Problem Analysis

Malaysian Computing Olympiad 2017

A. Cable Cars

Problem Summary :

There is an array of n integers that is a permutation of 1 to n. You have to choose a subarray. One of the ends must be the number n. The other end must satisfy the condition that it is the highest number among all numbers located between it and the number n. Find the number of suitable positions for the other end.

Constraints : N <= 1,000,000

Subtask 1 (17 Points)

Additional Constraints : $N <= 1,000$

For this subtask, N is quite small. Thus, we can solve it using brute force.

First, we find the position of the value N in the array in $O(N)$ time by simply iterating through all elements.

For each element of the array, iterate through all elements from it to the position of the value N. Check whether each element is smaller than the current element. If this is true, increment the answer by 1.

This solution takes $O(N^2)$ time.

Subtask 2 (83 Points)

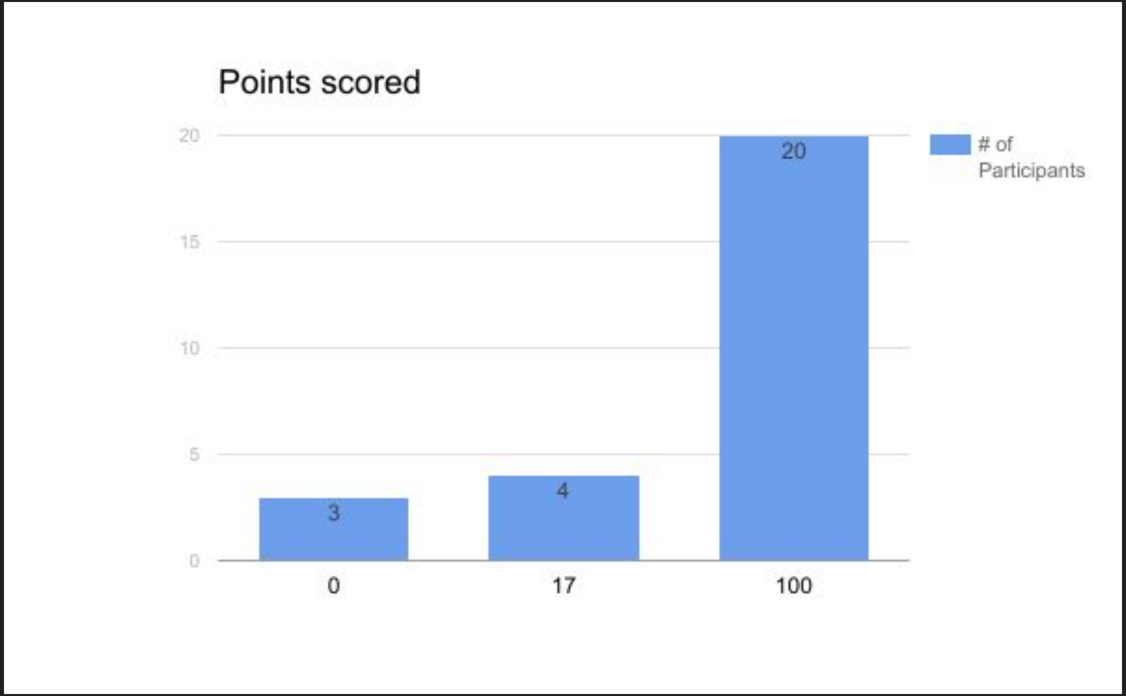
Additional Constraints : None

An O(N^2) solution cannot pass here. The trick is that we avoid iterating through all elements between N and the current element every time we want to determine if the current element can be an ending position.

Start from the position which contains N. Iterate from that position to the right. Maintain the maximum value of the elements visited so far (excluding N). To determine if the current element can be an endpoint, we just have to check whether it is larger than the maintained maximum value in O(1) time.

Do the same for the left side. Thus, we iterate through each element at most once. Thus, the algorithm takes O(N) time, which will pass.

Score Distribution



B. Travelling Salesman

Problem Summary :

There is a graph with N nodes and M bidirectional weighted edges.

Node i is assigned two numbers (s_i, e_i). It means that node i toggles between s_i minutes of daytime and e_i minutes of nighttime, and will starts its daytime on minute 0.

Example, $s_i = 4$, $e_i = 2$ means that minutes 0, 1, 2, 3, 6, 7, 8, 9, … will be daytime and 4, 5, 10, 11, … will be nighttime.

You start at node s. You can only leave a node when it is daytime in that node. For all nodes u, find the minimum amount of time required to reach u from s.

Constraints : N, M <= 200,000, w_i, s_i, $e_i <= 10^9$

Subtask 1 (13 Points)

Additional Constraints : The weights of all edges are 1, $e_i = 0$

Note that the conditions imply that the cities will not experience nighttime. Thus, the problem reduces to a direct SSSP (Single-Source Shortest Paths) problem. Our task is to find the distance of all nodes from the source vertex s.

Subtask 1 (13 Points)

Additional Constraints : The weights of all edges are 1, $e_i = 0$

Note that the conditions imply that the cities will not experience nighttime. Thus, the problem reduces to a direct SSSP (Single-Source Shortest Paths) problem. Our task is to find the distance of all nodes from the source vertex s.

This subtask can be easily solved with **BFS** (Breadth-First Search) algorithm as the weights of all edges are 1.

This algorithm works in O(N + M) time.

Subtask 2 (11 Points)

Additional Constraints : $e_i = 0$

This subtask is almost the same as Subtask 1, but now the weights of the graph are not necessarily equal to 1.

Subtask 2 (11 Points)

Additional Constraints : $e_i = 0$

This subtask is almost the same as Subtask 1, but now the weights of the graph are not necessarily equal to 1.

However, the cities will still not experience nighttime, so we only have to solve the SSSP problem on a weighted graph. This can be done by using **Dijkstra's Algorithm** instead of BFS in subtask 1.

This algorithm works in O(Mlog N) time.

Subtask 3 (14 Points)

Additional Constraints : $M = N - 1$

Now, there is no guarantee that no city will experience nighttime. However, we have the special condition that $M = N - 1$. What does it mean?

Subtask 3 (14 Points)

Additional Constraints : M = N - 1

Now, there is no guarantee that no city will experience nighttime. However, we have the special condition that M = N - 1. What does it mean?

Recall that we're given that the graph is connected. Thus, M = N - 1 precisely means that the given graph is a tree! Now, we just have to solve the problem on a tree.

We'll root the tree at s, the source vertex.

Subtask 3 (14 Points)

Let dp[i] denote the minimum amount of time needed to reach node i from the root, which is s. Clearly, dp[s] = 0.

Now, we'll do a **DFS** (Depth-First Search) on the tree. We'll calculate all the dp values of the ancestors of a node before processing the node.

Suppose we need to calculate the value of dp[u]. We already calculated dp[p], where p is the parent of u. How do we calculate dp[u]?

Subtask 3 (14 Points)

Let T = dp[p] and w be the weight of the edge between p and u.

Now, to go from the root to u, it is optimal to reach p as fast as possible, wait there until node p is experiencing daytime, and immediately move to u.

With this observation, the value of dp[u] can be calculated easily. First, we find T', the next moment of time where node p experiences daytime. This can be computed in constant time directly with the values of s_p and e_p. Now, we just set dp[u] = T' + w.

Thus, the dp values can be calculated with a single dfs and the solution works in O(N) time.

Subtask 4 (35 Points)

Additional Constraints : $s_i + e_i <= 10$, $s_i + e_i$ are equal for all i.

Now, we have to solve the general problem. However, the constraints on s_i and e_i are small in this subtask and we have the weird condition that $s_i + e_i$ are equal. How can we make use of this fact?

Subtask 4 (35 Points)

Additional Constraints : $s_i + e_i <= 10$, $s_i + e_i$ are equal for all i.

Now, we have to solve the general problem. However, the constraints on s_i and e_i are small in this subtask and we have the weird condition that $s_i + e_i$ are equal. How can we make use of this fact?

Observation : We only need to know what time we arrive at a vertex modulo $s_i + e_i$ to determine whether it’s daytime or nighttime.

This motivates us to construct an auxiliary graph.

Subtask 4 (35 Points)

For convenience, let $t = s_i + e_i$. t is the period of the day-night cycle for each vertex.

For each vertex u, construct t vertices in the auxiliary graph, labelled as (u, 0), (u, 1), …, (u, t - 1) respectively.

Now, for each edge u-v of the original graph with weight w,

- Add a directed edge from (u, x) to (v, (x + w)%t) with weight w for all $0 <= x < s_u$.
- Add a directed edge from (v, x) to (u, (x + w)%t) for all $0 <= x < s_v$.

Additionally, for each vertex u,

- Add a directed edge from (u, x) to (u, 0) with weight t - x for all $s_u <= x < t$.

Subtask 4 (35 Points)

Now, we apply Dijkstra's Algorithm from the node (s, 0) and we can find the shortest path from (s, 0) to every other node. From here, we can easily find the distance from s to all other nodes in the original graph.

Since we created <= tN nodes and the number of edges is <= tN + 2tM, the solution runs in O(t(N + M)log (tN)) time, which is sufficient for this subtask as t <= 10.

Subtask 5 (27 Points)

Additional Constraints : None

In this subtask, there are no longer any constraints on the values of s_i and e_i nor any restrictions on the graph. The previous solution would not work as $s_i + e_i$ might not be constant and the values of s_i and e_i are too large.

However, we can solve the full problem with just the Dijkstra’s Algorithm, but we need some slight modifications.

Subtask 5 (27 Points)

Let dist[u] be the minimum time required to travel from s to u.

In the original Dijkstra's Algorithm, when we process a new edge u-v with weight w, we check if dist[u] + w < dist[v]. If this is true, we let dist[v] be dist[u] + w and push (dist[v], v) into the priority queue.

We can actually do the same thing here, but with a minor change. The new distance is not necessarily dist[u] + w, as node u might be experiencing nighttime during minute dist[u]. Instead, we let T' be the next time after dist[u] for which node u is experiencing daytime. This can be calculated in O(1) time using the values of s_u and e_u. Now, the new distance is T' + w and we check if it's less than dist[v] as in the original Dijkstra's Algorithm.

Subtask 5 (27 Points)

Why does this work?

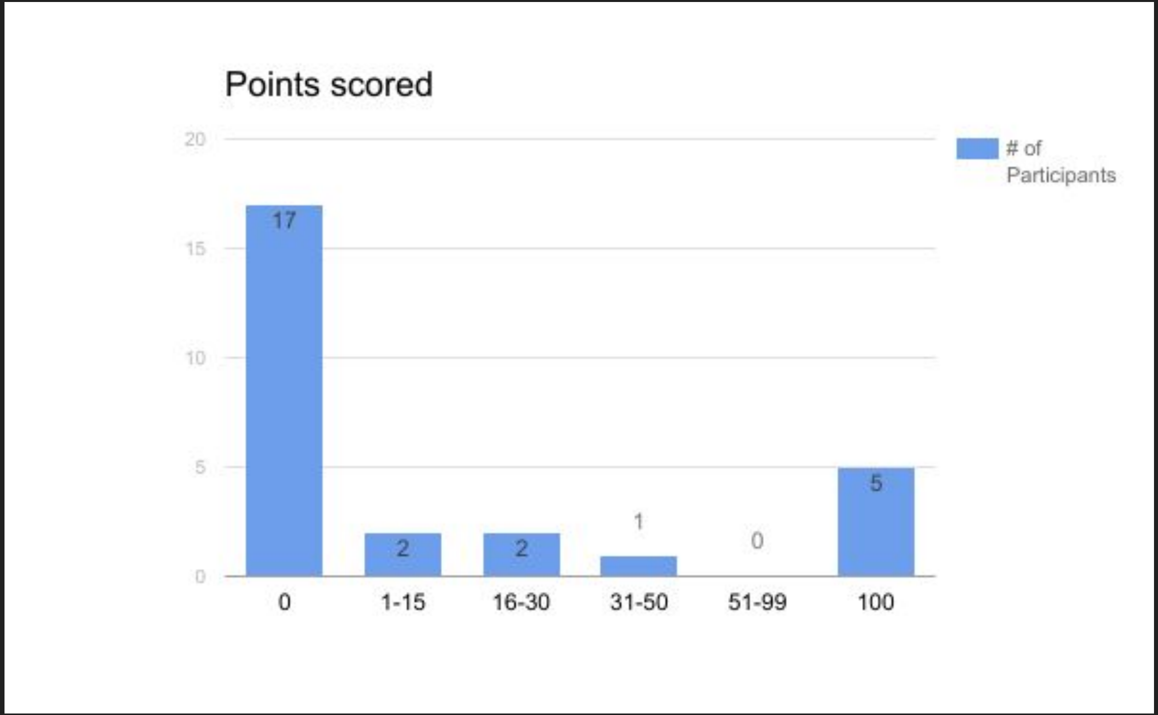
Note that there is no point waiting at a vertex during daytime as it is always beneficial to travel directly at that moment.

Thus, arriving at a vertex at time u is never worse than arriving at a vertex at time v if v > u.

This means that Dijkstra’s Algorithm will work correctly.

Time Complexity : O(Mlog N)

Score Distribution



C. Large City

Problem Summary :

There is a N x N grid where K of the cells are blocked. You start from the lower-left corner and can only go up or right. Find the number of cells that can possibly be reached.

Constraints : $N <= 10^9$, $K <= 300,000$

Subtask 1 (10 Points)

Additional Constraints : N <= 5000

In this subtask, the value of N is small enough for an $O(N^2)$ solution to be viable. In fact, we're able to store the whole grid in memory.

There are a few ways to tackle this subtask. The simplest way is probably via dynamic programming.

Let dp[i][j] be 1 if we can reach cell (i, j) and 0 otherwise. Initially, dp[1][1] = 1.

The transitions are simple :

dp[i][j] = 0 if (i, j) is blocked

dp[i][j] = dp[i - 1][j] OR dp[i][j - 1] otherwise, where x OR y is true if and only if at least one of x and y is true.

Time Complexity : $O(N^2)$

Subtask 2 (57 Points)

Additional Constraints : N <= 300,000

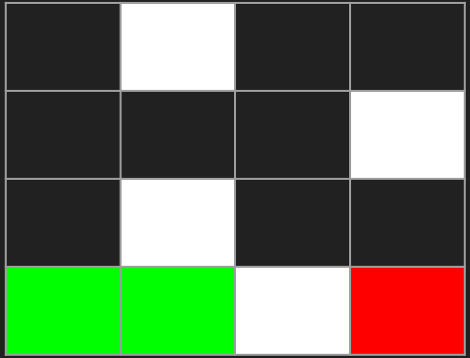
This subtask is the key to solve the whole problem. Once you get a solution for this subtask, solving the whole problem isn’t hard.

Looking at the constraints, we need a solution that is linear in terms of N and K.

The easiest way to solve this is to look at the reachable cells row by row.

Subtask 2 (57 Points)

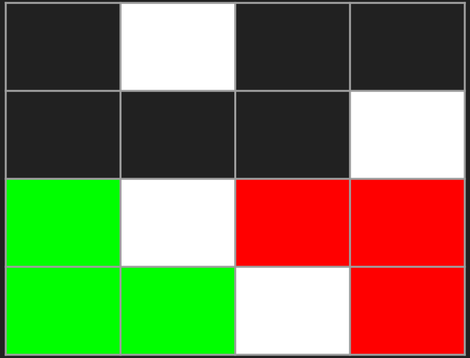
We'll maintain the set of reachable cells on each row starting from the bottom row and then update this set when we go to the next row.



Green cells = Reachable cells
White cells = Blocked cells
Red cells = Unreachable cells that are unblocked.

Subtask 2 (57 Points)

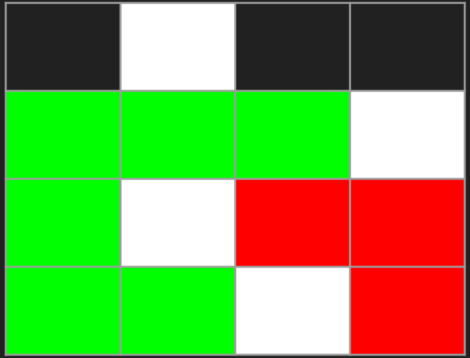
We’ll maintain the set of reachable cells on each row starting from the bottom row and then update this set when we go to the next row.



Green cells = Reachable cells
White cells = Blocked cells
Red cells = Unreachable cells that are unblocked.

Subtask 2 (57 Points)

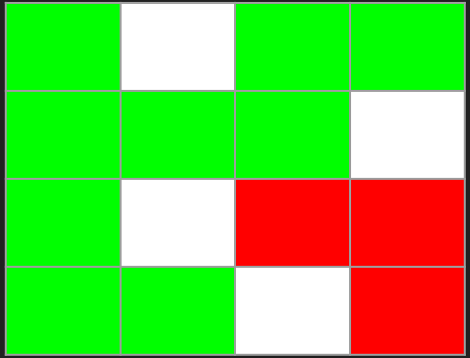
We’ll maintain the set of reachable cells on each row starting from the bottom row and then update this set when we go to the next row.



Green cells = Reachable cells
White cells = Blocked cells
Red cells = Unreachable cells that are unblocked.

Subtask 2 (57 Points)

We’ll maintain the set of reachable cells on each row starting from the bottom row and then update this set when we go to the next row.



Green cells = Reachable cells
White cells = Blocked cells
Red cells = Unreachable cells that are unblocked.

Subtask 2 (57 Points)

We only require the set of reachable cells of the last row to update the set of reachable cells of the next row.

However, we can’t store all the reachable cells separately, as it will take O(N) time per row. The trick here is to store the subsegment of reachable cells for each row instead.

For example, suppose the row of cells look like this : XOOXOOOXXOX where Os are reachable cells.

We can store this as a set of segments {[2, 3], [5, 7], [10, 10]} instead (the segments are 1-indexed).

Subtask 2 (57 Points)

Now, the question is : How can we update the set of reachable cells fast?

Let S be the set of reachable cells of the current row. For the next row, we first decompose the row into a set of segments of unblocked cells (so XOOXOOOXXOX where Os are unblocked cells becomes {[2, 3], [5, 7], [10, 10]} like before.) Denote this set by T.

Now, let I be the set of reachable cells for the next row. We want to compute I from S and T fast.

Subtask 2 (57 Points)

We iterate over the intervals of S one by one from left to right.

Suppose [l, r] is the current interval. Now, we iterate through the intervals of T. Let the current interval in T be [l’, r’]. There are a few cases :

Case 1 : r’ < l or l’ > r

In the former case, the current interval [l’, r’] is useless so we move to the next one. Similarly, we move to the next interval in S if l’ > r.

Subtask 2 (57 Points)

Case 2 : [l, r] is contained within [l’, r’]

In this case, note that all cells in [l, r’] in the next row are reachable. Thus, we push this into I and move on to the next interval of T and S.

Case 3 : l’ <= l <= r’ <= r

Again, we note that all cells in [l, r’] in the next row are reachable and move on to the next interval of T.

Case 4 : l <= l’ <= r <= r’

In this case, all cells in [l’, r’] in the next row are reachable and we may move on to the next interval of S and T.

Subtask 2 (57 Points)

Thus, if S contains X intervals, T contains Y intervals, then we can process each row in O(X + Y) time. Thus, the total time complexity is linear in the total number of intervals of reachable cells.

However, if we consider an interval of unblocked cells, we can see that it is impossible for there to be two disjoint intervals of unreachable cells in it, as if L is the leftmost reachable cell in that interval, all cells to the right of L in that interval must also be reachable.

Thus, the number of intervals of unreachable cells is bounded by the number of intervals of unblocked cells.

The number of intervals of unblocked cells is O(N + K), as each blocked cell can separate an interval of unblocked cells into at most two intervals.

Thus, the complexity of this solution is O(N + Klog K), where the log factor comes from sorting the blocked cells in increasing order of x-coordinate.

Subtask 3 (33 Points)

Additional Constraints : None

For this subtask, N can be up to 10^9, so we need to avoid iterating through all rows.

The key is to note that there are actually a lot of empty rows and we can actually skip most of them.

The solution for this subtask will be almost the same as the previous subtask, but we have to take care of the empty rows too.

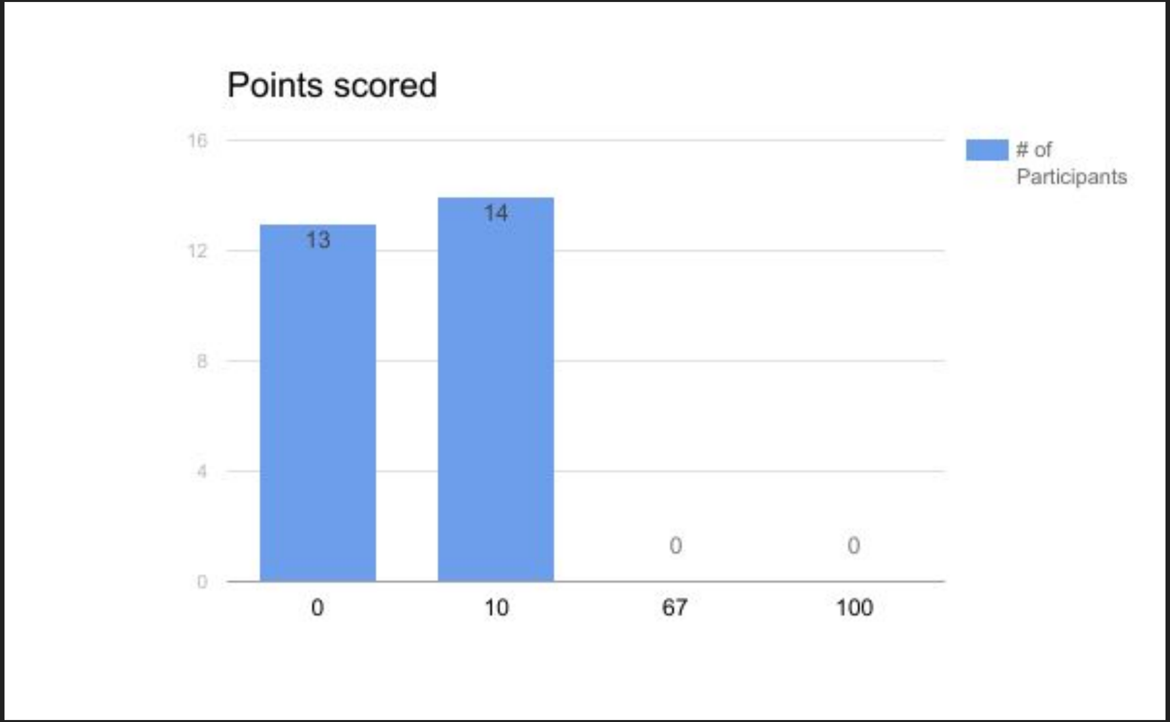
Subtask 3 (33 Points)

Suppose the set of intervals of reachable cells in the current row is S and we know the next few rows are empty.

Let L be the leftmost reachable cell in S. Then, the set of reachable cells in the next empty row is simply [L, N]. In fact, this applies to all the empty rows above it as well. Thus, we can skip through all the empty rows in the middle and directly go to the empty row directly below the next relevant row.

Thus, by skipping through all irrelevant empty rows, we arrive at a solution with O(Klog K) complexity, as we avoided wasting time on rows which does not contain a blocked cell.

Score Distribution



D. Newbie Hacker

Problem Summary :

Given a string S, a string X is called good if it can be cut into several pieces so that each piece is a prefix of S.

Given another string T, answer Q queries. Each query is of the form : “Is the substring [l..r] of T good?”.

Constraints : |S|, |T|, Q <= 300,000

Subtask 1 (15 Points)

Additional Constraints : |S|, |T| <= 1000, Q <= 10

For this subtask, we may solve for each query separately. We need to find a way to determine if a string X is good for a given S. Let n = |T|.

For each position in X, let dp[i] denote the largest index j such that the substring [i..j] is a prefix of S. The values of dp can be computed in $O(|X|^2)$ time naively.

Now, dp2[i] is 1 if the suffix X[i..n] is good and 0 otherwise.

Let dp2[n + 1] = 1 for convenience.

dp2[i] is 1 if and only if there exists a j such that dp[i] <= j and dp2[j + 1] is 1.

This solution works in $O(Q|T|^2)$ time, which will pass Subtask 1.

Subtask 2 (16 Points)

Additional Constraints : |S|, |T| <= 5000

Now, the number of queries can get very large. We need to find a way to answer queries quickly.

Observation : If a string X is good, then any prefix of X is also good.

Thus, for each i, if we can compute the maximal j such that T[i..j] is good, then we can answers queries in O(1) time.

Subtask 2 (16 Points)

It remains to compute these values. Let ans[i] denote the maximum j such that T[i..j] is good, where j = i - 1 if no such j exists.

For simplicity, set ans[n + 1] = n.

Let dp[i] denote the largest index j such that T[i..j] is a prefix of S, where j = i - 1 if no such indices exist.

We can precompute dp[] in $O(|T|^2)$ time naively.

Now, we have ans[i] = max(i - 1, ans[j + 1]) for all i <= j <= dp[i].

Thus, we may compute ans[] in $O(|T|^2)$ time too.

This gives us an $O(|T|^2 + Q)$ solution.

Subtask 3 (38 Points)

Additional Constraints : |S| <= 10000

There are no additional restrictions to the length of T now but the length of S is still small.

In fact, we will use the same method in the previous subtask, but optimize the operations we performed to make them faster.

There are two important operations that we need to be able to do quickly :

1. Compute the largest index j such that T[i..j] is a prefix of S for all i.
2. Compute the values of ans[] quickly.

We'll solve the second part first as it is easier.

Subtask 3 (38 Points)

Assume we're able to find the values of dp[] fast. Recall the recurrence for ans[] is

ans[i] = max(i - 1, ans[j + 1]) for all i <= j <= dp[i].

Note that we calculate the values of ans[] from right to left and the values of ans[i] can be computed as the maximum of a certain range of ans[] to the right of i.

Thus, a segment tree can help us here! Construct a segment tree on the ans[] array which is initialized with -1 in the beginning. Whenever we calculated the value of ans[i], we update the i-th element in the segment tree with ans[i]. To calculate ans[i], we just have to find the maximum value in the range [i + 1, dp[i] + 1] in segment tree. Thus, if our segment trees support point update and range maximum query, then this part can be done in O(|T|log |T|), which is fast enough.

Subtask 3 (38 Points)

It remains to calculate the values of dp[] fast. Recall that dp[i] refers to the largest index j such that T[i..j] is a prefix of S.

Since the length of S is small in this subtask, maybe we can try to compute this dp on S as well and use it to compute the values of dp[] on T fast.

Let X[i] denote the largest index j such that S[i..j] is a prefix of S. (again if j does not exist, let j = i - 1)

We can compute X[] naively in this subtask naively in $O(|S|^2)$ time as $|S| <= 10000$.

The question is : How do we calculate the values of dp[] using X[]?

Subtask 3 (38 Points)

Let’s calculate the values of dp[] from left to right. We’ll illustrate how to calculate dp[] with an example.

S = ABABCDA

T = ABABACBAB

Now, S[1] = 7, S[2] = 1, S[3] = 4, S[4] = 3, S[5] = 3, S[6] = 3, S[7] = 7.

Subtask 3 (38 Points)

Let’s calculate the values of dp[] from left to right. We’ll illustrate how to calculate dp[] with an example.

S = ABABCDA

T = ABABACBAB

Now, X[1] = 7, X[2] = 1, X[3] = 4, X[4] = 3, X[5] = 3, X[6] = 3, X[7] = 7.

dp[1] = 4. We maintain a variable ptr that points at position 4.

Subtask 3 (38 Points)

Let’s calculate the values of dp[] from left to right. We’ll illustrate how to calculate dp[] with an example.

S = ABABCDA

T = ABABACBAB

Now, X[1] = 7, X[2] = 1, X[3] = 4, X[4] = 3, X[5] = 3, X[6] = 3, X[7] = 7.

Now, we need to calculate dp[2]. However, there’s no need to iterate through the letters one by one as we can deduce that dp[2] = X[2] = 1 since the second letter of T is equal to the second letter of S by dp[1] = 4.

Subtask 3 (38 Points)

Let’s calculate the values of dp[] from left to right. We’ll illustrate how to calculate dp[] with an example.

S = ABABCDA

T = ABABACBAB

Now, X[1] = 7, X[2] = 1, X[3] = 4, X[4] = 3, X[5] = 3, X[6] = 3, X[7] = 7.

Now, suppose we’re calculating dp[3]. This time, X[3] = 4, so we know that the 3rd and 4th characters must match the first and second characters of S. However, we still can’t say dp[3] = 4, as it might be possible to continue further pass the position indicated by ptr. (which was previously equal to 4)

Subtask 3 (38 Points)

Let’s calculate the values of dp[] from left to right. We’ll illustrate how to calculate dp[] with an example.

S = ABABCDA

T = ABABACBAB

Now, X[1] = 7, X[2] = 1, X[3] = 4, X[4] = 3, X[5] = 3, X[6] = 3, X[7] = 7.

We move the pointer until the corresponding letters of S and T doesn’t match again. The new value of ptr is 6 and the value of dp[3] = 5.

Subtask 3 (38 Points)

Let’s calculate the values of dp[] from left to right. We’ll illustrate how to calculate dp[] with an example.

S = ABABCDA

T = ABABACBAB

Now, X[1] = 7, X[2] = 1, X[3] = 4, X[4] = 3, X[5] = 3, X[6] = 3, X[7] = 7.

To compute dp[4], we look at X[2] and deduce that dp[4] = 4 + 1 - 2 = 3.

We can continue this procedure until we find all the values of dp[].

Thus, this part works in O(|T|) time.

Subtask 3 (38 Points)

To summarize, we managed to :

1. Find the values of ans given dp in $O(|T|\log |T|)$ time.
2. Find the values of X in $O(|S|^2)$ time.
3. Find the values of dp using X in $O(|T|)$ time.

Thus, the algorithm works in $O(|S|^2 + |T|\log |T| + Q)$ time, which will work for this subtask.

Subtask 4 (31 Points)

Additional Constraints : None

Now, |S| is also large so we can't afford to use $O(|S|^2)$ time to compute X[].

It turns out that computing the value of X[] is a well-known task and it can be solved using **Z-Algorithm**.

Z-Algorithm computes for all i, a value Z[i] which denotes the length of the longest substring starting from position i that is some prefix of the entire string, which is exactly what we need. Thus, we can compute X[] using Z-Algorithm in O(|S|) time.

Subtask 4 (31 Points)

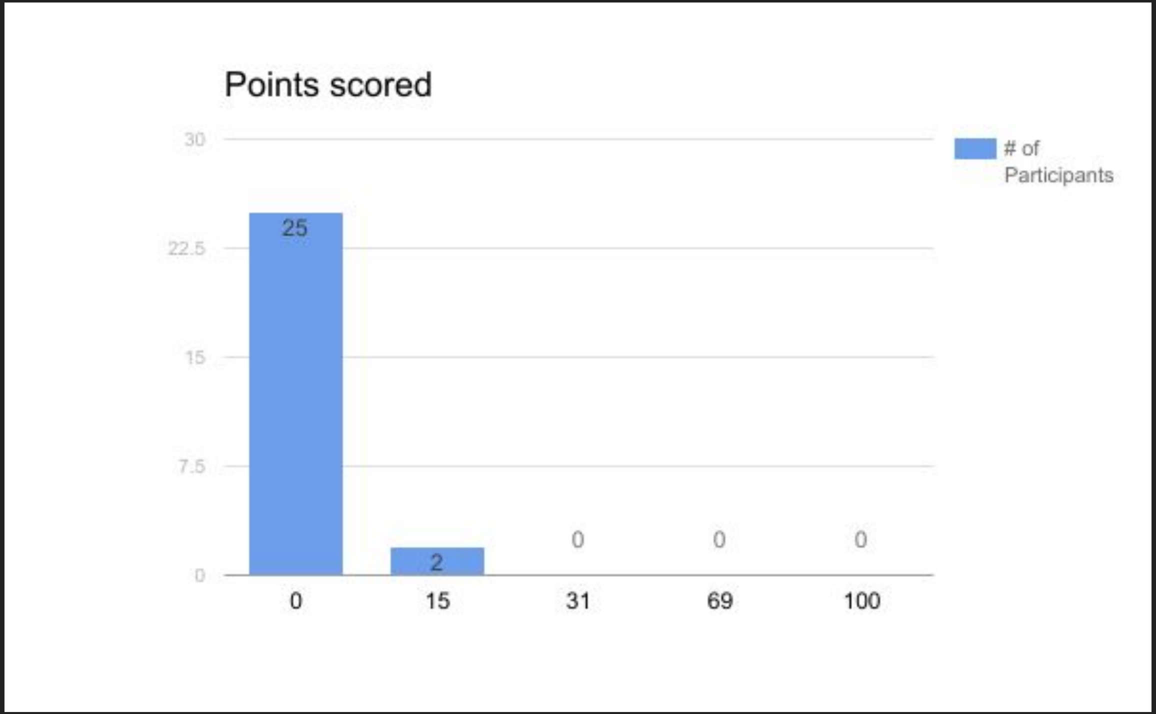
Thus, we can :

1. Find the values of ans given dp in O(|T|log |T|) time.
2. Find the values of X in O(|S|) time.
3. Find the values of dp using X in O(|T|) time.

The total time complexity reduces to O(|S| + |T|log |T| + Q), which is sufficient to get AC.

There’s a trick you can employ after knowing Z-Algorithm. You may append the string T after S with a dummy character ‘$’ in between (so the new string S’ = S + ‘$’ + T), and apply Z-Algorithm on the string S’ to find the values of dp[] directly.

Score Distribution



E. Magical Teleporter

Problem Summary :

There are N cities arranged in a row. Each city has a label L, R or B. The i-th city from the left is called city i. From city i, one can teleport to any other city if the label of city i is B, teleport to any city on the left if the label is L and teleport to any city on the right if the label is R.

Find the number of ways to start from city s and end in city e, while visiting every city **exactly once**.

Constraints : N <= 2000

Subtask 1 (8 Points)

Additional Constraints : N <= 10

Note that N is small in this subtask, so brute force might work.

We can try all possible permutation of cities to visit and for each permutation, determine whether it is possible to visit the cities in that order.

There are (N - 2)! possible permutations as the start and ending city is fixed. For each permutation, we can check its validity in O(N) time. Thus, this solution works in O(N(N - 2)!) time.

Subtask 2 (16 Points)

Additional Constraints : N <= 18

At first sight, this doesn’t seem much different from Subtask 1. However, (18 - 2)!*18 ~ 3*10^{14}, which is too much. We need a slightly harder approach.

On the other hand, an exponential solution could still work here. We’ll use bitmask dp to solve this subtask.

Subtask 2 (16 Points)

Let dp[S][i] denote the number of ways to visit all the cities in S exactly once and end at city i. The answer is dp[{1, 2, …, n}][e].

The base case of the dp is dp[{s}][s] = 1.

We will calculate the dp values in increasing order of size of S.

Suppose we found the value of dp[S][i]. Then, we try all possible ways to add a new city to the path. Suppose we try to add city x. Then, if we can teleport from i to x, we have dp[S ∪ {x}][x] += dp[S][i], where ∪ is the union operator.

Thus, we can compute all dp values in $O(2^n n^2)$ time.

Subtask 3 (55 Points)

Additional Constraints : s = 1, e = N

If you can solve this subtask, you’re very close to solving the full task. Exponential solutions are doomed to failure for N <= 2000. We have to find a polynomial-time solution.

Before this, we’ve been trying to append cities to the path one by one until we visit all cities. However, what if we add cities in another order?

Subtask 3 (55 Points)

Each path can be described by a permutation of N integers, denoting the sequence of cities visited.

Suppose N = 5 and so the final permutation has 5 integers :

_ _ _ _ _

Previously we have added numbers from left to right. Now, let us instead add the numbers 1 to N in this order.

Subtask 3 (55 Points)

Example :

Final Path : 4, 1, 3, 2, 5

_ _ _ _ _

_ 1 _ _ _

_ 1 _ 2 _

_ 1 3 2 _

4 1 3 2 _

4 1 3 2 5

Subtask 3 (55 Points)

Note that at each point of time, the permutation will consists of several “connected components”

For example,

9 6 _ 1 5 _ 3 _ 4 7 _ 8

contains 5 connected components (colored with different colors)

There are 3 types of connected components :

1. Starting component (The component that is stuck to the beginning of the permutation)
2. Ending component (The component that is stuck to the end of the permutation)
3. Free component (A component in the middle of the permutation)

Subtask 3 (55 Points)

The trick is we will do dynamic programming using the connected components. Let dp[i][j] denote the number of ways to add the numbers 1, 2, …, i to the permutation such that there are exactly j connected components.

Note that

1. At any time before the whole permutation is filled, the rightmost element of each connected component which is not the ending component must be either R or B. Otherwise, there is no way to add another element to the end of that component afterwards
2. If we add an element to the front of a component, it must be either L or B (since we’re adding the numbers in increasing order)

Subtask 3 (55 Points)

Let’s focus on this subtask first and thus we assume s = 1, e = n.

Denote the label of the i-th city by S[i].

Then, dp[1][1] = 1 if S[1] = R or B and the answer is 0 otherwise.

Suppose we already calculated the value of dp[i - 1][j]. Our task is to update the values of dp[i][] which involves dp[i - 1][j].

Note that when calculating dp we do not consider the order of the free components until we merge them together.

There are a few cases to consider :

Subtask 3 (55 Points)

Let val = dp[i - 1][j].

Case 1 : i = N

If j = 1, then dp[i][j] += val, as we can only append N in the end.

Otherwise, nothing happens as after placing N - 1 numbers we must have 1 component only.

Subtask 3 (55 Points)

Case 2 : i < N

Consider what could happen when we add the number i into the permutation. let f be the number of free components among the j components.

Let x = S[i], the label of the i-th city

- Create a new free component with i

dp[i][j+1] += val if x is R or B

- <start> + i

dp[i][j] += val if x is R or B

Subtask 3 (55 Points)

- <start> + i + <free>

dp[i][j - 1] += val*f if x is L or B

- i + <free>

dp[i][j] += val*f if x is L or B

- <free> + i

dp[i][j] += val*f if x is R or B

Subtask 3 (55 Points)

- <free> + x + <free>

dp[i][j - 1] += val*f*(f - 1) if x is L or B, since when merging free components the order of merging matters.

Thus, the dp transitions can be done in O(1) time and the total complexity of this solution is $O(N^2)$.

Subtask 4 (21 Points)

Additional Constraints : None

This is almost the same as Subtask 3, but now it is not guaranteed that s = 1, e = N, so we have more cases to consider.

Again, let val = dp[i - 1][j], x = S[i] and f denote the number of free components.

Firstly, suppose i = N.

This case is easy to handle. If s = N, we must add n at the beginning. If e = N, we must add N at the end and if neither holds, then we must merge the starting and ending components with N.

Subtask 4 (21 Points)

From now on, we assume i is not equal to N.

Case 1 : i = s

- Create a new component with i

dp[i][j + 1] += val if x = R or B

- i + <free>

dp[i][j] += f*val if x = L or B

Subtask 4 (21 Points)

Case 2 : i = e

- Create a new component

dp[i][j + 1] += val

- <free> + i

dp[i][j + 1] += f*val

Subtask 4 (21 Points)

Case 3 : i is not equal to s or e

If the starting component does not exist (i < s) or the ending component does not exist (i < e), ignore the cases that involve these.

- Create a new component

dp[i][j + 1] += val if x = R or B

- <start> + i

dp[i][j] += val if x = R or B

Subtask 4 (21 Points)

- <start> + i + <free>

dp[i][j - 1] += f*val if x = L or B

- i + <end>

dp[i][j] += val if x = L or B

- <free> + i + <end>

dp[i][j - 1] += f*val if x = L or B

Subtask 4 (21 Points)

- i + <free>

dp[i][j] += f*val if x = L or B

- <free> + i

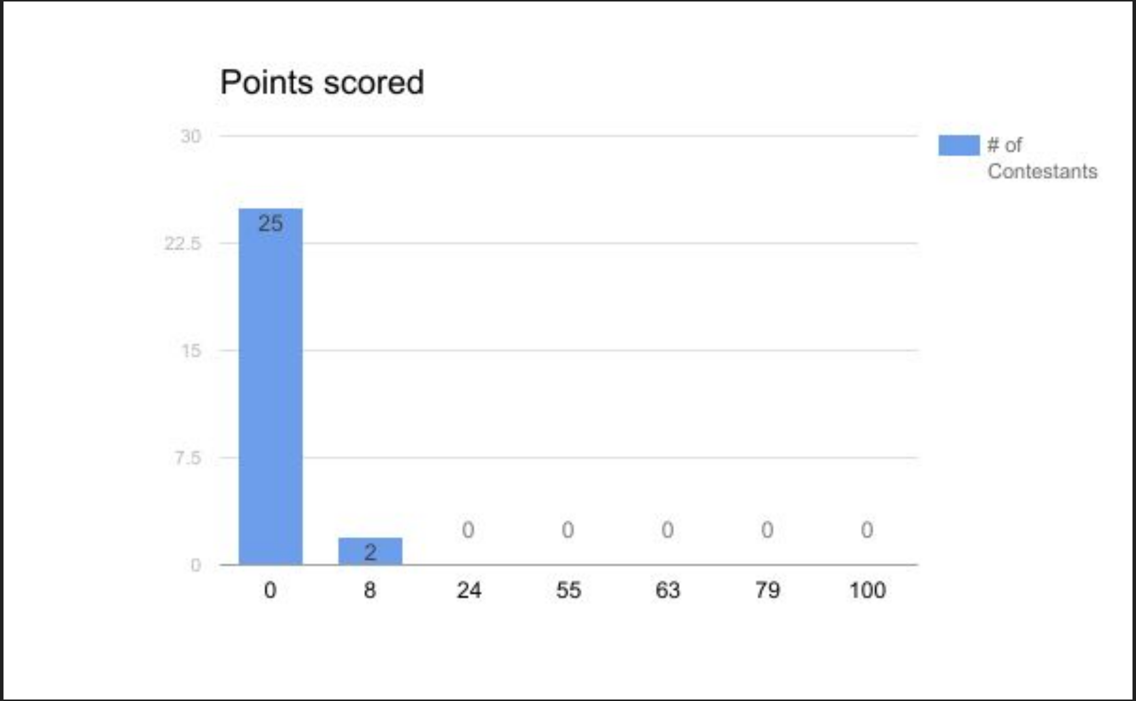
dp[i][j] += f*val if x = R or B

- <free> + i + <free>

dp[i][j - 1] += f*(f - 1)*val if x = L or B.

Thus, the transitions take O(1) time and the complexity of the solution is $O(N^2)$.

Score Distribution



F. Scientific Research

Problem Summary :

A word S of length 2N is called good if the last N characters can be rearranged to match the first N characters exactly. For example, ABCBAC is good whereas MCCMMC isn't.

An alien alphabet has K different letters. Find the number of good words of length 2N.

Constraints : $N <= 2000$, $K <= 10^9$.

Subtask 1 (6 Points)

Additional Constraints : $N <= 3$, $K <= 10$

N and K are both very small. This allows us to iterate through all possible strings of length 2N (there are K^{2N} such strings and check whether each string is valid.

The complexity of this solution is $O(K^{2N} * N)$.

Alternatively, it is possible to find the formula for the answer for N = 1, 2, 3. This solution will work in O(1).

Subtask 2 (12 Points)

Additional Constraints : N <= 300, K <= 3

The problem condition is equivalent to both halves of the string having the same multiset of letters. If we fix the multiset of letters S, and suppose there are f(S) different strings that can be formed using S. Then, there are a total of $f(S)^2$ good strings of length 2N that can be formed where both halves of the string uses the multiset of letters S.

This time, K is very small while N is bigger. This suggests that we need to find a $O(N^K)$ solution instead.

Note that each multiset can be described with a K-tuple $(a_1, a_2, \ldots, a_k)$ that denotes the frequency of each letter. We can iterate through all possible multisets S and add $f(S)^2$ to the answer as before. However, there’s a problem here. How can we find f(S) quickly?

Subtask 2 (12 Points)

f(S) is the number of distinct strings we can form using all the letters of S. Suppose (a_1, a_2, …, a_k) denote the frequencies of the different letters in the alphabet in S. Then,

$$f(S) = \frac{(a_1 + a_2 + \ldots + a_k)!}{a_1!a_2!\ldots a_k!}$$

To see why, there are (a_1 + a_2 + … + a_k)! to permute all the letters, if all of them are indistinguishable. But the order of the first type of letters doesn't matter, so we divide by a_1!. Similarly, we divide by all a_i! to account for the overcount.

We can precompute factorials in O(N) time. However, we still need to do division modulo 10^9 + 7.

Subtask 2 (12 Points)

To divide an integer A by an integer B modulo a prime P, where B is not divisible by P, we find the multiplicative inverse of B modulo P, which is often written as B^{-1} and multiply it by A. B^{-1} is actually equal to $B^{P - 2}$ modulo P so we only have to calculate $B^{P - 2}$. This can be done by binary exponentiation in O(log P)

Once we precomputed the factorials and inverse of factorials, we can calculate f(S) in O(K) time and thus the whole solution works in $O(K*N^K)$ time.

Subtask 3 (24 Points)

Additional Constraints : N, K <= 300

We cannot afford to use an exponential-time solution here, so we have to find a polynomial-time solution.

Our idea is still the same : Add up all the values of f(S)^2 for all tuples of nonnegative integers (a_1, a_2, …, a_k) whose elements sum of up to N.

Also, in the previous subtask, we noted that $f(S) = \frac{(a_1+a_2+...+a_k)!}{a_1!a_2!...a_k!}$

The numerator can be simplified to N!, as we know the sum of elements of a tuple is N. Thus, we can factor out (N!)^2 from all f(S)^2 and only focus on the denominator.

Subtask 3 (24 Points)

We now have to find the sum of $\frac{1}{(a_1!a_2!\ldots a_k!)^2}$ for all tuples (a_1, a_2, …, a_k) whose elements sum of up to N.

This might look like a tough math problem, but our solution does not use any advanced mathematics. We will solve this using dynamic programming.

Let dp[n][k] denote this sum when N = n, K = k.

dp[n][1] is just $\frac{1}{(a_1!)^2}$, which we precomputed in the beginning.

Suppose we already have the values (dp[0][k - 1], dp[1][k - 1], …, dp[n][k - 1]). How can we compute the values (dp[0][k], dp[1][k], …, dp[n][k])?

Subtask 3 (24 Points)

The idea is that we can iterate through all possible values of a_k and then go to other dp states.

More precisely,

$$dp[n][k] = (\frac{1}{0!})^2 dp[n][k-1] + (\frac{1}{1!})^2 dp[n-1][k-1] + ... + (\frac{1}{n!})^2 dp[0][k-1]$$

This comes from iterating through all possibilities for a_k and factoring out $(1/a_k)^2$ and replace the remaining sum with the corresponding dp values.

Thus, each transition takes $O(N)$ time and the whole solution takes $O(N^2K)$ time.

Subtask 4 (21 Points)

Additional Constraints : N <= 300

In this subtask, K can go up to 10^9 so we can’t even have a solution that runs linear in K. This means that we must drop the factor of K from our complexity somehow.

Let dp[n][k] denote the number of words of length 2n with an alphabet of size k.

Consider the first letter. Suppose it appeared exactly i times in the first half of the string (and thus also i times in the second half). Then, the number of ways to choose the positions for this letter is (n - 1)C(i - 1) * nCi. The remaining positions can be filled in dp[n - i][k - 1] ways. Finally, there are k ways to choose the first letter.

Thus, dp[n][k] = sum(k * (n - 1)C(i - 1) * nCi * dp[n - i][k - 1]).

Subtask 4 (21 Points)

Note that when $N = 1$ or $K = 1$, we can find the answer in $O(1)$ (they're K and 1 respectively)

Thus, only the states dp[n][k] when $1 <= n <= N$ and $\max(K - N, 1) <= k <= K$ will be visited. Also, the dp transitions take $O(N)$ time. Thus, the total complexity is $O(N^3)$, which passes this subtask.

Subtask 5 (37 Points)

Additional Constraints : None

We will use the same idea as in Subtask 3. The trick is that we have to deal with the large value of K here. Just now, we have went from K to K + 1 in $O(N^2)$ in the dp. What if we can move from K to 2K as well?

Subtask 5 (37 Points)

We will get the values of dp[][2K] from dp[][K].

To do this, fix split $(a_1, a_2, \ldots, a_{2k})$ into two halves and iterate over all possible sum of elements of the first half.

Suppose we fix the sum of the first half, $(a_1, a_2, \ldots, a_k)$ to be S. Then, the sum of the second half is N - S.

The sum we want to find over all such tuples is dp[S][K]*dp[N - S][K].

Thus, dp[N][2K] can be calculated as the sum of all dp[S][K]*dp[N-S][K] for all 0 <= S <= N. This means that we can calculate dp[][2K] from dp[][K] in $O(N^2)$ time.

Subtask 5 (37 Points)

To summarize,

- We can get dp[][K + 1] from dp[][K] in $O(N^2)$ time.
- We can get dp[][2K] from dp[][K] in $O(N^2)$ time.

Now, we need to get dp[N][K], where K is around 10^9. Call the two operations we can do above Operation A and B respectively.

Write K in binary (for example $K = 11 = 1011_2$)

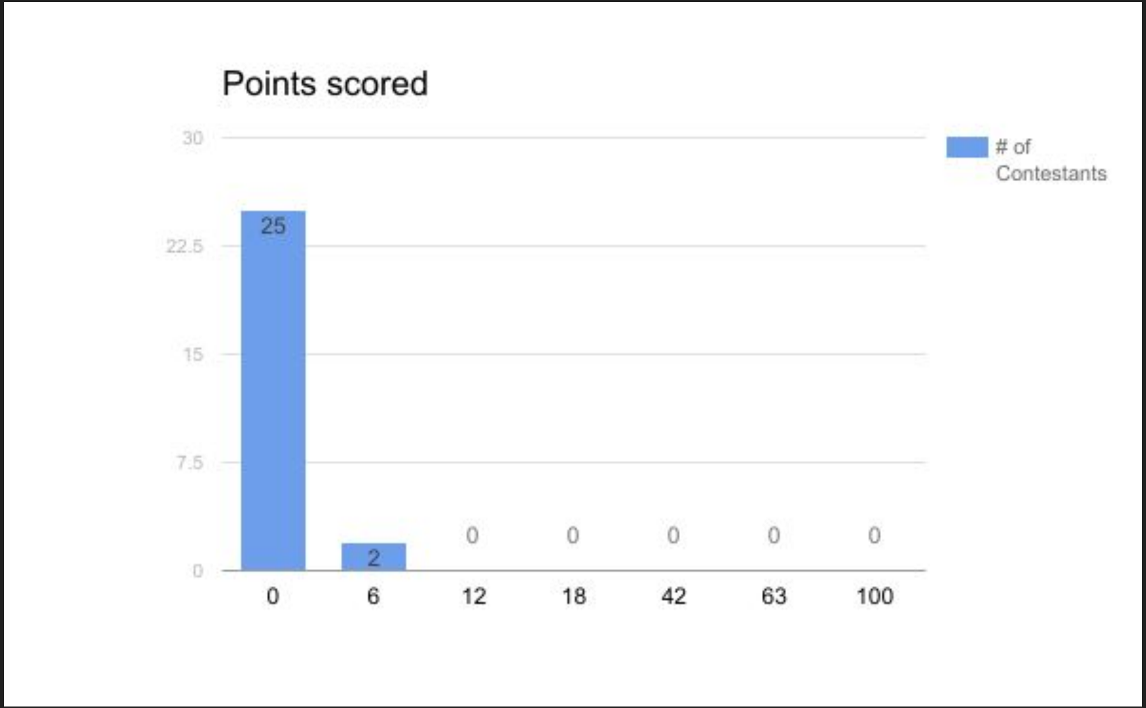
Start with the values of dp[][1]. Iterate through the binary digits of K from left to right, starting from the second digit. Apply Operation B once and if the current digit is 1, apply Operation A once. It is clear that after iterating through all digits we'll have the values of dp[i][K] for all 1 <= i <= N.

Subtask 5 (37 Points)

Note that we will always do O(log K) operations, each of which takes $O(N^2)$ time. Thus, the entire solution works in $O(N^2 \log K)$ time, which is sufficient to get AC.

In fact, it is possible to optimize this solution for it to work in O(Nlog Nlog K) time using Fast Fourier Transform, but it is out of the scope of IOI and thus we will not discuss it here.

Score Distribution



Contest Statistics

Malaysian Computing Olympiad 2017

Problem Credits

Problem A. Cable Car by *Christopher Boo*

Problem B. Travelling Salesman by *Zi Song Yeoh*

Problem C. Large City by *Zi Song Yeoh*

Problem D. Newbie Hacker by *Zi Song Yeoh*

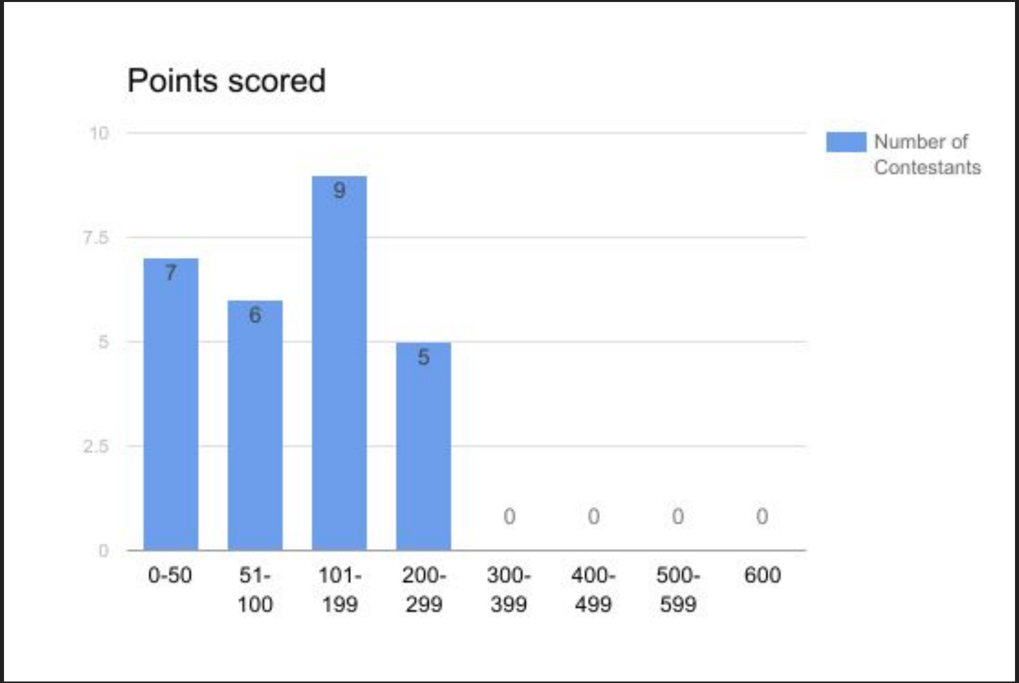
Problem E. Magical Teleporter by *Zi Song Yeoh*

Problem F. Scientific Research by *Zi Song Yeoh*

Contest Statistics

Problem	A	B	C	D	E	F
Mean Score	76.6	22.7	5.2	1.1	0.6	0.4
# of ACs	20	5	0	0	0	0
Max Score	100	100	10	15	8	6

Score Distribution


Points scored
Number of Contestants