Homework 2

CSE 202
Fall 2024

Due at: 11:59 PM Monday, October 21

For each problem, provide a high-level description of your algorithm. Please make sure to include the necessary details that are crucial for its correctness and efficiency. Prove its correctness and analyze its time complexity.

Background: graph reachability, BFS, DFS, bridges, cut vertices, DAG, topological ordering, 2-SAT, Dijkstra's algorithm, heap

---

Graded Problems

1. (20 points) You are given a directed graph (in adjacency list format) with n vertices and m edges. Each edge is colored either R, G or B. Additionally, you are provided with a source node s and a target node t. Design an algorithm to determine whether there exists a path from s to t such that no two consecutive edges along the path share the same color.

   High Level Description: To determine whether there exists a path from s to t such that no two consecutive edge along the path share the same color, we can design an algorithm by modifying breadth-first search. We keep track of each node respect to the color of the incoming edge starting from s, and explore its adjacent nodes while ensuring that no consecutive edge has the same incoming color.

   Algorithm: Each node in the graph has state represented by tuple (u, c), where u is the current node being explore, and c is the color of the edge that were used to reach the current node. We then use a queue for states to perform BFS. Begin with the starting node s with state $(s, None)$, we append it to the queue. While the queue is not empty, we pop the state (u, c) of node u from the front of the queue. For every adjacent edge (u, v) such that (v, c'), the state of node v is unvisited, if the color c' of its state (v, c') is different from the color c in (u, c), we append its state (v, c') to the queue and mark (v, c') as visited and continue doing so. When the queue is empty, if the target node t is marked visited, that indicate there exists a valid path from node s to t. If node t is not visited, there is no such a path.

Algorithm 1 Find Valid Path with Alternating Edge Colors

```
procedure FindPathWithColors(G, s, t)
    Q ← Empty Queue
    Visited ← Empty Set
    Enqueue (s, None) into Q
    Add (s, None) to Visited
    while Q is not empty do
        (u, c) ← Dequeue from Q
        if u = t then
            return There exist valid path
        end if
        for each neighbor (v, c′) of u do
            if c ≠ c′ then
                if (v, c′) ∉ Visited then
                    Enqueue (v, c′) into Q
                    Add (v, c′) to Visited
                end if
            end if
        end for
    end while
    return No valid path
end procedure
```

Runtime Analysis: Initialization takes $O(1)$, since BFS explore each nodes and its neighbors, as we considering all colors for each node, eacch node can be processed at most 3 times, that still takes $O(n)$. We also traverse through all edges once, which is O(m). Hence, the total time complexity is $O(n + m)$

Proof of Correctness:

Base Case:

At the start, the BFS initializes the queue with $(s, None)$, which represents the source node s without any prior edge color constraint. Thus, the first edge taken from s can be in any color. Therefore the algorithm can correctly start exploring all edges from s.

Inductive Steps:

During BFS, the algorithm explores each node u and checks for its adjacent nodes v. For each v, the algorithm ensure the edge from u to v is different in color from the edge to get node u.

Since our algorithms are using BFS, it explore each node by the distance from s, it will find the shortest valid path with alternative color if it exist. If such path exist, the algorithm will eventually explore node t and marked its state as visited. Using BFS also makes sure the algorithm explore all possible path from s to t while satisfy the condition of no two consecutive edges color along the path.

The algorithm will terminate when the queue is empty. Once queue is empty, if there is a valid path, the state of target node t should be marked visited. If the queue is empty without visited the state of t, that means we can't get to t from s Using BFS under the constraint, there is no valid path.

Therefore, the algorithm is correct and will find a valid path from s to t if such path exists.

2. (20 points) There are m constraints on n boolean variables $x_1, \ldots, x_n$. Each constraint is given by three indices i, j, k and three boolean values b_1, b_2, b_3. (The boolean values can be different for different constraints.) The constraint is satisfied if and only if at least two of the following conditions are met:

   (a) $x_i = b_1$

   (b) $x_j = b_2$

   (c) $x_k = b_3$

   To make the problem simpler, we can assume that i, j, k are always distinct. Design an algorithm that decide whether there exists an assignment such that all the constraints can be satisfied.

   We are given three-variable constraints of the form:

$$(x_i = b_1), \quad (x_j = b_2), \quad (x_k = b_3)$$

   The constraint is satisfied if at least two of these conditions hold. Our task is to represent this as a 2-SAT problem by constructing implications.

   For each three-variable constraint, we derive the implications for the possible cases:

Case 1: $x_i = b_1$ and $x_j = b_2$

If either $x_i \neq b_1$ or $x_j \neq b_2$, then $x_k = b_3$ must hold. The implications are:

$$\neg(x_i = b_1) \to (x_j = b_2) \wedge (x_k = b_3)$$

$$\neg(x_j = b_2) \to (x_i = b_1) \wedge (x_k = b_3)$$

Case 2: $x_i = b_1$ and $x_k = b_3$

If either $x_i \neq b_1$ or $x_k \neq b_3$, then $x_j = b_2$ must hold. The implications are:

$$\neg(x_i = b_1) \to (x_j = b_2) \wedge (x_k = b_3)$$

$$\neg(x_k = b_3) \to (x_i = b_1) \wedge (x_j = b_2)$$

Case 3: $x_j = b_2$ and $x_k = b_3$

If either $x_j \neq b_2$ or $x_k \neq b_3$, then $x_i = b_1$ must hold. The implications are:

$$\neg(x_j = b_2) \to (x_i = b_1) \wedge (x_k = b_3)$$

$$\neg(x_k = b_3) \to (x_i = b_1) \wedge (x_j = b_2)$$

Then for each constraint, we have:

$$f = ((x_j = b_2) \vee (x_k = b_3)) \wedge ((x_i = b_1) \vee (x_k = b_3)) \wedge ((x_i = b_1) \vee (x_j = b_2))$$

For all constraints, we have:

$$F = \Pi_{c=1}^{m} \left((x_{c_j} = b_{c_2}) \vee (x_{c_k} = b_{c_3})\right) \wedge ((x_{c_i} = b_{c_1}) \vee (x_{c_k} = b_{c3})) \wedge \left((x_{c_i} = b_{c_1}) \vee (x_{c_j} = b_{c_2})\right)$$

Now, for each constraint, whenever there are at least two conditions are met, the constraints can be satisfied.
With the conjunctive expression for all constraint clauses, we can convert the solution of the original question to a **2SAT** solution.

3. (15 points) You are given an array A of n integers. Your goal is to construct a min-heap of A: i.e., an array B that consists of every element of A, and satisfies

$$B[i] \leq B[2i] \quad \text{and} \quad B[i] \leq B[2i+1],$$

whenever the indices exist. (We are using 1-indexing in this problem.) Show a linear-time algorithm that completes this task.

Hint: you may use the fact that the infinite series $\sum_{n=0}^{\infty} n2^{-n}$ *converges to a constant.*

For array A, We traverse the array A from index $n/2$ to 1 to call the heapify method, since it's 1-indexing. We assume the latter half of array A is the leaf node that does not need to be iterated. In other words, we start the iteration from the bottom of the heap.

Algorithm 2 Min-Heap

procedure HEAPIFY(B, i)
    $left \leftarrow 2 \times i$
    $right \leftarrow 2 \times i + 1$
    $smallest \leftarrow i$
    if $B[left] < B[smallest]$ **then**
        $smallest \leftarrow left$
    end if
    if $B[right] < B[smallest]$ **then**
        $smallest \leftarrow right$
    end if
    if $smallest \neq i$ **then**
        Swap $B[I]$ and $B[smallest]$
        Call HEAPIFY($B, smallest$)
    end if
end procedure

We can easily know that there are approximately $\frac{n}{2^{h+1}}$nodes at height h. For each node at height h, the time complexity of the heapify operation is $O(h)$, since the heapify process may adjust nodes up to that height.

The total time complexity is the sum of the heapify costs for all nodes, which can be expressed as:

$$\text{Total Complexity} = \sum_{h=0}^{\log n} \left(\frac{n}{2^{h+1}} \cdot h \right)$$

The hint says such an infinite series is convergent. So we can conclude that such an algorithm has a linear time complexity.

4. (25 points) There are n segments sitting on the real axis. Each segment L_i is of the form $[a_i, b_i]$ where $a_i \leq b_i$, i.e., a closed interval. We say two segments L_i, L_j *intersect* if $[a_i, b_i] \cap [a_j, b_j]$ is non-empty. You do not know what a_i, b_i are for every segment, but you learn m conditions, each of which tells you exactly one of the following for some pair of indices $1 \leq i, j \leq n$:

   - either L_i and L_j intersect, or
   - L_i and L_j do not intersect *and* $b_i < a_j$, i.e. L_i lies before L_j

   Design an algorithm that takes as input these m conditions, and decides whether such n segments exist (i.e., there exist $L_i = [a_i, b_i]$ for all $1 \leq i \leq n$ satisfying all the m conditions).

   We shall solve this problem by reducing it to a topological sort problem on a directed graph. First, we shall create a directed graph based on m given conditions.

Directed graph creation

Node creation: For each segment L_i, we create two nodes in the graph:

- a_i: Represents the start of the segment L_i.
- b_i: Represents the end of the segment L_i.

Edge creation: For each segment L_i, we add a directed edge from a_i to b_i:

$$a_i \to b_i$$

This edge enforces the constraint that the start of the segment must occur before the end of the segment.

For each intersection condition between segments L_i and L_j, we add two directed edges:

$$a_i \to b_j \quad \text{(Segment } L_i \text{ starts before segment } L_j \text{ ends)}$$
$$a_j \to b_i \quad \text{(Segment } L_j \text{ starts before segment } L_i \text{ ends)}$$

These edges ensure that both segments overlap, as required by the intersection condition.

For each non-intersection (comes before) condition $b_i < a_j$, we add a directed edge:

$$b_i \to a_j$$

This edge enforces that segment L_i must end before segment L_j starts, ensuring that the two segments do not intersect.

Reduction to topological sort:

Yes to Yes: The Graph is a DAG for a Valid Assignment of Intervals
We now prove that if there is a valid assignment of n intervals, the resulting graph must be a Directed Acyclic Graph (DAG), and we will be able to sort it topologically.

Proof: Suppose there exists a valid assignment of intervals such that all n intervals are arranged on the real line without violating any of the given conditions (either intersection or "comes before"). We claim that the resulting directed graph must be a DAG.

Revisiting the structure of the graph:

- For each interval $L_i = (a_i, b_i)$, we add a directed edge $a_i \to b_i$, enforcing that the start of an interval must precede its end.
- For every pair of intersecting intervals L_i and L_j, we add two directed edges: $a_i \to b_j$ and $a_j \to b_i$, representing the mutual constraint that the intervals must overlap.
- For every non-intersecting ("comes before") condition, we add a directed edge $b_i \to a_j$, enforcing that L_i must end before L_j starts.

Assume for the sake of contradiction that the graph contains a cycle. A cycle implies that we have a sequence of directed edges that eventually loops back to a starting node. This would correspond to a contradiction in the assignment of intervals because it would mean that some interval both starts before and ends after another interval, and vice versa, violating either the intersection or the "comes before" condition. Such a contradiction would imply that no valid assignment exists.

Since we assumed that a valid assignment of intervals exists, no such cycle can form in the graph. Therefore, the graph must be acyclic.

Once we have established that the graph is a DAG, we know that it can be topologically sorted.

Thus, if there is a valid assignment of intervals, the resulting graph must be a DAG, and we will be able to sort it topologically. QED

No to No: A DAG Implies a Valid Assignment of Intervals

Proof by contradiction: Assume that the graph is a DAG, and we have performed a successful topological sort of the nodes. Suppose, for the sake of contradiction, that it is not possible to assign a valid set of n intervals that satisfies all the given conditions.

This means there must be some condition in the set of m conditions (either intersection or "comes before") that cannot be satisfied in the current assignment of intervals.

Let's consider the topological sort process. For a node u representing an interval, the sort processes u only when its incoming edge count is zero, meaning all constraints related to u have been satisfied by previously processed nodes. If there were a condition m that couldn't be satisfied, it would imply that the topological sort encountered a node whose constraints were not respected, which leads to a contradiction:

- **Intersection Condition**: If there's an intersection condition between nodes u and v, the directed edges $u \to v$ and $v \to u$ ensure that both intervals overlap. When node u is processed, its incoming edge count being zero guarantees that node v, and any other nodes it depends on, have already been processed. This ensures that the intersection condition is satisfied. If this weren't true, it would contradict the fact that the topological sort only processes a node when all incoming edges (dependencies) are resolved.
- **"Comes Before" Condition**: If there's a "comes before" condition (i.e., u must come before v), the edge $u \to v$ enforces this order. When node u is processed, all preceding nodes (those with edges pointing into u) have already been placed, ensuring that u respects the order imposed by the "comes before" condition. Any violation of this would again contradict the correctness of the topological sort, where nodes are processed only after their dependencies are fully resolved.

Since the topological sort ensures that each node is processed only when all its constraints are satisfied (represented by incoming edge count = 0), the existence of a condition that cannot be satisfied forms a contradiction.

Thus, a successful topological sort guarantees that there is at least one valid way to assign all n intervals, satisfying all the conditions. QED

Algorithm 3 Check Valid Interval Assignment

procedure ValidAssignment($n, m, conditions$)
    $G \leftarrow$ Empty directed graph
    $in\_degree \leftarrow$ array of size $2n$ initialized to 0
    for $i \in \{1, \ldots, n\}$ **do**
        Add node a_i (start of interval L_i) to G
        Add node b_i (end of interval L_i) to G
        Add directed edge $(a_i \rightarrow b_i)$ to G
        $in\_degree[b_i] \leftarrow in\_degree[b_i] + 1$
    end for
    for all $(L_i, L_j, \text{type}) \in conditions$ **do**
        if type = "intersection" **then**
            Add directed edge $(a_i \rightarrow b_j)$ to G
            $in\_degree[b_j] \leftarrow in\_degree[b_j] + 1$
            Add directed edge $(a_j \rightarrow b_i)$ to G
            $in\_degree[b_i] \leftarrow in\_degree[b_i] + 1$
        else if type = "comes before" **then**
            Add directed edge $(b_i \rightarrow a_j)$ to G
            $in\_degree[a_j] \leftarrow in\_degree[a_j] + 1$
        end if
    end for
    Perform Topological Sort on the Graph
    Check if topological sort succeeds
    if Topological sort fails **then**
        return False
    else
        return True
    end if
end procedure

Runtime Analysis

The overall time complexity of the algorithm is $O(n+m)$, where n is the number of intervals and m is the number of conditions. The breakdown is as follows:

Graph Construction: For each interval L_i, we create two nodes a_i and b_i, resulting in $2n$ nodes. For each "comes before" condition, we add one edge, and for each "intersection" condition, we add two edges, giving m edges in total.
Time Complexity: $O(n+m)$

Topological Sort: The topological sort processes all $2n$ nodes and m edges.
Time Complexity: $O(n+m)$

Validity Check: After the sort, we check if all nodes were processed to detect cycles.
Time Complexity: $O(n)$

Total Time Complexity: The total runtime is $O(n+m)+O(n+m)+O(n) = O(n+m)$.

5. (20 points) You are given an undirected graph with n vertices and m edges (in adjacency list format), as well as a source node s and a target t. Each edge has length either 1 or 2. Give a linear time[1] algorithm that computes the length of the shortest path from s to t.

Let's first consider the simplest case: finding the shortest path in an undirected graph with edge weights of 1 (or equivalently scaled, so that all edge weights are equal). We can use BFS for this purpose. Starting from the source node (the source node is the first layer, corresponding to distance 0), for all nodes at each layer, we can determine that the shortest distance from the source node to these nodes is equal to the path length of the BFS traversal.

Proof:

By contradiction, for a pair of nodes (u, v), assume node v is reached from node u during the i-th phase of BFS. The shortest path from u to v should be of length i. Now suppose there exists a path from u to v with a length less than i. In the BFS algorithm, all intermediate nodes along this path must have been visited between the 0-th and the $(i-1)$-th phases. Therefore, node v should have been reached before the i-th phase, which contradicts our assumption. Thus, the proof is complete.

When dealing with a more complex case, namely finding the shortest path in an undirected graph with edge weights of 1 or 2, we can use a similar approach as in the equal-weight undirected graph, specifically BFS, to track the shortest path from the source node to each node. An intuitive transformation is to add a virtual node in the middle of an edge with weight 2. Then, we can do global BFS and record their distance at the same time.

However, the implementation of above virtual nodes is complicated so we use two queues to simulate the process. For nodes that can be reached from the current node

[1] Here, linear time means $O(n+m)$.

via an edge of weight 1, we place them in the queue for the next expansion. For nodes that can be reached from the current node via an edge of weight 2, we place them in the queue for the expansion after the next one (note that the next expansion queue includes nodes that can be reached from the current node via edges of weight 1, while the subsequent expansion queue includes nodes that can be reached from the next nodes via edges of weight 1 as well as nodes that can be reached from the current node via edges of weight 2).

The correctness of this algorithm is that even if we revisit the node visited from the current node along the edge with weight 2 in the next expansion, it will not cause it to be reentered into the queue because the conditional judgment is not satisfied(the condition is $dist[node] + w < dist[nei]$). The condition judgment here actually acts like a visited array, so that for all nodes, we will only queue them up once.

The above case analysis ensures that no node needs to be re-enqueued during BFS. For every two expansion queues, we can apply a similar case analysis to ensure that each node is visited only once. Therefore, in an undirected graph with edge weights of 1 and 2, BFS can be used to find the shortest path, as shown by the algorithm in the next page.

Regarding time complexity, since every node is enqueued and dequeued only once in all the queues (in practice, we only use two queues and, after each expansion, directly place the old subsequent execution queue in the current phase's next execution queue), the algorithm runs in linear time. Essentially, in this special type of graph with edge weights of 1 and 2, we utilize the specific conditions to replace the insert-search operations (which take $O(\log m)$) in traditional shortest path algorithms with simple enqueue-dequeue operations. This reduces the time complexity from $O((n+m)\log n)$ to $O(n+m)$.

Algorithm 4 Shortest Path in Graph with Weights 1 and 2

```
procedure ShortestPath(n, edges, s, t)
    Initialize adjacency list adj for n nodes
    for each edge (u, v, w) in edges do
        Append (v, w) to adj[u]
        Append (u, w) to adj[v]                    ▷ Since the graph is undirected
    end for
    Initialize distance array dist with size n, set all values to ∞
    Set dist[s] ← 0
    Initialize queues q_1 and q_2
    Enqueue s to q_1
    while q_1 is not empty or q_2 is not empty do
        Initialize new queues new_q1 and new_q2
        size ← size of q_1
        for i ← 1 to size do
            node ← Dequeue from q_1
            for each neighbor (nei, w) in adj[node] do
                if dist[node] + w < dist[nei] then
                    dist[nei] ← dist[node] + w
                    if w = 1 then
                        Enqueue nei to new_q1
                    else
                        Enqueue nei to new_q2
                    end if
                end if
            end for
        end for
        q_1 ← new_q1
        while q_2 is not empty do
            Enqueue from q_2 to q_1
        end while
        q_2 ← new_q2
    end while
    return dist[t]
end procedure
```

Leetcode Question:

Yuchen:

42. Trapping Rain Water

Solved

Hard | Topics | Companies

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:

32.6K | 303

Submissions

Status	Language	Runtime	Memory
Accepted Oct 10, 2024	Python3	116 ms	18.9 MB

84. Largest Rectangle in Histogram

Solved

Hard | Topics | Companies

Given an array of integers `heights` representing the histogram's bar height where the width of each bar is `1`, return *the area of the largest rectangle in the histogram*.

Example 1:

17.6K | 142

Submissions

Status	Language	Runtime	Memory
Accepted Oct 11, 2024	Python3	662 ms	31.6 MB

394. Decode String

Solved

Medium | Topics | Companies

Given an encoded string, return its decoded string.

The encoding rule is: `k[encoded_string]`, where the `encoded_string` inside the square brackets is being repeated exactly `k` times. Note that `k` is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain

12.9K | 98

Submissions

Status	Language	Runtime	Memory
Accepted Oct 11, 2024	Python3	38 ms	16.6 MB

3319. K-th Largest Perfect Subtree Size in Binary Tree

Solved

Medium | Topics | Companies | Hint

You are given the `root` of a **binary tree** and an integer `k`.

Return an integer denoting the size of the k^{th} **largest perfect binary** subtree, or `-1` if it doesn't exist.

A **perfect binary tree** is a tree where all leaves are on the same level, and every parent has two children.

78 | 20

Submissions

Status	Language	Runtime	Memory
Accepted Oct 19, 2024	Python3	25 ms	17.4 MB

Botao:

Description

1545. Find Kth Bit in Nth Binary String

Solved

Medium | Topics | Companies | Hint

Given two positive integers `n` and `k`, the binary string `Sn` is formed as follows:

- `S1 = "0"`
- `Si = Si - 1 + "1" + reverse(invert(Si - 1))` for `i > 1`

Where `+` denotes the concatenation operation, `reverse(x)` returns the reversed string `x`, and `invert(x)` inverts all the bits in `x` (`0` changes to `1` and `1` changes to `0`).

For example, the first four strings in the above sequence are:

- `S1 = "0"`
- `S2 = "011"`
- `S3 = "0111001"`
- `S4 = "011100110110001"`

Return *the* `kth` *bit in* `Sn`. It is guaranteed that `k` is valid for the given `n`.

Example 1:

1.4K | 113

Editorial | Solutions

Test Result | Testcase | Submissions

Status	Language	Runtime	Memory	Notes
Accepted Oct 18, 2024	Python3	0 ms	16.5 MB	

Description

2530. Maximal Score After Applying K Operations

Solved

Medium | Topics | Companies | Hint

You are given a **0-indexed** integer array `nums` and an integer `k`. You have a **starting score** of `0`.

In one **operation**:

1. choose an index `i` such that `0 <= i < nums.length`,
2. increase your **score** by `nums[i]`, and
3. replace `nums[i]` with `ceil(nums[i] / 3)`.

Return *the maximum possible* ***score*** *you can attain after applying* ***exactly*** `k` *operations.*

The ceiling function `ceil(val)` is the least integer greater than or equal to `val`.

Example 1:

```
Input: nums = [10,10,10,10,10], k = 5
Output: 50
Explanation: Apply the operation to each array element exactly once. The final score is 10 + 10 + 10 + 10 + 10 = 50.
```

Example 2:

```
Input: nums = [1,10,3,3,3], k = 3
```

826 | 198

Editorial | Solutions

Test Result | Testcase | Submissions

Status	Language	Runtime	Memory	Notes
Accepted Oct 13, 2024	Python3	718 ms	31.3 MB	

Description

670. Maximum Swap

Solved

Medium | Topics | Companies

You are given an integer `num`. You can swap two digits at most once to get the maximum valued number.

Return *the maximum valued number you can get.*

Example 1:

```
Input: num = 2736
Output: 7236
Explanation: Swap the number 2 and the number 7.
```

Example 2:

```
Input: num = 9973
Output: 9973
Explanation: No swap.
```

Constraints:

- $0 <= num <= 10^8$

4K | 217

Editorial | Solutions

Test Result | Testcase | Submissions

Status	Language	Runtime	Memory	Notes
Accepted Oct 16, 2024	Python3	36 ms	16.4 MB	

Description

51. N-Queens

Solved

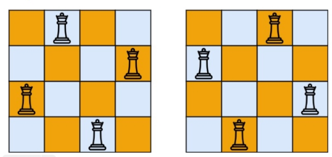
Hard | Topics | Companies

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the* ***n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



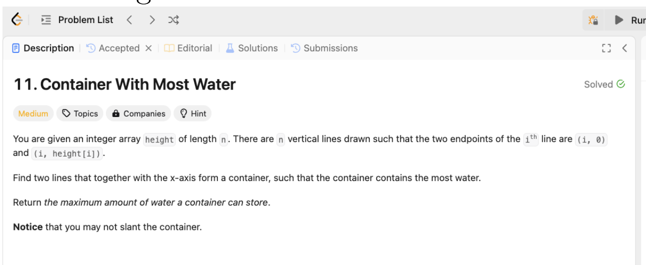
12.7K | 96

Editorial | Solutions

Test Result | Testcase | Submissions

Status	Language	Runtime	Memory	Notes
Accepted Oct 12, 2024	Python	30 ms	11.8 MB	

Yuheng:



Problem List

Run

Description | Accepted | Editorial | Solutions | Submissions

11. Container With Most Water

Solved

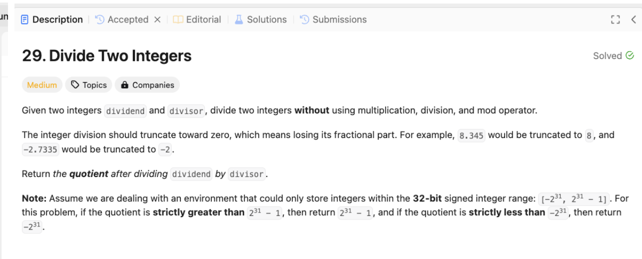
Medium | Topics | Companies | Hint

You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the i^{th} line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.



Description | Accepted | Editorial | Solutions | Submissions

29. Divide Two Integers

Solved

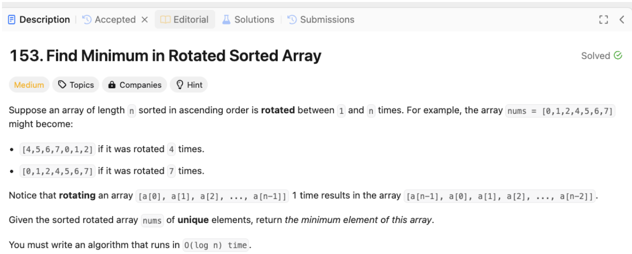
Medium | Topics | Companies

Given two integers `dividend` and `divisor`, divide two integers **without** using multiplication, division, and mod operator.

The integer division should truncate toward zero, which means losing its fractional part. For example, `8.345` would be truncated to `8`, and `-2.7335` would be truncated to `-2`.

Return *the **quotient** after dividing* `dividend` *by* `divisor`.

Note: Assume we are dealing with an environment that could only store integers within the **32-bit** signed integer range: $[-2^{31}, 2^{31} - 1]$. For this problem, if the quotient is **strictly greater than** $2^{31} - 1$, then return $2^{31} - 1$, and if the quotient is **strictly less than** -2^{31}, then return -2^{31}.



Description | Accepted | Editorial | Solutions | Submissions

153. Find Minimum in Rotated Sorted Array

Solved

Medium | Topics | Companies | Hint

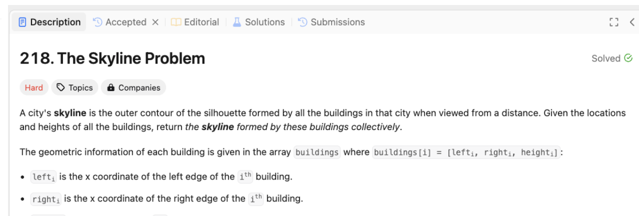
Suppose an array of length `n` sorted in ascending order is **rotated** between `1` and `n` times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated `4` times.
- `[0,1,2,4,5,6,7]` if it was rotated `7` times.

Notice that **rotating** an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of **unique** elements, return *the minimum element of this array*.

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



Description | Accepted | Editorial | Solutions | Submissions

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.

Peng Yao:

3327. Check if DFS Strings Are Palindromes

Solved

Hard | Topics | Companies | Hint

You are given a tree rooted at node 0, consisting of `n` nodes numbered from `0` to `n - 1`. The tree is represented by an array `parent` of size `n`, where `parent[i]` is the parent of node `i`. Since node 0 is the root, `parent[0] == -1`.

You are also given a string `s` of length `n`, where `s[i]` is the character assigned to node `i`.

Consider an empty string `dfsStr`, and define a recursive function `dfs(int x)` that takes a node `x` as a parameter and performs the following steps in order:

- Iterate over each child `y` of `x` **in increasing order of their numbers**, and call `dfs(y)`.
- Add the character `s[x]` to the end of the string `dfsStr`.

Note that `dfsStr` is shared across all recursive calls of `dfs`.

You need to find a boolean array `answer` of size `n`, where for each index `i` from `0` to `n - 1`, you do the following:

- Empty the string `dfsStr` and call `dfs(i)`.
- If the resulting string `dfsStr` is a palindrome, then set `answer[i]` to `true`. Otherwise, set `answer[i]` to `false`.

Return the array `answer`.

30 | 17

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	1403 ms	376.6 MB	

127. Word Ladder

Solved

Hard | Topics | Companies

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- s_k `== endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* `beginWord` *to* `endWord`*, or* `0` *if no such sequence exists.*

12.3K | 150

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	113 ms	23.5 MB	

2127. Maximum Employees to Be Invited to a Meeting

Solved

Hard | Topics | Companies | Hint

A company is organizing a meeting and has a list of `n` employees, waiting to be invited. They have arranged for a large **circular** table, capable of seating **any number** of employees.

The employees are numbered from `0` to `n - 1`. Each employee has a **favorite** person and they will attend the meeting **only if** they can sit next to their favorite person at the table. The favorite person of an employee is **not** themself.

Given a **0-indexed** integer array `favorite`, where `favorite[i]` denotes the favorite person of the i^{th} employee, return *the* ***maximum number of employees*** *that can be invited to the meeting.*

1.1K | 14

Submissions

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	362 ms	318.5 MB	

332. Reconstruct Itinerary

Solved

Hard | Topics | Companies

You are given a list of airline `tickets` where `tickets[i] = [from_i, to_i]` represent the departure and the arrival airports of one flight. Reconstruct the itinerary in order and return it.

All of the tickets belong to a man who departs from `"JFK"`, thus, the itinerary must begin with `"JFK"`. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.

- For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.

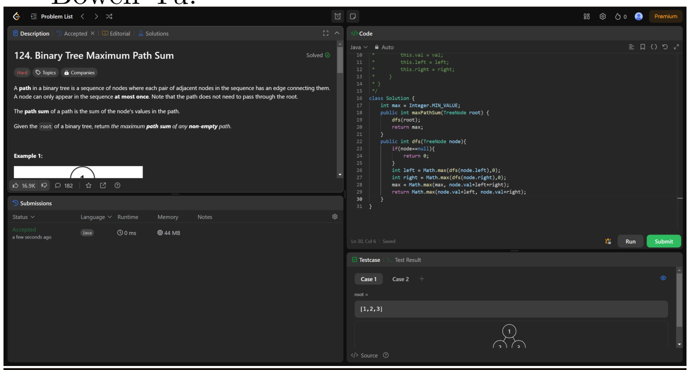
You may assume all tickets form at least one valid itinerary. You must use all the tickets once and only once.

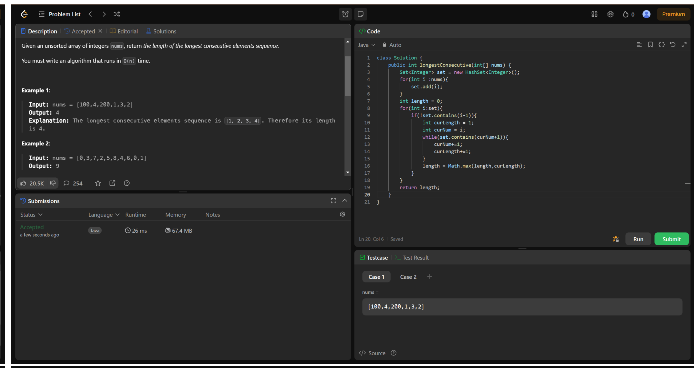
6K | 112

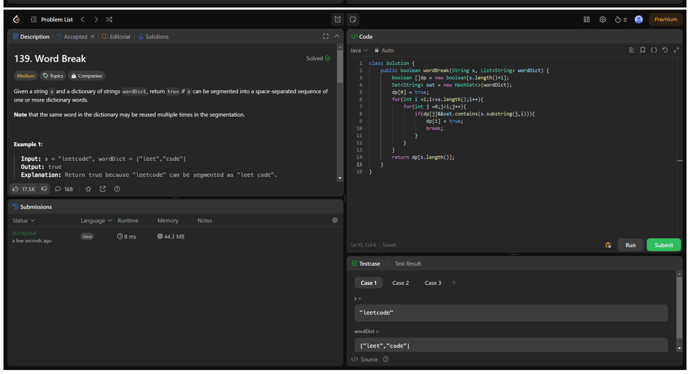
Submissions

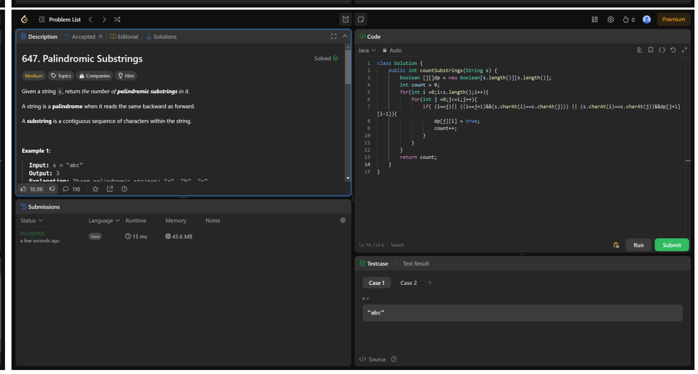
Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	C++	1 ms	18.1 MB	

Bowen Yu:


124. Binary Tree Maximum Path Sum
Hard
Solved
Accepted


Accepted


139. Word Break
Medium
Solved
Accepted


647. Palindromic Substrings
Medium
Solved
Accepted