New Zealand Informatics Competition 2022
Round 1 Solutions

July 10, 2022

Overview

Questions

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

`my_ints = list(map(int, input().split()))`

Notice that we used `list()`. This is because `map()` returns us a special generator object, not a list. However, generator objects are easily converted to lists.

We suggest having a go at some of the NZIC Practice Problems.

Move on to the next question

If you are spending too much time on a question, move on. There could be easy subtasks waiting in the next question. Often, you will think of a solution to your problem while working on another question. It also helps to come back to a question with a fresh pair of eyes.

Take time to read the questions

Don't underestimate the value of taking time to read and understand the question. You will waste exponentially more time powering off thinking you have the solution only to discover you missed something obvious.

In the real world, it is very rare to be provided with a problem in a simplistic form. Part of the challenge with these contests is reading and understanding the question, then figuring out what algorithm will be needed. Before submitting, check to make sure you have done everything the question asks you to do.

Test your code first!

It is much more time efficient to test your solutions on your own computer first. It can take a while to wait for the server to run your code. It is far more efficient to test it yourself first, submit it, then start reading the next question while you wait for your previous solution to be marked by the server.

While testing, remember to add some edge case tests. For example, if a question specifies "$1 \leq N \leq 10$" then try out an input where $N = 1$. Think of other tricky inputs that might break your code. Look at the feedback the server is giving you.

Use a rubber duck

https://en.wikipedia.org/wiki/Rubber_duck_debugging
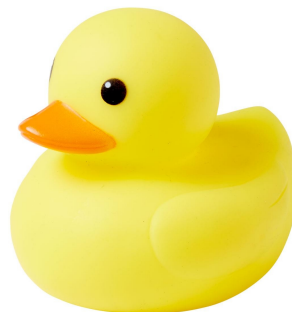
Flower Garden

Subtask 1

In this subtask, all flowers start off as the same colour. To ensure that no two adjacent flowers share the same colour, we must replace every other flower. If there are N flowers, then this requires $N/2$ replacements, rounded down.

```python
print(int(input()) // 2)
```

Subtask 2

In this subtask, it is guaranteed that no flower shares the same colour as both of is neighbours. Here we can simply count the number of pairs of adjacent flowers of the same colour – for each pair, we must replace one of these flowers with a different colour. Note that we don't care about what the actual replacements are, and it is always possible to replace a flower with one that does not share the same colour as any of its neighbours, since there are three possible colours, but each flower has at most two neighbours.

```python
N = int(input())
s = input()
replacements = 0
for i in range(1, N):
    if s[i] == s[i - 1]:
        replacements += 1
print(replacements)
```

Subtask 3

Our solution for subtask 2 does not work if there are three or more flowers of the same colour in a row. For example, with `RRR`, there are two pairs of adjacent flowers of the same colour, but we can resolve both with a single replacement of the middle flower. We can fix this by skipping a flower after counting a pair, because we can always replace the second flower in the pair with a colour that is not the same as the next flower's colour.

```python
N = int(input())
s = input()
replacements = 0
i = 1
while i < N:
    if s[i] == s[i - 1]:
        replacements += 1
        i += 1
    i += 1
print(replacements)
```

Alternatively, we can use Python's `str.count()` method, since it only counts non-overlapping substrings.

```python
N = int(input())
s = input()
print(s.count("RR") + s.count("BB") + s.count("GG"))
```

Another solution is to treat the garden as multiple cases of subtask 1 -– we divide the N flowers into several groups such that each group only contains a series of adjacent flowers of a single colour. Then for each group, we need to make $X/2$ replacements, where X is the size of the group.

```python
import itertools
N = int(input())
s = input()
replacements = 0
for colour, group in itertools.groupby(s):
    replacements += len(list(group)) // 2
print(replacements)
```

Lost Shoes

https://train.nzoi.org.nz/problems/1229

Note: all solutions for this problem are in Python.

Subtask 1

In this subtask, all shoes are green. This means that we only have to check if there are an even number of green shoes - if the number of green shoes is even then they must be all paired up. Because each green shoe is a single letter (G), we can just count the length of the input we get, and that will be the number of green shoes.

To check if that number is odd, we can use the *modulo* operator. In many languages (such as Python, C++ and Java), the *modulo* operator is the % symbol. $x\%y$ divides x by y and returns the remained. For example, $5\%2$ returns 1 and $4\%2$ returns 0. If we want to check if x is even, we can just check the value of $x\%2$, since if the remainder of x divided by 2 is 0 then x must be even. Otherwise, it must be odd.

To solve this subtask, we can just check if the length of the input modulo 2 is 0, and then output the corresponding answers:

```python
1  if len(input()) % 2 == 0:
2      print("All paired up!")
3  else:
4      print("A Green shoe has no partner.")
```

Subtask 2

In this subtask, there's a few more different kinds of shoes. We can't just count the length of the input anymore since not all the shoes are the same color. However,

we can use Python's built-in `str.count()` method to count the colors for us (or `std::count` if you use C++). We could just count the number of times each color appears for each of the colors, and check if it's even:

```
1   input_string = input()
2
3   paired_up = True
4
5   green_count = input_string.count("G")
6   if green_count % 2 != 0:
7       print("A Green shoe has no partner.")
8       paired_up = False
9
10  black_count = input_string.count("B")
11  if black_count % 2 != 0:
12      print("A Black shoe has no partner.")
13      paired_up = False
14
15  red_count = input_string.count("R")
16  if red_count % 2 != 0:
17      print("A Red shoe has no partner.")
18      paired_up = False
19
20  mustard_count = input_string.count("M")
21  if mustard_count % 2 != 0:
22      print("A Mustard shoe has no partner.")
23      paired_up = False
24
25
26  if paired_up:
27      print("All paired up!")
```

Note that we also use a variable `paired_up` to keep track of whether all the colors have been paired up or not.

This code will pass Subtask 2, but there's a lot of repeated code in there. A single typo in any of them could easily lead cause you to fail a test case. We can simplify this code by just keeping a list of the color letters and names. Then we can just loop over the list and do our count for each color:

```
1   input_string = input()
2   colors = [("G", "Green"), ("B", "Black"), ("R", "Red"), ("M",
    ↪   "Mustard")]
3
4   paired_up = True
```

```
5   for color_letter, color_name in colors:
6       count = input_string.count(color_letter)
7       if count % 2 != 0:
8           print("A", color_name, "shoe has no partner.")
9           paired_up = False
10
11  if paired_up:
12      print("All paired up!")
```

Subtask 3

If we try our Subtask 2 approach (simply counting how many times each color appears) on the Subtask 3 test cases, we'll get a wrong answer! Why? The issue is that 3 colours (Black, Brown, and Blue) all include 'B' in their colour code. If we count the number of 'B's in the input, then we'll end up counting all the 'B's in 'Br' and 'Bl' as well!

We can solve this problem by subtracting the number of 'Br's and 'Bl's from the total number of 'B's. This gets rid of all the Brown and Blue shoes we incorrectly counted as Black, and will give us the total number of Black shoes. To keep track of the counts, we can use a *dictionary* to store the counts for each color. After populating the dictionary with the counts of each color, we subtract the count for Black shoes and then use the same approach as Subtask 2.

```
1   input_string = input()
2   colors = [("G", "Green"), ("B", "Black"), ("R", "Red"), ("Bl", "Blue"),
    ↪  ("Br", "Brown"), ("M", "Mustard")]
3   counts = {}
4
5   paired_up = True
6   for color_letter, color_name in colors:
7       count = input_string.count(color_letter)
8       counts[color_letter] = count
9
10  counts["B"] -= counts["Bl"] + counts["Br"]
11
12  paired_up = True
13  for color_letter, color_name in colors:
14      count = counts[color_letter]
15      if count % 2 != 0:
16          print("A", color_name, "shoe has no partner.")
17          paired_up = False
```

```
18
19  if paired_up:
20      print("All paired up!")
```

Alternative solutions

We could instead count up each color manually by looping through each character in the input string. To differentiate between 'B', 'Bl' and 'Br', we can attempt to 'look ahead' at the next character. This solution is a bit more efficient, but more complicated, than the model solution.

```python
colours = ["Green","Black","Red","Blue","Brown","Mustard"]
shoe_colours = ['G','B','R','Bl','Br','M']
shoe_counts = [0,0,0,0,0,0]
shoes_in = input()

i = 0
while i < len(shoes_in):
    if shoes_in[i] == 'G':
        shoe_counts[0] +=1
    elif shoes_in[i] == 'R':
        shoe_counts[2] +=1
    elif shoes_in[i] == 'M':
        shoe_counts[5] +=1
    elif shoes_in[i] == 'B':
        if  i + 1 < len(shoes_in) :  # can look ahead safely
            if shoes_in[i+1] == 'l':
                shoe_counts[3] +=1
                i +=1
            elif shoes_in[i+1] == 'r':
                shoe_counts[4] +=1
                i+=1
            else:
                shoe_counts[1] +=1
        else: # last one is B
            shoe_counts[1] +=1
    i +=1

all_paired_up = True
for i in range(6):
    if shoe_counts[i] % 2 != 0:
        all_paired_up = False
        colour_of_odd = colours[i]
        print("A",colour_of_odd, "shoe has no partner.")

if all_paired_up:
    print("All paired up!")
```

Camping Trip

Subtask 1

In this subtask, $t_i \leq t_{i+1}$ for all i. This means that all of the good camping days form a continuous interval. Let G be the number of good camping days. Then the largest interval that has more good days than bad days will have G good days, and up to $G - 1$ bad days, which is a total of $2G - 1$ days. But there are a few edge cases we need to watch out for – if $G = 0$, then our answer is 0, and if $2G - 1 > N$, then our answer is N (since the trip cannot go over N days long).

```
1   N,A,B = map(int, input().split())
2   t = list(map(int, input().split()))
3   good = 0
4   for x in t:
5       if A <= x <= B:
6           good += 1
7   if good == 0:
8       print(0)
9   else:
10      print(min(N, good * 2 - 1))
```

Subtask 2

This subtask can be solved using a brute-force approach. For each possible starting day, and each possible ending day, we count the number of good and bad days within the interval to determine if the trip is valid. The time complexity of this solution is $O(N^3)$.

```
1  N,A,B = map(int, input().split())
2  t = list(map(int, input().split()))
3  best = 0
4  for start in range(N):
5      for end in range(start, N):
6          good = 0
7          bad = 0
8          for x in t[start:end+1]:
9              if A <= x <= B:
10                 good += 1
11             else:
12                 bad += 1
13         if good > bad:
14             best = max(best, end - start + 1)
15 print(best)
```

Subtask 3

This subtask is also solvable using brute-force, but we need a faster solution. Notice that the innermost for-loop in our subtask 2 solution is recomputing a lot of the same information. By eliminating this loop, we can improve the time complexity to $O(N^2)$

```
1  N,A,B = map(int, input().split())
2  t = list(map(int, input().split()))
3  best = 0
4  for start in range(N):
5      good = 0
6      bad = 0
7      for end in range(start, N):
8          if A <= t[end] <= B:
9              good += 1
10         else:
11             bad += 1
12         if good > bad:
13             best = max(best, end - start + 1)
14 print(best)
```

Subtask 4

Instead of counting good and bad days separately, let's assign a value of 1 for good days and -1 for bad days. Then we can determine if an an interval is valid by computing the sum of values within the interval – if the sum is greater than 0, then the interval contains more good days than bad days.

Let's consider the the length of the longest valid interval that ends on day i. Let t be the sum of values from day 1 to day i. If $t > 0$, then the answer is i. Otherwise, we need to subtract some values from the start of the interval to make t positive. Let $f(x)$ be the length of the smallest possible interval that starts on day 1 and has a total value of x. If $t \leq 0$, then to make t positive, we need to subtract a total value of $t - 1$. Thus, the answer must be $i - f(t - 1)$, assuming $f(t - 1) < i$, otherwise no valid trip exists ending on day i. We can record all possible values of $f(x)$ while considering all possible ending days in a single loop over the N days, so the time complexity of this solution is $O(N)$.

```
1   N,A,B = map(int, input().split())
2   t = list(map(int, input().split()))
3   total = 0
4   best = 0
5   f = {}
6   for i in range(1, N+1):
7       total += 1 if A <= t[i-1] <= B else -1
8       if total > 0:
9           best = max(best, i)
10      elif total - 1 in f:
11          best = max(best, i - f[total - 1])
12      if total not in f:
13          f[total] = i
14  print(best)
```

Holiday Shopping

The step towards solving this problem is assembling a representation of the graph / shopping mall. The representation we get in the input is simply a list of all of the edges in the graph. As you will soon see, this representation isn't very easy (or efficient) to work with. Instead, we create an adjacency list of the graph that we can use later on.

An adjacency list keeps track of all the vertices (nodes) which share an edge with each other (neighbours). We keep a list for each node, containing all the neighbours of the node. For instance, if the vertex/node numbered 0 has two neighbours, 1 and 2, the adjacency list should look like:

```
1  0: 1, 2
2  1: 0
3  2: 0
```

(Note that 0 is also listed as neighbours of 1 and 2, since we can travel both ways along the edges in this problem.)

To assemble the adjacency list, we need to create a 2-dimensional array in Python. Then, the first dimension of the array refers to each vertex, and the second dimension stores all of that vertex's neighbours. For example, the above adjacency list stored in Python should be:

```
1  [
2      [1, 2], # Index 0
3      [0],    # Index 1
4      [0]     # Index 2
5  ]
```

The code snippet below is one way to do construct an adjacency list in Python.

```
1  N, E = map(int, input().split())
2  adj_list = [[] for i in range(N)]
```

```
3  for i in range(E):
4      A, B = map(int, input().split())
5      adj_list[A].append(B)
6      adj_list[B].append(A)
```

(You may have stored your neighbours in a different way - if you are interested, look up adjacency matrices, and compare this strategy to an adjacency list. Which would be better for this problem, and why?)

Now that we've stored the graph, we can focus on completing the problem.

Subtask 1

The basic strategy we use to solve this problem is moving through the graph systematically, starting at the start vertex and finishing when we reach the exit vertex.

To solve the problem, we need to keep track of various things.

1. We need to know which vertices we have already visited, so we need a list `visited` which stores `True` if the vertex has been visited, `False` if not.

2. We need a list of vertices to explore next. We can delete these as we visit them, and add when we discover new neighbours. This going to be a first-in-first-out structure called a queue (in Python we can use a `deque`).

3. As we explore the graph, we need to know how far away from the start room we are. Since we are only interested in the distance to the exit, our solution below uses a queue maintaining `(room, distance)` tuples / pairs, but note that other implementations might store the distances in a separate list instead.

We can now traverse (explore) the graph using an algorithm called Breadth-First Search (BFS). There are some good resources available online if you are interested in learning more about BFS. The algorithm description is as follows:

1. Add starting vertex to the queue.

2. While the queue has vertices remaining in it:

   (a) Take the vertex and its current distance value from the top of the queue, and store it (make sure it is removed from the queue!)

   (b) If the vertex has already been visited, then continue back to 2.

   (c) Set the visited value of the vertex to True.

(d) For each of the vertex's neighbours, add the neighbour to the queue with the current distance increased by one.

The code for the full solution is shown below. Note that there are other algorithms you can use to solve this problem!

Python Subtask 1 Solution

```python
from collections import deque

N, E = map(int, input().split())
adj = [[] for i in range(N)]
for i in range(E):
    A, B = map(int, input().split())
    adj[A].append(B)
    adj[B].append(A)

S,M = map(int,input().split())  # Ignore these for this subtask as S = 0

q = deque()
q.append((0, 1))  # Start at node 0 with a 'distance' of 1 (as we need
    ↪   to count node 0 towards our visited count)
visited = [False] * N

while len(q) > 0:
    node, dist = q.popleft()
    if visited[node]:
        continue
    visited[node] = True

    if node == N - 1:
        # at this point, it is guaranteed that `dist`
        # is the shortest distance from `0` to `node`
        print(dist)
        break

    for neighbour in adj[node]:
        q.append((neighbour, dist+1))
else:
    print("SELF_ISOLATE")
```

C++ Subtask 1 Solution

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

using namespace std;

vector<int> adj[100001];
bool visited[100001];

int N, E, S, M;
int a, b, s;

void solve() {
    queue<pair<int, int>> q;
    q.push(make_pair(0, 1));

    while (!q.empty()) {
        auto cur = q.front();
        q.pop();

        int node = cur.first;
        int dist = cur.second;

        if (visited[node]) {
            continue;
        }
        visited[node] = true;

        if (node == N-1) {
            cout<<dist<<endl;
            return;
        }

        for (auto neighbour : adj[cur.first]) {
            q.push(make_pair(neighbour, cur.second + 1));
        }
    }

    cout<<"SELF_ISOLATE"<<endl;
}
```

```
42
43  int main() {
44      cin>>N>>E;
45      for (int i=0; i<E; i++) {
46          cin>>a>>b;
47          adj[a].push_back(b);
48          adj[b].push_back(a);
49      }
50
51      cin>>S>>M;
52
53      solve();
54  }
```

Subtask 2

For this subtask M is always either 1 or 0. If M is 1, then we have to avoid visiting any node that's currently occupied. If M is 0, then as we're allowed to be within 0 edges of any node, we can actually still visit any node we want.

How do we ensure we don't visited any node that's currently occupied? Recall that for Subtask 1 we maintained a list to keep track of all the nodes we've already visited. In our BFS algorithm, we avoid visiting any node that's already been visited. That means that, if M = 1, we can just set all of the occupied nodes as 'visited' before starting our BFS, and the BFS will avoid those nodes! That's all we need to complete Subtask 2.

Python Subtask 2 Solution

```
1   from collections import deque
2
3   N, E = map(int, input().split())
4   adj = [[] for i in range(N)]
5   for i in range(E):
6       A, B = map(int, input().split())
7       adj[A].append(B)
8       adj[B].append(A)
9
10  S,M = map(int,input().split())
11
12  visited = [False] * N
13  for x in range(S):
```

```python
14          s = int(input())
15          if M == 1:
16              visited[s] = True
17
18  q = deque()
19  q.append((0, 1))  # Start at node 0 with a 'distance' of 1 (as we need
    ↪  to count node 0 towards our visited count)
20
21  while len(q) > 0:
22      node, dist = q.popleft()
23      if visited[node]:
24          continue
25      visited[node] = True
26
27      if node == N - 1:
28          # at this point, it is guaranteed that `dist`
29          # is the shortest distance from `0` to `node`
30          print(dist)
31          break
32
33      for neighbour in adj[node]:
34          q.append((neighbour, dist+1))
35  else:
36      print("SELF_ISOLATE")
```

C++ Subtask 2 Solution

```cpp
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <algorithm>
5
6  using namespace std;
7
8  vector<int> adj[100001];
9  bool visited[100001];
10
11  int N, E, S, M;
12  int a, b, s;
13
14  void solve() {
15      queue<pair<int, int>> q;
16      q.push(make_pair(0, 1));
```

```
17
18      while (!q.empty()) {
19          auto cur = q.front();
20          q.pop();
21
22          int node = cur.first;
23          int dist = cur.second;
24
25          if (visited[node]) {
26              continue;
27          }
28          visited[node] = true;
29
30          if (node == N-1) {
31              cout<<dist<<endl;
32              return;
33          }
34
35          for (auto neighbour : adj[cur.first]) {
36              q.push(make_pair(neighbour, cur.second + 1));
37          }
38      }
39
40      cout<<"SELF_ISOLATE"<<endl;
41  }
42
43  int main() {
44      cin>>N>>E;
45      for (int i=0; i<E; i++) {
46          cin>>a>>b;
47          adj[a].push_back(b);
48          adj[b].push_back(a);
49      }
50
51      cin>>S>>M;
52
53      for (int i=0; i<S; i++) {
54          cin>>s;
55          if (M > 0) {
56              visited[s] = true;
57          }
58      }
59
```

```
60    solve();
61  }
```

Subtask 3

For the full solution, we have to avoid going within M-1 edges of any occupied node. We could first compute all of the nodes that are within M-1 edges of any occupied nodes, set all of those as visited, and then use our BFS as before. But how do we know which nodes are within M-1 edges of an occupied node? We could try to do an individual BFS from every occupied node, marking every node we visit as occupied, and stopping our BFS when we go past M-1 edges. In the worst case, M will be large and so each BFS will be $O(E)$. Because we do this for every shopper, the total complexity will be $O(E * S)$ - if both E and S are roughly 100 000, then this will be too slow.

You might be tempted to share the 'visited' list between the BFS calls for each shopper. This will ensure you visit each node at most once, as a node marked as visited in the BFS for one shopper won't be visited in any later BFS calls from other shoppers. However, this trick has a problem. Consider the test case below:

```
1   13 14
2   0 1
3   1 2
4   2 3
5   3 4
6   4 5
7   1 6
8   6 7
9   7 8
10  8 9
11  9 10
12  5 10
13  10 11
14  6 11
15  11 12
16  2 3
17  4
18  9
```

The graph in the test case can be visualised below:
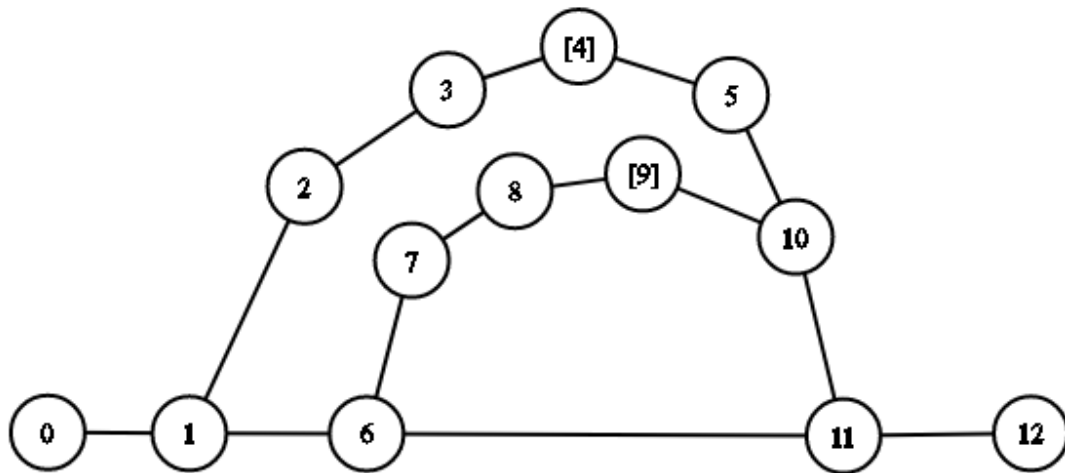
Nodes 4 and 9 are occupied by shoppers. If we BFS from node 4 first, then we will mark nodes 2, 3, 4, 5 and 10 as visited. Then, when we BFS from node 9, we will mark nodes 7, 8 and 9 as visited. We won't visit node 10 as it's already been visited. However, this prevents us from marking node 11 as visited, even though it's only 2 edges away from node 9! So even though it's actually impossible in this case to get to the end node (because we can't go through node 11), our proposed solution will output an incorrect answer of 5.

(You might notice that we would have gotten the correct answer had we done our first BFS from node 9 and then 4. However, because the numbering of the nodes is arbitrary, had nodes 4 and 9 been swapped around then you would still have the same problem.)

So how do we solve that problem in a way that's fast enough to pass? We can think of BFS as maintaining a 'frontier' of nodes that expands outwards as we keep traversing the graph. Each node in the 'frontier' (i.e. the queue) will be the same distance (plus or minus one) as every other node in the frontier. When we start, the 'frontier' is just the starting node. But what if we started out with a frontier containing *all* of the occupied nodes? In that case, the frontier would expand outwards from all of the starting nodes. We would effectively be performing S BFS traversals *in parallel*! But because it's still a single BFS, it would complete in $O(E)$ time! To achieve this, all we have to do is to modify our original BFS to start with a queue containing every shopper. Then, we do our BFS as normal, stopping whenever we go past $M - 1$ nodes. Finally, we still have to do one last BFS from node 0 to find out the actual distance.

Python Subtask 3 Solution

```python
from collections import deque

N, E = map(int, input().split())
adj = [[] for i in range(N)]
for i in range(E):
    A, B = map(int, input().split())
    adj[A].append(B)
    adj[B].append(A)

S,M = map(int,input().split())

visited = [False] * N
q = deque()
for x in range(S):
    s = int(input())
    q.append((s, 0))

# Find all of the nodes we aren't allowed to visit
while len(q) > 0:
    node, dist = q.popleft()

    if dist == M:
        continue

    if visited[node]:
        continue
    visited[node] = True

    for neighbour in adj[node]:
        q.append((neighbour, dist+1))

# Find the distance between node 0 and N-1
q.append((0, 1))
while len(q) > 0:
    node, dist = q.popleft()
    if visited[node]:
        continue
    visited[node] = True

    if node == N - 1:
        # at this point, it is guaranteed that `dist`
```

```
42              # is the shortest distance from `0` to `node`
43              print(dist)
44              break
45
46          for neighbour in adj[node]:
47              q.append((neighbour, dist+1))
48      else:
49          print("SELF_ISOLATE")
```

C++ Subtask 3 Solution

```
1   #include <iostream>
2   #include <vector>
3   #include <queue>
4   #include <algorithm>
5
6   using namespace std;
7
8   vector<int> adj[100001];
9   bool visited[100001];
10
11  int N, E, S, M;
12  int a, b, s;
13
14  void precomp(vector<int> people) {
15      queue<pair<int, int>> q;
16
17      for (auto person : people) {
18          q.push(make_pair(person, 0));
19      }
20
21      while (!q.empty()) {
22          auto cur = q.front();
23          q.pop();
24          if (cur.second == M) {
25              continue;
26          }
27          visited[cur.first] = true;
28
29          for (auto neighbour : adj[cur.first]) {
30              if (!visited[neighbour]) {
31                  q.push(make_pair(neighbour, cur.second + 1));
32              }
```

```
33              }
34          }
35  }
36
37  void comp() {
38      queue<pair<int, int>> q;
39      q.push(make_pair(0, 1));
40
41      while (!q.empty()) {
42          auto cur = q.front();
43          q.pop();
44          if (cur.first == N-1) {
45              cout<<cur.second<<endl;
46              return;
47          }
48          visited[cur.first] = true;
49
50          for (auto neighbour : adj[cur.first]) {
51              if (!visited[neighbour]) {
52                  q.push(make_pair(neighbour, cur.second + 1));
53              }
54          }
55      }
56
57      cout<<"SELF_ISOLATE"<<endl;
58  }
59
60  int main() {
61      cin>>N>>E;
62      for (int i=0; i<E; i++) {
63          cin>>a>>b;
64          adj[a].push_back(b);
65          adj[b].push_back(a);
66      }
67
68      vector<int> people;
69      cin>>S>>M;
70      for (int i=0; i<S; i++) {
71          cin>>s;
72          people.push_back(s);
73      }
74
75      precomp(people);
```

```
76      comp();
77  }
```

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.