NZIC Round 1 2019

Questions

1. **House of Cards:** Maths or a simple loop
2. **Adding Teas:** A list and a while loop
3. **Tea Party:** Nested loops and storing information
4. **Modern Art:** 2d arrays and coordinate compression
5. **Twilight Sparkle's Magical Research:** Fast exponentiation

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.

There are a few common errors that students make in all contests. These can be easily avoided if students can

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.

There is no point in "optimising"

It's true that Python is often a bit slower than a compiled language such as C++. However, we try our best to design the problems so the full solution is achievable in both C++ and Python. In this round, all of the model solutions were written in Python. Trying to find small efficiency improvements by reducing the number of calculations you do per cycle isn't going to get you more marks. What will help you is thinking about a better way to solve the problem in general, i.e. a better algorithm. By "better" we usually mean so that it uses less memory or takes less time to run. For example, choosing a good algorithm was the difference between the program taking more than 3 hours vs taking a few milliseconds in the Modern Art problem. This is how we differentiate the top students. If you'd like to be one of those top students then have a play with some of the different approaches to the problems discussed below.

Test code

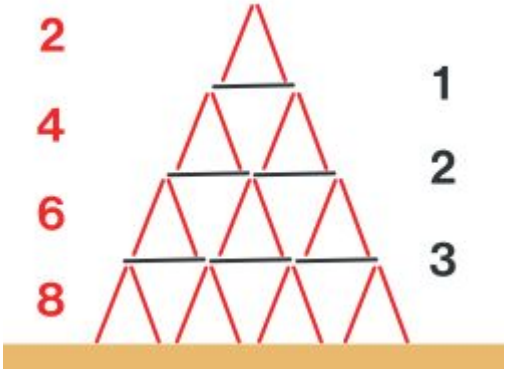
Many students submitted broken or syntactically incorrect code as initial solutions. This wastes time while you wait for the server to run your code. It is far more efficient to test it yourself first.

Test using edge cases

However, the example cases are by no means a complete test. You need to think of some other simple tests to make sure your program behaves. Part of being is good programmer is learning to think of all the edge cases and making sure your code handles them as you would expect. These might include the lowest and highest number you could receive as input.

House of Cards

https://train.nzoi.org.nz/problems/869



If a tower is N high, we need to calculate the number of cards needed. Let's break it down...

1. If we go through each level starting at 1 and going up to N, the number of angled cards (red ones) at each level is always 2 times which ever level we are at. e.g. level 2 has 4 angled cards.
2. The number of cards on the bottom is always equal to the level we are on, except for the last level. Therefore, one possible solution is as follows

```
n = int(input())

result = 0
for story in range(1, n + 1):
  red = 2 * story
  grey = story
  result += red + grey

# Bottom story doesn't have a platform
result -= n

print(result)
```

This solution will get 100%, but out of interest let us see if we can improve it...

From step 1. above, we know that the number of red cards is twice the floor number. Let's call the floor number i and the number of red cards on that floor r. Then we can say $r = 2i$. Additionally, we can look at 2. above. It would be nice to have an equation just like we do for r that also works for g (the grey horizontal cards). However, the last line makes that difficult. Instead, let us restate 2. to say the number of cards on the floor above is the same as the current floor level minus 1. e.g. the first floor has zero cards on the floor above it. This way, we can say $g = i - 1$, which works for all floors! That means the number of cards on each floor is $r + g = 2i + i - 1 = 3i - 1$.

The last useful fact to remember is that the sum of numbers from 1 to N can be calculated with $N(N+1)/2$ (more info). That means, we can calculate the total cards for the tower by replacing i with $N(N+1)/2$. This gives us

$$\begin{aligned} K &= 3\left(\frac{N(N+1)}{2}\right) - N \\ &= \frac{\left(3N^2 + 3N - 2N\right)}{2} \\ &= \frac{N\left(3N + 3 - 2\right)}{2} \\ &= \frac{N\left(3N + 1\right)}{2} \end{aligned}$$

Therefore, a much simpler program would be

```
n = int(input())

print(n * (3 * n + 1) // 2)
```

Then there is also this one liner, because Python...

```
print((lambda n: n * (3 * n + 1) // 2)(int(input())))
```

Adding Teas

https://train.nzoi.org.nz/problems/872

Python examples

```
gcepls = [0, 0, 0, 0, 0, 0]
while True:
   tea = input()
   if tea[0] == "G":
       gcepls[0] += int(tea[2:])
   elif tea[0] == "C":
       gcepls[1] += int(tea[2:])
   elif tea[0] == "E":
       gcepls[2] += int(tea[2:])
   elif tea[0] == "P":
       gcepls[3] += int(tea[2:])
   elif tea[0] == "L":
       gcepls[4] += int(tea[2:])
   elif tea[0] == "S":
       gcepls[5] += int(tea[2:])
   else:
       break

totals = map(str, gcepls)
print(" ".join(totals))
```

```
teas = {'G': 0, 'C': 1, 'E': 2, 'P': 3, 'L': 4, 'S': 5}
pantry = [0 for _ in range(6)]
ln = input()

while 'D' not in ln:
   t, n = ln.split()
   pantry[teas[t]] += int(n)
   ln = input()

print(' '.join(map(str, pantry)))
```

C++ example

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

using namespace std;

int main()
{
    map<char, int> tea;
    char t;
    int x;
    cin >> t;
```

```
    while (t != 'D') {
        cin >> x;
        tea[t] += x;
        cin >> t;
    }

    for (auto ch : (string)"GCEPLS") cout << tea[ch] << ' ';
    cout << endl;
}
```

Tea Party

https://train.nzoi.org.nz/problems/870

Subtask 1

For subtask 1, it is sufficient to "simulate" each party, by looping over each person's favourite tea and subtracting from the host's stock. If the host's stock reaches 0, then we increase the count of disgruntled party goers instead. Note that instead of storing each person's six tea stocks, it is easier to just simulate each party as we read in the stocks and store the count of disgruntled party goers, even if that person isn't a potential host – in the worst case, K can be as large as N anyway. The time complexity of this solution is $O(N^2)$, since for each of the N (potential) parties, we iterate over N tea preferences $N \times N = N^2$.

Python example

```
teas = "GCEPLS"
N, K = map(int, input().split())
fav_teas = []
parties = []

for x in range(N):
   name, fav_tea = input().split()
   fav_teas.append(fav_tea)

for x in range(N):
   line = input().split()
   host = line[0]
   stocks = list(map(int, line[1:]))
   missed = 0
   for tea in fav_teas:
       tea_index = teas.index(tea)
       if stocks[tea_index] == 0:
           missed += 1
       else:
           stocks[tea_index] -= 1
   parties.append((host, missed))

for x in range(K):
   host = input()
   for party in parties:
       if party[0] != host:
           continue
       if party[1] == 0:
           print(host, "Successful")
       elif party[1] <= 2:
           print("{} Mildly Successful ({})".format(host, party[1]))
       else:
           print("{} Disaster ({})".format(host, party[1]))
       break
```

Full Solution

Instead of storing each person's favourite tea separately, we can precompute counts of how many people prefer each of the six tea types. Then to simulate a party, we only need to loop through the 6 tea types instead of the N tea preferences. For each tea type, the number of disgruntled party goers is equal to the difference (if positive) between the number of people who prefer that tea, and the stock of that tea. This allows us to simulate all of the parties in $O(N)$ time.

However, there is still one area from our last solution that is too slow – since we stored the outcome of each party in a list, we had to iterate over up to N parties to find the answer for each host, which takes $O(NK)$ time in total, or $O(N^2)$ in the worst case where $N = K$. Instead, we can use a dictionary (**std::unordered_map** in C++), which allows us to store (key, value) pairs, and then later lookup the value associated with some key in $O(1)$ time (on average). A **std::map** would also work in C++, which has an $O(logN)$ lookup time.

These two improvements reduce the overall complexity of our solution to $O(N)$.

Python example

```
teas = "GCEPLS"
N, K = map(int, input().split())
fav_teas = [0] * 6
parties = {}

for x in range(N):
    name, fav_tea = input().split()
    tea_index = teas.index(fav_tea)
    fav_teas[tea_index] += 1

for x in range(N):
    line = input().split()
    host = line[0]
    stocks = list(map(int, line[1:]))
    missed = 0
    for i in range(6):
        missed += max(0, fav_teas[i] - stocks[i])
    parties[host] = missed

for x in range(K):
    host = input()
    if parties[host] == 0:
        print(host, "Successful")
    elif parties[host] <= 2:
        print("{} Mildly Successful ({})".format(host, parties[host]))
    else:
        print("{} Disaster ({})".format(host, parties[host]))
```

C++ example

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

int n,k,stock;
char pref;
string name;
string teas = "GCEPLS";
int favs[256];
unordered_map<string, int> answers;

int main() {
   cin >> n >> k;
   for(int i = 0; i < n; i++) {
       cin >> name >> pref;
       favs[pref]++;
   }
   for(int i = 0; i < n; i++) {
       cin >> name;
       for(char t : teas) {
           cin >> stock;
           answers[name] += max(0, favs[t]-stock);
       }
   }
   for(int i = 0; i < k; i++) {
       cin >> name;
       cout << name << " ";
       if(answers[name] == 0) cout << "Successful\n";
       else if(answers[name] <= 2)
           cout << "Mildly Successful (" << answers[name] << ")\n";
       else cout << "Disaster (" << answers[name] << ")\n";
   }
}
```

Modern Art

https://train.nzoi.org.nz/problems/891

This problem might appear to have an obvious solution, but only one student received full marks during the contest. Remember, if your solution is reaching the time limit or memory limit then there is probably a better way to solve the problem.

Subtask 1 (75%)

For the first subtask, the height and width of the canvas are limited to 100 cm. We can simulate the canvas using a 2-dimensional array (or list) and filling in required squares for each paint throw.

Common mistakes

- Not colouring in the correct cells.
- Attempting to colour in and/or count cells which are outside of the canvas area.

Python example

```
H, W, N = map(int, input().split())

# create an empty canvas
canvas = []
for _ in range(H):
   canvas.append([None] * W)

# paint the canvas
for _ in range(N):
   ln = input().split()
   # convert first 3 bits to integers
   a, b, spread = map(int, ln[:3])
   colour = ln[3]

   # upper left corner of the painted square
   x = max(a - spread, 0)
   y = max(b - spread, 0)

   # lower right corner of the painted square
   u = min(a + spread, W - 1)
   v = min(b + spread, H - 1)

   # fill in between the two coordinates
   for i in range(x, u + 1):
       for j in range(y, v + 1):
           canvas[j][i] = colour
```

```
# count the number of coloured cells
col = input().strip()
num_col = 0
for i in range(H):
   num_col += canvas[i].count(col)

print(num_col)
```

C++ example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
   int height, width, n_throws;
   cin >> height >> width >> n_throws;

   vector<vector<char>> canvas(width, vector<char>(height, -1));
   for (int i = 0; i < n_throws; ++i) {
       int zx, zy, spread;
       cin >> zx >> zy >> spread;
       char colour;
       cin >> colour;

       // calculate square bounds
       int x = max(0, zx - spread);
       int y = max(0, zy - spread);
       int u = min(width - 1, zx + spread);
       int v = min(height - 1, zy + spread);

       // fill square
       for (int j = x; j <= u; j++) {
           for (int k = y; k <= v; k++) {
               canvas[j][k] = colour;
           }
       }
   }

   // count total colour
   char colour;
   cin >> colour;
   int total = 0;
   for (const auto& r : canvas) {
       total += count(r.begin(), r.end(), colour);
   }
   cout << total << endl;
}
```

Full Solution

For full marks we are going to need a different approach. The width and height of the canvas can be up to 1 million cm. What's the issue here?

Exercise. Can you calculate how much memory you will need to store a 1 million x 1 million 2-dimensional list/array? Assume that each cell stores a single character (which is 1 byte). *Answer at the end of this section.*

A 1 million by 1 million array/list takes too long for a computer to process and consumes too much memory. We are only allowed 1 second and 12 megabytes of space to run our solution. What can we do?

One technique is to use what's called coordinate compression. This works by ignoring all the cells inside each coloured square and only looking at the corner coordinates (i.e. the places where the squares begin and end). Since there are only 50 paint throws, we only need to care about 100 start and end points - easy as for a computer. But how do we do that?

1. Create a compressed 2d array/list. This only needs to be 100 x 100 in size since we have a maximum of 50 paint throws giving 2 * 50 = 100 beginning and ending coordinates.
2. We calculate the same beginning and ending coordinates as for subtask 1.
3. Add those coordinates, along with the associated colour, to a list so that we retain the order of the paint throws.
4. Also add those coordinates to a rows and columns list. Once sorted, the index of each coordinate in these lists will tell us the corresponding row and column position on our compressed canvas.
5. Sort the rows and columns lists.
6. Iterate through the paint throws and colour the appropriate cells on our compressed canvas.
7. Iterate through our compressed canvas. When a cell contains the colour we care about, calculate the actual area of that cell using the difference between the previous row and column coordinates, adding it to the total painted area.

The code examples below use this method.

Another student solution was to store a list of squares, defined by their top left and bottom right coordinates. For each new square, calculate if an intersection exists. If the new square does intersect, split the old square into pieces and remove the area covered by the new square. Additionally, one could also use event queues where each coordinate is stored as an event in a list and processed sequentially. There are often numerous ways to solve a problem :)

Python example

```
H, W, N = map(int, input().split())

# create an empty compressed canvas (2 * 50 = 100 max length and width)
canvas = []
for _ in range(100):
    canvas.append([None] * 100)

to_process = []
rows = set()  # use sets to remove duplicates
columns = set()

# paint the canvas
for _ in range(N):
    ln = input().split()
    # convert first 3 bits to integers
    a, b, c = map(int, ln[:3])
    colour = ln[3]

    # upper left corner of the painted square
    x, y = max(a - c, 0), max(b - c, 0)
    # lower right corner of the painted square
    u, v = min(a + c + 1, W), min(b + c + 1, H)

    to_process.append((x, y, u, v, colour))
    rows |= {y, v}  # add y and v to the rows set
    columns |= {x, u}

# convert to lists and order the coordinates so we can index them
rows = sorted(list(rows))
columns = sorted(list(columns))

# fill compressed canvas
for x, y, u, v, colour in to_process:
    for i in range(rows.index(y), rows.index(v)):
        for j in range(columns.index(x), columns.index(u)):
            canvas[i][j] = colour

# count the number of coloured cells
col = input().strip()
total = 0
for i in range(len(rows)):
    for j in range(len(columns)):
        if canvas[i][j] == col:
            # calculate square cm between coordinates
            total += (rows[i + 1] - rows[i]) * (columns[j + 1] -
columns[j])

print(total)
```

C++ example

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

using namespace std;

struct rect {
   int x, y, u, v;
   char colour;
};

vector<int> row, col;

int r (int val) {
   return lower_bound(row.begin(), row.end(), val) - row.begin();
}
int c (int val) {
   return lower_bound(col.begin(), col.end(), val) - col.begin();
}

char canvas[100][100];

int main() {
   int height, width, n_throws;
   cin >> height >> width >> n_throws;

   row = {0, height};
   col = {0, width};
   vector<rect> to_process;
   for (int i = 0; i < n_throws; ++i) {
       int zx, zy, spread;
       cin >> zx >> zy >> spread;
       char colour;
       cin >> colour;

       // calculate square bounds
       int x = max(0, zx - spread);
       int y = max(0, zy - spread);
       int u = min(width, zx + spread + 1);
       int v = min(height, zy + spread + 1);

       to_process.push_back({x, y, u, v, colour});
       col.push_back(x);
       col.push_back(u);
       row.push_back(y);
       row.push_back(v);
   }

   sort(row.begin(), row.end());
   sort(col.begin(), col.end());
   row.erase(unique(row.begin(),row.end()), row.end());
   col.erase(unique(col.begin(),col.end()), col.end());
```

```
    for(rect p : to_process) {
        for(int i = r(p.y); i < r(p.v); i++) {
            for(int j = c(p.x); j < c(p.u); j++) {
                canvas[i][j] = p.colour;
            }
        }
    }

    char colour;
    cin >> colour;
    long long res = 0;
    for(size_t i = 0; i < row.size() - 1; i++) {
        for(size_t j = 0; j < col.size() - 1; j++) {
            if(canvas[i][j] == colour) {
                long long r = row[i + 1] - row[i];
                long long c = col[j + 1] - col[j];
                res += r * c;
            }
        }
    }
    cout << res;
}
```

Exercise Answer:

A 1 million x 1 million grid has 1 trillion cells. If each cell uses 1 byte, then that's 1 trillion bytes or 931 GB. That's a fair bit of RAM. Additionally, as an estimate, a computer can generally do between 10 million to 100 million useful things per second with a compiled language like C++. So, it would probably also take more than 3 hours to finish it's calculations - assuming it had enough memory! The above C++ solution took 0.004 seconds and used less than 256 kB of memory. The Python solution took still only took 0.072 seconds and used about 3 MB of memory. It's amazing what you can achieve by using a better algorithm.

Twilight Sparkle's Magical Research

https://train.nzoi.org.nz/problems/868

Subtask 1

For the first subtask, it's enough to simulate the spell with a *for* loop:

```
u, v = 1, 1
for i in range(n):
    u, v = a*u + b*v, c*u + d*v
print(u, v)
```

If you've coded the Fibonacci sequence before, then this solution may seem familiar. In fact, the spell a=0, b=1, c=1, d=1 will generate Fibonacci numbers (try it!).

Subtask 2

For the second subtask, the numbers start to get very large.

In C++, this would make the variables overflow and give a wrong answer. In Python, the program might run out of memory instead.

Luckily, the problem statement gives us a way out. It tells us to only give the "last 5 digits" of the answer. We can do that using the modulo (%) operator:

```
print(u % 10000, v % 10000)
```

That stops the final answer from getting too large. But the numbers might get too big before that point. We want to prevent overflow all the way through the program, not just at the end.

Let's put another modulo inside the loop:

```
for i in range(n):
    u, v = (a*u + b*v) % 10000, (c*u + d*v) % 10000
```

These changes are enough to pass subtask 2.

Subtask 3

In the final subtask, the number of times to cast the spell can go up to a billion (10^9). A good rule of thumb is that a computer takes about 1 second to execute a billion instructions. Since

each step of the loop has many instructions, there's no way that our previous solution can finish in time!

It's clear that we can't use a simple loop anymore. We need something better.

Combining spells

If we cast two different spells, one after the other, on the same pile of ukuleles and vuvuzelas, what does the result look like?

We can work it out with a bit of algebra. Suppose that the two spells are (a_1, b_1, c_1, d_1) and (a_2, b_2, c_2, d_2) respectively.

Then, after the first spell, we'll have

$$u' = a_1 u + b_1 v$$

ukuleles, and

$$v' = c_1 u + d_1 v$$

vuvuzelas.

And after the second spell, we'll have

$$\begin{aligned} u'' &= a_2 u' + b_2 v' \\ &= a_2(a_1 u + b_1 v) + b_2(c_1 u + d_1 v) \\ &= (a_1 a_2 + c_1 b_2)u + (b_1 a_2 + d_1 b_2)v \end{aligned}$$

ukuleles, and

$$\begin{aligned} v'' &= c_2 u' + d_2 v' \\ &= c_2(a_1 u + b_1 v) + d_2(c_1 u + d_1 v) \\ &= (a_1 c_2 + c_1 d_2)u + (b_1 c_2 + d_1 d_2)v \end{aligned}$$

vuvuzelas.

There's something interesting about the equations for u'' and v''. If you squint a bit, the right-hand sides look like another spell. Indeed, rather than treating the two spells as separate things, we can combine them into a larger spell with the parameters:

$$\begin{aligned} a_3 &= a_1 a_2 + c_1 b_2 \\ b_3 &= b_1 a_2 + d_1 b_2 \\ c_3 &= a_1 c_2 + c_1 d_2 \\ d_3 &= b_1 c_2 + d_1 d_2 \end{aligned}$$

Using these equations, let's write a function to combine spells, and a function to cast them:

```
modulo = 10000
```



```
def combine_spells(s1, s2):
    a1, b1, c1, d1 = s1
    a2, b2, c2, d2 = s2
    return (
        (a1 * a2 + c1 * b2) % modulo,
        (b1 * a2 + d1 * b2) % modulo,
        (a1 * c2 + c1 * d2) % modulo,
        (b1 * c2 + d1 * d2) % modulo)

def cast_spell(s, u, v):
    a, b, c, d = s
    return ((a * u + b * v) % modulo, (c * u + d * v) % modulo)
```

Now, instead of applying the spell over and over again, we can build up a really big spell and apply it all at once!

```
s = (a, b, c, d)
big_spell = (1, 0, 0, 1)
for i in range(n):
    big_spell = combine_spells(big_spell, s)

u, v = cast_spell(big_spell, 1, 1)
print(u, v)
```

Exercise. On paper, work out what this code does when $n = 0$. Does that explain why `big_spell` is initialized to `(1, 0, 0, 1)`?

That's cool, but it isn't any faster than the original solution. We're still looping *n* times; we're just building up the spell *n* times instead of applying it. But this change opens up a neat trick which can make the solution much faster.

Exponentiation by squaring

Let S be an arbitrary spell.

Our original solution, casting the spell n times, might look like this:

$$S(S(S(S(u,\ v))))$$

When we're building up the spell and casting it in one go, it might look like this instead:

$$(((S \cdot S) \cdot S) \cdot S)(u,\ v)$$

In other words, we're combining the spells from left to right.

But what if we combine the spells in a different order?

$$((S \cdot S) \cdot (S \cdot S))(u,\ v)$$

This should give us the same answer—but since $(S \cdot S)$ appears twice, we can compute it once and use it both times, cutting the run time in half! Moreover, on larger values of n, this halving trick can be used multiple times, leading to an $O(\log n)$ speedup.

This algorithm is called *exponentiation by squaring*, and is the key to solving subtask 3.

```
def spell_power(s, n):
    if n == 0:
        return (1, 0, 0, 1)
    r = spell_power(s, n//2)
    r = combine_spells(r, r)
    if n % 2 == 1:
        r = combine_spells(r, s)
    return r

u, v = cast_spell(spell_power(s, n), 1, 1)

print(u, v)
```

Full solution (Python)

```
#!/usr/bin/env python3

modulo = 10000

def combine_spells(s1, s2):
    a1, b1, c1, d1 = s1
    a2, b2, c2, d2 = s2
    return (
        (a1 * a2 + c1 * b2) % modulo,
        (b1 * a2 + d1 * b2) % modulo,
        (a1 * c2 + c1 * d2) % modulo,
        (b1 * c2 + d1 * d2) % modulo)

def cast_spell(s, u, v):
    a, b, c, d = s
    return ((a * u + b * v) % modulo, (c * u + d * v) % modulo)

def spell_power(s, n):
    if n == 0:
        return (1, 0, 0, 1)
    r = spell_power(s, n//2)
    r = combine_spells(r, r)
    if n % 2 == 1:
        r = combine_spells(r, s)
    return r

s = tuple(map(int, input().strip().split()))
n = int(input())

u, v = cast_spell(spell_power(s, n), 1, 1)

print(u, v)
```