NZIC Round 2 Solutions 2019

Questions

1. **More Tea Please:** Multiplication and output
2. **Quality Tea:** While loop
3. **Shadow's Battle:** Simulation
4. **Totem Pole:** Greedy / Simulation / Data Structures
5. **An Average Problem:** Ad-hoc

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 at the end of this document.

Resources

Ever wondered what the error messages mean?
http://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:
http://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Other resources: http://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:
           tea = int(input(prompt))
           if tea < 0 or tea > 100:
               print('That was not a valid integer!')
           else:
               return tea
       except ValueError:
           print('Not a valid integer')
...
```

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

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.

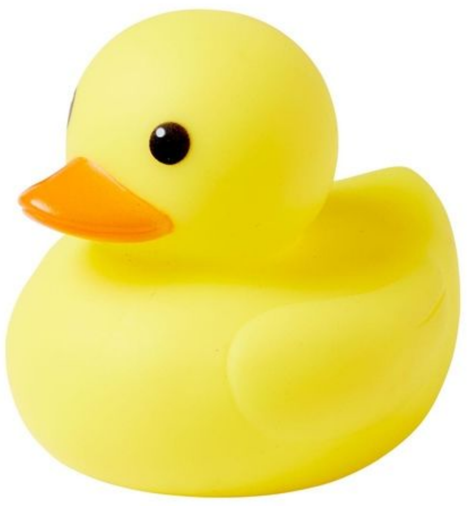
Test your code first!!!

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 < N <= 10" then try out an input where N = 10. 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



More Tea Please

https://train.nzoi.org.nz/problems/880

There are 24 boxes, each with 10 teabags, so 24 * 10 = 240 tea bags per box.

Python 3 Examples

```python
# example 1
print("Ginger", int(input()) * 240)
print("Chamomile", int(input()) * 240)
print("Earl Gray", int(input()) * 240)
print("Peppermint", int(input()) * 240)
print("Lemon", int(input()) * 240)
print("Strawberry", int(input()) * 240)
```

```python
# example 2
teas = ["Ginger", "Chamomile", "Earl Gray",
       "Peppermint", "Lemon", "Strawberry"]

for tea in teas:
   print(tea, int(input()) * 240)
```

```python
# example 3 - one line because why not
print("\n".join(map(" ".join, zip(["Ginger", "Chamomile", "Earl Gray",
"Peppermint", "Lemon", "Strawberry"], [str(int(input()) * 240) for _ in
range(6)]))))
```

C++ Examples

```cpp
// example 1
#include <bits/stdc++.h>
using namespace std;

int main() {
    int G, C, E, P, L, S;
    cin >> G >> C >> E >> P >> L >> S;
    cout << "Ginger " << G * 240 << endl;
    cout << "Chamomile " << C * 240 << endl;
    cout << "Earl Gray " << E * 240 << endl;
    cout << "Peppermint " << P * 240 << endl;
    cout << "Lemon " << L * 240 << endl;
    cout << "Strawberry " << S * 240 << endl;

    return 0;
}
```

```cpp
// example 2
#include <iostream>
using namespace std;

string tea[] = {"Ginger", "Chamomile", "Earl Gray",
                "Peppermint", "Lemon", "Strawberry"};

int main() {
    for (int i = 0; i < 6; i++) {
        int x;
        cin >> x;
        cout << tea[i] << " " << x * 240 << '\n';
    }
}
```

Quality Tea

https://train.nzoi.org.nz/problems/1023

For the first subtask, if all three quality levels are less than K then a fault is detected. Otherwise, no fault is detected. We can solve this by reading each line in and checking if all of them are less than K.

```
# 40% subtask
N, K = map(int, input().split())

# read in quality values
quals = []
for _ in range(3):
   quals.append(int(input()))

# check if all qualities less than K
if quals[0] < K and quals[1] < K and quals[2] < K:
   print("Fault Detected")
   print(*quals)
else:
   print("No Fault")
```

For the full solution, we need to handle more than three products. We must check that the previous three qualities in a row are less than K. Some competitors lost points because they forgot to print "No Fault" if the end of the input was reached and no three successive quality values were less than K.

Python Examples

```
# example 1
N, K = map(int, input().split())

quals = []
for _ in range(N):
   q = int(input())
   if q < K:
       quals.append(q)
   else:
       quals.clear()  # makes the list empty again

   if len(quals) == 3:
       print("Fault Detected")
       print(*quals)  # this * syntax is called argument unpacking
       break
else:
   print("No Fault")
```

Extra for interest: argument unpacking explained:
https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists

```python
# example 2
info = input()
N = int(info[0])
K = int(info[1])
faults = 0
products = []

for i in range(N):
   product = int(input())
   if product < K:
       faults += 1
       products.append(product)
   elif faults < 3:
       faults = 0
       products = []

if faults >= 3:
   print("Fault Detected")
   print("{} {} {}".format(products[0], products[1], products[2]))
else:
   print("No Fault")
```

C++ Example

```cpp
// example 1
#include <iostream>
using namespace std;

int main() {
   int n_products, threshold;
   cin >> n_products >> threshold;

   int count = 0;
   int stuff[3];
   for (int i = 0; i < n_products; ++i) {
       int t;
       cin >> t;

       if (t < threshold) {
           stuff[count++] = t;
       } else {
           count = 0;
       }

       if (count == 3) {
           cout << "Fault Detected" << endl;
           cout << stuff[0] << " " << stuff[1] << " " << stuff[2] << endl;
```

```
            return 0;
        }
    }

    cout << "No Fault" << endl;
}
```

Shadow's Battle

https://train.nzoi.org.nz/problems/885

Make sure you understand the problem description above before looking at the solution!

To solve this problem we can 'play out' or simulate the battle until it ends. To do this, we need to know the initial values of each variable, and our end criteria. The initial values of our variables - for instance, Shadow's health - are stated in the problem description. Shadow begins with 60 health, and the players with 10. Note that you will need to keep track of the players' health points separately in an array or list. The end criteria for the game is when either Shadow dies or all the other players die, because then we can declare our winner.

To implement this, you could use a while loop that checks after each turn if Shadow and the players are still alive. eg. **`while shadow_alive and players_alive`.** Then at the end of the while loop, we should set **`shadow_alive`** and **`players_alive`** to **`True`** or **`False`**, depending on the health of Shadow and the players.

Within our while loop, we take turns playing for Shadow and the group of players. You can also keep track of whose turn it is using a boolean variable. Shadow starts first, so initialize this to **`True`**. At the end of each turn, we can set **`shadows_turn = not shadows_turn`** to toggle whose turn it is.

During Shadow's turn:

Shadow makes an attack on the adjacent player(s) around her. To determine which players these are, we find the closest player on each side of Shadow. One way to do this is to create two new variables, **`left`** and **`right`**, which represent the index of the left player and right player. To start, we set these indices to the location of shadow. We move these indices outwards from shadow until we find a player who has a health greater than zero. These players are the two which receive damage. When one of these players dies, in the next turn we will continue moving the left or right indices out until we find a new player or reach the end of the list.

```
while left > 0 and H[left] <= 0:
    left -= 1
while right < 199 and H[right] <= 0:
    right += 1
```

A value of 199 is used since this is the maximum index right can be in a list 200 items long. It must be 200 long since

During the players turn:

Shadow’s health will decrease by the combined damage of all living players. One way to do this is to sum up the damage of all living players for each turn. A more computationally efficient way would be to sum up the total damage before starting the simulation then, when a player dies, subtract their damage contribution from the total. This saves having to iterate through the list each time.

After taking either Shadow’s or the players’ turn, we complete death checks as discussed above and print the required output depending on **`shadow_alive`** and **`players_alive.`**

Python Example

```python
N = int(input())
H = [0] * 200  # each player's health
D = [0] * 200  # how much damage each player can do
p_dam = 0  # total damage all players can do together

# read in players
for _ in range(N):
   pos, dam = map(int, input().split())
   D[pos - 1] = dam  # player damage
   H[pos - 1] = 10  # player health
   p_dam += dam  # collective player damage

# read in shadow
s_pos, s_dam = map(int, input().split())
s_health = 60
s_pos -= 1

# start two indices at shadow's position
left, right = s_pos, s_pos  # inclusive

shadow_alive = True
players_alive = True
shadows_turn = True  # shadow starts

while shadow_alive and players_alive:

   if shadows_turn:
       # find the index of next player to the left and right of shadow
       while left > 0 and H[left] <= 0:
           left -= 1
       while right < 199 and H[right] <= 0:
           right += 1

       # apply damage to these players
```

```
        H[left] -= s_dam
        H[right] -= s_dam

        # if they died, remove their contribution to total damage
        if H[left] <= 0:
            p_dam -= D[left]
        if H[right] <= 0:
            p_dam -= D[right]

        # if the players can do no more damage, they all must be dead
        if p_dam <= 0:
            players_alive = False

    # players turn
    else:
        s_health -= p_dam

        # check if shadow died
        if s_health <= 0:
            shadow_alive = False

    shadows_turn = not shadows_turn  # switch turns


# print results
if shadow_alive:
    print("Shadow wins!")
else:
    print("We win! Players alive: ", end='')
    for i in range(200):
        if H[i] > 0:
            print(i + 1, end=' ')
    print()
```

C++ Example

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

#define LEN 200

int main() {
    int N, p_dam = 0;
    cin >> N;  // number of players
```

```cpp
// each player's health and damage they can deal
vector<int> health(LEN), damage(LEN);

// read in players
for (int i = 0; i < N; i++) {
    int pos, dam;
    cin >> pos >> dam;
    damage[pos - 1] = dam;
    health[pos - 1] = 10;
    p_dam += dam;
}

// read in shadow
int s_pos, s_dam, s_health = 60;
cin >> s_pos >> s_dam;
s_pos -= 1;  // since our vector is 0 indexed

// start two iterators at shadow's position
vector<int>::iterator left, right;
left = right = health.begin() + s_pos;  // inclusive pointers

bool shadow_alive, players_alive, shadows_turn;
shadow_alive = players_alive = shadows_turn = true;

// simulate
while (shadow_alive && players_alive) {
    if (shadows_turn) {
        // find closest players to shadow
        while (left > health.begin() && *left <= 0)
            left--;
        while (right < health.end() && *right <= 0)
            right++;

        // apply damage to these players
        *left -= s_dam;
        *right -= s_dam;

        // if they died, remove their contribution
        if (*left <= 0)
            p_dam -= damage[left - health.begin()];
        if (*right <= 0)
            p_dam -= damage[right - health.begin()];

        // if players still deal collective damage then some
        // must be alive
        players_alive = p_dam > 0;
    }
```

```
        else {
            // players turn
            s_health -= p_dam;  // shadow takes collective damage
            shadow_alive = s_health > 0;
        }

        shadows_turn = !shadows_turn;
    }

    // output simulation results
    if (shadow_alive) {
        cout << "Shadow wins!" << endl;
    } else {
        cout << "We win! Players alive:";
        for (auto it = health.begin(); it != health.end(); it++) {
            if (*it > 0) {
                // + 1 to go from 0 to 1 based indexing
                cout << ' ' << it - health.begin() + 1;
            }
        }
        cout << endl;
    }
}
```

Totem Pole

https://train.nzoi.org.nz/problems/1022

Subtasks 1 & 2 (Python)

For subtasks 1 and 2 we can do a direct simulation. For each position record if it has a carving, and loop over the positions to find where a person will make their carving.

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

# records if a position has a carving
carved = [False] * (T + 1)
# The positions have 1-based indexes, so use a list of length T+1.
# [False] creates a list with one element, and multiplying a list
# by a number repeats the list that many times in Python.

# number of people who made carvings
count = 0
for h in map(int, input().split()):

   # limit h to top of pole to prevent index errors
   h = min(h, T)

   # find the next available position to make a carving
   while h >= 1 and carved[h]:
       h -= 1

   # increment the counter if the person makes a carving
   if h >= 1:
       carved[h] = True
       count += 1

print(count)
```

Python uses 0-based indexing, which means the valid indexes in a list of length N are 0 to $N-1$. The carving positions are 1 to T, which makes it slightly awkward to use them as indexes. This program uses a neat trick: it records the status of each carving position in a list called `carved` of length $T+1$. The extra slot means it can use the indexes 1 to T, effectively making the list 1-based (index 0 is never used, so this trick "wastes" a tiny amount of memory). Another option is to make the carving positions 0-based by subtracting 1 from all the heights.

This program takes at most $N \times T$ "steps" to run because it has an outer for-loop that iterates over the N people, and for each person the inner while-loop iterates over a list of length T. The while-loop won't always iterate over all T elements because it starts from h and stops early when it finds an available carving position, so in fact the number of step will be smaller.

The time limit is 1 second. A good rule of thumb for Python is 1 million operations per second, so when N and T are 1,000 the program runs within the time limit. But when N and T are 100,000 (for subtask 3) the program is too slow.

Subtasks 1 & 2 (C++)

This is a C++ version of the Python program.

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

int main() {
   int n, t;
   cin >> n >> t;
   // records if a position has a carving (1-based)
   vector<bool> carved(t + 1);

   // the number of people who made carvings
   int count = 0;
   for (int i = 0; i < n; i++) {
       int h;
       cin >> h;
       // limit h to top of pole to prevent index errors
       if (h > t) {
           h = t;
       }
       // find where the person will make their carving
       while (h >= 1 && carved[h]) {
           h--;
       }
       if (h >= 1) {
           // the person will make a carving at height h
           carved[h] = true;
           count++;
       } else {
           // the person won't make a carving
       }
   }
   cout << count << "\n";
}
```

Sidebar: out-of-range indexes in C++

The problem statement implicitly allows people to be taller than the totem pole (as the sample data demonstrates), but it's easy to miss this and leave out the if-statement that limits h to T.

Because of that the test data for subtask 1 intentionally does not have such test cases besides the sample.

Surprisingly, the C++ program passes the sample data even without the if-statement. In the sample T is 4 and there is a person of height 6, so without the if-statement the program would try to access index 6 in a vector of length $T+1$ = 5 (valid indexes: 0 to 4). In many languages including Python and Java that would display an error message and terminate the program, but in C++ accessing an array/vector with an index that is out of range causes "undefined behaviour". Usually the program will either crash with a segmentation fault or read/write an arbitrary piece of memory (which can cause completely unpredictable behaviour, for example it could change the value of an unrelated variable).

There are various ways to improve bounds checking in C++:

- For vectors, use **`at()`** instead of **`[]`**, for example **`carved.at(h)`** instead of **`carved[h]`**. (This is inconvenient to type, so it is not commonly used.)
- If compiling with GCC, add the option **`-D_GLIBCXX_DEBUG`**, for example
  ```
  g++ -D_GLIBCXX_DEBUG program.cpp
  ```
  to enable error checking in the standard library. This will make **`[]`** on vectors check bounds, among other things.
- If compiling with GCC, add the options **`-g -fsanitize=address`** to enable AddressSanitizer, a memory error detector. It will detect out of bound array indexes in many cases, among other things. (The option **`-g`** generates debugging information, which adds line numbers to error messages.)

Subtask 3

There are several ways to find the next available carving position more efficiently:

- Use a sorted set to keep track of the available carving positions. This allows the next available carving position below a height to be found and removed in $O(log(T))$. A C++ implementation is given in Appendix 1. (Python's standard library does not include a sorted set type, so this approach can't be used.)
- Use a range tree to track the highest available carving position in each range.
- Use a Fenwick tree to efficiently count the number of available carving positions up to each height, and use a binary search to find the highest available position.
- Use a disjoint-set data structure to track contiguous groups of carvings and record the height of the lowest carving in each group.

There is also a completely different and much simpler solution, which makes use of the following observation: The process where each person places a carving at the highest position they can reach results in the greatest possible number of carvings, regardless of the people's order (proof given in Appendix 2).

The optimal number of carvings can be found by sorting the people from shortest to tallest and having each person make a carving at the lowest available position. This is simple to implement:

Python

```python
n, t = map(int, input().split(" "))
hs = list(map(int, input().split(" ")))
hs.sort()
count = 0 # number of carvings (made at heights 1..count)
for h in hs:
   if h > count and count < t:
       count += 1
print(count)
```

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
   int n, t;
   cin >> n >> t;
   vector<int> heights;
   for (int i = 0; i < n; i++) {
       int h;
       cin >> h;
       heights.push_back(h);
   }
   sort(heights.begin(), heights.end());

   int count = 0; // number of carvings (made at heights 1..count)
   for (auto h : heights) {
       if (h > count && count < t) count++;
   }

   cout << count << endl;
}
```

Appendix 1: Sorted set solution (C++)

This program uses a sorted set to keep track of the available carving positions.

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

int main() {
   int n, t;
   cin >> n >> t;
   vector<int> all; // to build the numbers 1 to T
   all.reserve(t);
   for (int i = 1; i <= t; i++) { all.push_back(i); }
   set<int> available(all.begin(), all.end());
   int count = 0;
   for (int i = 0; i < n; i++) {
       int h;
       cin >> h;
       // We want to find the biggest available position <= h.
       // This operation is not supported directly, but we can use
       // upper_bound() to find the first element > h then go back one.
       set<int>::iterator pos = available.upper_bound(h);
       if (pos == available.begin()) {
           // no available position
       } else {
           pos--; // go back one element
           available.erase(pos);
           count++;
       }
   }
   cout << count << "\n";
}
```

Appendix 2: Proof of observation (Extra)

"The process where each person places a carving at the highest position they can reach results in the greatest possible number of carvings, regardless of the people's order."

The result of the process can be represented as a mapping from people to carving heights (or -1 if a person does not make a carving).

Let M be the mapping produced by the process on the line of people P1..Pn, and let X be an optimal mapping on those people (i.e. with the most carvings possible). This proof shows that X can be modified to be equal to M without decreasing the number of carvings in X, which means M has at least as many carvings as X and hence M is optimal.

Let Pi be the first person for which M(Pi) != X(Pi). We must have M(Pi) > X(Pi) because the mappings are the same up to Pi and the process chooses M(Pi) to be the highest available carving position. We show that X(Pi) can be changed to M(Pi).

If there is a later person Pj for which X(Pj) == M(Pi), swap X(Pi) and X(Pj). This is permissible because:

- Pi can carve at height M(Pi) and M(Pi) == X(Pj) so Pi can carve at height X(Pj)
- Pj can carve at height X(Pj) and X(Pj) == M(Pi) > X(Pi) so Pj can carve at height X(Pi)

If there is no person Pj for which X(Pj) == M(Pi), we can simply set X(Pi) to M(Pi).

It's not possible for an earlier person Pj to have X(Pj) == M(Pi), because X and M are equal up to Pi so X(Pj) == M(Pj) != M(Pi).

Hence we can change X(Pi) to M(Pi), and with repetition make X equal to M. Since the number of carvings in X does not decrease at any point, M must have at least as many carvings as X.

An Average Problem

https://train.nzoi.org.nz/problems/893

Subtask 1

If the mean, median, and mode are all the same, it is always possible to generate a valid sequence – we can just output that number N times.

```python
n,mean,median,mode = map(int,input().split())
if mean == median == mode:
    print("Possible")
    print("{} ".format(mean)*n)
```

Subtask 2

For $N = 1$, it is clearly only possible if the mean, median, and mode are equal, and in this case we can just use the same solution as subtask 1.

For $N = 3$, the mode has to occur at least twice (otherwise it cannot be the most frequent value), and the median has to occur at least once. But if the median is not the same as the mode, then our sequence would be [median, mode, mode], and the “middle” element would be the mode, not the required median. Therefore, the median must be equal to the mode – if not, it is impossible to generate a solution. So, our solution will always uses the median twice, which satisfies both the median and mode criteria. For the last element, we calculate the required value to satisfy the mean criterion: $N \times mean - 2 \times median$. If this value doesn’t fit within the valid range (1 to 1,000,000), it is impossible to generate a valid sequence.

```python
# continuing from the subtask 1 solution
elif n == 1:
    print("Impossible")
elif n == 3:
    x = n*mean - 2*median
    if median != mode or x < 1 or x > 1000000:
        print("Impossible")
    else:
        print("Possible")
        print(median, median, x)
```

Subtask 3

For this subtask, it is guaranteed that the median is always equal to the mode. Let’s forget about the mean criterion for now. Instead, let’s try to generate a sequence satisfying the median and mode criteria with the lowest mean possible. To satisfy the median criterion, we need the upper half of the sequence to be equal to or above the median, so it's easy to see that we should take the median $\lceil N/2 \rceil$ times.[1] This also satisfies our mode criterion, so we can then take 1 (the lowest possible value) $\lfloor N/2 \rfloor$ times to complete our "lowest mean" sequence. Let x be the sum of this sequence. We can similarly generate a sequence with the highest mean possible by taking the median $\lceil N/2 \rceil$ times, and 1,000,000 (the highest possible value) $\lfloor N/2 \rfloor$ times – let y be the sum of this sequence. Then it is possible to generate a valid sequence if and only if $x \leq mean \times n \leq y$.

```
if median == mode:
    x = median*(n//2+1) + 1*(n//2)
    y = median*(n//2+1) + 1000000*(n//2)
    if x <= mean*n <= y:
        print("Possible")
    else:
        print("Impossible")
```

You may have noticed that both the "lowest mean" and "highest mean" sequences use the median $\lceil N/2 \rceil$ times. In fact, it is possible to generate any sequence using the median exactly $\lceil N/2 \rceil$ times, satisfying the median and mode criteria with an integer sum between x and y – if we start at the "lowest mean" sequence, we can repeatedly increment each of the remaining $\lfloor N/2 \rfloor$ elements to increase the sum by 1 each time, until we reach the "highest mean" sequence. However, this can be too slow, so instead of incrementing by 1, we will shift elements all the way up to 1,000,000 in one go, stopping once we reach the required total to satisfy the mean criterion.

```
if median == mode:
    solution = [median] * (n//2+1)
    remaining = [1] * (n//2)
    total = sum(solution+remaining)
    if total > mean*n:
        print("Impossible")
        sys.exit()

    # shift remaining elements upwards until we reach the required total
    for x in range(len(remaining)):
        total -= remaining[x]
        remaining[x] = min(1000000, mean*n - total)
        total += remaining[x]
        if total == mean*n:
            break

    if total != mean*n:
        print("Impossible")
    else:
        print("Possible")
```

[1] Note that ⌈x⌉ denotes the ceiling of x (rounded up to the nearest integer), and ⌊x⌋ denotes the floor of x (rounded down to the nearest integer).

```
        print(' '.join(map(str,solution+remaining)))
```

Another working strategy is to take the median $\lceil N/2 \rceil$ times, then calculate the required sum of the remaining elements: $N \times mean - \lceil N/2 \rceil \times median$ – let this value be r. We can divide r by $\lfloor N/2 \rfloor$ to get the average value of the remaining elements. If this is an integer, we can just take it $\lfloor N/2 \rfloor$ times to complete our sequence. Otherwise, we use the floor instead, and distribute the remainder of $r / \lfloor N/2 \rfloor$ across the remaining $\lfloor N/2 \rfloor$ elements.

```
if median == mode:
    solution = [median] * (n//2+1)
    r = n*mean - median*(n//2+1)
    quotient = r // (n//2)  # note that this fails when n == 1
    remainder = r % (n//2)
    solution += [quotient] * (n // 2 - remainder)
    solution += [quotient+1] * remainder
    if min(solution) < 1 or max(solution) > 1000000:
        print("Impossible")
    else:
        print("Possible")
        print(' '.join(map(str,solution)))
```

Subtask 4

Our subtask 3 solution relies on the observation that we can always take the mode (or median) $\lceil N/2 \rceil$ times, but this no longer holds if the median is not equal to the mode. Nevertheless, suppose we knew how many times the mode occurs in our solution – let this value be k. To make things simpler, we'll also assume that the median is not equal to the mode. Now, let's once again generate a sequence that satisfies the median and mode criteria, with the lowest mean possible.

We'll start off by taking the mode k times, and the median once. To satisfy the median criterion, we need $\lfloor N/2 \rfloor$ elements to be equal to or below the median, and $\lfloor N/2 \rfloor$ elements to be equal to or above the median. Clearly, we want to take the value 1 as many times as possible – but not any more than $k-1$ times (unless 1 is the mode). Then we do the same for values 2, 3, 4, etc. until we've taken $\lfloor N/2 \rfloor$ elements below the median. Similarly, we take values from the median upwards until we've taken $\lfloor N/2 \rfloor$ values above or equal to the median.

We can use a similar process to generate a sequence with the highest mean possible. Again, let x and y denote the sums of the lowest and highest mean sequences. Then, if $x \le mean \times n \le y$, a valid sequence exists that uses the mode k times. Finally, we will brute force across all possible values of k to determine whether a valid sequence exists. As there are $O(N)$ possible values of k, and each sequence takes $O(N)$ time to generate, the overall time complexity of this solution is $O(N^2)$.

```python
from collections import Counter

def generate(mode_freq, p1, p2, direction):
    """Generates either the minimum (or maximum) mean sequence for
    the given mode frequency (minimum if direction == 1, maximum
    if direction == -1). p1 and p2 are the lowest (or highest) elements
    we can take on each side of the median."""

    sequence = [mode] * mode_freq + [median]
    counts = Counter(sequence)

    # take n/2 elements left of the median
    for x in range(n//2 - (mode_freq if mode < median else 0)):
        while counts[p1] >= mode_freq-1: p1 += direction
        counts[p1] += 1
        sequence.append(p1)

    # take n/2 elements right of the median
    for x in range(n//2 - (mode_freq if mode > median else 0)):
        while counts[p2] >= mode_freq-1: p2 += direction
        counts[p2] += 1
        sequence.append(p2)

    assert(min(p1,p2) >= 1 and max(p1,p2) <= 1000000)
    return sequence


n,mean,median,mode = map(int,input().split())

if median == mode:
    # refer to subtask 3 solution
else:
    # try each possible mode frequency
    for mode_freq in range(2, n//2+1):
        try:
            x = sum(generate(mode_freq, 1, median, 1))
            y = sum(generate(mode_freq, median, 1000000, -1))
        except AssertionError:
            continue
        if x <= n*mean <= y:
            print("Possible")
            break
    else:
        print("Impossible")
```

To actually generate the sequence, we can use a similar idea to our subtask 3 solution. Let's start from the lower bound sequence from above. Elements in the lower half of the sequence can be shifted up to the median, and elements in the upper half can be shifted up to 1,000,000. We will start by shifting elements in the upper half to allow elements in the lower half to move to the median later. We also need to ensure that we never shift any of the mode elements.

```
def solve(mode_freq):
    sequence = generate(mode_freq, 1, median, 1)
    sequence.sort(reverse=True)
    counts = Counter(sequence)
    total = sum(sequence)
    p = 1000000  # shift upper half elements up to 1000000

    for index, x in enumerate(sequence):
        if index == n//2: p = median  # shift lower half elements up to the median
        if x == mode: continue  # do not move any mode elements

        # shift one element from x up to p
        counts[x] -= 1
        total -= x
        while counts[p] >= mode_freq-1: p -= 1
        sequence[index] = min(p, mean*n-total)
        counts[sequence[index]] += 1
        total += sequence[index]

    print("Possible")
    print(' '.join(map(str,sequence)))
```

Full Solution

Instead of actually generating entire lowest/highest mean sequences for each possible mode frequency k, notice that we only need the sums of each sequence to determine if k is valid. Furthermore, almost all of the elements form two contiguous ranges on each side of the median, with each value occurring exactly $k-1$ times in the sequence. Also recall that the sum of numbers from 1 to N can be calculated with $N(N+1)/2$. Using this formula, we can calculate the sum of elements in each contiguous range in $O(1)$ time!

Once we've found a valid k, we can then use our solution from subtask 4 to actually generate a sequence. This reduces the overall time complexity of our solution to $O(N)$.

```
def series_sum(a, b):
    """Returns the sum of numbers from a to b (inclusive)."""
    if a > b: return series_sum(b, a)
    return b*(b+1)//2 - a*(a-1)//2


def calc(area, height, mx, start_pos):
    """Take 'area' elements starting from 'start_pos',
    where we can take each value up to 'height' times
    (maximum if mx == 1, minimum if mx == -1)."""

    subtotal = 0
    width = area // height
    remainder = area % height
    end_pos = start_pos - width*mx + mx

    # mode intersects with the current range, we need to avoid double counting modes
    if start_pos*mx >= mode*mx and mode*mx >= end_pos*mx:
        end_pos -= mx
        subtotal -= mode * height

    if width: subtotal += series_sum(start_pos, end_pos) * height
    if remainder and end_pos-mx == mode: end_pos -= mx
    if remainder and end_pos-mx != mode: subtotal += remainder * (end_pos-mx)

    return subtotal, end_pos, remainder


def find_min_max(mode_freq, mx):
    """Finds and returns the minimum or maximum sum possible.
    (maximum if mx == 1, minimum if mx == -1)"""

    total = mode * mode_freq
    height = mode_freq-1  # max number of each value we can take

    area = n//2  # required number of values
    if mode*mx > median*mx: area -= mode_freq  # to avoid double counting modes
    subtotal, end_pos, remainder = calc(area, height, mx,
                                    1000000 if mx == 1 else 1)
    total += subtotal

    area = n//2+1  # one extra to include the median
    if mode*mx < median*mx: area -= mode_freq  # to avoid double counting modes
    if remainder and end_pos-mx == median:
        # left/right sides overlap at the median, move the remainder to the right
        # side instead to simplify calculations
        cur_total -= remainder * median
        area += remainder
    total += calc(area, height, mx, median)[0]

    return total


n,mean,median,mode = map(int,input().split())

if median == mode:
    # refer to subtask 3 solution
else:
    # try each possible mode frequency
    for mode_freq in range(2, n//2+1):
        # check that we have enough space on each side of the median
        height = mode_freq-1
        right_spaces = (1000000-median) * height + height-1
        left_spaces = (median-1) * height + height-1
        if mode < median: left_spaces += 1
        if mode > median: right_spaces += 1
        if min(left_spaces, right_spaces) < n//2: continue

        x = find_min_max(mode_freq, -1)
        y = find_min_max(mode_freq, 1)
        if x <= mean*n <= y:
            solve(mode_freq)  # refer to subtask 4 solution
            break
    else:
        print("Impossible")
```

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 do about 10 million to 100 million useful operations per second. For example, if the problem specifies a run time of 1 second and an input N as big as 100 thousand, then you automatically know an **O(N^2)** algorithm will not work since 100 thousand squared is 10 billion operations.

Time complexity

The time taken by a program is usually estimated on the number of processor operations. For example, an addition $a+b$ or a compare $a<b$ 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[1] of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when N is 8 but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very very large before a computer starts to struggle.

O(N) time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm **max()** will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually if you have one loop that iterates N times your algorithm is **O(N)**.

O(N^2) time means the number of operations is proportional to N^2. For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N-1$ items. In total, $N(N-1)$ checks are done. This expands to N^2-N. For Big O, we always take the most significant term as the dominating factor, which gives **O(N^2)**. This is generally

[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.

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 $N \leq 10K$.