2019 SBU ACM Programming Selection Contest

(Stony Brook University, Sep 19, 2019)

1 Fridge Magnets

If we do not have u and n as constraints, this is an alphabet counting problem and the answer is just to count the alphabet which appeared most in the input. But since u and n are interchangeable, let's say `cnt` is the max count of alphabets apart from u's and n's. Therefore, the answer is $\max(\texttt{cnt}, \lceil \frac{u+n}{2} \rceil)$.

2 Xordan and Πper

The answer is $\sqrt{N}$, where N is the length of the string. The initial string does not matter for the answer. The first approach is to simply simulate the XORs with binary arrays in a loop. The second approach is to realize that the only bits being flipped are the positions with a square number as an index. The reason being that each index is being XORed with a 1 (XORs with zeroes do not change the value) equal to the number of factors of that index. Since square numbers have an odd number of factors, they will be the only ones flipped in the end.

3 Jericho Turnpike

In this problem we are supposed to find a window of minimum length that contains at least one branch of all of favorite stores. Since the number of different types of stores are at max 100 we can store indexes of each type of store. We can have something like vector<int>storeIndexes[100]. Once we have the indices mapped to the correct store, we sort each of the vectors. Then, we create a min-heap of size K and store first index from each store. We calculate the min and max distance. After this we simply pop the smallest index and add the next smallest index for the popped store. We keep updating the min and max value and keep track of the min-window. One can find the working code here `http://ideone.com/XDKo5k`.

An alternate approach that avoids the use of a min-heap is to use an array to store the frequencies of each type of store while maintaining a sliding window. We initialize an interval starting from the first store and iterate forwards until we have seen at least one of each store. We then increment the starting point until we no longer contain at least one of each store, and then increment the ending point until we do. We keep iterating this process, keeping track of the minimum interval at all times, until we reach the end.

Conceptually, it is simple, but implementing this idea in a brute-force manner will lead to a timeout. It is very important that checking whether each store has been seen at least once and checking whether we no longer have at least one of each store be done in constant time. A naive loop through the frequency array at every point will be too slow. To check if we have seen every point at least once, we keep a counter variable that increments by 1 every time we see a new store. Once we reach k, the number of stores, we know we have seen everything. To check if we no longer have at least one of every point while increasing the starting point, we know this occurs every time a frequency for any store reaches zero. By performing the checks in this way, we get a linear time solution in the number of stores.

4 Dish Jenga

This can be done with a greedy approach. We first note that given N dishes, the minimum number of dishes that need to be moved is at most $N-1$ (i.e. when the dishes are stacked from smallest to largest). We see that we do not need to care about the order in which the dishes are removed and each dish need only be moved at most once.

To have some intuition for the solution, let us first consider the largest and second largest dishes. We can see that every dish underneath the largest must be moved; otherwise, the largest will never reach the bottom of the stack. If the second largest dish was below the largest, we see that everything except the largest must be moved! This is so because in order for the second largest dish to be above the largest, everything above must be cleared away first. However, if the second largest dish was above the largest, we would only need to move the dishes between their two locations. This is enough to ensure the second largest dish will come directly after the largest. This idea can be extended to all the dishes in decreasing order.

In the end, we simply have the count the number of dishes between each consecutive dish in decreasing order. Thus, we need to first (reverse) sort and sum up all the dishes between each consecutive element in the sorted stack as found in the original stack. Since the solution is short, a reference is provided in Python 3 for clarity.

```python
N = int(input())
arr = [int(i) for i in input().split()]
sorted_arr = sorted(a, reverse=True)
ans = 0
curr = 0
for i in range(N):
    if arr[i] != sorted_arr[curr]:
        ans += 1
    else:
        curr += 1
print(ans)
```

5 Not all who wander, wander away from ice-cream

One interesting observation of this dynamic problem is that we don't have to account for negative indices and East direction is symmetric to West and North is to South. All it maters is that how far are you in terms of Manhattan distance from the initial location (0,0). Another interesting observation is that by changing a North to a South reduces the travel distance by 2 units (i.e: changing a North to South is equals to North $\rightarrow$ South $\rightarrow$ South). Therefore, depending on the distance you are from the origin, you'd need $\frac{(abs(x)+abs(y))}{2}$ modifications. Note that, if you are at odd length distance, there is no way you can reach to the origin since you are not allowed to insert/delete any character.

```
int rec(int idx, int rem){
  if(idx == n) return 0;
  int &ret = dp[idx][rem];
  if(ret != -1) return ret;
  ret=0;
  int x=0, y=0;
  for(int i=idx; i<n; i++){
    if(s[i] == 'N') ++y;
    if(s[i] == 'S') --y;
    if(s[i] == 'E') ++x;
    if(s[i] == 'W') --x;
    if((abs(x) + abs(y)) & 1) continue;
    if((abs(x) + abs(y)) / 2 > rem) continue;
    ret = max(ret, 1 + solve(i+1, rem - (abs(x) + abs(y))/2));
  }
  return ret;
}
```

6 Taggart Transcontinental

This problem has two major algorithmic facets: binary search and computing a minimal spanning tree (MST) for a given graph. If you (properly) understand what these mean, then with the least amount of mental manipulation, the solution to this question becomes apparent. In this case, the explanation is done. Thus, in what follows, what I do is explain these concepts by the way of the example of this problem.

So, you're still Dagny Taggart, and in some configuration of input, as in the problem, suppose that you've solved the problem, and the output is not *IMPOSSIBLE*. What does this mean?

Well, what you've done is gone and chosen a collection of railroad segments to use. Let's denote the set of segments that you've chosen by $S = \{e_i : i \in I\}$ for I some index set (the meaning of this sentence is solely the introduction of notation). Let's say that rail segment e_i has a maintenance cost of c_i and supports speeds up to s_i.

For this collection of segments, the maximal speed that they all support is given by the expression $\min \{s_i : i \in I\}$. Notice that had you excluded some segments from you collection

S, then the value of this expression would not be lower than the value of this expression involving all edges.

This collection must satisfy the property that every city is connected to any other city through some sequence of segments. With this restriction in mind, suppose that it's possible to remove a segment from S while preserving this property. Since the cost of every edge is positive, removing this edge will result in a cheaper but not slower (in fact potentially faster) network. Thus let's say that you modify your solution by discarding away edges that you're allowed to discard until there are no more edges that you're allowed to discard.

The structure of your collection S is now very special. In graph theoretic terms, what you have is a connected tree. This is defined by the property that any two cities are connected by one, and one only, sequence of rail segments that doesn't involve backtracking.

Ok, so what does this tell us about how to solve the problem?

Well, first let's consider a simpler problem in which you just don't care about all of this maxmin speed nonsense, that is, you just want to connect the cities in a cheapest way possible. This problem is called the **minimal spanning tree problem**. The way that you solve this simpler problem turns out to be the following.

Imagine that one of the cities is your central station. Well, it's inexpensive to just operate a service in which you don't send any trains whatsoever. Suppose you do this. What will happen is that your brother will come and yell at you for not providing any rail service to some cities, provided that there is more than one city. Say that you ask your brother to make his complaint explicit. Say that you say to him that you want a list of cities that you are not servicing. Then say that he comes back with a list. How are you to correct this issue? Well, by adding more segments. In the first instance, add a cheapest segment that helps this problem in the sense that it creates some way of traveling to a city on this list to the central station (in later instances, not necessarily a direct one).

Now, because you want to annoy your brother, say you say to him that you've solved the problem. If he still wants to complain, then the above process repeats, and repeats, until there's left nothing about which he can complain.

It turns out that this process will actually yield a network of minimal cost! The process which which I've just described is called **Prim's algorithm**.

Now, let's begin returning to the general problem in which you do care about speeds, but by first going through one special intermediate case. In this case, we'll make the problem that of just finding any network that satisfies the following three constraints: the cities are connected, the cost is not greater than our budget, and the minimal speed is not less than some fixed speed constraint.

In staring to solve this problem, we can obviously begin by excluding from our consideration any edges that are too slow for our needs, since including them would violate the third constraint. Subsequent to this, any remaining edge is fair game. Now we run Prim's algorithm, which will either be unable to connect all the cities, or will return with a network of minimal cost. Thus a network satisfying all the constraints exist if and only if Prim's algorithm connects all the cities and the total cost is not greater than out budget. This solves this second simpler problem.

Now finally, for the solution to the question as posed, one thing that we could do is start with speed zero as the constraint in the above problem and try to solve it. If we can't, output *IMPOSSIBLE*. Otherwise, try to solve the problem with the speed constraint being

1. If this works, try 2, and so on. This process eventually stops since the input obviously can't support arbitrarily large speeds, provided that there is more than one city. Thus you find the largest speed.

Now this is a bit inefficient - trying speeds one by one. So what you do is you search for the speed point at which this special problem stops having a solution. If you know that this point necessarily exists between two numbers, then try solving the sub-problem for the integer closest to the middle of that range; the result of this test will tell you that you can confine your search to a range that's (to rounding) half the length of the original range. This process is called **binary search**. It terminates when your range fully hones in on a single number.

The problem setter's solution to this problem is thus to run binary search in this way starting from the interval 0 to one greater than the upper bound for speeds given in the problem. I output *IMPOSSIBLE* when the search hones in on the number 0, since all s_i are greater than zero, so 0 can't be this point, so the search doing this proves that the hypothesis that (the point at which the sub-problems stop having a solution is in my starting interval) is false. Conversely, if the process hones in another number, then that implies that the sub-problem had an affirmative solution for some points in the interval, thus the turning point is in the interval, by the construction of my bounds, thus the number to which the honing happened is the solution to the problem as posed.

7 Convergence

The essence of the code is the Floyd–Warshall algorithm for finding shortest paths $w[i, j]$ between each pair of (i, j). For each enumerated k, the algorithm tries to use the shortest path between (i, k) and (k, j) to update the shortest path between (i, j). Thus, $p[i, j]$ is the largest index in the shortest path between i and j, except i and j.

We can use Dijkstra's algorithm to compute the shortest paths and trace the largest indices. The special cases like multiple shortest paths, unconnected graph and originally shortest paths may be tricky in coding.