ++;
58              if (isUnOfficial[competitor]) {
59                  cout<<"UNOFFICIAL"<<endl;
60              } else {
61                  cout<<getSum(tree, competitor)<<endl;
62              }
63          } else {
64              isUnOfficial[competitor] = !isUnOfficial[competitor];
65              if (isUnOfficial[competitor]) {
66                  update(tree, competitor, -1);
67              } else {
68                  update(tree, competitor, 1);
69              }
70          }
71      }
72
73      assert(q >= 1);
74  }
```

Tim Jhomas Returns

https://train.nzoi.org.nz/problems/problem-link-here

Subtask 1

For subtask 1, the network of spies forms a line - this means the temporary connection must be used to go directly back to Tim. Therefore, we should always open our temporary connection between Tim, and the spy that is the furthest in terms of standard connections from Tim.

Notice that the indexes of the spies in this subtask are in sequential order. This means that spies with larger indexes are further away from Tim. Additionally, you can only reach a spy through normal connections if there aren't any compromised spies with an index smaller than it. Therefore, for each day you keep track of the spy with the smallest index who is compromised. Let's say that index is i. If we use our temporary connection between Tim and spy $i-1$, then every spy between 0 (Tim) and $i-1$ will be able to receive the message. That gives a total of $(i-1)-0+1=i$ spies that can receive the message. Therefore, we will always output the index of the closest compromised spy.

Python Subtask 1 Solution

```
N, D = map(int, input().split())
for i in range(N-1):
    input() # we don't need to store the links in this subtask, since we know what they're going to be already

print(N)

furthest_spy = N
for i in range(D):
```

```
9       furthest_spy = min(furthest_spy, int(input()))
10      print(furthest_spy)
```

Subtask 2

For future terminology - we will root the tree at node 0 - or Tim's node. This means that we define a node's depth as the distance from that node to the root - with node 0 having a depth of zero. A leaf node is one that doesn't have any neighbours who are deeper than itself.

Let's say two nodes are in the same "branch" if the first node in the path from 0 to them are the same. For the temporary connection, you have to connect two spies in two different branches - if you connect spies in the same branch, you would have no way of connecting back to spy 0 without reusing spies.

Notice it is never optimal to use a temporary connection on a non-leaf node - you can always visit more spies by using the temporary connection on a descendant leaf node instead. Therefore, the biggest collection of spies before the first day would be achieved by connecting the two deepest spies that are in differing branches. This can be done in $O(N)$ time - traverse the tree, and for each branch, find its deepest node. Then, get the two deepest branches, then add their depths and add 1 (don't forget to include Tim) to get the answer. The only edge case is if there is only one branch - in which case you have to use a temporary connection to get back to Tim, which means the answer is just the deepest node plus 1 - which is also $O(N)$ time. We only need to do this once, because there are zero extra days - so we can complete this in linear time.

Python Subtask 2 Solution

```
1   import sys
2   sys.setrecursionlimit(10**6)
3
4   N, D = map(int, input().split())
5
6   adj = [[] for i in range(N)]
7
8   for i in range(N-1):
9       a, b = map(int, input().split())
10      adj[a].append(b)
11      adj[b].append(a)
12
13  def two_deepest_branches(cur, parent, adj_list):
```

```
    branches = [0]
    for neighbour in adj[cur]:
        if neighbour != parent:
            branches.append(
                max(two_deepest_branches(neighbour, cur, adj_list)) + 1
            )
    branches.sort()
    return branches[-2:]

print(sum(two_deepest_branches(0, -1, adj)) + 1)
```

Subtask 3

For this subtask, we can reuse the solution for subtask 2 - but just repeat it after each day a spy gets compromised. Remember how the network of spies forms a tree - this means there is only one path between any two nodes (property of a tree - this is true for any tree). Notice how if a spy gets compromised - it's children might as well also be compromised - as the only way to reach them from node zero would be with the temporary connection - but then it would be impossible to get the message back. This means we can just traverse the tree and repeat the solution for subtask 2, but just remember to not visit nodes that have been disabled. The algorithm from subtask 2 runs in $O(N)$ times, and you will have to repeat it D times for D days - which means this runs in $O(ND)$, which is sufficient for this subtask.

Python Subtask 3 Solution

```
import sys
sys.setrecursionlimit(10**6)

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

adj = [[] for i in range(N)]
deleted = set()

for i in range(N-1):
    a, b = map(int, input().split())
    adj[a].append(b)
    adj[b].append(a)

def two_deepest_branches(cur, parent, adj_list, deleted):
```

```
    branches = [0]
    for neighbour in adj[cur]:
        if neighbour != parent and neighbour not in deleted:
            branches.append(
                max(two_deepest_branches(neighbour, cur, adj_list,
                ↪  deleted)) + 1
            )
    branches.sort()
    return branches[-2:]

print(sum(two_deepest_branches(0, -1, adj, deleted)) + 1)

for i in range(D):
    k = int(input())
    deleted.add(k)
    print(sum(two_deepest_branches(0, -1, adj, deleted)) + 1)
```

Subtask 4

For this subtask $O(ND)$ will time out, as N and D go up to 50000. Remember, if there is only one branch in the tree - the best solution is just the deepest node. For this subtask, Tim only has one neighbour - which means there is only one branch. This means for each day, we just need to find the deepest node that isn't deactivated, and has no deactivated ancestors - or in other words, has a path from it to Tim without encountering compromised spies.

Doing this naively will result in $O(N)$ every day, and $O(ND)$ overall, which is too slow. However, we can use a priority queue, set or similar data structure to do this in $\log(N)$ time. This data structure will store all the nodes and their depths, and find the current deepest one in $\log(N)$ time. However, we also need to be able to remove nodes as well. If we use a set - we can remove a node in $\log(N)$ time. There are N nodes, so it takes worst case overall $N \log(n)$ to remove nodes, and $D \log(N)$ to find the deepest node - which is fast enough.

Python Subtask 4 Solution

```
import heapq
import sys
sys.setrecursionlimit(10**6)

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

```

```
adj = [[] for i in range(N)]
rooted_adj = [[] for i in range(N)]
depths_nodes = []
deleted = set()

for i in range(N-1):
    a, b = map(int, input().split())
    adj[a].append(b)
    adj[b].append(a)

def root_and_find_depths(cur, parent, adj_list, rooted_adj, depth):
    heapq.heappush(depths_nodes, (-depth, cur))

    for neighbour in adj[cur]:
        if neighbour != parent:
            rooted_adj[cur].append(neighbour)
            root_and_find_depths(neighbour, cur, adj_list, rooted_adj,
            ↪  depth + 1)

def delete_node(cur, rooted_adj, deleted):
    deleted.add(cur)
    for neighbour in rooted_adj[cur]:
        if neighbour not in deleted:
            delete_node(neighbour, rooted_adj, deleted)

root_and_find_depths(0, -1, adj, rooted_adj, 0)

print(-depths_nodes[0][0] + 1)

for i in range(D):
    k = int(input())
    delete_node(k, rooted_adj, deleted)
    while (depths_nodes[0][1] in deleted):
        heapq.heappop(depths_nodes)
    print(-depths_nodes[0][0] + 1)
```

Subtask 5

Let B be the number of branches. We can perform the solution in subtask 4 for all of the B branches. Notice after each day we only need to update one branch - this means updates can be done in overall $N \log(N)$. However, naively finding the largest branch takes up to $O(B)$. This has to be done every day - which is

too slow. However, we can use a set to store the max depth of each branch - and get the 2 largest branches, which takes $\log(N)$ to update and query. When nodes get deleted, we recalculate the max depth of the affected branch by using it's set/priority queue structure (discussed in Subtask 4). This gives a worst case $D\log(N)$ performance - which is sufficient for the full solution.

C++ Subtask 5 Solution

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

using namespace std;

vector<vector<int>> adj_list = vector<vector<int>>(50001);
unordered_set<int> deleted;
vector<int> branch_of_node = vector<int>(50001);
vector<int> parent_of_node = vector<int>(50001);
vector<int> depth_of_node = vector<int>(50001);

// Compare nodes by their depths.
// Extra condition when depths equal is to guarantee uniqueness within sets
struct cmp_nodes {
    bool operator() (int a, int b) const {
        if (depth_of_node[a] == depth_of_node[b]) {
            return a > b;
        } else {
            return depth_of_node[a] > depth_of_node[b];
        }
    }
};

vector<set<int, cmp_nodes>> nodes_of_branch = vector<set<int, cmp_nodes>>(50001);
set<int, cmp_nodes> deepest_in_branch;

void find_depths_of_branch(int branch, int cur, int parent, int depth) {
    branch_of_node[cur] = branch;
    parent_of_node[cur] = parent;
    depth_of_node[cur] = depth;

    nodes_of_branch[branch].insert(cur);

    for (auto neighbour : adj_list[cur]) {
```

```
34          if (neighbour != parent) {
35              find_depths_of_branch(branch, neighbour, cur, depth + 1);
36          }
37      }
38  }
39
40  void delete_node(int cur) {
41      nodes_of_branch[branch_of_node[cur]].erase(cur);
42      deepest_in_branch.erase(cur);
43      deleted.insert(cur);
44
45      for (auto neighbour : adj_list[cur]) {
46          if (neighbour != parent_of_node[cur] && deleted.find(neighbour)
            ↪  == deleted.end()) {
47              delete_node(neighbour);
48          }
49      }
50  }
51
52  // Get sums of depths of two deepest branches, or depth of deepest
    ↪  branch if only one exists
53  int two_deepest() {
54      int answer = 0;
55      auto deepest = deepest_in_branch.begin();
56      answer += depth_of_node[*deepest];
57      deepest++;
58      if (deepest != deepest_in_branch.end()) {
59          answer += depth_of_node[*deepest];
60      }
61      return answer;
62  }
63
64  int main() {
65      int n, m, a, b, k;
66      cin>>n>>m;
67
68      for (int i=0; i<n-1; i++){
69          cin>>a>>b;
70          adj_list[a].push_back(b);
71          adj_list[b].push_back(a);
72      }
73
74      for (auto branch : adj_list[0]) {
```

```
75          find_depths_of_branch(branch, branch, 0, 1);
76          deepest_in_branch.insert(*nodes_of_branch[branch].begin());
77      }
78
79      cout<<two_deepest() + 1<<endl;
80
81      for (int i=0; i<m; i++) {
82          cin>>k;
83          delete_node(k);
84
85          if (nodes_of_branch[branch_of_node[k]].begin() !=
            ↪  nodes_of_branch[branch_of_node[k]].end()) {
86
                ↪  deepest_in_branch.insert(*nodes_of_branch[branch_of_node[k]].begin());
87          }
88
89          cout<<two_deepest() + 1<<endl;
90      }
91  }
```

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.