Homework

Solutions

1. (33 points) **Tree Cutting.** You are given a tree of n vertices, rooted at vertex 0. Consider the following game: every turn, you pick a vertex uniformly at random from the remaining tree and cut the node, together with the node's subtree. The procedure stops when the whole tree is cut (i.e., the root is cut).

   Design an algorithm that computes the expected total number of cuts when the procedure stops, in linear time.

   Note. If you are able to solve the case where the graph is a line: $0 \to 1 \to 2 \to \cdots \to n-1$ in at most linear time, you will get 20 points. This can be a good starting point.

   Hint. (Linearity of expectation). Define e_v as a random variable indicating whether v is ever chosen before the whole tree is cut.

   First, let us look at the simpler case where the graph is a line: $0 \to 1 \to 2 \to \cdots \to n-1$.

   Like the course material, we define e_v, $e_v = 1$ if node v is selected before any of its ancestors (nodes 0 to $v-1$), and $e_v = 0$ otherwise. The objective, which is the expected total number of cuts required to remove the entire tree, is lists as follows:

$$E = \mathbb{E}\left[\sum_{v=0}^{n-1} e_v\right].$$

   The reason that we can calculate the objective in such way is because for each node, we either make it responsible for one of these cuts or it is already been removed as part of a subtree when an ancestor is cut.

   Now we need to calculate $\mathbb{E}[e_v]$ for each node:

$$E = \mathbb{E}\left[\sum_{v=0}^{n-1} e_v\right] = \sum_{v} \mathbb{E}[e_v].$$

   Under the simpler case, we have the circumstance that the set of node from root 0 to node v, inclusive, has depth(v)+1 nodes. Since each node is equally likely to be selected at each turn (uniform random selection), the probability that node v is the first among these $v+1$ nodes to be selected is:

$$\mathbb{P}(e_v = 1) = \frac{1}{v+1},$$

   which means, the probability that node v is removed by ancestors is:

$$\mathbb{P}(e_v = 0) = 1 - \frac{1}{v+1}.$$

As a result, the expected total number of cuts is:

$$E = \sum_{v=0}^{n-1} \mathbb{E}[e_v] = \sum_{v=0}^{n-1} \frac{1}{v+1}.$$

It is easily seen that the time we need to calculate E is O(N).

Second, let us look at the general case. We now have an arbitrary rooted tree with n nodes. For each node v, we define depth(v) as its depth (the number of edges from the root 0 to node v).

Same as the simpler case, we define e_v, $e_v = 1$ if node v is selected before any of its ancestors (nodes 0 to $v-1$), and $e_v = 0$ otherwise.

The objective is also the same:

$$E = \mathbb{E}\left[\sum_{v=0}^{n-1} e_v\right].$$

To calculate the probability that a node would be selected to remove trees, which means not to be removed because of its ancestors, now we only need to consider the path from node 0 to node v, in other words, this node to be selected under this path:

$$\mathbb{P}(e_v = 1) = \frac{1}{\text{depth}(v)+1}.$$

To explain more specifically, since these depth(v)+1 nodes(ancestors and v) are the only ones that can prevent v from being selected first in its ancestral line, and selection is uniform. *Note: That is exactly what I mean by path :)*

As a result, the expected total number of cuts is:

$$E = \sum_{v} \mathbb{E}[e_v] = \sum_{v} \frac{1}{\text{depth}(v)+1}.$$

Now the algorithm is also obvious, we just need to pre-compute the depth of each nodes and preform the *Linearity of expectation*. I list the steps as follows:

1. Perform DFS to computes depth for all nodes.
2. For each node, compute $\frac{1}{\text{depth}(v)+1}$ and sum them up.
3. The sum obtained is the expected number of cuts required to remove the entire tree.

All the steps is O(N) so we get a linear time approach.

2. (33 points) **Picking Teams.** You have n people that you want to divide into two teams, A and B. Every person needs to be included in exactly one of the teams. The i-th person has two associated prices:

   - a_i – the price to include them in team A
   - b_i – the price to include them in team B

   You also learn that there are m pairs of friends among these n people, which are specified by (u_j, v_j, c_j)[1] where we guarantee that $u_j \neq v_j$. If persons u_j and v_j are assigned to different teams, you have to pay the price c_j. You can assume all a_i, b_i, c_j are positive.

   Use a network flow algorithm to find an optimal way to divide these people into teams A and B, i.e. paying the minimum total cost. You can use any network flow algorithm, and as long as the graph you construct is polynomial-sized in n and m, you will receive full credit for time complexity.

   Answer:

   To solve the problem, we construct a flow network as follows:

   Graph Construction

   1. Nodes:
      - Add a source node s and a sink node t.
      - For each person i, create a node p_i.
   2. Edges:
      - Add an edge (s, p_i) with capacity a_i (cost of assigning i to team A).
      - Add an edge (p_i, t) with capacity b_i (cost of assigning i to team B).
      - For each pair of friends (u_j, v_j, c_j), add an undirected edge (p_{u_j}, p_{v_j}) with capacity c_j (penalty for assigning u_j and v_j to different teams).

   Correctness of the Reduction

   Claim: We claim that he minimum s-t cut of the graph corresponds to the minimum cost of dividing people into teams A and B.

   Proof. 1. Assigning person i to team A corresponds to "cutting" the edge (s, p_i), incurring a cost of a_i. Similarly, assigning i to team B corresponds to cutting the edge (p_i, t), incurring a cost of b_i. 2. If two friends u_j and v_j are assigned to different teams, the edge (p_{u_j}, p_{v_j}) is cut, incurring a penalty of c_j. 3. By the max-flow min-cut theorem, the value of the minimum s-t cut equals the maximum flow in the network, which minimizes the total cost. □

   Algorithm and Pseudocode We solve the problem by running a max-flow algorithm such as Edmonds-Karp. Here is the pseudocode:

   ```
   Input: Costs a_i, b_i for each person i, penalties c_j for m friend pairs
   Output: Minimum total cost of dividing people into teams
   ```

[1] We assume the same pair of friends will only appear once; but a person may have multiple friends.

```
1. Construct a graph G as follows:
   - Add source node s and sink node t.
   - Add edges (s, p_i) with capacity a_i.
   - Add edges (p_i, t) with capacity b_i.
   - Add edges (p_u, p_v) with capacity c_j for all friend pairs (u, v).

2. Run Edmonds-Karp to compute the maximum flow.
3. The minimum cost is the value of the minimum s-t cut,
    which is the same as the maximum flow.

Return minimum cost.
```

Complexity Analysis

1. Graph Size: - There are $n + 2$ nodes (one for each person, plus s and t). - There are $2n + m$ edges:

- n edges from s to p_i,
- n edges from p_i to t,
- m edges between pairs of friends.

Therefore, the graph size is polynomial in n and m.

2. Time Complexity: - Using Edmonds-Karp, the time complexity is $O(VE^2)$, where V is the number of vertices and E is the number of edges. - In our case, $V = O(n)$ and $E = O(n + m)$, so the complexity becomes:

$$O((n)(n + m)^2).$$

- Since the graph is polynomial-sized in n and m, this algorithm satisfies the time complexity requirements.

3. (33 points) **Meet in the Middle.** We know the knapsack problem is NP-hard when volumes are allowed to be very large. In this question, you will consider a similar problem, which is also NP-hard.

You get n real numbers (positive or negative) whose absolute values may be very large. Design an algorithm running in expected time $O(2^{n/2} \times n)$ that decides whether there is a subset of numbers that sum up to 0.

Hint. Use a hash table to store all possible subset sums for the first $\frac{n}{2}$ elements.

Answer.
To solve this question efficiently, we can split the n numbers into two half, where A consist the first $\frac{n}{2}$ elements, and B consist the last $\frac{n}{2}$ elements. The key idea is to divide the problem into two smaller subset, solve each independently, then merge the results.

First, we compute all the possible subset sum of A, and store the result in a hash table,

where the key is the subset sum and the value can be anything, for example the list of subsets producing this subset sum. For elements in B, we just compute the subset sums are store in an array. If there are subset sum equals to 0 in either A or B, we can return there are a subset of number that sum up to 0 immediately. If not, for each of the computed subset sums s in B, we check if there is a complementary sum $-s$ in A using the hash table. If there is such $-s$ exist in hash table, there is a subset of numbers that sum up to 0 in the given n real numbers, else no such subset exist.

Algorithm summary

1. Split the input array *nums* into two halves:
   1.1 A (first $n/2$ elements) and B (remaining $n/2$ elements).
2. Compute all subset sums for A:
   2.1 Calculate all possible subset sums of A.
   2.2 Store the subset sums in a hash table, where the key is the subset sum, the value is the list of subsets that produce this sum(or anything).
3. Compute all subset sums for B:
   3.1 Calculate all possible subset sums of B.
   3.2 Store the subset sums in an array.
4. Check for subsets summing to 0:
   4.1 Check if 0 exists as a subset sum in either A or B.
   4.2 If 0 exists, return immediately that a subset exists with a sum of 0.
5. Find complementary sums:
   5.1 For each subset sum s in B, check if its complement $-s$ exists in the hash table of A.
   5.2 If such a complement exists, return that a subset exists with a sum of 0.
6. If no subset sums to 0 in either A, B, or across A and B, Return no such subset exists.

Proof of correctness

To prove completeness, we must show that the algorithm examines all possible subsets of the given n numbers and guarantees to find the subset with a sum of 0 if such a subset exists. Let $S(A)$ be all the subset sums of A and $S(B)$ be all the subset sums of B.

Case 1: The subset is entirely in A.

The algorithm explicitly computes all possible subset sums of A. If any subset of A sums to 0, then $0 \in S(A)$, and the algorithm will detect this in step 4.1.

Case 2: The subset is entirely in B.

The algorithm explicitly computes all possible subset sums of A. If any subset of B sums to 0, then $0 \in S(B)$, and the algorithm will detect this in step 4.1.

Case 3: The subset is split between in A and B.

Any subset of the given n numbers can be expressed as the union of a subset of A and a subset of B. For such a subset, the total sum is the sum of the subset sums from A and B:

$$\textbf{Subset Sum} = s_A + s_B$$

where $s_A \in S(A) \quad \text{and} \quad s_B \in S(B)$

Our algorithm iterate over all $s_B \in S(B)$ and checks whether $-s_B \in S(A)$. If such s_B exist, that means:

$$s_A + s_B = 0 \implies s_A = -s_B$$

Which ensures that all possible subset formed by combining the subset of A and B are checked.

Since all subset of the given n real numbers are either:
1. Fully in A
2. Fully in B
3. Split in A and B

Our algorithm has explicitly check all the possible subsets of the given n numbers, the algorithm is complete and correct.

Time Complexity Analysis

1. Split the input real numbers into two halves takes $O(1)$
2. Compute all subset sums for A:
   - 2.1 There are $\frac{n}{2}$ elements in A, so the number of subset in A is $2^{n/2}$, there are at most n element in a subset, so calculate all possible subset sums of A takes $O(2^{n/2} \cdot n)$.
   - 2.2 Store the subset sums in a hash table takes $O(2^{n/2})$ since there is $2^{n/2}$ subset sums in A.
3. Compute all subset sums for B:
   - 3.1 There are $\frac{n}{2}$ elements in B, so the number of subset in B is $2^{n/2}$, there are at most n element in a subset, so calculate all possible subset sums of B takes $O(2^{n/2} \cdot n)$.
   - 3.2 Store the subset sums in an array takes $O(2^{n/2})$ since there is $2^{n/2}$ subset sums in B.
4. Check if 0 exists as a subset sum in either A or B takes $O(1)$.
5. Find complementary sums:

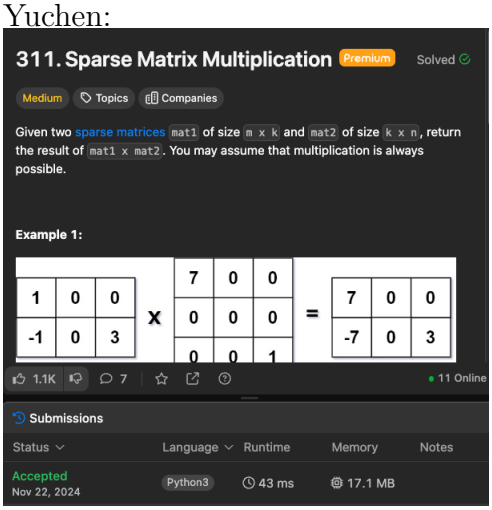
5.1 Hash table lookup takes $O(1)$ on average, there are $2^{\frac{n}{2}}$ subsets in B, so for each subset sum s in B, check if its complement $-s$ exists in the hash table of A takes $O(2^{\frac{n}{2}})$.

So the total Time Complexity is:

$$O(2^{n/2} \cdot n) + O(2^{n/2}) + O(2^{n/2}) = O(2^{n/2} \cdot n)$$

Coding Problem Record

Yuchen:



311. Sparse Matrix Multiplication Premium Solved

Medium Topics Companies

Given two sparse matrices `mat1` of size `m x k` and `mat2` of size `k x n`, return the result of `mat1 x mat2`. You may assume that multiplication is always possible.

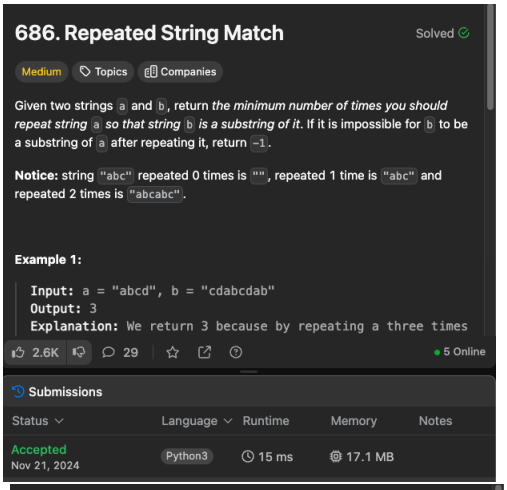
Example 1:

1	0	0		7	0	0		7	0	0
-1	0	3	x	0	0	0	=	-7	0	3
				0	0	1				

1.1K 7 11 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted Nov 22, 2024	Python3	43 ms	17.1 MB	



686. Repeated String Match Solved

Medium Topics Companies

Given two strings `a` and `b`, return *the minimum number of times you should repeat string `a` so that string `b` is a substring of it*. If it is impossible for `b` to be a substring of `a` after repeating it, return `-1`.

Notice: string `"abc"` repeated 0 times is `""`, repeated 1 time is `"abc"` and repeated 2 times is `"abcabc"`.

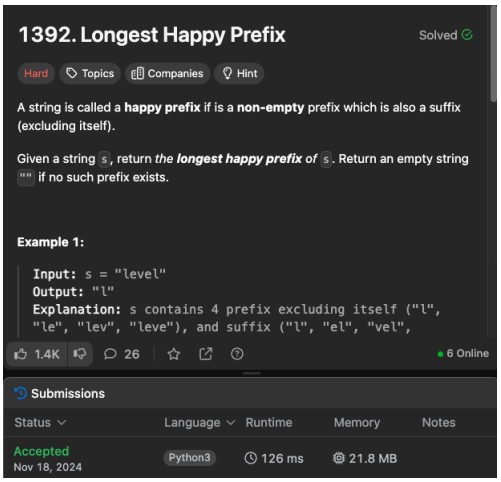
Example 1:

```
Input: a = "abcd", b = "cdabcdab"
Output: 3
Explanation: We return 3 because by repeating a three times
```

2.6K 29 5 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted Nov 21, 2024	Python3	15 ms	17.1 MB	



1392. Longest Happy Prefix Solved

Hard Topics Companies Hint

A string is called a **happy prefix** if is a **non-empty** prefix which is also a suffix (excluding itself).

Given a string `s`, return *the **longest happy prefix** of* `s`. Return an empty string `""` if no such prefix exists.

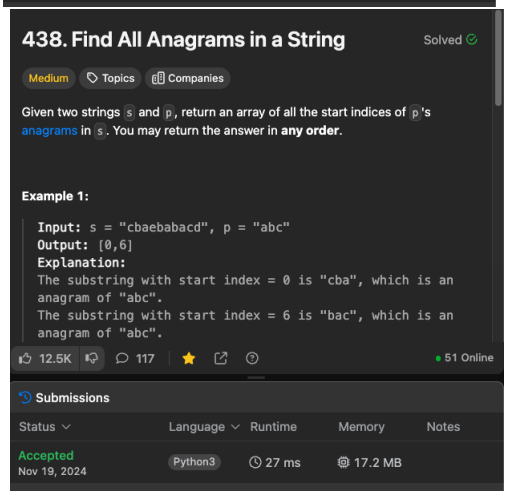
Example 1:

```
Input: s = "level"
Output: "l"
Explanation: s contains 4 prefix excluding itself ("l",
"le", "lev", "leve"), and suffix ("l", "el", "vel",
```

1.4K 26 6 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted Nov 18, 2024	Python3	126 ms	21.8 MB	



438. Find All Anagrams in a String Solved

Medium Topics Companies

Given two strings `s` and `p`, return an array of all the start indices of `p`'s anagrams in `s`. You may return the answer in **any order**.

Example 1:

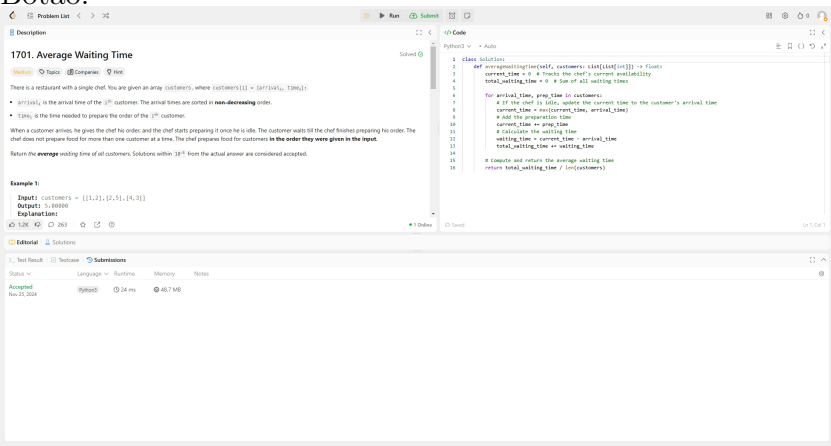
```
Input: s = "cbaebabacd", p = "abc"
Output: [0,6]
Explanation:
The substring with start index = 0 is "cba", which is an
anagram of "abc".
The substring with start index = 6 is "bac", which is an
anagram of "abc".
```

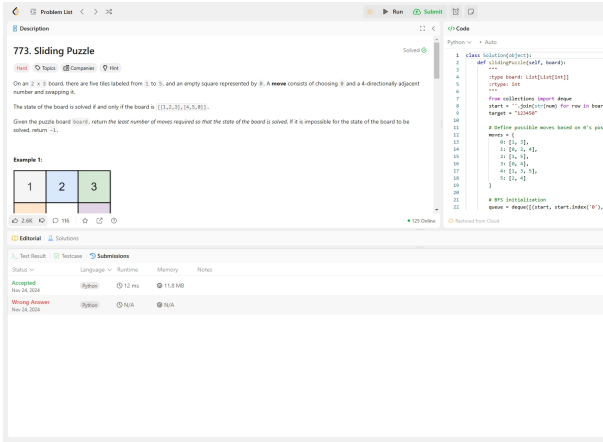
12.5K 117 51 Online

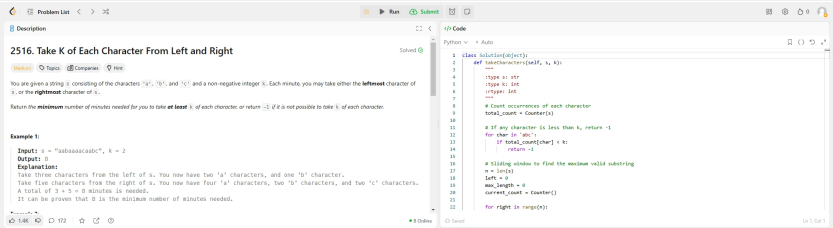
Submissions

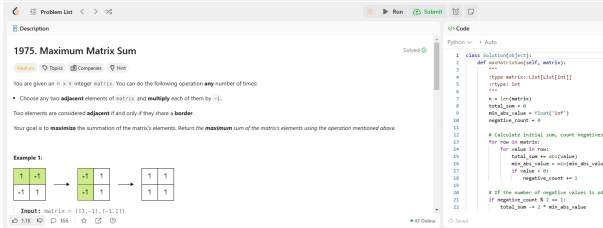
Status	Language	Runtime	Memory	Notes
Accepted Nov 19, 2024	Python3	27 ms	17.2 MB	

Botao:


1701. Average Waiting Time
class Solution:
def averageWaitingTime(self, customers: List[List[int]]) -> float:
current_time = 0 # Tracks the chef's current availability
total_waiting_time = 0 # Sum of all waiting times
for arrival_time, prep_time in customers:
# If the chef is idle, update the current time to the customer's arrival time
current_time = max(current_time, arrival_time)
# Add the preparation time
current_time += prep_time
# Calculate the waiting time
waiting_time = current_time - arrival_time
total_waiting_time += waiting_time
Compute and return the average waiting time
return total_waiting_time / len(customers)


773. Sliding Puzzle
1
2
3


Accepted
Nov 25, 2024
Python3
24 ms
48.7 MB


Accepted
Nov 24, 2024
Python
12 ms
11.8 MB
Wrong Answer
Nov 24, 2024
Python
N/A
N/A

2516. Take K of Each Character From Left and Right
class Solution(object):
def takeCharacters(self, s, k):
:type s: str
:type k: int
:rtype: int
# Count occurrences of each character
total_count = Counter(s)
# If any character is less than k, return -1
for char in 'abc':
if total_count[char] < k:
return -1
# Sliding window to find the maximum valid substring
n = len(s)
left = 0
max_length = 0
current_count = Counter()
for right in range(n):
Accepted
Nov 20, 2024
Python
1027 ms
15.7 MB
Accepted
Nov 20, 2024
Python
1020 ms
15.6 MB
Accepted
Nov 20, 2024
Python
1062 ms
15.6 MB

1975. Maximum Matrix Sum
class Solution(object):
def maxMatrixSum(self, matrix):
:type matrix: List[List[int]]
:rtype: int
n = len(matrix)
total_sum = 0
min_abs_value = float('inf')
negative_count = 0
# Calculate initial sum, count negatives
for row in matrix:
for value in row:
total_sum += abs(value)
min_abs_value = min(min_abs_value
if value < 0:
negative_count += 1
# If the number of negative values is od
if negative_count % 2 == 1:
total_sum -= 2 * min_abs_value
Accepted
Nov 22, 2024
Python
83 ms
17.6 MB

Yuheng:

Description | Accepted ✕ | Editorial | Solutions | Submissions

128. Longest Consecutive Sequence

Solved

Medium | Topics | Companies

Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in `O(n)` time.

Example 1:

```
Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.
```

Description | Accepted ✕ | Editorial | Solutions | Submissions

212. Word Search II

Solved

Hard | Topics | Companies | Hint

Given an `m x n board` of characters and a list of strings `words`, return *all words on the board.*

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

Example 1:

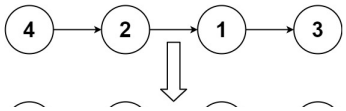
Description | Accepted ✕ | Editorial | Solutions | Submissions

148. Sort List

Medium | Topics | Companies

Given the `head` of a linked list, return *the list after sorting it in* ***ascending order***.

Example 1:



45. Jump Game II

Medium | Topics | Companies

You are given a **0-indexed** array of integers `nums` of length `n`. You are initially positioned at `nums[0]`.

Each element `nums[i]` represents the maximum length of a forward jump from index `i`. In other words, if you are at `nums[i]`, you can jump to any `i + j` where:

- `0 <= j <= nums[i]` and
- `i + j < n`

Peng Yao:

32. Longest Valid Parentheses

Solved

Hard Topics Companies

Given a string containing just the characters `'('` and `')'`, return *the length of the longest valid (well-formed) parentheses substring*.

12.6K 80 13 Onli

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	0 ms	10.3 MB	

87. Scramble String

Solved

Hard Topics Companies

We can scramble a string s to get a string t using the following algorithm:

1. If the length of the string is 1, stop.
2. If the length of the string is > 1, do the following:
   - Split the string into two non-empty substrings at a random index, i.e., if the string is `s`, divide it to `x` and `y` where `s = x + y`.
   - **Randomly** decide to swap the two substrings or to keep them in the same order. i.e., after this step, `s` may become `s = x + y` or `s = y + x`.
   - Apply step 1 recursively on each of the two substrings `x` and `y`.

Given two strings `s1` and `s2` of **the same length**, return `true` if `s2` is a scrambled string of `s1`, otherwise, return `false`.

3.4K 82 6 On

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	135 ms	34.7 MB	

132. Palindrome Partitioning II

Solved

Hard Topics Companies

Given a string `s`, partition `s` such that every substring of the partition is a palindrome.

Return *the* ***minimum*** *cuts needed for a palindrome partitioning of* `s`.

5.6K 56 6 Onli

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	55 ms	13.9 MB	

115. Distinct Subsequences

Solved

Hard Topics Companies

Given two strings s and t, return *the number of distinct* ***subsequences*** *of s which equals* t.

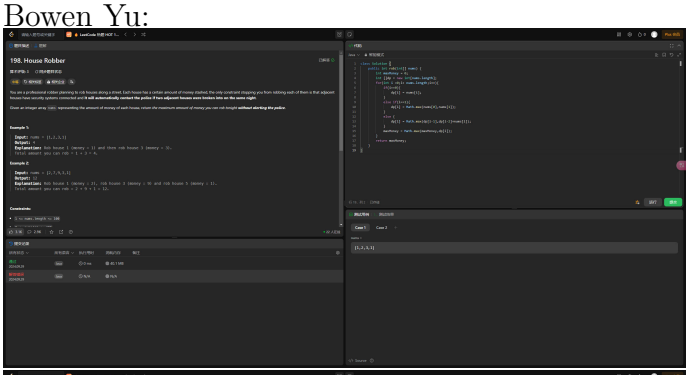
The test cases are generated so that the answer fits on a 32-bit signed integer.

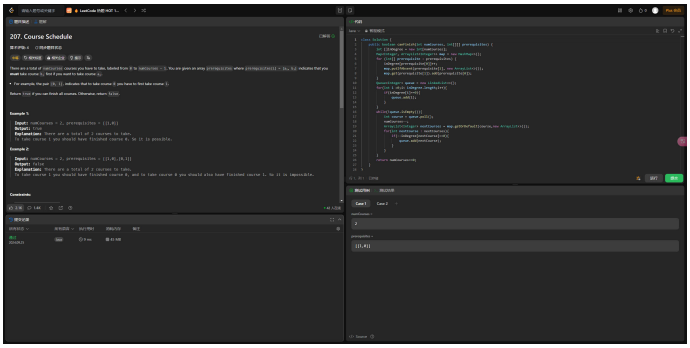
6.8K 118 27 Onli

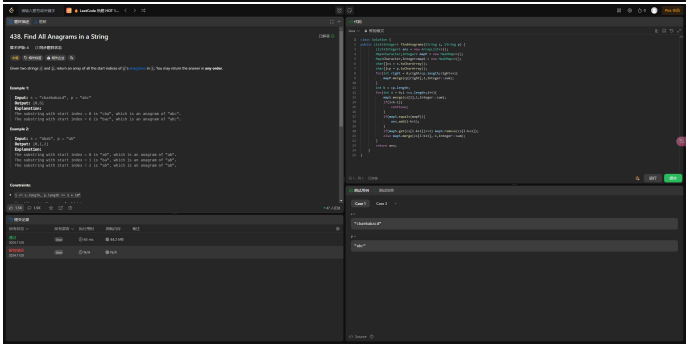
Submissions

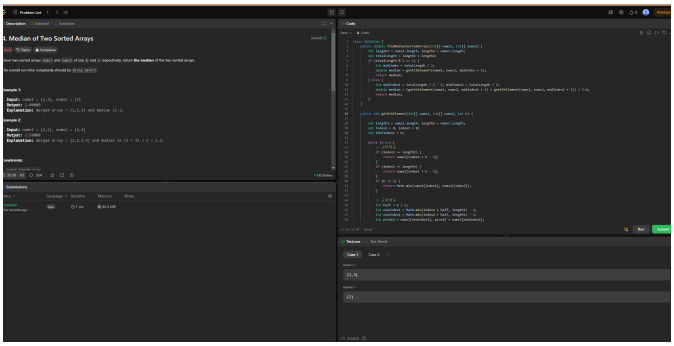
Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	27 ms	13.1 MB	

Bowen Yu:


198. House Robber


207. Course Schedule


438. Find All Anagrams in a String


4. Median of Two Sorted Arrays