New Zealand Informatics Competition 2020
Round 2 Solutions

May 1, 2020

Overview

Questions

1. **Mother Hubbard**
2. **Bear Hunt**
3. **Imbalance**
4. **Minceraft**
5. **Packing Boxes**

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.

© NZOI 2020

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.

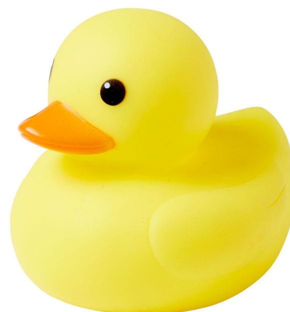
Test your code first!

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



© NZOI 2020

Mother Hubbard

https://train.nzoi.org.nz/problems/1164

Since children dropped off are already represented as negative numbers, we can simply sum up all of the input numbers to find the difference between the number of children picked up and dropped off. If this sum is zero, we have ended the day with all 10 children. Otherwise, the sum must be a negative number, the negation of which is the number of children left behind.

Python Solution

```
N = int(input())
total = 0
for _ in range(N):
    total += int(input())

if total == 0:
    print("She's got them all")
else:
    print(-total)
```

One Line Python Solution

```
print((lambda total: -total if total else "She's got them
↪  all")(sum(int(input()) for _ in range(int(input())))))
```

C++ Solution

```
#include <iostream>
using namespace std;

int main ()
```

```
{
    int N, total = 0; cin >> N;
    while (N--) {
        int d; cin >> d;
        total += d;
    }
    if (total) cout << -total << '\n';
    else cout << "She's got them all\n";
}
```

© NZOI 2020

Bear Hunt

https://train.nzoi.org.nz/problems/1165

This problem is similar to the previous question, but the output format is more involved.

Python Solution

```
N = int(input())
total = sum(int(input()) for _ in range(N))
h, m = divmod(total, 60)
s = "It took"
if h:
    s += f" {h} hours" if h > 1 else " 1 hour"
if m:
    if h:
        s += " and"
    s += f" {m} minutes" if m > 1 else " 1 minute"
print(s)
```

Note that `h, m = divmod(total, 60)` is shorthand for

```
h = total // 60
m = total % 60
```

Imbalance

https://train.nzoi.org.nz/problems/1166

Subtask 1

Clearly, it is always optimal to move a keyboard from the largest box to the smallest box at any point in time. In other words, for each minute, we want to remove a keyboard from the largest box, and then add one to the smallest box.

```
N, T = map(int, input().split())
boxes = list(map(int, input().split()))
for x in range(T):
    boxes.append(max(boxes) - 1)  # remove a keyboard from largest box
    boxes.remove(max(boxes))
    boxes.append(min(boxes) + 1)  # add a keyboard to the smallest box
    boxes.remove(min(boxes))
print(max(boxes) - min(boxes))
```

However, there is an edge case that caught many students out – if all boxes contain the same number of keyboards, the imbalance is zero and cannot be reduced any further, but some solutions might continue trying to move keyboards between boxes sub-optimally. For example, take a look at the following code:

```
N, T = map(int, input().split())
boxes = list(map(int, input().split()))
for x in range(T):
    boxes.sort()
    boxes[-1] -= 1  # remove a keyboard from the largest box
    boxes[0] += 1  # add a keyboard to the smallest box
print(max(boxes) - min(boxes))
```

At a glance, this solution may appear to be equivalent to the first one. However, it can fail if, at the very last minute, all of the boxes contain the same number of

keyboards, since it will always pick two different boxes (when $N > 1$), causing the imbalance to increase. To fix this, we could sort the list again after adding each keyboard, or check that all boxes do not contain the same number of keyboards before attempting to move a keyboard.

Finding the minimum or maximum box takes $O(N)$ time, or $O(N \log N)$ time if we sort the list. Since we repeat this T times, the overall time complexity is either $O(TN)$ or $O(TN \log N)$.

Subtask 2

When $T \leq 100\,000$, it is too slow to perform an $O(N)$ search every time we want to find the minimum or maximum box. A useful observation here is that the order in which we add or remove keyboards from boxes does not matter – it can be easier to first add T keyboards to the smallest boxes, and then remove T keyboards from the largest boxes.

If we sort the boxes, we can then use a nested loop to add T keyboards to the smallest boxes. If we are careful to ensure that the list of boxes always remains sorted, it becomes much easier to find the smallest box at any point in time. We then use a similar process to remove T keyboards from the largest boxes. The tine complexity of this solution is $O(N + T)$

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

boxes.sort()
time = T
while time > 0:
    for i in range(N):
        # to keep the list sorted, we restart if we reach a
        # box that is not smaller than the previous box
        if time == 0 or (i > 0 and boxes[i-1] <= boxes[i]):
            break
        boxes[i] += 1
        time -= 1

boxes.sort(reverse=True)
time = T
while time > 0:
    for i in range(N):
        if time == 0 or (i > 0 and boxes[i-1] >= boxes[i]):
            break
```

© NZOI 2020

```
            boxes[i] -= 1
            time -= 1

print(max(boxes) - min(boxes))
```

There are also other solutions that use various data structures. In Python, we can use priority queues (heaps), which allow us to both add elements and find (and remove) the smallest element in $O(\log N)$ time. When we want to find the largest element, we can simply invert all the values to become negative. In C++, we can use a multiset instead. The time complexity of these solutions are roughly $O(T \log N)$.

Python

```
import heapq

N, T = map(int, input().split())
boxes = list(map(int, input().split()))

heapq.heapify(boxes)
for x in range(T):
    min_box = heapq.heappop(boxes)
    heapq.heappush(boxes, min_box + 1)

boxes = [-x for x in boxes]  # make all elements negative
heapq.heapify(boxes)  # now we can get the max box easily

for x in range(T):
    max_box = heapq.heappop(boxes)
    heapq.heappush(boxes, max_box + 1)

boxes = [-x for x in boxes]  # make positive again
print(max(boxes) - min(boxes))
```

C++

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

int n,t,x;

int main() {
    cin >> n >> t;
```

```
8       multiset<int> s;
9       for (int i = 0; i < n; i++) {
10          cin >> x;
11          s.insert(x);
12      }
13      while (t--) {
14          int high = *--s.end();
15          s.erase(--s.end());
16          s.insert(high-1);
17          int low = *s.begin();
18          s.erase(s.begin());
19          s.insert(low+1);
20      }
21      cout << *s.rbegin() - *s.begin() << '\n';
22  }
```

Subtask 3

When T can be at most 1 billion, it is too slow to individually move keyboards between boxes T times, even if each move can be done in constant time. While there are a few different approaches we could take to solve the full problem, the simplest method uses an observation that the number of keyboards inside each box is limited to 100 000. If we store counts of how many of each box size we have, then adding T keyboards to the smallest boxes can be done in a single loop which iterates over these counts. The time complexity of this solution is $O(N + \max(x_i))$

```
1   N, T = map(int, input().split())
2   boxes = list(map(int, input().split()))
3   counts = [0]*100010
4   for x in boxes:
5       counts[x] += 1
6
7   time = T
8   for idx in range(1, 100000):
9       delta = min(counts[idx], time)
10      counts[idx+1] += delta
11      counts[idx] -= delta
12      time -= delta
13
14  time = T - time  # we might not have used all of our time
15  for idx in range(100000, 1, -1):
```

© NZOI 2020

```
16       delta = min(counts[idx], time)
17       counts[idx-1] += delta
18       counts[idx] -= delta
19       time -= delta
20
21   # find the largest and smallest box sizes where count > 0
22   largest = max(idx for idx, count in enumerate(counts) if count > 0)
23   smallest = min(idx for idx, count in enumerate(counts) if count > 0)
24   print(largest - smallest)
```

© NZOI 2020

Minceraft

https://train.nzoi.org.nz/problems/1167

Subtask 1

In this subtask we only have one type of mince which means every item requires some amount of this initial type. As each raft only depends on either previous rafts or the initial mince type we can compute how much mince each raft requires.

```
vector<int> costs(r_rafts, 0);
for (int r = 0; r < r_rafts; ++r) {
    int c_count; cin >> c_count;
    for (int i = 0; i < c_count; ++i) {
        int type, amount;
        cin >> type >> amount;
        --type;

        if (type >= m_minces)
            amount *= costs[type - m_minces];

        costs[r] += amount;
        if (costs[r] >= MAX)
            costs[r] = MAX;
    }
}
```

Note that the total mince required for a raft may exceed the size of a 32 bit integer which means the costs may overflow. Because the maximum amount of mince is 10^9 we can check to see if we exceed this value and if so, mark it as too big. However, we did not include test cases that included an overflow in the computation of the final answer so this was not penalised.

Finally, we just need to divide the amount of mince we have by the cost of the last

raft.

```
int amount = mince_amount / costs.back();
cout << amount << '\n';
```

Subtask 2

Here each raft can require multiple types of initial minces. We can use the same approach as Subtask 1 except we store the amounts required for every initial mince type instead of just one mince type.

```
vector<vector<int>> costs(r_rafts, vector<int>(m_minces, 0));
```

We also need to change the body of the nested loop to add on all the required mince amounts from the rafts that are used.

```
if (type < m_minces) {
    costs[r][type] += amount;
    if (costs[r][type] >= MAX)
        costs[r][type] = MAX;
} else {
    type -= m_minces;
    for (int k = 0; k < m_minces; ++k) {
        i64 addend = (i64) costs[type][k] * amount;
        if (costs[r][k] + addend < MAX)
            costs[r][k] += addend;
        else costs[r][k] = MAX;
    }
}
```

To calculate the final amount we need to find the minimum quotient of dividing an initial mince amount by the raft cost for that type. In other words, we find the mince type that bottlenecks the final amount the most.

```
int total = INT_MAX;
for (int i = 0; i < n_minces; ++i) {
    if (costs.back()[i] == 0) continue;
    total = min(total, initial[i] / costs.back()[i]);
}
cout << total << '\n';
```

Note that it is possible the last raft does not use one of the initial mince types and so that amount will be zero. We need to skip the type if it is zero otherwise it will result in an undefined division.

© NZOI 2020

Time Complexity

For the first subtask we iterate once for every mince type in each raft. This means our algorithm runs in $O(R \times C_{max})$ time where C_{max} denotes the maximum amount of mince types per raft (in this problem, five).

In the second subtask we may need to add on all the mince amounts from a previous raft for a mince type. This brings our complexity to $O(M \times R \times C_{max})$ as we need to iterate for each mince type in the raft.

Memory Complexity

For the first subtask we need to store an array of integers where each integer represents the amount of mince required for a raft. Hence, the memory complexity is $O(R)$. The second subtask is similar except we store all the mince types and amounts for each raft and so the complexity becomes $O(M \times R)$. In a highly optimised implementation this is roughly half a megabyte.

© NZOI 2020

Packing Boxes

https://train.nzoi.org.nz/problems/1132

Subtask 1

For the first subtask, we can simulate putting boxes into the car, keeping track of how much space we have used in each row using a list of integers.

For each of the N boxes, we must search through up to N rows to find which row the next box is pushed to. Thus, the time complexity of this solution is $O(N^2)$

Python Solution

```
N, K = map(int, input().split())
width = [0]  # total width of boxes in each row

for i in range(N):
    w = int(input())  # get the next box width

    # add a new row if necessary
    if w + width[-1] > K:
        width.append(0)

    # push the box back until the gap is too small
    row = len(width) - 1
    while row > 0 and w <= K - width[row-1]:
        row -= 1
    width[row] += w

print(len(width))
```

Subtask 2

For larger values of N, our subtask 1 solution is too slow. Every time a new box is placed in the car, we have to check a large number of rows to see how far back we can push the box. However, we can speed up the gap finding process by keeping track of the furthest row we can push boxes to, for each possible box width. Since $K \leq 10$, there are only up to 10 possible widths to keep track of.

Now instead of searching through up to N rows for each box, we can lookup the next row for any box width in constant time. Then, for each of the N boxes, we must update at most K values of this lookup table, so the time complexity of this solution is $O(NK)$.

Python Solution

```python
N, K = map(int, input().split())

next_row = [0] * (K+1)  # lookup table for K possible widths
width = [0] * N  # we require at most N rows
sol = 0

for i in range(N):
    w = int(input())

    row = next_row[w]  # get the next row from the lookup table
    width[row] += w  # add the box to this row
    sol = max(sol, row+1)  # keep track of the answer (total rows used)

    # If this row has `gap` space left, then all boxes of width `gap+1`
    # or greater cannot be pushed past the current row. Thus, we update
    # the next row for these widths to be at least `row+1`.
    gap = K - width[row]
    for i in range(gap+1, K+1):
        next_row[i] = max(next_row[i], row+1)

print(sol)
```

Subtask 3

For larger values of K, updating our lookup table starts taking too much time. Notice that our lookup table always contains several *ranges* of widths, where the

© NZOI 2020

next row for each width in the range is the same. Instead of keeping track of the next row for each width individually, we can store these ranges as an ordered map (keyed by the start of each range), allowing it to be updated much faster. In C++, `std::map` can be used. In Java, the equivalent is `TreeMap`. Unfortunately, Python does not offer a built-in structure with similar lookup characteristics.

We will insert at most N ranges into our map, each of which takes $O(\log N)$ time, so the time complexity of this solution is $O(N \log N)$. For the more experienced students, note that this problem can also be solved using a segment tree.

C++ Solution

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

int N,K,w,sol;
int width[250010];
map<int,int> next_row;

int get_next_row(int x) {
    // find the largest key not larger than x
    return (--next_row.upper_bound(x))->second;
}

int main() {
    cin >> N >> K;
    next_row[1] = 0;
    while(N--) {
        cin >> w;
        int row = get_next_row(w);
        width[row] += w;
        sol = max(sol, row+1);
        int gap = K - width[row];
        if(get_next_row(gap+1) < row+1) {
            next_row[gap+1] = row+1;  // add a new range into the map
            auto it = next_row.upper_bound(gap+1);
            while(it != next_row.end() && it->second <= row+1) {
                it = next_row.erase(it);  // delete outdated ranges
            }
        }
    }
    cout << sol << '\n';
}
```

© NZOI 2020

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.[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

[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.

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 $N \leq 10\,000$.

© NZOI 2020