Programming League 2021
Preliminary Round Editorial

Contest Team

May 2021

Contents

1 Introduction

The problems are sorted by categories, Closed Category then Open Category.
Contests link:
Programming League Contest 2021 - Preliminary Round [Closed]
Programming League Contest 2021 - Preliminary Round [Open]

2 Meow Array

Author: Amos Tan Li Sheng

Abridged Statement

Given an array of integers, find the second smallest element, the median, and the second-largest element of the array.

Solution

We use any built-in sorting method (e.g. Merge Sort or Heap Sort) to sort the array, which is faster than Bubble Sort. The second smallest element is the second element of the sorted array and the second-largest element is the second last element of the sorted array.

Denote the size of array as N. If N is odd, the median is $(\frac{N+1}{2})^{th}$ element of the sorted array; if N is even, the median is the average of $(\frac{N}{2})^{th}$ element and $(\frac{N}{2}+1)^{th}$ element. Be aware of the precision error for the case when N is even.

Time Complexity: $O(NlogN)$

3 Meow Tree

Author: Amos Tan Li Sheng

Abridged Statement

Given a rooted tree, find the depth of all nodes and the number of leaves of the tree.

Solution

We use Depth-First Search (DFS) once to traverse all the nodes from the root. We set the depth of the root node to be 0, and the depth of every other node to be 1 + the depth of its parent.

This also can be solved by using Breath-First Search (BFS). Note that the total depth of the tree may exceed 32-bit.

To check if a node is a leaf, we can check if the degree of node (the number of connected edges) is 1.

Time Complexity: $O(N)$

4 Meow String

Author: Amos Tan Li Sheng

Abridged Statement

Given a string S. For each query, find the number of occurrence of `"ME"` in the substring of the S, starting from index L and ending at index R.

Solution

We cannot count the number of occurrence of `"ME"` for the substring of each query naively because the number of operations is too large.
(Time complexity : $O(NQ)$)

Let us use our knowledge of basic Dynamic Programming and prefix sums to solve this problem. We can set $dp(i)$ as the number of occurrence of `"ME"` from index 1 to index i.

$$dp(i) = dp(i-1) + (S[i-1] == \text{`}\texttt{M}\text{'}\,\&\&S[i] == \text{`}\texttt{E}\text{'})$$

Then for each query, the number of occurrence of `"ME"` is $dp(R) - dp(L)$, which can be calculated in $O(1)$ time.

Time Complexity: $O(N + Q)$

5 Meow Products

Author: Amos Tan Li Sheng

Abridged Statement

Given the prices of Product A, Product B, Product C, and Product D as a, b, c, and d respectively, find the least amount of money to purchase x Product As, y Product Bs, and z Product Cs.
Note that you could turn 3 Product Ds into one Product A, one Product B, and one Product C and it is okay to purchase more than the required number of products.

Solution

We need to check whether is it optimal to purchase Product D. Note that if we do, we always purchase the number of Product D in multiples of 3, which we will then turn each of 3 Product Ds into one Product A, one Product B, and one Product C.

There are two cases where we would purchase Product Ds.

The first case is when $3d < a + b + c$, it is cheaper to buy 3 Product Ds than buying one Product A, one Product B, and one Product C at their own prices.

The other case is when one of the products (excluding Product D) is too expensive. Take Product A as an example. If $3 \times d < a$, we will buy 1 Product A at the cost of 3 Product Ds, and buy the leftover products at their own prices.

We will check for these 2 cases and obtain the minimum price possible.

This problem can also be solved with brute force on the number of Product D to be bought (in multiples of 3) and to purchase the remaining products at their own prices, which will take $O(max(x, y, z))$ time.

Time Complexity: $O(1)$

6 Bad Genius Meow

Author: Chooi He Lin

Abridged Statement

Given an array of integers, each representing the number of cats sitting for the exam in the i^{th} dorm. Find the minimum energy usage of the Robot Cheetah Meow to deliver all the exam papers. The Robot Cheetah Meow will travel from its charging station to the a^{th} dorm, take one exam paper, move from the a^{th} dorm to the b^{th} dorm, place the paper at b^{th} dorm, and move back to the charging station.

Solution

Denote the position of the charging station as the k^{th} dorm. For each Meow who takes exam in the i^{th} dorm, we requires Robot Cheetah Meow to send the exam paper from the i^{th} dorm to the 1^{st} dorm (Genius Meow) and deliver back to the i^{th} dorm. Here is the details for the delivery.

Send the paper from i^{th} dorm to Genius Meow
k^{th} dorm (charging station) $\Rightarrow$ i^{th} dorm (get paper) $\Rightarrow$ 1^{st} dorm (Genius Meow) $\Rightarrow$ k^{th} dorm (charging station)

Get the paper from Genius Meow to i^{th} dorm
k^{th} dorm (charging station) $\Rightarrow$ 1^{st} dorm (get paper) $\Rightarrow$ i^{th} dorm (the Meow) $\Rightarrow$ k^{th} dorm (charging station)

Either it is to send the paper to Genius Meow or to get the paper from Genius Meow, the energy needed for Robot Cheetah Meow is $|k-i|+|i-1|+|k-1|$. If i is larger than k, the energy needed will be $2 \times (i-1)$. In contrast, if k is larger than i the energy needed will be $2 \times (k-1)$. This equals to $2 \times (max(i,k)-1)$.

It is optimal to choose the charging station to be placed between the 1^{st} dorm and the k^{th} dorm and the energy consumed for either sending paper or getting paper at the i^{th} dorm is $2 \times (i-1)$. Since, the location of charging station is fixed (once placed, it cannot be moved), it is optimal to place the charging station at the 1^{st} dorm. From what we derived previously, $2 \times (max(i,k)-1)$, we can bolster our statement to place the charging station at the 1^{st} dorm ($k = 1$), the energy needed for either sending paper or getting paper for each dorm is $2 \times (max(i,k)-1) \to 2 \times (i-1)$. For the entire delivery for the paper (sending and getting it back), the energy needed is $2 \times (i-1) + 2 \times (i-1) = 4 \times (i-1)$. There are a_i students in the i^{th} dorm, hence the total energy needed for i^{th} dorm is $4 \times (i-1) \times a_i$. The solution for this problem is $4 \times \sum_{i=1}^{n}(i-1) \times a_i$.

Time Complexity: $O(N)$

7 The Poly-tical Question

Author: Ong Jack Min

Abridged Statement

Given the coefficients of a polynomial and 2 possible x. Find the x that yields the highest value for the equation modulo 32000011.

Solution

There are 2 key concepts required to be able to pass the time constraint given in the question.

1. $O(N)$ evaluation of polynomial equation can be done using Horner's method.
2. Modulo is distributive under multiplication:

$$(a \bmod c) \times (b \bmod c) = (ab \bmod c)$$

We should also make sure the data type used can store the intermediate values without overflow. We must also make sure that the answer is always positive as the modulo operator can yield negative value in some languages.

Time Complexity: $O(N)$

8 Catastrophe

Author: Ong Jack Min

Abridged Statement

Given a multiset (set but without the unique condition) of integers N, a floored-multiply function that maps every element to 2.718 floored of itself, i.e. $x \to \lfloor 2.718x \rfloor$. And a resolve operation that splits all elements x where $x > C$ into the elements $\lfloor \frac{x}{2} \rfloor$ and $\lceil \frac{x}{2} \rceil$. Find the size of the multiset after M iterations of the composite of these 2 operations.

Solution

The first thing to identify when we read the bounds is that it is never a good idea to store the elements of the multiset because the cat growth rate is exponential. The key inference here is that the number of unique elements in the multiset is at most $\lfloor 2.718C \rfloor$. Thus, if we only store the cardinality of the elements (number of times the element occurs in the multiset), the multiset can be represented as a map (associative array/hashmap/hashtable/dictionary) of key value pairs where the key is the unique element and the value is its cardinality.

The two operations can easily be modified to operate on a set of key value pairs. The floored-multiply function becomes $(key, value) \to (\lfloor key \times 2.718 \rfloor, value)$ and the resolve operation becomes split all pairs $(key, value)$ where $key > C$ into the pairs $(\lfloor \frac{x}{2} \rfloor, value)$ and $(\lceil \frac{x}{2} \rceil, value)$.

The operation to reduce the multiset from its element representation to cardinality map representation takes $O(N)$ time. We then should cast the resolve operation once to maintain an assumption made by the floored-multiply operation. The resolve function need only iterate from $max(\lfloor 2.718C \rfloor, \lfloor 2.718 \times 100 \rfloor)$ to $C + 1$ as the operation only acts on elements greater than or equal to C.

The floored-multiply function then need only go through C iterations as it is safe to assume that the resolve operation reduces the unique elements to less than C before the floored-multiply function. We then cast the resolve operation to rehouse the cats and sum the values from keys 1 to C to obtain the number of houses for the month.

The 3 functions will be cast M times yielding a time complexity of $O(MC)$ for this stage. Thus, the total time complexity is $O(N + MC)$.

We should also note that we must use big integer data pointers to store the value in our map as the worst case sum can contain 85 decimal digits and requires 282 bits to store.

As the elements are all integers with value at most $\lfloor 2.718C \rfloor$, a hashtable with no hash function would suffice as the map data structure. This is simply a C++ Vector/Java Array/Python List. We can use two Vectors to represent the multiset and iterate with one as the left multiset before the operations and the other as the right multiset after the operations. We then swap left and right after each iteration and clear the right vector at the start of the next iteration. We can also achieve the desired results with one Vector by enumerating the resolve operation and floored-multiply function in descending order. We can prove this works by realizing that the dependencies always point to the left of the number line in the partial order. Thus, a descending order resolution always forms a topological order.

Time Complexity: $O(N + MC)$

9 Meow-tain View (Easy Version)

Author: Amos Tan Li Sheng

Abridged Statement

Given N mountains arranged in a row with their unique heights, find the number of mountains that can be viewed from the i^{th} mountain's peak. At the i^{th} mountain, you can only view the j^{th} mountain if and only if there is no mountain taller than the j^{th} mountain between the i^{th} mountain and j^{th} mountain (non-inclusive of the i^{th} and j^{th} mountains). You can always view the i^{th} mountain itself when you are viewing from the i^{th} mountain.
This is the easy version of the problem by having $N \leq 10^3$.

Solution

Since N is small, we could implement a $O(N^2)$ solution.

Let us use two pointers: the 1^{st} pointer points to the current mountain and the 2^{nd} pointer points the mountain to view.

Now, we look at the case when the 2^{nd} pointer points to the left. When we have our first pointer pointing to i^{th} mountain, let the 2^{nd} pointer points to the mountains to its left: $(i-1)^{th}$ mountain, $(i-2)^{th}$ mountain, $(i-3)^{th}$ mountain, ..., 1^{st} mountain.

We keep track of the maximum height that we encounter. If the height of mountain pointed to by the 2^{nd} pointer is shorter that the maximum height we have encountered previously, this means we can't view the mountain of the 2^{nd} pointer from the mountain of the 1^{st} pointer. In order words, there is a taller mountain in between the 1^{st} and 2^{nd} pointers that blocks our view.

If the height of mountain is greater than the maximum height that we have encountered, this means from the mountain that first pointer points, you could view the mountain that the second pointer points without any mountain blocking our view.

We can use the same approach for when the 2^{nd} pointer points to the right of the 1^{st} pointer. We also need to add 1 to number of mountains that can be viewed because the you can view the mountain itself.

Time Complexity: $O(N^2)$

10 Meow-tain View (Hard Version)

Author: Amos Tan Li Sheng

Abridged Statement

Given N mountains arranged in a row with their unique heights, find the number of mountains that can be viewed from the i^{th} mountain's peak. At the i^{th} mountain, you can only view the j^{th} mountain if and only if there is no mountain taller than the j^{th} mountain between the i^{th} mountain and j^{th} mountain (non-inclusive of the i^{th} and j^{th} mountains). You can always view the i^{th} mountain itself when you are viewing from the i^{th} mountain.
This is the hard version of the problem by having $N \leq 2 \times 10^5$.

Solution

Since N is large, $O(N^2)$ will definitely give us Time Limit Exceeded. Notice that when we stand on a mountain, and view the mountains either on our left or on our right, the heights of all the mountains we can see are increasing. Hence we can keep the heights in a Stack.

We need to iterate from left to right, and from right to left.
From left to right, all the heights stored in the Stack are the mountains that we could view to the left of the current mountain. Because the number of heights in the stack equals the number of mountains to the left that we can see, let us add the size of the stack to the answer for the current mountain. Then we consider if current mountain's height will block the view of the previous heights when viewed from right to left. If the height of the mountain at the top of the stack is lower than the height of current mountain, the current mountain will obviously block it when viewed from the right, and so we need to remove it. After that, we will push current height to the Stack.

The idea is the same when finding the number of mountains we can view to the right of us (iterate from right to left). We also need to add 1 to number of mountains that can be viewed because the you can view the mountain itself.

Time Complexity: $O(N)$

11 Meow Party

Author: Amos Tan Li Sheng

Abridged Statement

Given N friends, each of them having the unique *meowness* values. There are Q queries, type 1 query is to merge the two friends' group while type 2 is to query who has the lowest value and highest value in the group.

Solution

This problem can be solved using Disjoint Set Union Data Structure.

For each disjoint set, let us store the values *lowest* and *highest*, denoting the highest and lowest meowness value among the cats in each set.

For queries of type 1, we will modify the *unite* function when merging 2 disjoint sets as such:
$lowest_{New} = (meowness[lowest_A] < meowness[lowest_B]?lowest_A : lowest_B)$
$highest_{New} = (meowness[highest_A] > meowness[highest_B]?highest_A : highest_B)$
where A and B are the representatives of the 2 sets we want to merge.

For queries of type 2, we can just find the representative of the given cat, and output who has the *lowest* value and and who *highest* value.

Remember to use either path-compression or small-to-large merging when implementing the Disjoint-Set Union functions to bring down the time-complexity from $O(N^2)$ to $O(NlogN)$.

Time Complexity: $O(NlogN)$