New Zealand Informatics Competition 2021
Round 2 Solutions

May 12, 2021

Overview

Questions

1. **Ducks in a Row**
2. **Chris' Ducks**
3. **More Ducks in a Row**
4. **Duck Latin**
5. **Ponderous Pondering Ducks**

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.

```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 2021

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.

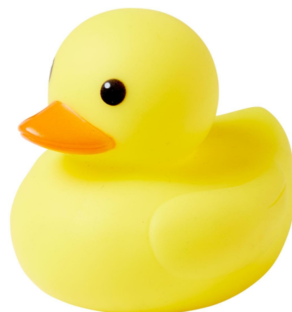
Test your code first!

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 2021

Ducks in a Row

https://train.nzoi.org.nz/problems/1216

Subtask 1

In this subtask, we only have ducks of a large size. That means the first line will all be 'L' and the other two lines will all be underscores. Each line will have N_L characters.

Python Subtask 1 Solution

```
s = int(input())
m = int(input())
l = int(input())

print('L' * l)
print('_' * l)
print('_' * l)
```

C++ Subtask 1 Solution

```
#include <iostream>

using namespace std;

int main() {
    int s, m, l;
    cin >> s >> m >> l;

    cout << string(l, 'L') << '\n';
    cout << string(l, '_') << '\n';
```

```
11     cout << string(l, '_') << '\n';
12 }
```

Subtask 2

We know that each line is going to have as many characters as the largest amount of ducks. So, let's just consider a single line.

If the line needs to have n characters and we have d ducks for this line, that means the amount of underscores in this line will be $n - d$. That means, we should have $\frac{n-d}{2}$ underscores on either side of the ducks. If $n - d$ is even (that is, divides perfectly) then it all works out but what if $n - d$ is odd? We can't have half an underscore on either side! In this case, we need to round down the number of underscores on the left side and round up the amount on the right side.

For example, let's say we have $d = 2$ ducks for a line and it needs to be $n = 5$ characters long. Halving $n - d = 3$ gives us 1.5. The number of underscores on the left will be 1 and the amount on the right will be 2.

Python Subtask 2 Solution

```
1  s = int(input())
2  m = int(input())
3  l = int(input())
4
5  def line(n, d, character):
6      # '//' is the integer division operator.
7      # It rounds down the result of the division.
8
9      underscores = n - d
10     left = underscores // 2
11     right = underscores - left
12
13     output = '_' * left
14     output += character * d
15     output += '_' * right
16     print(output)
17
18 n = max([s, m, l])
19 line(n, l, 'L')
20 line(n, m, 'M')
21 line(n, s, 'S')
```

© NZOI 2021

C++ Subtask 2 Solution

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

using namespace std;

void line(int n, int d, char character) {
    // Division on integers rounds down in C++.
    int underscores = n - d;
    int left = underscores / 2;
    int right = underscores - left;

    cout << string(left, '_');
    cout << string(d, character);
    cout << string(right, '_');
    cout << '\n';
}

int main() {
    int s, m, l;
    cin >> s >> m >> l;

    int n = max({s, m, l});
    line(n, l, 'L');
    line(n, m, 'M');
    line(n, s, 'S');
}
```

© NZOI 2021

Chris' Ducks

https://train.nzoi.org.nz/problems/1123

Subtask 1

In this subtask, we don't need to worry about handling days where any duck is allowed. We can keep track of how many ducks we have for each type and decrement the tally for the corresponding type for each day. If we ever try to decrement a tally that is zero then we know we've run out of ducks and so that will be the last day. To calculate the number of remaining ducks we add up all the remaining tallies for our duck types.

Python Subtask 1 Solution

```
l_days = int(input())
[nr, np, nd, ng] = map(int, input().split())
s = input()

last_day = 0
for i in range(l_days):
    if s[i] == 'R':
        if nr == 0: break
        nr -= 1
    elif s[i] == 'P':
        if np == 0: break
        np -= 1
    elif s[i] == 'D':
        if nd == 0: break
        nd -= 1
    elif s[i] == 'G':
        if ng == 0: break
```

```
        ng -= 1
    last_day = i + 1

print(last_day)
print(nr + np + nd + ng)
```

C++ Subtask 1 Solution

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

int main() {
    int l;
    cin >> l;
    string wants;
    array<int,4> ducks {};
    for (int i = 0; i < 4; ++i) {
        cin >> ducks[i];
    }
    cin.ignore();
    getline(cin, wants);
    int days = 0;

    for (char type : wants) {
        if (type == 'R') {
            if (ducks[0] < 1) break;
            --ducks[0];
        } else if (type == 'P') {
            if (ducks[1] < 1) break;
            --ducks[1];
        } else if (type == 'D') {
            if (ducks[2] < 1) break;
            --ducks[2];
        } else if (type == 'G') {
            if (ducks[3] < 1) break;
            --ducks[3];
        }
        ++days;

    }

```

© NZOI 2021

```
35      cout << days << '\n' << ducks[0] + ducks[1] + ducks[2] + ducks[3] <<
        ↪  '\n';
36  }
```

Subtask 2

When we encounter a day where we can give any duck (except for goose) to Chris, we might try to find a way to assign a particular duck. For example, we might give Chris a rubber duck if we have one available. However, on a later day, we might need a rubber duck when we don't have any left and so we might have been better off giving Chris a plush duck. While this method of managing the assignment of ducks is doable, there is a simpler way.

Let's change the problem slightly and imagine that instead of giving Chris a duck on each day, we just give Chris all the ducks he wants on say, the 10th day. We know all the ducks with specific types that Chris wants so we can remove those immediately. That only leaves the days where Chris doesn't care about the type of duck we give. As long as we have enough of these ducks remaining ($R + P + D$, remember, geese don't count) then everything is fine and dandy!

For every possible number of days we can check if we can fulfill the above criteria. The last day where we can will be our chosen result.

Python Subtask 2 Solution

```
1   l_days = int(input())
2   [nr, np, nd, ng] = map(int, input().split())
3   s = input()
4
5   def test(l_days, nr, np, nd, ng, s, n_days):
6       n_wild = 0
7       for i in range(n_days):
8           if s[i] == 'R': nr -= 1
9           elif s[i] == 'P': np -= 1
10          elif s[i] == 'D': nd -= 1
11          elif s[i] == 'G': ng -= 1
12          elif s[i] == '.': n_wild += 1
13
14      if (
15              nr >= 0 and np >= 0 and
16              nd >= 0 and ng >= 0 and
17              nr + np + nd >= n_wild
```

© NZOI 2021

```
18          ):
19              # Make sure to actually subtract the wildcard ducks!
20              return nr + np + nd + ng - n_wild
21      else:
22          # Return -1 if it's not possible to
23          # give for n_days consecutive days
24          return -1
25
26
27  # Iterate in reverse so we can break immediately
28  # when we find a valid number of days. Remember that
29  # Python ranges are exclusive so we need to add one.
30  for n_days in reversed(range(l_days + 1)):
31      remaining = test(l_days, nr, np, nd, ng, s, n_days)
32      if remaining != -1:
33          print(n_days)
34          print(remaining)
35          break
```

Subtask 3

Imagine giving ducks for 500,000 days (over a thousand years)! Our solution for Subtask 2 is too slow because for each day, we need to iterate through all the previous days. Fortunately, there is a faster way.

We don't need to recalculate the tallies for every possible number of days. As long as we keep a running total, we can easily check if we still meet our criteria or break out if we don't. You can imagine it as if you "owe" a certain amount of ducks to Chris. As long you have enough ducks to fill what you owe, everything works out!

By the way, you could also adapt Subtask 2's solution to binary search over the target number of days. We leave this as an exercise to the reader.

Python Subtask 3 Solution

```
1  l_days = int(input())
2  [nr, np, nd, ng] = map(int, input().split())
3  s = input()
4
5  n_wild = 0
6  last_day = 0
```

© NZOI 2021

```
7  for i in range(l_days):
8      available = nr + np + nd
9      if s[i] == 'R':
10         if nr == 0: break
11         if available - 1 < n_wild: break
12         nr -= 1
13     elif s[i] == 'P':
14         if np == 0: break
15         if available - 1 < n_wild: break
16         np -= 1
17     elif s[i] == 'D':
18         if nd == 0: break
19         if available - 1 < n_wild: break
20         nd -= 1
21     elif s[i] == 'G':
22         if ng == 0: break
23         ng -= 1
24     elif s[i] == '.':
25         if available < n_wild + 1: break
26         n_wild += 1
27     last_day = i + 1
28
29 print(last_day)
30 print(nr + np + nd + ng - n_wild)
```

C++ Subtask 3 Solution

```
1  #include <array>
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int l;
7      cin >> l;
8      string wants;
9      array<int,4> ducks {};
10     int quackers_left = 0;
11     for (int i = 0; i < 4; ++i) {
12         cin >> ducks[i];
13         if (i != 3) {
14             quackers_left += ducks[i];
15         }
16     }
```

© NZOI 2021

```
17      cin.ignore();
18      getline(cin, wants);
19      int days = 0;
20
21      for (char type : wants) {
22          if (type != 'G' && quackers_left <= 0) break;
23
24          if (type == 'R') {
25              if (ducks[0] < 1) break;
26              --ducks[0];
27              --quackers_left;
28          } else if (type == 'P') {
29              if (ducks[1] < 1) break;
30              --ducks[1];
31              --quackers_left;
32          } else if (type == 'D') {
33              if (ducks[2] < 1) break;
34              --ducks[2];
35              --quackers_left;
36          } else if (type == 'G') { // goose is not a duck
37              if (ducks[3] < 1) break;
38              --ducks[3];
39          } else { // wildcard
40              --quackers_left;
41          }
42          ++days;
43
44      }
45
46      // return number of days, number of ducks left (incl. geese)
47      cout << days << '\n' << quackers_left+ducks[3] << '\n';
48  }
```

© NZOI 2021

More Ducks in a Row

https://train.nzoi.org.nz/problems/1212

Subtask 1

In this subtask, there are only two ducks. If we pick a row or column 'outside' of both ducks, it will always result in a further distance than picking a row or column between the two ducks. This makes sense - both ducks will have to move an extra distance to that row or column, whereas if we pick somewhere between the two ducks, both ducks are moved towards each other so the total distance is reduced. Additionally, if we pick, for example, a row between on on the two ducks, it always results in the same distance. It is always equal to the distance in rows between the two ducks. Try it for yourself and see why this is! The same logic holds for columns as well. That means that we have two choices - either the best distance is the distance in rows, or the distance in columns between the two ducks. We simply calculate both and output the smaller number.

Python Subtask 1 Solution

```
n = int(input())
c1, r1 = list(map(int, input().split()))
c2, r2 = list(map(int, input().split()))
c_distance = abs(c1-c2)
r_distance = abs(r1-r2)
print(min(c_distance, r_distance))
```

C++ Subtask 1 Solution

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

using namespace std;
```

```
int main() {
    int n, x1, y1, x2, y2;
    cin>>n;
    cin>>x1>>y1;
    cin>>x2>>y2;
    cout<<min(abs(x1-x2), abs(y1-y2))<<endl;
}
```

Subtask 2

In this subtask, there are 200 or less ducks, and every duck's row and column is between 0 and 400. Clearly, choosing a row or column outside of that range will not give us the minimum distance. Instead, we can just try every row and column within that range and calculate the distance for each. We will output whichever of those distances is the smallest.

Python Subtask 2 Solution

```
def solve(vals):
    for i in range(401):
        total = 0
        for val in vals:
            total += abs(val - i)
        return total

cols = []
rows = []
n = int(input())

for i in range(n):
    col, row = map(int, input().split())
    cols.append(col)
    rows.append(row)

print(min(solve(cols), solve(rows)))
```

C++ Subtask 2 Solution

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

```

© NZOI 2021

```
3  using namespace std;
4
5  long long find_distance(vector<int> vals) {
6      long long best = LLONG_MAX;
7      for (int i=0; i <= 400; i++) {
8          long long total = 0;
9          for (auto val: vals) {
10             total += abs(val - i);
11         }
12         best = min(best, total);
13     }
14     return best;
15 }
16
17 int main() {
18     vector<int> cols;
19     vector<int> rows;
20     int n, col, row;
21
22     cin>>n;
23     for (int i=0; i<n; i++) {
24         cin>>col>>row;
25         cols.push_back(col);
26         rows.push_back(row);
27     }
28
29     cout<<min(find_distance(cols), find_distance(rows))<<endl;
30 }
```

Subtask 3

If we tried the previous approach for the final subtask, we would have to iterate through $1,000,000,000$ rows and columns. For **each** of those we would then iterate through $60,000$ ducks. That means we would have to do on the order of $2 * 1,000,000,000 * 60,000 = 120,000,000,000,000$ operations! That's far more than any single computer can do in one second (as of April 2021).

Clearly, we need a better method than trying all of the possible rows and columns. Let's say there are 7 ducks, and we start at a column left of all of the ducks - column -1, for example. Now let's move right one column. What happens to the total distance?
There are 7 ducks to the right of us. Therefore, when we move right one column,

© NZOI 2021

the distance from **each** duck to us decreases by 1. In other words, the total distance decreases by $7 * 1 = 7$.
If we keep moving right, the distance keeps decreasing by 7 each time until we reach the first duck. When we move one column left past the first duck, there is now 1 duck on our left, and 6 ducks on our right. So, our distance to the duck on our left increases by 1, and our distance to each of the other ducks still decreases by 1. Therefore, total total distance decrease is $-1 + 6 * 1 = 5$. This is still a net decrease, so it makes sense to keep going further right.
Similarly, past the second duck our distance increases by 3 each column. However, when we get to the middle duck - the fourth duck - things change. There are now an equal number (3) of ducks to our left and our right. That means that if we keep going right, there will be more ducks on our left than on our right. In other words, if we move further right, the total distance will increase. Since we know this column is has a smaller distance than all the columns to the left, and all the columns to the right must increase the distance, this column must be the column that minimises the total distance!
There's still one catch - since ducks can appear on top of each other, what if there are several 'duplicate' columns in the middle? In that case, the assumption that there are an equal number of ducks to the left and right may not apply, as some of the ducks aren't to our left or right, but at the same position as us. However, if we go to the right of all of those columns, then there will be more columns on the left than the right, so the distance still decreases. That means the best column is still any of those duplicates in the middle.
In other words, the optimal column is the *median* (middle element) of all the ducks' columns. If we have an even number of columns, then any point in between or on the *two* 'middle' ducks will give the same minimal distance. The exact same logic applies to the rows. Therefore, we will simply find the distance if we choose the median row, the distance if we choose the median column, and output the lesser of those two.

Python Subtask 3 Solution

```
def solve(vals):
    vals.sort()
    total = 0
    median = vals[len(vals)//2]
    for val in vals:
        total += abs(val - median)
    return total

cols = []
```

© NZOI 2021

```
10 rows = []
11 n = int(input())
12
13 for i in range(n):
14     col, row = map(int, input().split())
15     cols.append(col)
16     rows.append(row)
17
18 print(min(solve(cols), solve(rows)))
```

C++ Subtask 3 Solution

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  long long solve(vector<int> vals) {
6      sort(vals.begin(), vals.end());
7      long long total = 0;
8      int median = vals[vals.size()/2];
9      for (auto val: vals) {
10         total += abs(val - median);
11     }
12     return total;
13 }
14
15 int main() {
16     vector<int> cols;
17     vector<int> rows;
18     int n, col, row;
19
20     cin>>n;
21     for (int i=0; i<n; i++) {
22         cin>>col>>row;
23         cols.push_back(col);
24         rows.push_back(row);
25     }
26
27     cout<<min(solve(cols), solve(rows))<<endl;
28 }
```

© NZOI 2021

Duck Latin

https://train.nzoi.org.nz/problems/1221

This problem is an example of *cryptography* - using codes to make messages difficult for someone else to read. In cryptography, we use a code to *encode plaintext* (the message we want to send, eg. in plain English) into *ciphertext* (the scrambled message that we actually send). In this problem, we essentially do the opposite - given some *ciphertext*, what *plaintext* could it have originally been? In other words, how many ways are there to *decode* the *ciphertext*?
There are two different encodings - Duck and Goose Latin. Each of these encodings also have two different rules. To make the following explanation more concise, let's name the different rules:

- Duck Latin where the word starts with a consonant - DUCKCONS
- Duck Latin where the word starts with a vowel - DUCKVOWEL
- Goose Latin where the word contains no vowels - GOOSECONS
- Duck Latin where the word contains vowels - GOOSEVOWEL

Subtasks 1 and 2

In the first subtask, each word ends in 'onk'. Let's work backwards and see which encoding rules could have resulted in a words ending in 'onk'. GOOSECONS is an obvious candidate - if the original work was only consonants then we would end up with a word that ends in 'onk'. To check this case, we can check if every letter (excluding the 'onk') is a consonant - if it is, then this is one possible decoding. For example, if the encoded word is 'jklonk', then the original word could have been 'jkl'.
Both DUCKCONS and DUCKVOWEL cannot result in words that end in 'onk', as they both result in words that end in 'ck' (a word that ends in 'uack' also ends in 'ck').

gooseVOWEL can also result in a word that ends in 'onk'! for example, the word 'honk' encoded with gooseVOWEL results in 'holfonk'. We will therefore also need to check if the word could have been encoded this way. For a word to have be encoded in gooseVOWEL, it must have contained vowels. Additionally, after each vowel, we add 'lf' and that vowel again. How do we check that?
The first vowel in the ciphertext must be followed with 'lf' and the vowel again. That means that the second vowel doesn't need to be followed with the 'lf' pattern, as it was added in the encoding process. For example, when 'honk' is encoded into 'holfonk', the second vowel in 'holfonk' isn't followed by the 'lf' pattern. However, the next vowel (if it exists) does need to be followed by that pattern - eg. 'hoonk' → 'holfoolfonk'.
Thus, we can iterate through the word - if we find a vowel, we check for the 'lf' pattern. If it's valid, we can then skip past the four-letter pattern and repeat. If it's not, then the ciphertext can't have been encoded with gooseVOWEL.
You may have noticed that this solution is actually valid for both subtasks 1 and 2! Unfortunately, the test data for Subtask 1 did not contain cases that tested for the gooseVOWEL encoding, so solutions that only checked gooseCONS would also pass. Subtask 1 and 2 are actually effectively equivalent, as they both require checking for gooseVOWEL and gooseCONS.

Python Subtask 1 & 2 Solution

```
1  VOWELS = {'a','e','i','o','u'}
2
3  def is_vowel(c):
4      return c in VOWELS
5
6  def gooseCONS(S, L):
7      if L <= 3 or S[-3:] != "onk":
8          return 0
9      elif all(map(lambda c: not is_vowel(c), S[:-3])):
10         return 1
11     else:
12         return 0
13
14 def gooseVOWEL(S, L):
15     index = 0
16     valid = False
17     while index < L:
18         if is_vowel(S[index]):
19             if S[index+1 : index+4] == 'lf' + S[index]:
20                 valid = True
```

© NZOI 2021

```
                index += 4
            else:
                valid = False
                break
        else:
            index += 1

    return valid


L = int(input())
S = input()
print(gooseCONS(S, L) + gooseVOWEL(S, L))
```

C++ Subtask 1 & 2 Solution

```
#include <iostream>
#include <cassert>
#include <string>
#include <unordered_set>
#include <algorithm>

using namespace std;

unordered_set<int> VOWELS {'a','e','i','o','u'};

inline bool is_vowel(char c) {
    return VOWELS.find(c) != VOWELS.end();
}

int gooseCONS(string S, int L) {
    if (L <= 3 || S.compare(L-3, 3, "onk") != 0) {
        return 0;
    }
    if (all_of(S.begin(), S.end()-3, [](char c){return !is_vowel(c);}))
    ↪ {
        return 1;
    } else {
        return 0;
    }
}

int gooseVOWEL(string S, int L) {
```

© NZOI 2021

```
27      int index = 0;
28      bool valid = false;
29      while (index < L) {
30          if (is_vowel(S[index])) {
31              if (index+3 < L && S[index+1] == 'l' && S[index+2] == 'f' &&
                 ↪  S[index+3] == S[index]) {
32                  valid = true;
33                  index += 4;
34              } else {
35                  valid = false;
36                  index = L;
37              }
38          } else {
39              index += 1;
40          }
41      }
42      return valid;
43  }
44
45  int main() {
46      int L;
47      string S;
48      cin>>L>>S;
49      cout << gooseCONS(S, L) + gooseVOWEL(S, L) << endl;
50  }
```

Subtasks 3 and 4

Now we need to check for the cases where DUCKVOWEL and DUCKCONS could be used. If the ciphertext ends in 'uack', then DUCKCONS could have been used. Working backwards from the ciphertext, how could we generate all the possible plaintexts?
In DUCKCONS we take consonants at the start of the word and place them at the back of the word, then add 'uack'. Therefore, working backwards, we should first remove the 'uack'. Then, the decodings must come from taking consonants at the end of the word and moving them back to the start of the word. More specifically, we should only consider consonants after the last vowel in the word.
Why? Consider the ciphertext 'aintruack'. We first get rid of the 'uack' to get 'aintr'. Then, working back from the last consonants, we can get:

- 'aintr' → 'raint'

© NZOI 2021

- 'aintr' → 'train'
- 'aintr' → 'ntrai'

But we can't keep going and get 'intra', because then the word would start with a vowel, so DUCKCONS would not apply. What that means is that we can just count all the consonants between the last vowel before the 'uack' and the 'uack'. That should be all the possible plaintext words.
But there's a catch! Take for example the ciphertext 'totuack'. We will calculate one possible decoding - 'tto'. But the plaintext 'tto', when encoded with DUCKCONS, will be 'ottuack' and not 'totuack'. This means that our method is wrong! Why? DUCKCONS shifts **all** consonants before the first vowel to the end of the word. Because the ciphertext starts with the consonant 't', it means that not all the consonants before the first vowel were shifted. Therefore, the ciphertext must start with a vowel in order to have been encoded with DUCKCONS.
But there's yet another catch! There actually is a case when a ciphertext starts with a consonant - if the plaintext word only contains consonants. In that case, a DUCKCONS encoding wouldn't shift the consonants at all - after all, there is no vowel to shift all the consonants behind. Therefore, in order to be valid DUCKCONS, the ciphertext must either:

- Start with a vowel, or
- Contain nothing but consonants before the 'uack'

For ciphertext that ends in 'ck', we can use a very similar approach. We count the number of vowels between the last consonant (excluding 'ck') and the 'ck'. We also need to make sure that the ciphertext starts with a consonant, or contains only vowels.
Combining all these methods with those in the previous subtask, we can solve Subtask 4! Subtask 3 was intended for solutions that were able to find all valid plaintext but were not efficient enough. For example, some solutions relied on actually generating all the possible plaintext words. This was not sufficiently efficient to solve Subtask 4.

Python Subtask 3 & 4 Solution

```
VOWELS = {'a','e','i','o','u'}

def is_vowel(c):
    return c in VOWELS

def duckCONS(S, L):
```

```
7       if L <= 4 or S[-4:] != "uack":
8           return 0
9   
10      done, all_consonants = False, False
11      index = L-5
12      while not done:
13          if index < 0:
14              all_consonants = True
15              done = True
16          elif is_vowel(S[index]):
17              done = True
18          else:
19              index -= 1
20  
21      if all_consonants:
22          return 1
23      elif not is_vowel(S[0]):
24          return 0
25      else:
26          return L - 5 - index
27  
28  
29  def duckVOWEL(S, L):
30      if L <= 2 or S[-2:] != "ck":
31          return 0
32  
33      done, all_vowels = False, False
34      index = L-3
35      while not done:
36          if index < 0:
37              all_vowels = True
38              done = True
39          elif not is_vowel(S[index]):
40              done = True
41          else:
42              index -= 1
43  
44      if all_vowels:
45          return 1
46      elif is_vowel(S[0]):
47          return 0
48      else:
49          return L - 3 - index
```

```python
50
51 def gooseCONS(S, L):
52     if L <= 3 or S[-3:] != "onk":
53         return 0
54     elif all(map(lambda c: not is_vowel(c), S[:-3])):
55         return 1
56     else:
57         return 0
58
59 def gooseVOWEL(S, L):
60     index = 0
61     valid = False
62     while index < L:
63         if is_vowel(S[index]):
64             if index+3 < L and S[index+1 : index+4] == 'lf' + S[index]:
65                 valid = True
66                 index += 4
67             else:
68                 valid = False
69                 break
70         else:
71             index += 1
72
73     return valid
74
75
76 L = int(input())
77 S = input()
78 assert(len(S) == L)
79 print(duckCONS(S, L) + duckVOWEL(S, L) + gooseCONS(S, L) + gooseVOWEL(S,
   ↪  L))
```

C++ Subtask 3 & 4 Solution

```cpp
1 #include <iostream>
2 #include <cassert>
3 #include <string>
4 #include <unordered_set>
5 #include <algorithm>
6
7 using namespace std;
8
9 unordered_set<int> VOWELS {'a','e','i','o','u'};
```

© NZOI 2021

```
10
11  inline bool is_vowel(char c) {
12      return VOWELS.find(c) != VOWELS.end();
13  }
14
15  int duckCONS(string S, int L) {
16      if (L <= 4 || S.compare(L-4, 4, "uack") != 0) {
17          return 0;
18      }
19      bool done = false;
20      bool all_consonants = false;
21      int index = L-5;
22      while (!done) {
23          if (index < 0) {
24              all_consonants = true;
25              done = true;
26          } else if (is_vowel(S[index])) {
27              done = true;
28          } else {
29              index--;
30          }
31      }
32      if (all_consonants) {
33          return 1;
34      } else if (!is_vowel(S[0])) {
35          return 0;
36      } else {
37          return L - 5 - index;
38      }
39  }
40
41  int duckVOWEL(string S, int L) {
42      if (L <= 2 || S.compare(L-2, 2, "ck") != 0) {
43          return 0;
44      }
45      bool done = false;
46      bool all_vowels = false;
47      int index = L-3;
48      while (!done) {
49          if (index < 0) {
50              all_vowels = true;
51              done = true;
52          } else if (!is_vowel(S[index])) {
```

© NZOI 2021

```
53                  done = true;
54              } else {
55                  index--;
56              }
57          }
58          if (all_vowels) {
59              return 1;
60          } else if (is_vowel(S[0])) {
61              return 0;
62          } else {
63              return L - 3 - index;
64          }
65      }
66
67      int gooseCONS(string S, int L) {
68          if (L <= 3 || S.compare(L-3, 3, "onk") != 0) {
69              return 0;
70          }
71          if (all_of(S.begin(), S.end()-3, [](char c){return !is_vowel(c);}))
            ↪  {
72              return 1;
73          } else {
74              return 0;
75          }
76      }
77
78      int gooseVOWEL(string S, int L) {
79          int index = 0;
80          bool valid = false;
81          while (index < L) {
82              if (is_vowel(S[index])) {
83                  if (index+3 < L && S[index+1] == 'l' && S[index+2] == 'f' &&
                    ↪  S[index+3] == S[index]) {
84                      valid = true;
85                      index += 4;
86                  } else {
87                      valid = false;
88                      index = L;
89                  }
90              } else {
91                  index += 1;
92              }
93          }
```

© NZOI 2021

```
94        return valid;
95    }
96
97    int main() {
98        int L;
99        string S;
100       cin>>L;
101       cin>>S;
102       assert(S.length() == L);
103       cout << duckCONS(S, L) + duckVOWEL(S, L) + gooseCONS(S, L) +
          ↪   gooseVOWEL(S, L) << endl;
104   }
```

© NZOI 2021

Ponderous Pondering Ducks

https://train.nzoi.org.nz/problems/1220

Subtask 1

In this subtask, H and W are small so we can simulate the entire grid and the ducks' movement upon it. For each duck, we need to check if there's a duck already at the starting position $(0, 0)$ and if the duck collides with any existing ducks after moving a square.

Python Subtask 1

```
[h_height, w_width] = map(int, input().split())
n_ducks = int(input())

occupied = [([False] * w_width) for _ in range(h_height)]
for d in range(n_ducks):
    # This syntax splits a list into the first element and the rest
    [n_strides, *steps] = list(map(int, input().split()))
    is_up = True

    x, y = 0, 0
    collision = occupied[0][0]
    for step in steps:
        # Move the duck and check for collisions
        for offset in range(step):
            if is_up: y += 1
            else: x += 1

            if occupied[y][x]:
                collision = True
```

```
            is_up = not is_up

    # Occupy the final position
    occupied[y][x] = True
    if collision: print("OUCH")
    else: print("smooth swimming")
```

Subtask 2

Our "grid" in this subtask is guaranteed to have a height of one. In other words, it's a line. That means we only need worry about the ducks' movement towards the right.

Let's say there's a duck 10 squares right from the starting position. If the final position of the next duck is 10 squares or greater then we know it must have collided with the first duck (there's no other path it could have taken!). Hence, all we need to do is keep track of the current leftmost duck and compare its position with subsequent ducks.

Python Subtask 2

```
[h_height, w_width] = map(int, input().split())
n_ducks = int(input())

x_left = (10 ** 9) + 1
for d in range(n_ducks):
    [n_strides, *steps] = map(int, input().split())

    x = 0
    for i in range(n_strides):
        x += steps[i * 2 + 1]

    if x >= x_left:
        print("OUCH")
    else:
        print("smooth swimming")
        x_left = x
```

© NZOI 2021

Subtask 3

In this subtask, H and W are large so, unlike Subtask 1, we can't simulate the entire grid. However, N is small. Instead of simulating each grid square we can store a list of occupied positions. Whenever a duck moves a leg of its route, we check if it collides with any existing ducks.

Each route leg (for example, 2 up or 3 right) has a starting position and an ending position. The line connecting these two positions is either vertical or horizontal. A collision occurs if any duck is sitting on this line.

Python Subtask 3

```
1  [h_height, w_width] = map(int, input().split())
2  n_ducks = int(input())
3
4  occupied = []
5  for d in range(n_ducks):
6      [n_strides, *steps] = map(int, input().split())
7      is_up = True
8
9      x, y = 0, 0
10     collision = False
11     for step in steps:
12         if is_up:
13             new_y = y + step
14             for other_y, other_x in occupied:
15                 if other_x != x: continue
16                 if y <= other_y <= new_y:
17                     collision = True
18             y = new_y
19         else:
20             new_x = x + step
21             for other_y, other_x in occupied:
22                 if other_y != y: continue
23                 if x <= other_x <= new_x:
24                     collision = True
25             x = new_x
26
27         is_up = not is_up
28
29     occupied.append((y, x))
30     if collision: print("OUCH")
```

© NZOI 2021

```
31      else: print("smooth swimming")
```

Subtask 4

In this subtask the number of ducks is much larger, so we can't just check against every possible duck's position.

We know that our duck moves along a series of horizontal and vertical lines. To check whether this duck will collide with any other duck on those lines, we can *binary search* that line. That is, for every row and column, we can keep a list of all the ducks on that row or column. When we want to see if our duck will collide with any other duck on a specific row, we binary search that list to see if there are any ducks between our starting and ending position.

A simple way of doing this is to binary search for the *closest* duck that is *further* away than the starting point. If that duck is closer or at the same spot as our ending point, then there must be a collision. Thankfully, most languages come with libraries/modules that can do this for us. For example, python has a `bisect` module that provides a `bisect_left` function. Note that in order for a binary search to work, the input list must be in sorted order. Therefore, when adding each duck to its corresponding row and column, we must insert it is sorted order. The `bisect` module provides a `insort` method that does this for us.

So we've now made our collision checking much more time efficient, but we've introduced a big problem. If we have to store a sorted list for every column and row, and there are 1,000,000,000 columns and rows, then we will easily run out of memory - even if all the lists are empty. However, note that many of those columns and rows won't ever be occupied by a duck. There are only 10,000 ducks, so there are only 10,000 different possible columns and 10,000 different rows that can ever be occupied. To save on memory, we can have a `map` (or `dictionary` in Python) that maps between row/column indexes and the lists for those rows/columns. That way, we only need to store duck positions for rows and columns that actually have ducks in them.

What's the time complexity of this solution? Let M be the maximum value of m_i. For every N ducks we do at most M 'steps', and for each step we binary search for ducks. We can binary search through N ducks at most, so this step which takes $O(log(N))$ time. So far our complexity is $O(N * M * log(H))$.

However, for each duck we also insert it into to corresponding row and column. This is an insertion into a list, which can cost $O(n)$ time at worst if the list contains n items. This adds an extra $O(N^2)$ cost. Therefore, our total complexity is $O(N * M * log(H) + N^2)$.

© NZOI 2021

Python Subtask 4 Solution

```
import bisect
from collections import defaultdict

[h_height, w_width] = map(int, input().split())
n_ducks = int(input())

s = ""
columns, rows = {}, {}
for d in range(n_ducks):
    [n_strides, *steps] = map(int, input().split())
    is_up = True

    x, y = 0, 0
    collision = False
    for step in steps:
        if is_up:
            new_y = y + step
            if x in columns and not collision:
                p = bisect.bisect_left(columns[x], y)
                if p != len(columns[x]):
                    other_y = columns[x][p]
                    if y <= other_y <= new_y:
                        collision = True
            y = new_y
        else:
            new_x = x + step
            if y in rows and not collision:
                p = bisect.bisect_left(rows[y], x)
                if p != len(rows[y]):
                    other_x = rows[y][p]
                    if x <= other_x <= new_x:
                        collision = True
            x = new_x
        is_up = not is_up

    # Insert and keep sorted
    if x not in columns: columns[x] = []
    if y not in rows: rows[y] = []
    bisect.insort(columns[x], y)
    bisect.insort(rows[y], x)

```

© NZOI 2021

```
42     if collision: s += "OUCH\n"
43     else: s += "smooth swimming\n"
44 print(s)
```

Extra

If you're using a language that has tree-based set data structures, such as C++ with `set` or Java with `TreeSet`, then we can actually improve a little on the Python solution. Instead of keeping lists for rows and columns, we can keep `set`s or `TreeSet`s instead. These structures are implemented as binary search trees, which not only allow us to search in $O(log(n))$ time, but allow insertion of elements in $O(log(n))$ time as well. Using those instead of Python lists brings our complexity down to $O(N * M * log(N))$.
We can't use Python `set`s because, while those allow us to search for exact matches, they don't have the ability to find the *closest* or *next* item to the one we searched. We depend on that ability in our solution. If you're curious, Python `set`s are based on a totally different approach - a *hashtable* - which you can read about here.

C++ Extra

```
1  #include <iostream>
2  #include <unordered_map>
3  #include <set>
4
5  using namespace std;
6
7  int main() {
8      // If you are using C++ style input/output this
9      // will improve its throughput significantly
10     cin.tie(nullptr);
11     ios::sync_with_stdio(false);
12
13     int h_height, w_width, n_ducks;
14     cin >> h_height >> w_width >> n_ducks;
15
16     unordered_map<int, set<int>> columns, rows;
17     for (int duck = 0; duck < n_ducks; ++duck) {
18         int n_strides; cin >> n_strides;
19
20         int x = 0, y = 0;
21         bool collision = false;
```

© NZOI 2021

```
        for (int stride = 0; stride < n_strides; ++stride) {
            int up, right; cin >> up >> right;

            // Upwards
            int new_y = y + up;
            auto it_up = columns[x].lower_bound(y);
            if (it_up != columns[x].end())
                if (y <= *it_up && *it_up <= new_y)
                    collision = true;
            y = new_y;

            // Rightwards
            int new_x = x + right;
            auto it_right = rows[y].lower_bound(x);
            if (it_right != rows[y].end())
                if (x <= *it_right && *it_right <= new_x)
                    collision = true;
            x = new_x;
        }

        rows[y].insert(x);
        columns[x].insert(y);
        if (collision) cout << "OUCH\n";
        else cout << "smooth swimming\n";
    }
}
```

© NZOI 2021

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 $O(N)$.

$O(N^2)$ 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 $N \lesssim 10\,000$. 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.

© NZOI 2021