A. VowelScript

We simply need to check if the first or the last character of the string is a vowel or not.

Time complexity - $O(n)$for reading the string.

B. Pattern Matching

For every cell (i, j) that has a 1, we need to check if all its diagonal neighbours have 1, if we can find at least one such cell, the answer is YES, otherwise NO.

Time complexity - $O(n^2)$ as we need to check each cell once in the worst case.

C. Cryptocurrency

The final price of each cryptocurrency will be $p_i + x \cdot a_i$. First we need to find the index with the maximum final price, if there are multiple such indexes, the smallest of them will be the answer. Also note that you need to use $long\ long\ int$ to store the final price to avoid overflow.

Time complexity - $O(n)$ as we need to iterate on the array to find the final price of each cryptocurrency.

D. Count Pairs

Let $f(x) = sumOfDigits(x)$

We need to find the number of pairs (i, j) such that
$a[i] - f(a[j]) = a[j] - f(a[i])$
Let's rewrite this as $a[i] + f(a[i]) = a[j] + f(a[j])$

Now observe that we can match any two pairs with the same value of $a[i] + f(a[i])$. So if there are x indexes with the value $a[i] + f(a[i])$, the number of pairs will be ${}^{x}C_{2} = x \cdot (x - 1) / 2$. So we can use something like $std::map$ to count the frequency of each $a[i] + f(a[i])$ and finally add the number of pairs to get the answer. Note that the final answer may not fit in 32-bit integer.

Instead of using $std :: map$ we can also maintain a global array $c[\,]$ of size $10^7 + 100$ for storing count of $a[i] + f(a[i])$ for each test case. We take size $10^7 + 100$ because the sum of digits will always be less than 100 and we make it global so that we won't have to declare it again for every test case which will make it slow.

For each test case we can iterate on the array $a[\,]$ and increase $c[s]$ by 1 where $s = a[i] + f(a[i])$ and for calculating the answer we can again iterate array $a[\,]$ and if $c[s] > 0$ then we add $c[s] \cdot (c[s] - 1) / 2$ to our answer and update $c[s] = 0$. This will allow us to reset the array $c[\,]$ with iterating n times for each test case.

Time complexity - $O(n \log n \log (a_{max}))$ or $O(n \log (a_{max}))$ depending on the implementation.

E. Yet another query task

d_i is nothing but the smallest prime factor of a_i. Notice the constraint of a_i is $\leq 10^6$. So we can use Sieve of Eratosthenes to quickly calculate the smallest prime factor of each integer $\leq 10^6$. Now for each query (l, r) we need to find the sum of d_i from l to r, i.e. $d_l + d_{l+1} + \ldots + d_{r-1} + d_r$.

You can learn more about Sieve of Eratosthenes here:
https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html

Now lets create an array pf, where $pf[i]$ tells us the sum $d_1 + d_2 + \ldots + d_i$. Now, for each query (l, r) $(l \leq r)$ -

$pf[r] = d_1 + \ldots + d_{l-1} + d_l + \ldots + d_r$

$pf[l-1] = d_1 + d_2 + \ldots + d_{l-1}$

Now notice that $pf[r] - pf[l-1] = d_l + \ldots + d_r$ which is the answer to each query. So if we pre calculate the array pf, the answer of each query will be $pf[r] - pf[l-1]$. We can get this in $O(1)$. This technique is called prefix sums.

Time complexity - $O(M \log\log M)$ $(M = max(a_1, a_2, \ldots, a_n)$ due to sieve of eratosthenes.

F. Modulo Teleportation

We can teleport from $(a, 0)$ to $(b, 0)$ **iff** $b\%(b - a) = 0$. This means that $(b - a)$ should be a factor of b.

Let $x = (b - a)$, which is a factor of b
$\Rightarrow a = b - x$
$\Rightarrow a = b - some\ factor\ of\ b$

So lets calculate an array dp, where $dp[i]$ tells us the number ways to reach i. Since we start at $(1, 0)$ $dp[1]$ will be 1. So now if x is a factor of b, and the number of ways to reach $b - x$ is $dp[b - x]$, we can teleport from $b - x$ to b and the number of ways to reach b will increase by $dp[b - x]$.

So, for each i from 2 to 10^5, we need to find the factors of i, and increase $dp[i]$ by $dp[i - factor]$.

Now, to find the factors of an integer -
If f is a factor of n, n / f will also be a factor of n. So if we find half of the factors of n, we can know all the factors of n due to this property. So till what value should we check? Only the first $\sqrt{n}$ integers. Because if f_1 and f_2 are factors of n and $n = f_1 \cdot f_2$, at least one of them will be $\leq \sqrt{n}$, if both are $> \sqrt{n}$, $f_1 \cdot f_2$ would be greater than n.

More on this here :
https://stackoverflow.com/questions/5811151/why-do-we-check-up-to-the-square-root-of-a-prime-number-to-determine-if-it-is-pr

Now, the answer of each query n will be $dp[n]$, so we can answer queries in $O(1)$. This technique is called dynamic programming.

Time complexity - $O(n\sqrt{n})$ as we need to find the factor of each integer.