Homework 3

Group Members

Yuchen Gong, Yuheng Ge, Yao Peng, Bowen Yu, Botao Zhang

Solutions

1. (25 points) Suppose each edge of an undirected graph G is associated with a cost function $f_e(t) = a_e t^2 + b_e t + c_e$ where $a_e > 0$ (think of t as a time parameter and $f_e(t)$ as the length of e at time t). Given the graph G and the values $\{a_e, b_e, c_e\}$ as input, give an algorithm that returns a value t at which the minimum spanning tree has a minimum cost (over all time t). Assume that the arithmetic operations on $\{a_e, b_e, c_e\}$ can be done in constant time per operation. Any correct algorithm with time complexity polynomial in $|V|$ and $|E|$ gets full credits; i.e., $|V|^a \cdot |E|^b$ for constant a, b.

   Answer:

   Our objective is to find the time t at which the minimum spanning tree of an undirected graph G has the lowest possible cost over time. The cost of edge will be change respect to time t, this means the structure of MST can be different with different t value. Recalling the interesting properties we discuss in class, when the weight of edges changes, if the sorted order of edges weight remains, the MST remains the same. This means rather than computing the cost of MST for every possible t, we only need to compute the cost whenever the relative order of the edge weight changes.

   One possible solution is to first identify the critical points where the order of edges might change due to their cost function. Therefore, for each pair of edges (e, e') in graph G, we find the critical point at $f_e(t) = f_{e'}(t)$ where the order of edge weight could change. Once we have all the critical point we can sort them in ascending order.

   After we have sort the critical point, for each critical point t, we sort all the edges according to their cost function $f_e(t)$, then compute the MST using Kruskal's algorithm at current t. Let the total cost of the MST at time t be $C_t(t)$. Notice that in a time interval (t_{i-1}, t_i), the structure of MST remains the same until time $t > t_i$, so we define The MST cost function within each interval as:

   $$C_{t_i}(t) = \sum_{e \in MST} (a_e t^2 + b_e t + c_e)$$

   which is also a quadratic function.

   In addition, in each time interval, the structure of MST remains the same, but the total cost might changes, so for each t, we calculate the local minimum of each time interval by differentiating the quadratic equation $C_{t_i}(t)$ respect to t and solve $C'_{t_i}(t) = 0$. If such local minimum exist at some time t', calculate the cost $C_{t_i}(t')$. Additionally, we also evaluate $C_{t_i}(t)$ at each critical points. Finally, after we tracked all the minimum MST cost across each each interval and at the interval boundaries, we sort all the total cost $C_{t_i}(t)$ in order, and the smallest value is the global minimum MST cost, return the corresponding time t.

Proof of Correctness:

Claim 1: The only times at which the MST structure may change are at the critical points, where the order of edge weights may change. Within each interval (t_{i-1}, t), the MST structure remains stable and can be determined by the edge order at t_i

Proof: Giving the properties in lecture, the MST structure will change only when the order of edge weights changes. Consider any two edges e and e' in graph G with time-dependent cost function $f_e(t) = a_e t^2 + b_e t + c_e$, the order of weight change only happens when $f_e(t) = f_{e'}(t)$, solving this equation for t gives us the critical points where a reordering might occur.

The algorithm finds all critical points by solving$f_e(t) = f_{e'}(t)$ for each pair (e, e') and sorting them in ascending order. This ensures that we capture every possible change in the order of edge weights. Hence, every potential change in the MST structure is being evaluate.

Claim 2: Within each interval (t_{i-1}, t), the MST structure remains stable and can be determined by the edge order at t_i.

Proof: Using similar properties, Kruskal's algorithm depends solely on the relative order of edge weights, not their exact values. Since the order of edge weights is stable within each interval,the MST structure remains constant in each interval as well.

By computing the MST structure at the right endpoint t_i of each interval (t_{i-1}, t), the algorithm ensures that this structure is valid for the entire interval.

Claim 3: The algorithm correctly finds the minimum MST cost within each interval (t_{i-1}, t)

Proof: Within each interval (t_{i-1}, t) the MST structure is stable, so the total cost of the MST is a quadratic function of t is given by:

$$C_{t_i}(t) = \sum_{e \in MST} (a_e t^2 + b_e t + c_e)$$

Since $C_{t_i}(t)$ will also be a quadratic function,it is continuous and differentiable. The minimum of a quadratic function on an interval occurs either at a local minimum within the interval or at the endpoints of the interval.

The algorithm differentiates $C_{t_i}(t)$ with respect to t and compute $C'_{t_i}(t) = 0$ to find the any local minimum within the interval (t_{i-1}, t). If a local minimum t', exists within the interval, it is added to the list of candidate times. The algorithm also considers the endpoints t_{i-1} and t, ensuring that the minimum MST cost within the interval (t_{i-1}, t) is correctly evaluated.

Claim 4: The global minimum MST cost occurs at one of the evaluated candidate times, either at a critical point or a local minimum within an interval.

Proof: Since we have identified all intervals(t_{i-1}, t) and calculated the MST cost function $C_{t_i}(t)$ within each interval, we have accounted for every possible configuration of the MST across all times t, By evaluating the cost function $C_{t_i}(t)$ at every local minimum and at every critical point, the algorithm ensures that the global minimum MST cost is included among the evaluated values.

The final step of the algorithm compares all calculated MST costs and selects the smallest one and return the corresponding time t. Therefore, the algorithm correctly identifies the global minimum MST cost over time, as it evaluates every possible configuration that could yield a lower cost.

Time Complexit analysis:

Let E be the number of edges in graph G

For each pair of edges (e, e'), we solve $f_e(t) = f_{e'}(t)$, to find the critical point, there are $O(|E|^2)$ pairs of edges, solving the equation takes constant time $O(1)$, thus finding critical point takes $O(|E|^2)$ time. Sorting this critical point takes $O(|E|^2 \log |E|)$.

For each critical point, we calculate the cost $f(e)$ for each edge, which is $O(|E|)$, sorting these edges takes $O(|E| \log |E|)$, running Kruskal's algorithm takes $O(|E| \log |E|)$. Since we repeat this for each critical point and there are $O(|E|^2)$ critical points, the total time for this step is $O(|E|^3 \log |E|)$.

There are $O(|E|^2)$ time interval, finding local minimum of each takes $O(E)$ since we need to define the MST cost function, which is a sum of quadratic function for each edge in MST. So finding local minimum takes total of $O(|E|^3)$. For the Final calculation of all MST cost at all local minima and interval boundaries, there are $O(|E|^2)$ possible time, finding the minimum value takes $O(|E|^2)$ by iterating through the list of MST cost.

Hence the final time complexity is $O(|E|^3 \log |E|)$.

2. (25 points) **Periodic scheduling**. There are n tasks. Each task takes one unit of time to perform, and the machine can process at most 1 task in each time slot. The requirement is that the task i should be scheduled once in each time period p_i. For example, if task i has a period p_i, we must schedule it in each time-window of the form $[p_i j + 1, \ldots, p_i(j+1)]$. Let $L = LCM(p_1, p_2, \ldots, p_n)$ (LCM stands for least common multiple). Answer the following questions.

   (a) (10 points) Prove that if $\sum_i \frac{1}{p_i} > 1$, there is no periodic schedule (i.e., you can not find a schedule for the first L time slots).

   First, consider the expression $\sum_i \frac{1}{p_i} > 1$, we multiply both sides by L:

$$\sum_i \frac{L}{p_i} > L$$

   Then, consider the term $\frac{L}{p_i}$, it presents the total time slots that $task_i$ needs.

   By summing over all p_i, $\sum_i \frac{L}{p_i}$ means that total time slots that all tasks need.

   If $\sum_i \frac{1}{p_i} > 1$, which means $\sum_i \frac{L}{p_i} > L$, then the total demand of time for all tasks exceeds the capacity L, then there will be no periodic schedule.

   (b) (15 points) Suppose $\sum_i \frac{1}{p_i} \leq 1$, then prove the following greedy algorithm can find a feasible schedule for the first L time slots: At each time slot, schedule the unfulfilled task with the nearest deadline (if there are several, do it in an arbitrary order).

   Hint: consider the first task that can not be fulfilled by the algorithm, and prove this case can not happen if $\sum_i 1/p_i \leq 1$.

   According to the greedy algorithm, at each time slot, schedule the unfulfilled task with nearest deadline, the there will be at most L time slots needed since it at most schedule one task at each time slot.

   Since $\sum_i \frac{1}{p_i} \leq 1$, then $\sum_i \frac{L}{p_i} \leq L$ which means that there will be enough time slots to schedule all the tasks.

   Case 1:$\sum_i \frac{1}{p_i} = 1$

   $\sum_i \frac{1}{p_i} = 1$, then $\sum_i \frac{L}{p_i} = L$ which implies that the total demand over L is equal to L. Every slot is occupied with no idle time.

   Let t_k be the first task that can not be fulfilled by the algorithm.
   Let d_k be the deadline of t_k.($d_k < L$)

Contradiction:
Since t_k missed its deadline, it means that from the beginning(slot 0) to d_k, there are not enough time slots to schedule the tasks since slot 0 to d_k are all occupied and t_k is still not scheduled. In other words, the total demand of time slots from slot 0 to d_k exceeds to system capacity from slot 0 to d_k.

However, given $\sum_i \frac{1}{p_i} = 1$, the system capacity should exactly matches the total demand of all tasks. Otherwords, the total demand of time slots from slot 0 to d_k exactly matches the system capacity from slot 0 to d_k

Therefore, the assumption that t_k is the first task that can not be fulfilled by the algorithm contradicts with the condition that $\sum_i \frac{1}{p_i} = 1$. It means that there should be enough space to schedule all tasks without missing any deadlines.

Case 2:$\sum_i \frac{1}{p_i} < 1$
$\sum_i \frac{1}{p_i} < 1$, then $\sum_i \frac{L}{p_i} < L$ which means that the total demand time of all tasks is less than L.There will be some idle slots.

Let t_k be the first task that can not be fulfilled by the algorithm.
Let d_k be the deadline of t_k.$(d_k < L)$

Contradiction:
Since t_k missed its deadline, it means that from the beginning(slot 0) to d_k, there are not enough time slots to schedule the tasks that have deadlines in this interval. In other words, the total demand of time from slot 0 to d_k for those tasks exceeds the system capacity from slot 0 to d_k.

However, given $\sum_i \frac{1}{p_i} < 1$, the system capacity is greater than the total demand of all tasks. In other words, the system capacity from slot 0 to d_k either exactly matches the total demand of time from slot 0 to d_k or exceeds it. It is impossible for the total demand within this interval to exceed the system's capacity.

Therefore, the assumption that t_k is the first task that can not be fulfilled by the algorithm contradicts with the condition that $\sum_i \frac{1}{p_i} < 1$. It means that there should be enough space to schedule all tasks without missing any deadlines.

Conclusion:
Under the condition $\sum_i \frac{1}{p_i} \leq 1$, the greedy algorithm successfully schedules all tasks without missing any deadlines, which confirms that a feasible schedule exists for the first L time slots.

3. (20 points) Given a list of n natural (i.e., positive integers) numbers $d_1, \ldots, d_n$, show how to decide in polynomial time[1] whether there exists an undirected graph whose degrees are precisely the number $d_1, \ldots, d_n$ (G should not contain multiple edges between the same pair of vertices or "loop" edges with both endpoints equal to the same node). *Hint.* Assume $d_1 \geq d_2 \geq \cdots \geq d_n$, and try to obtain an undirected graph G' whose nodes have degrees $d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n$; the problem of deciding whether G exists reduces to whether G' exists. From there, obtain your greedy algorithm.

Answer:

Here is the algorithmn to decide whether such graph exists.

Algorithm 1 Algorithm for Graph Realizability

Require: A degree sequence $D = [d_1, d_2, \ldots, d_n]$ where $d_i \in \mathbb{N}$ for all i

Sort D in non-increasing order initially
while D is not empty **do**
    Remove any zeros from D
    if D is empty **then**
        return `True` {If all degrees are zero, the sequence is realizable}
    end if
    $d \leftarrow D[1]$ {Take the largest degree}
    Remove $D[1]$ from D
    if $d > |D|$ **then**
        return `False` {Not enough vertices to connect to}
    end if
    for $i = 1$ to d **do**
        $D[i] \leftarrow D[i] - 1$ {Decrement the degree of the next d vertices}
        if $D[i] < 0$ **then**
            return `False` {Negative degree means it's not realizable}
        end if
    end for
end while
return `True`

The initial sort is $O(n \log n)$. The while loop iterates up to n times. Inside the while loop, the inner loop only decrements without sorting, achieving an amortized $O(n^2)$ complexity overall.

We will prove the correctness of the algorithm using mathematical induction.

Base Case

For $n = 1$, the only possible degree sequence is $[0]$, which corresponds to a single isolated node without any edges. This sequence is trivially realizable as a simple

[1] For this problem, you only need to give an $\mathcal{O}(n^2)$ algorithm. With some data structures, you can speed it up to quasi-linear.

graph (a single vertex with no edges), and the algorithm correctly terminates, returning `True`.

Inductive Hypothesis

Assume that for any degree sequence of length less than n, the algorithm correctly determines whether there exists a simple graph with the specified degree sequence.

Inductive Step

Now, consider a degree sequence $D = [d_1, d_2, \ldots, d_n]$, sorted in non-increasing order such that $d_1 \geq d_2 \geq \cdots \geq d_n$.

1. **Checking Validity**: If $d_1 \geq n$, then it is impossible to construct a simple graph with this degree sequence, since the maximum degree of a vertex cannot exceed the total number of other vertices in the graph. In this case, the algorithm correctly returns `False`.

2. **Reduction Step**: If $d_1 < n$, we take the largest degree d_1, remove it from the sequence, and decrement the degrees of the next d_1 vertices by 1. This corresponds to connecting the vertex with degree d_1 to the next d_1 vertices, thereby reducing their degrees.

3. **Reordering and Checking Negatives**: After decrementing the degrees of these d_1 vertices, we record the greatest d in the rest of the sequence. If any degree becomes negative, it indicates that the original degree sequence is not realizable as a simple graph, and the algorithm returns `False`.

4. **Applying the Inductive Hypothesis**: By removing the degree d_1 and reducing the next d_1 degrees, we have effectively reduced the length of the sequence by one. According to our inductive hypothesis, the reduced degree sequence can be checked by the algorithm. If the reduced sequence is realizable, then the original sequence is also realizable; otherwise, it is not.

5. **Termination Condition**: The algorithm repeats the above steps until the sequence becomes empty or all degrees are zero. If all degrees are reduced to zero, the algorithm returns `True`, indicating that the original sequence can be realized as a simple graph. This is because a degree sequence with all zeros corresponds to an empty graph, which is trivially realizable.

Conclusion

Thus, by mathematical induction, such algorithm correctly determines whether a given degree sequence can be realized as a simple undirected graph.

4. (10 points) Given two arrays $A[1 \dots n]$ and $B[1 \dots n]$ whose entries are between 0 to n, design an algorithm to output s_k for every k between 0 and $2n$. s_k is defined as the number of distinct (i, j) pairs such that $A[i] + B[j] = k$. Your algorithm should have a quasi-linear time complexity, i.e., at most $\mathcal{O}(n \log n)$.

Proof:

Let a_{freq} and b_{freq} be two frequency arrays, where:

$$a_{\text{freq}}[i] = \text{the number of occurrences of } i \text{ in } A,$$
$$b_{\text{freq}}[j] = \text{the number of occurrences of } j \text{ in } B.$$

Each entry $a_{\text{freq}}[i]$ counts how many times the value i appears in array A, and similarly, each entry $b_{\text{freq}}[j]$ counts occurrences in array B.

To compute s_k, which is the number of distinct (i, j) pairs such that $A[i] + B[j] = k$, we use the convolution which is defined as:

$$s_k = \sum_{i+j \equiv k \pmod{2n}} a_{\text{freq}}[i] \cdot b_{\text{freq}}[j].$$

Here, s_k represents the total number of ways to select elements $A[i]$ and $B[j]$ from arrays A and B, respectively, such that $A[i] + B[j] = k \pmod{2n+1}$. This is because $a_{\text{freq}}[i] \cdot b_{\text{freq}}[j]$ gives the number of ways to select elements with values i and j that add up to k.

One last thing to prove is that the modulo operation in the convolution formula does not affect the correctness of our result. The modulo operation implies that any sums exceeding the range $[0, 2n]$ would wrap back into this range. However, since each entry in the arrays A and B ranges from 1 to n, the maximum possible sum $A[i] + B[j]$ is $n + n = 2n$. This means no sums will exceed $2n$, so no wrap-around occurs.

Additionally, we padded each frequency array to length $2n + 1$, covering the full range from 0 to $2n$. Values in the range $[n + 1, 2n]$ of each frequency array are effectively zero because no element in A or B can produce values in this range by itself. As a result, any products involving these padded zeros contribute nothing to the final sum for any index s_k in s. Therefore, even with the modulo operation, the convolution result remains unaffected and correctly represents the number of ways to achieve each sum k from 0 to $2n$.

By the convolution theorem, we can compute s_k by transforming both frequency arrays to the frequency domain using the Fast Fourier Transform (FFT), performing an element-wise multiplication, and then transforming back to the time domain with the Inverse FFT:

$$s = \text{IFFT}(\text{FFT}(a_{\text{freq}}) \cdot \text{FFT}(b_{\text{freq}})).$$

This allows us to derive the psuedo code below:

Runtime Analysis:

Initializing the frequency arrays and counting occurrences for each element in A and B takes $\mathcal{O}(n)$ time.

Algorithm 2 Count Pairs with Given Sum Using Convolution

Require: Arrays A and B of length n

Ensure: Array c of length $2n+1$ where $c[k]$ is the number of pairs (i, j) such that $A[i] + B[j] = k$

Initialize a_freq and b_freq of size $2n+1$ to all zeros
for each element x in A **do**
    $a\_freq[x] \leftarrow a\_freq[x] + 1$
end for
for each element y in B **do**
    $b\_freq[y] \leftarrow b\_freq[y] + 1$
end for
Use FFT to obtain convolution $s \leftarrow \text{IFFT}(\text{FFT}(a\_freq) \cdot \text{FFT}(b\_freq))$
return s

Performing the FFT on a_{freq} and b_{freq}, the element-wise multiplication, and the Inverse FFT each take $\mathcal{O}(n \log n)$ time.

Summing up the values in the resulting array c also takes $\mathcal{O}(n)$ time.

Since the FFT operations dominate the overall complexity, the total time complexity is $\mathcal{O}(n \log n)$, which satisfies the quasi-linear time requirement.

5. (20 points) Given an array $A[1 \dots n]$ of n distinct numbers between 1 to n, compute the number of inversions for every $A[1 \dots i]$. Two array elements $A[i]$ and $A[j]$ form an inversion if $A[i] > A[j]$ and $i < j$. You should output n different answers, one for each $A[1 \dots i]$. The expected time complexity is $\mathcal{O}(n \log n)$.

   Answer:

   To compute the number inversions with the expected time complexity, through the course material, we can use two methods: divide and conquer(merge sort) and segment tree. We choose the second one because we need to output n different answers, one for each $A[1 \dots i]$.

   The main idea is to process each element of the array sequentially and, for each element $A[i]$, count the number of elements that are less than $A[i]$ and have already been processed. This count, $inversions\_with\_A_i$, corresponds to the number of inversions that $A[i]$ forms with previous elements.

   It can be easily seen that the results we want, which correspond to n different answers, one for each $A[1 \dots i]$, can be computed by adding both $inversions\_with\_A_i$ and already accumulated result $inversions\_count[i-1]$. As a result, when we iterate the original array A, we can sequentially get what we want.

   With the help of segment tree, we can easily do the counting operation in $O(logn)$ time complexity and for every update/modification, we also have $O(logn)$ time complexity, which can result the expected time complexity. More specifically, we can use a segment tree to:

   (a) **Query** the number of elements less than $A[i]$ that have already been inserted into the segment tree.

   (b) **Update** the segment tree by inserting $A[i]$ into it.

   The algorithm is shown as follows:

Algorithm 5 Counting Inversions Using Segment Tree

```
procedure CountInversions(n, A)
    Initialize segment tree st of size n
    Initialize inversion count array inversion_count of size n with all values 0
    for i ← 0 to n − 1 do
        Decrement A[i] by 1 to make it 0-based
    end for
    for i ← 0 to n − 1 do
        inversions_with_A_i ← st.query(A[i] + 1, n − 1)
        inversion_count[i] ← (i > 0 ? inversion_count[i − 1] : 0) + inversions_with_A_i
        st.update(A[i])
    end for
    for i ← 0 to n − 1 do
        Output inversion_count[i]
    end for
end procedure
procedure Update(node, l, r, index)
    if l = r then
        Increment tree[node] by 1
        return
    end if
    mid ← (l+r)/2
    if index ≤ mid then
        Update left child 2 × node
    else
        Update right child 2 × node + 1
    end if
    tree[node] ← tree[2 × node] + tree[2 × node + 1]
end procedure
procedure Query(node, l, r, ql, qr)
    if ql > r or qr < l then
        return 0
    end if
    if ql ≤ l and qr ≥ r then
        return tree[node]
    end if
    mid ← (l+r)/2
    return Query left child + Query right child
end procedure
```

Since we process each element of the array (total n elements), the time complexity for processing all elements is:

$$O(n \log n)$$

This is due to performing $O(\log n)$ operations for each of the n elements.

Coding Problem Record

Yuchen:

3334. Find the Maximum Factor Score of Array

Solved

Medium | Topics | Companies | Hint

You are given an integer array `nums`.

The **factor score** of an array is defined as the *product* of the LCM and GCD of all elements of that array.

Return the **maximum factor score** of `nums` after removing **at most** one element from it.

Note that *both* the LCM and GCD of a single number are the number itself, and the *factor score* of an **empty** array is 0.

Example 1:

Input: `nums = [2,4,8,16]`

65 | 8

Submissions

Status	Language	Runtime	Memory	Notes
Accepted Oct 31, 2024	Python3	47 ms	16.8 MB	

295. Find Median from Data Stream

Solved

Hard | Topics | Companies

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

- For example, for `arr = [2,3,4]`, the median is `3`.
- For example, for `arr = [2,3]`, the median is `(2 + 3) / 2 = 2.5`.

Implement the MedianFinder class:

- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within 10^{-5} of the actual answer will be accepted.

12.1K | 80

Submissions

Status	Language	Runtime	Memory	Notes
Accepted Oct 25, 2024	Python3	128 ms	38.7 MB	

621. Task Scheduler

Solved

Medium | Topics | Companies | Hint

You are given an array of CPU `tasks`, each labeled with a letter from A to Z, and a number `n`. Each CPU interval can be idle or allow the completion of one task. Tasks can be completed in any order, but there's a constraint: there has to be a gap of **at least** `n` intervals between two tasks with the same label.

Return the **minimum** number of CPU intervals required to complete all tasks.

Example 1:

Input: `tasks = ["A","A","A","B","B","B"], n = 2`

Output: 8

10.7K | 144

Submissions

Status	Language	Runtime	Memory	Notes
Accepted Oct 25, 2024	Python3	432 ms	18.5 MB	

654. Maximum Binary Tree

Solved

Medium | Topics | Companies

You are given an integer array `nums` with no duplicates. A **maximum binary tree** can be built recursively from `nums` using the following algorithm:

1. Create a root node whose value is the maximum value in `nums`.
2. Recursively build the left subtree on the **subarray prefix** to the **left** of the maximum value.
3. Recursively build the right subtree on the **subarray suffix** to the **right** of the maximum value.

Return *the **maximum binary tree** built from* `nums`.

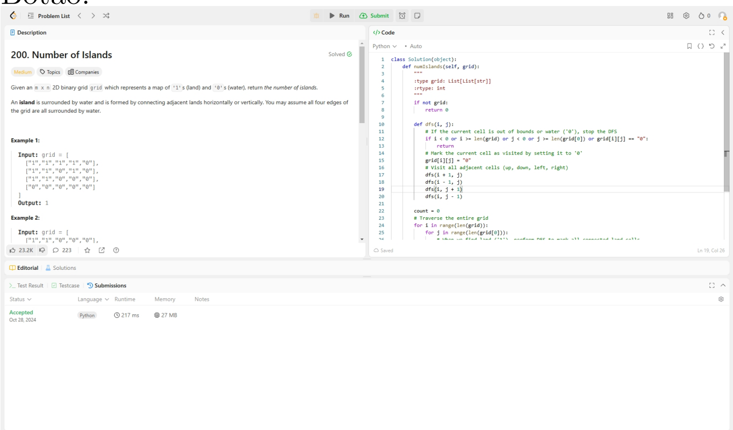
Example 1:

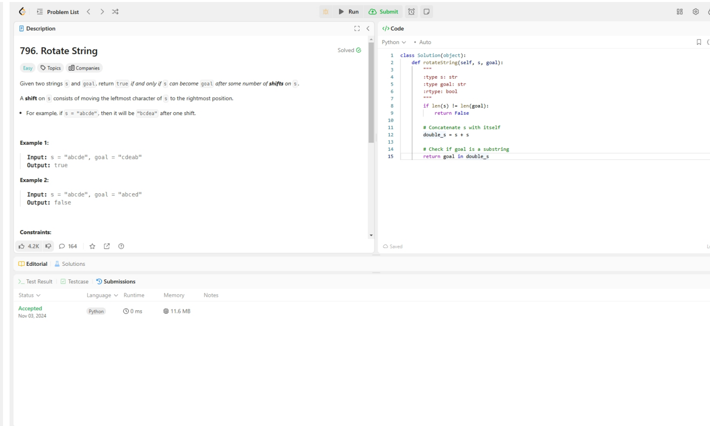
5.3K | 26

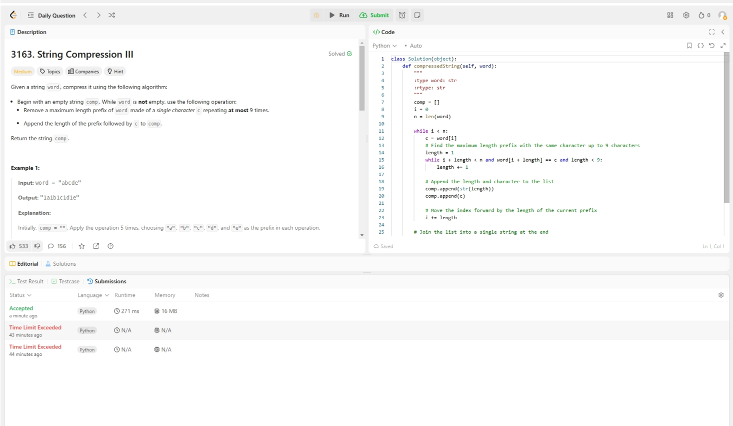
Submissions

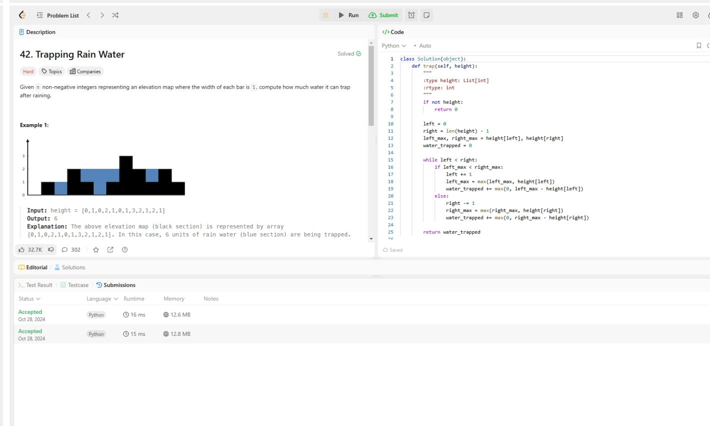
Status	Language	Runtime	Memory	Notes
Accepted Oct 24, 2024	Python3	23 ms	17 MB	

Botao:


200. Number of Islands
Accepted


796. Rotate String
Accepted


3163. String Compression III
Accepted
Time Limit Exceeded
Time Limit Exceeded


42. Trapping Rain Water
Accepted
Accepted

Yuheng:

Description | Accepted × | Editorial | Solutions | Submissions

164. Maximum Gap

Solved

Medium | Topics | Companies

Given an integer array `nums`, return *the maximum difference between two successive elements in its sorted form*. If the array contains less than two elements, return `0`.

You must write an algorithm that runs in linear time and uses linear extra space.

Example 1:

264. Ugly Number II

Medium | Topics | Companies | Hint

An **ugly number** is a positive integer whose prime factors are limited to `2`, `3`

Given an integer `n`, return *the* n^{th} ***ugly number***.

Example 1:

274. H-Index

Solved

Medium | Topics | Companies | Hint

Given an array of integers `citations` where `citations[i]` is the number of citations a researcher received for their i^{th} paper, return *the researcher's h-index*.

According to the definition of h-index on Wikipedia: The h-index is defined as the maximum value of `h` such that the given researcher has published at least `h` papers that have each been cited at least `h` times.

Example 1:

```
Input: citations = [3,0,6,1,5]
Output: 3
Explanation: [3,0,6,1,5] means the researcher has 5 papers in total and each of them had received 3, 0,
```

410. Split Array Largest Sum

Hard | Topics | Companies

Given an integer array `nums` and an integer `k`, split `nums` into `k` non-empty **minimized**.

Return *the minimized largest sum of the split*.

A **subarray** is a contiguous part of the array.

Example 1:

```
Input: nums = [7,2,5,10,8], k = 2
Output: 18
```

Peng Yao:

218. The Skyline Problem

Solved

Hard | Topics | Companies

A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return *the **skyline** formed by these buildings collectively*.

The geometric information of each building is given in the array `buildings` where `buildings[i] = [left_i, right_i, height_i]`:

- `left_i` is the x coordinate of the left edge of the `i^th` building.
- `right_i` is the x coordinate of the right edge of the `i^th` building.
- `height_i` is the height of the `i^th` building.

6K | 30

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	10 ms	19.7 MB	

327. Count of Range Sum

Solved

Hard | Topics | Companies

Given an integer array `nums` and two integers `lower` and `upper`, return *the number of range sums that lie in* `[lower, upper]` *inclusive*.

Range sum `S(i, j)` is defined as the sum of the elements in `nums` between indices `i` and `j` inclusive, where `i <= j`.

2.3K | 18

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	313 ms	206.1 MB	

315. Count of Smaller Numbers After Self

Solved

Hard | Topics | Companies

Given an integer array `nums`, return *an integer array* `counts` *where* `counts[i]` *is the number of smaller elements to the right of* `nums[i]`.

8.9K | 35

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	368 ms	248.5 MB	

493. Reverse Pairs

Solved

Hard | Topics | Companies | Hint

Given an integer array `nums`, return *the number of **reverse pairs** in the array*.

A **reverse pair** is a pair `(i, j)` where:

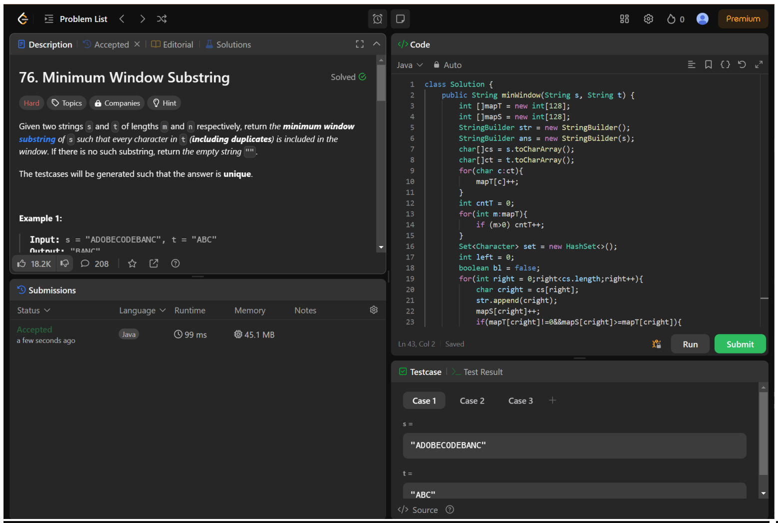
- `0 <= i < j < nums.length` and
- `nums[i] > 2 * nums[j]`.

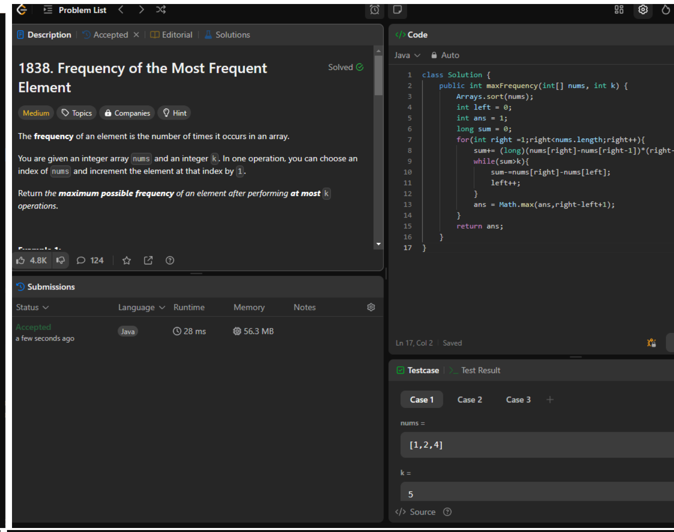
6.3K | 59

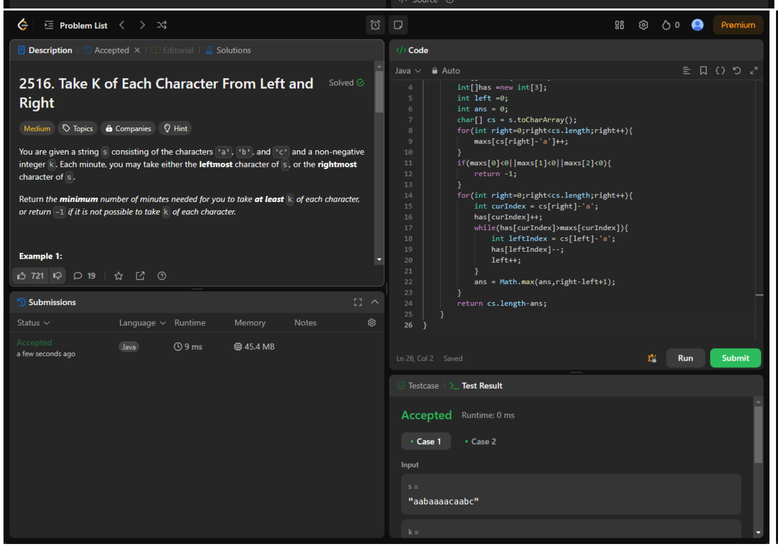
Submissions

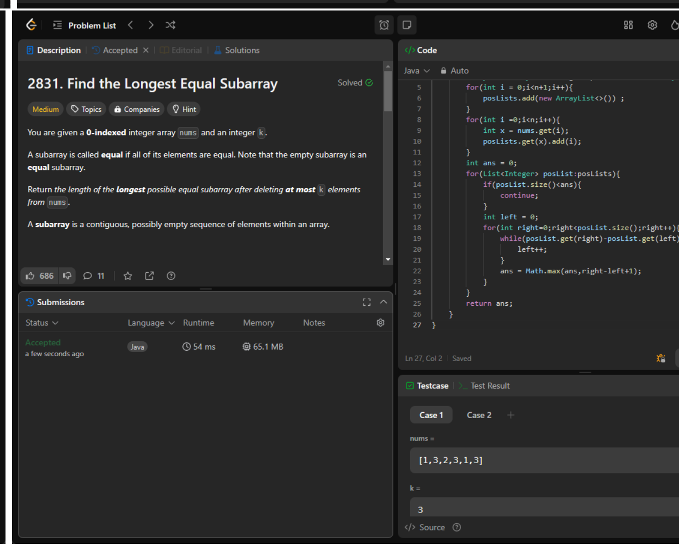
Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	236 ms	109.5 MB	

Bowen Yu:


Problem List
Description
Accepted
Editorial
Solutions
76. Minimum Window Substring
Solved
Hard
Topics
Companies
Hint
Code
Java
Auto
Submissions
Accepted
a few seconds ago
99 ms
45.1 MB
Run
Submit
Testcase
Test Result
Case 1
Case 2
Case 3
"ADOBECODEBANC"
"ABC"


Problem List
1838. Frequency of the Most Frequent Element
Solved
Medium
Topics
Companies
Hint
Code
Java
Auto
Submissions
Accepted
a few seconds ago
28 ms
56.3 MB
Testcase
Test Result
Case 1
Case 2
Case 3
[1,2,4]
5


Problem List
2516. Take K of Each Character From Left and Right
Solved
Medium
Topics
Companies
Hint
Code
Java
Auto
Submissions
Accepted
a few seconds ago
9 ms
45.4 MB
Run
Submit
Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
"aabaaaacaabc"


Problem List
2831. Find the Longest Equal Subarray
Solved
Medium
Topics
Companies
Hint
Code
Java
Auto
Submissions
Accepted
a few seconds ago
54 ms
65.1 MB
Testcase
Test Result
Case 1
Case 2
[1,3,2,3,1,3]
3