Problem A. Is it a pattern?

Simple explanation: This problem is a simple problem on recursion. We divide our string into 2 equal parts at each step and check if they are palindromes, then we repeat the same on the 2 subproblems (i.e the 2 substrings). If our condition gets violated at any stage we return false. Finally we stop at base case i.e length 0 or 1. and return true.

Proof of correctness:

Induction hypothesis:

P(n) : The string s is a palindrome, and the string formed by the first $\left\lfloor \frac{|s|}{2} \right\rfloor$ and the last $\left\lfloor \frac{|s|}{2} \right\rfloor$ characters of s are both $pattern - j$.

Base case: P(0) is true (by definition)

Inductive step: We will use the strong form of induction

Assume P(1),P(2),...P(n) is true.

We have to show this implies P(n+1) is true

By Induction hypothesis, we have the strings formed by the first $\left\lfloor \frac{|s|}{2} \right\rfloor$(P(n/2)) and the last $\left\lfloor \frac{|s|}{2} \right\rfloor$(P(n/2)) characters of s both $pattern - j$ and also they are reverse of each other by definition.

Thus P(n+1) is true. Thus proved. q.e.d.

Analysis of time complexity:

Draw the recursion tree. By standard divide and conquer algorithm this tree has log(n) levels. And since we check if strings are palindromes or not at each level, we do O(n) work at each level, thus thus total time complexity is O(nlogn).

In some cases, while checking if strings are palindromes or not, one may pass the entire string instead of passing a reference to that string, and since strings are data structures passed by value thus one ends up having a time complexity of O(n^2)(that will give you partial points)

Problem B. Game Over

Naive Approach $O(N^2)$ **(50 points)** : Traversing through the given array, we simulate the game starting from every position. We initialize an array **visited** at every position with all zeroes initially and keep marking the positions visited in the simulation with a non zero number. If we reach a position marked with a non zero number in the visited array and hence previously visited in the game simulation, it implies the game simulation has a loop and the starting position is **not good** or **bad**. If the game simulation leads to an end, it is a **good** starting position. We can count the good starting positions and store them in another array to print the answer.

Optimized Approach $O(N)$ **(100 points)** : We consider a starting position **good** when the game simulation starting from that position comes to an end. It is observed that if a starting position is good, then every position visited in the game simulation starting from this position is also a good starting position. A starting position is considered **not good** or **bad** if the game simulation forms a loop and never comes to an end. It can be observed that if a starting position is bad, then every position visited in the game simulation is also a bad starting position.

We will use another array **visited** of the same size as the given array to mark the unvisited positions as 0, good starting positions as 1 and bad starting positions as -1. We start by looping through the given array and simulating the game starting from unvisited positions only. As the game simulation proceeds, every unvisited position visited is marked -1. If we reach a position previously visited and marked -1, it is either a newly discovered loop or a bad starting position identified in previous game simulations. Hence, we break out of the simulation and leave the positions visited in this simulation as -1. If we reach a position previously visited and marked 1 (i.e., a good starting position), it is assured that the game will come to an end on proceeding from here. Hence, we break out of the simulation and mark all positions newly visited in this game simulation as 1. If no such positions are visited, the game will eventually end since the number of visitable positions are finite and we mark all the positions visited in this simulation 1 in the visited array.

Proceeding this way, we will have identified all the positions in the given array as good or bad and marked them on the visited array. Iterating through the visited array and counting and printing all the positions marked with 1 will list all good starting positions.

Problem C. Poor Man's Chess

For $N = 2$,it is trivial to see that Alice can place the rook just above or just to the left of the bottom-rightmost square forcing Bob to land on it. For$N \geq 3$, if Alice places the rook at the second last row or last row or the second last column or the last column then she loses as then Bob clearly has the ability to place the rook just above or just to the left of the losing square. Hence Alice loses the game if she plays any of the above-mentioned 4 moves. For $N = 3$, she has no choice but to play either of the 4 moves, hence she loses for $N = 3$. Now for $N \geq 3$, we prove by induction that Alice always loses.

Proposition: For $N \geq 3$, Alice always loses.

Base case: For $N = 3$ proved above.

Induction step: We show that if for any k ($k \geq 3$) proposition holds, it holds for $k + 1$ as well. If Alice sends the rook to the above-mentioned 4 positions she loses. Else if Alice moves the rook to the p^{th} column or row$(2 < p < k + 1)$, Bob can move the rook to p^{th} row or column respectively causing the game to become that a $p \times p$ game in which Alice begins. And from induction hypotheses, we know that Alice loses this game. Hence by induction, we have proved that Alice loses this game.

Bonus Question: Try the above game with $M \times N$ dimensions of grid with $max(M, N) > 1$

Problem D. Ehab Likes to Travel

Let's consider a path a_1, a_2, ..., a_{k-1}, a_k.$(k > 1)$

It can be observed that $T_{a_k} = T_{a_{k-1}} +$ Length of road connecting a_{k-1} and a_k (For $k > 1$) and $T_1 = 0$.

Define $Mincost_i$ as the minimum travelling cost for a path starting at node i and terminating at some leaf.

Consider the folowing cases for a_k -

1. a_k is a leaf: In this case $Mincost_{a_k} = T_{a_k}$.

2. a_k is not a leaf: Let a_k be connected to nodes b_1, b_2, ..., b_j and a_{k-1}.

$Mincost_{a_k} = min(Mincost_{b_r} \text{ for } 1 \leq r \leq j) + T_{a_k}$

We can use Depth First Search to calculate T_i and $Mincost_i$ for all nodes.

The answer is equal to $Mincost_1$.

Time Compexity: $O(n/nlogn)$ depending upon the data structure used for storing length of roads.

Note: If memoization is not used for Tax values,we can calculate it as-

$T_{a_k} = \sum_{i=1}^{k-1}$ Length of road connecting a_i and a_{i+1} (For $k > 1$) and $T_1 = 0$.

In this case Time Compexity: $O(n^2)$ which can give partial points for this problem.

Problem E. Valuable Strings

Naive Approach $O(N^3)$: Iterate over all possible substrings and check if the substring has a vowel or not. There are $O(N^2)$ substrings and it takes $O(N)$ time to iterate each substring hence the complexity becomes cubic

Optimised Naive Approach $O(N^2)$: Instead of iterating over each substring to check the occurrence of a vowel we can maintain a prefix frequency of vowels array and use this pre-computed array to check if the frequency of vowels in a substring from position l to r is more than 0 or not. prefix[i] stores the frequency of vowels from index 0 to index i. This can be precomputed in a single traversal of the array. The number of vowels from l to r is prefix[r] - prefix[$l-1$] if $l \neq 0$ or prefix[r] if $l = 0$. The string is assumed to be 0-indexed here.

Optimised Approach $O(N)$:

Approach 1: The idea is to calculate the number of valuable substrings ending at a particular index for each index and taking the sum over all indices. We iterate over the string and at each index calculate the number of valuable strings ending at that index. Since the ending position is fixed we need to calculate the number of starting positions possible for each index. If s is a possible starting index then all indices before s are also valid. Hence, we need the greatest s for which the substring from s to the current index is valuable. This value of s is the latest occurrence of a vowel till the current index. This can be maintained and updated as we traverse over the string. If $last$ stores the index of the last occurrence of a vowel, then for a given end index there are $last + 1$ possible valuable substrings index ending at the current index.

Approach 2: Instead of counting the number of substrings having at least one vowel, we will count the number of distinct substrings each vowel is present in. The total number of distinct substrings containing a particular character of a string, say at the i^{th} index (0 indexing) is given by $(i+1) \times (n-i)$, where n is the length of string. This is explained as follows. We know that a substring is a contiguous part of the original string. That means, for a particular character, the substrings containing it will extend both on the right and(or) the left-hand side of that character. There are $(i+1)$ and $(n-i)$ characters on the left and the right-hand side of the i^{th} character respectively(including the i^{th} character itself). Hence, the number of substrings will be given by the product of the two, i.e., $(i+1) \times (n-i)$ ($\because$, there are $i+1$ choices on the left and $n-i$ choices on the right).

Now, we will iterate through the string and take the summation of the number of substrings at each occurrence of vowel. We will also need to maintain a variable l. Suppose i^{th} index has a vowel. Then, l will store the index of the vowel present just before the vowel at index i. We do this because while calculating the number of substrings for the i^{th} vowel, we don't want to include the l^{th} vowel, otherwise, it will lead to overcounting (the substrings containing the i^{th} and the l^{th} vowel will be counted twice). The variable l will be initialized to -1. It will be updated to i after calculating the number of substrings for the i^{th} vowel.

So the answer is given by $\Sigma(i-l) \times (n-i)$ for all index i at which a vowel is present.

Problem F. Study Groups

To calculate the number of different ways in which the teacher can partition the class, we need to find out how many choices do we have for placing each partition line.

One thing we can observe from the problem statement is that since every partition must have exactly k-students, we can place a partition line between seat numbers a_m and $a_{m'}$ ($m < m'$) , such that $\sum_{i=1}^{i=m} a_i$ is a multiple of k , $a_m = a_{m'} = 1$, and there are no students sitting between these 2 seats. This can be done in $a_{m'} - a_m$ ways. There is no other way to place a partition line following the given constraints.

So by the fundamental principle of counting, the final answer will be $\prod(a_{m'} - a_m)$, for all valid m and m'. This can be implemented in *one-pass* by maintaining a count of the number of seats that have been occupied up to seat a_i, and by storing the location of the last seat that was occupied by a student. Hence the complexity for the ideal solution will be O(n).

One corner case that needs to be dealt with separately is when $\sum_{i=1}^{i=n} a_i = 0$. In this case, there exists no way of partitioning the class such that every partition has exactly k-students (as $1 \leq k \leq 10^5$). Therefore the answer will be 0 when none of the seats is occupied.

Problem G. Is time travel possible?

Any odd integer n can be written as $n = 2p + 1$, where p is an integer. Whereas any even integer m can be written as $m = 2q$, where q is an integer. Therefore, $n^4 = 16p^4 + 32p^3 + 24p^2 + 8p + 1$ and $m^4 = 16q^4$.

Hence, for any two odd integers n_1 and n_2, $n_1^4 - n_2^4 = 8 * [2(p_1^4 - p_2^4) + 4(p_1^3 - p_2^3) + 3(p_1^2 - p_2^2) + (p_1 - p_2)]$ and for any two even integers m_1 and m_2, $m_1^4 - m_2^4 = 8 * [2(q_1^4 - q_2^4)]$.

Let the total number of odd and even integers given in the array be o and e respectively. Then the required answer will be $\binom{o}{2} + \binom{e}{2}$.