Task 100 – Mysterious Letter

This is a very simple implementation task. To solve it we need to do exactly what the problem instructs us to – we need to count the number of pairs of same letters that contain exactly ***N*** letters between them. There are numerous ways to implement this, so we will not stick to a particular solution and instead cover some tips and common pitfalls:

- While iterating through the input string, we must be careful not to go out of bounds while looking for a suitable pair
- Since it is stated in the task that the pairs ***(x, y)*** and ***(y, x)*** are considered to be same, we can simply look through the letters that are in front of our current index and completely disregard the letters we have already passed
- Ba careful to also cover the case of ***N*** being larger than the actual length of the array

*The time complexity of the solution is **O(n)**.*

Task 200 – Endless Run

This task relies on implementing a deterministic finite automaton, with rules for its state changes being the mappings from tuples to numbers depicted in the problem description. Similar to the previous task, there are many ways to implement this so we will once again stick to a couple of pointers that might be of interest:

- We will begin with calculating the first row of the grid using the initial state as a base
- Be careful with implementing the edge cases correctly – improper handing might cause *array index out of bounds* type of errors to occur
- To avoid covering the edge cases with additional conditionals, one possible way to not encounter them whatsoever is to increase the number of columns of the matrix representing the grid from the task by ***2***. These will be the two outer columns and will be prefilled with ***1s*** so that afterwards we can simply apply the rules starting from index ***1*** to index ***n***.

The time complexity of the solution is ***O(n · d).***

Task 300 – The Perfect Build

As we gather from the problem description, we are required to find the combination of weapons bought with our initial gold so that it provides us with the highest amount of power with respect to the type of hero we get.

Intuitively, we might consider generating all of the possible combinations (using DFS for example), consider the ones that fit the gold restrictions and choose the one which provides a highest amount of power. However this is not an option due to the fact that the number of items can go up to ***50***, meaning there can be 2^{50} combinations in the worst case, which is way above the time complexity restriction of the problem.

Instead, we can go for a more convenient approach and deduce that the problem of finding the best possible combination for our initial sum of gold can be split into several subproblems:

Let ***f(x)*** represent the maximum amount of power we can attain for ***x*** gold, given the current items and ***p(x)*** represent the power the tuple of attributes ***x*** brings given our hero's class type (meaning ***p(x)*** changes based on what our hero's main attribute is as explained in the problem).

It is obvious that

$$f(n) = max(f(n - c_1) + p(a_1), f(n - c_2) + p(a_2), \dots , f(n - c_i) + p(a_i))$$

where c_i represents the cost of item ***i*** and a_i represents a tuple of the attributes item ***i*** has.

Since the same can be done for all of the subproblems, we can continue in this fashion until we run out of gold. At this point we might notice that there are a lot of the subprolems that keep repeating themselves (basically lower values of gold that are needed to calculate higher amounts of gold), so it should become obvious that the optimal solution involves dynamic programming. In particular, this is a very textbook version of the Knapsack Problem . In our case, the knapsack and the amount of weight it can hold is represented by the gold we have, the

weight of the items is represented by their cost and the value of each item is represented by ***p(x)***.

Considering this has now turned into a very straightforward dynamic programming problem, we can compute the answer easily. For convenience, we may also put the items and their attribute values in a separate data structure and instead of calculating how much power each item gives, we can add its attributes to our current ones and then apply ***p(x)*** on the attribute sums.

*The time complexity of the solution is **O(N · G)**, where **N** represents the number of items and **G** represents the initial amount of gold we get.*

Task 400 – Let There Be Light

The fourth task for this round is a difficult and complicated version of a known problem – maze traversal. Our maze is **3D** and contains switches which are connected to each other – flipping a switch triggers the flip of the one that follows it as well.

Since we are required to find the shortest number of steps to turn all of the switches on, it is obvious that for traversing through the maze we will use BFS.

To begin with, let us define all of the states we will need before we begin implementing our algorithm:

- **x – coordinate** of our current position (on the current floor we are on)
- **y – coordinate** of our current position (on the current floor we are on)
- the floor we are currently on
- the state of each switch (***1*** for "on" and ***0*** for "off")

This means that for this solution we will use a 4-state **BFS**. Here are some additional technical details about the movement and actions we can do in the **3D** plane:

- Moving left/right or up/down is handled by adjusting the ***x*** and ***y*** coordinates (***x*** for left/right and ***y*** for up/down)
- Moving up/down a floor is handled by the third state
- To move vertically we exclusively need to be on a ***'U'*** or ***'D'*** character on the current floor we are on
- Stepping on an ***'U'*** or ***'D'*** character does not mean we need to move vertically – we also need to take the other viable neighbours into consideration

With this, we have enclosed our movement actions completely and we can move anywhere in the house safely as long as we do not hit a wall or go out of walls of the house (we need to bear these cases in mind).

The only thing left to do is figure out an efficient way to handle the states of the switches. Since there are at most ***10*** light switches, it is obvious that we have 2^{10}

combinations when it comes to the fourth state. Since running **BFS** every time we switch a light may result in a lot of iterations of the traversal, that is obviously completely out of the question. Instead, we need a structure that allows us to quickly switch between states, while retaining our current moves as well as the other states (our horizontal and vertical position in the house).

For this purpose, we can use ***bitmasks***. I will explain the usage here without going into too much detail, since this topic in itself is worth a separate post.

Shortly, as we mentioned before we can label the light switches that are "on" as ***1*** and the ones switched "off" as ***0***. This means that we can represent them as a binary string. For example, having ***4*** lights out of which the first and last ones are switched on would render the string ***"1001"***. Now instead of having each state for the switches have its own string hash (which would increase the complexity by a lot when trying to compare states) we may notice that each of these strings can be represented as a unique integer. The example mentioned above would be the integer ***9***, only represented binary. We will call this integer a **bitmask**.

This means that if we can change bits in the **bitmask** really quickly, we may save up a lot of time of computation. One such quick operation is using logical **XOR** combined with **bitwise shifting**.

Namely, the operation ***1 << x*** means that we are shifting the number ***1 x*** times, which in other words means we are calculating 2^x. Now, wanting to invert a certain bit in the **bitmask** would basically mean doing:

bitmask ^= 1 << n

which is the main formula used with **bitmasks**.

What this does is it shifts the ***1*** to our wanted position and then performs a logical **XOR** operation. Since all of the other bits in the right hand side of the equation are zeros, the only inverted bit would be the n_{th} one, which is exactly what we need.

So to sum up, every time we watch to flip a switch we will perform the **bitmask** operation for ***i*** and ***i + 1***, where i is the current switch we are examining (of course taking into account if ***i + 1*** actually exists or not).

Finally, the terminating states of the search are the following:

- The queue is empty, meaning a solution does not exist
- We have achieved a **bitmask** that contains no zeros in its binary representation

The second terminating state can be either checked by converting the **bitmask** to binary and comparing it, or by simply comparing it with $2^k - 1$ (or $(1 << k) - 1$ for short) where ***k*** is the total number of switches in the house.

The time complexity of the solution is $O(L \cdot W \cdot F \cdot 2^S)$*, where* ***L*** *represents the length of the house,* ***W*** *represents the width of the house,* ***F*** *represents the number of floors in the house and* ***S*** *represents the number of switches in the house.*

Task 500 – Stairs to Razor Tower

At first sight, this seems like a fairly easy combinatorics problem, however because of the very high constraints of ***n*** it becomes dramatically more difficult.

If ***f(n)*** represents the solution for a given ***n***, then we immediately notice an interesting recurrence relation that states:

$$f(n) = f(n-1) + f(n-3)$$

This is in fact a very famous sequence called Narayana's cows sequence. With regards to it, we will discuss two solutions and their limitations.

Solution 1:

I want to shortly summarize why a combinatorics approach will not work for a very large ***n*** since a lot of the competitors tried this approach, resulting in ***0*** points.

Basically, the solution to the problem would be to find all unique pairs ***(a, b)*** such that

$$a + 3b = 100$$

where ***a*** is the number of ***1*** steps you do and b is the number of ***3*** steps you do.

Notice how for each pair there are multiple ways you can shuffle the order of the steps so you get more valid combinations. For example, if ***n = 4*** and we consider the pair ***(1, 3)*** (which is a valid one), we can either do a ***1*** step and then a ***3*** step or we can do a ***3*** step and then a ***1*** step.

If we represent a certain pair through ***b***, we would get that the pair equals ***(n – 3b, b)*** and the number of ways to get ***n – 3b 1*** steps and ***b 3*** steps is:

$$\frac{(n-2b)!}{b!(n-3b)!}$$

Since we can have a maximum of $\left\lfloor \frac{N}{3} \right\rfloor$ ***3*** steps (otherwise we would go over ***N***) the total number of ways to get to ***N*** would be:

$$\sum_{0}^{\lfloor N/3 \rfloor} \frac{(n-2b)!}{b!(n-3b)!}$$

This agrees with **Narayana's cows sequence** as well.

However, this approach is very computation heavy since the large constraints will cause multiplications of enormously large numbers. Even if we do optimize as much as we can by using tree factorials and by switching the equation around a bit we will get nowhere near the desired time complexity for the large inputs.

<u>Solution 2:</u>

We have now realized that we need a solution that is much more optimized and much less computation heavy.

First of all, let's notice how we can turn the recurrence relation into a matrix equation.

Let us denote $a_n = f(n)$ (the solution for our problem for ***n*** or the n_{th} term in **Narayana's cows sequence**). We are trying to find a matrix ***A***, such that the following figure holds:

1) $$\begin{bmatrix} a_{n-1} \\ a_n \\ a_{n+1} \end{bmatrix} \quad A \; = \; \begin{bmatrix} a_n \\ a_{n+1} \\ a_{n+2} \end{bmatrix}$$

The point of this equation is find the value of the next a_{n+2} from the left-hand side vector by multiplying it with some constant matrix.

Since it is obvious from the formula that we simply need to move the values of a_n and a_{n+1} up by one position, the first two columns of ***A*** will be ***(0, 1, 0)*** and

(0, 0, 1) (since the first element of the right-hand side vector is the second element in the left-hand side vector and the second one in the right side vector is the third one in the left-hand side one, hence the zeros in all other positions).
For calculating the value of the third column, we can use our previous recurrence formula, so since $a_{n+2} = a_{n-1} + a_{n+1}$ the third column will be ***(1, 0, 1)***.

Thus, the value of the matrix is

$$A = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}.$$

We now know that for matrix ***A***, figure ***1)*** holds.

Lemma 1. For an arbitrary number $n \in \mathbb{N}$, the following figure holds:

2) $$\begin{bmatrix} a_n \\ a_{n+1} \\ a_{n+2} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} A^n$$

Proof:

We can prove this by simple mathematical induction.

Let us note that vector (a_0, a_1, a_2) equals ***(1, 1, 1)***.

1. Let us see if the statement holds for ***n = 1***.

We then have:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} A = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} A = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

We can calculate that $a_{n+2} = 2$ by hand, so we can confirm that indeed $(a_n, a_{n+1}, a_{n+2}) = (1, 1, 2)$ so the figure holds for this step of the induction.

2. Let us assume that for $n = k$

$$\begin{bmatrix} a_k \\ a_{k+1} \\ a_{k+2} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} A^k$$

is valid.

This will serve as our ***induction hypothesis***.

3. If figure ***2)*** is true for $n=k$, let us show that it is also true for $n=k+1$:

$$\begin{bmatrix} a_{k+1} \\ a_{k+2} \\ a_{k+3} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} A^{k+1} \Leftrightarrow \begin{bmatrix} a_{k+1} \\ a_{k+2} \\ a_{k+3} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} A^k A$$

If we plug in our induction hypothesis in this equation we will get that

$$\begin{bmatrix} a_{k+1} \\ a_{k+2} \\ a_{k+3} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} A^k A \Leftrightarrow \begin{bmatrix} a_{k+1} \\ a_{k+2} \\ a_{k+3} \end{bmatrix} = \begin{bmatrix} a_k \\ a_{k+1} \\ a_{k+2} \end{bmatrix} A$$

We know that this is true from figure ***1)***, so we know that figure ***2)*** is true for $n=k+1$ as well. With this, we can conclude our mathematical induction and have successfully proved that figure ***2)*** indeed holds.

Considering we already know that $a_0=a_1=a_2=1$, we can derive from ***Lemma 1*** that to find a_n all we need to do is find A^n.

At this point we can use the exponentiation by squaring method and get the desired matrix in very low time complexity (making sure we use proper modular arithmetic along the way to avoid overflow).

Now since we can get the value of the left-hand vector we can easily see what the value of a_n is, which is the solution to our problem.

*The time complexity of the solution is **O(logn)**.*