ASSIGNMENT-7

1.Height of Binary Tree After Subtree Removal Queries

You are given the root of a binary tree with n nodes. Each node is assigned a unique value
from 1 to n. You are also given an array queries of size m.You have to perform m
independent queries on the tree where in the ith query you do the following:

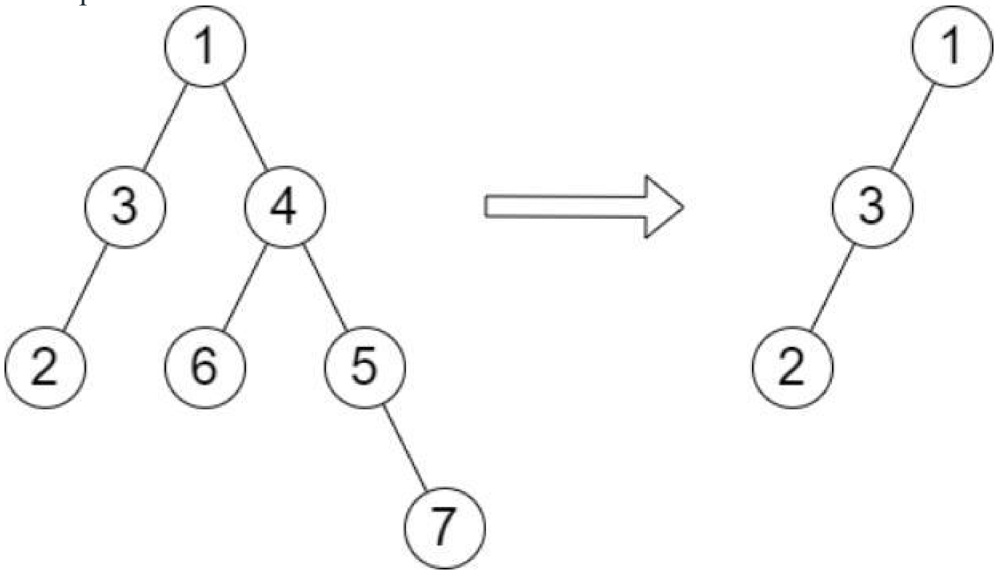
- Remove the subtree rooted at the node with the value queries[i] from the tree. It is guaranteed that queries[i] will not be equal to the value of the root.

Return *an array* answer *of size* m *where* answer[i] *is the height of the tree after performing*
the ith *query*.

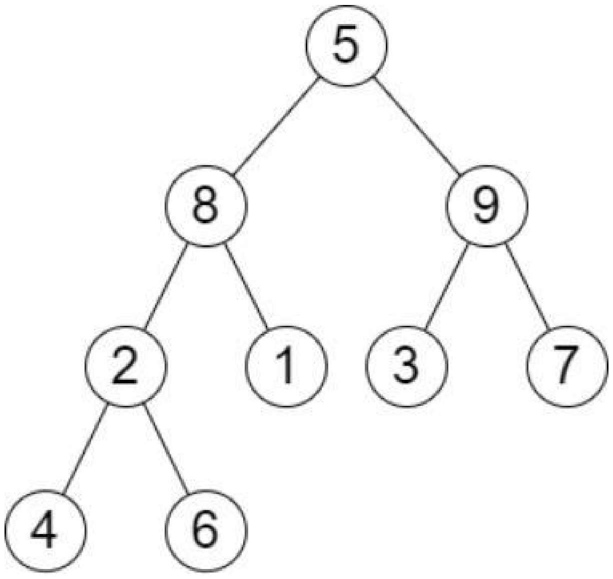
Note:

- The queries are independent, so the tree returns to its initial state after each query.
- The height of a tree is the number of edges in the longest simple path from the root to
some node in the tree.

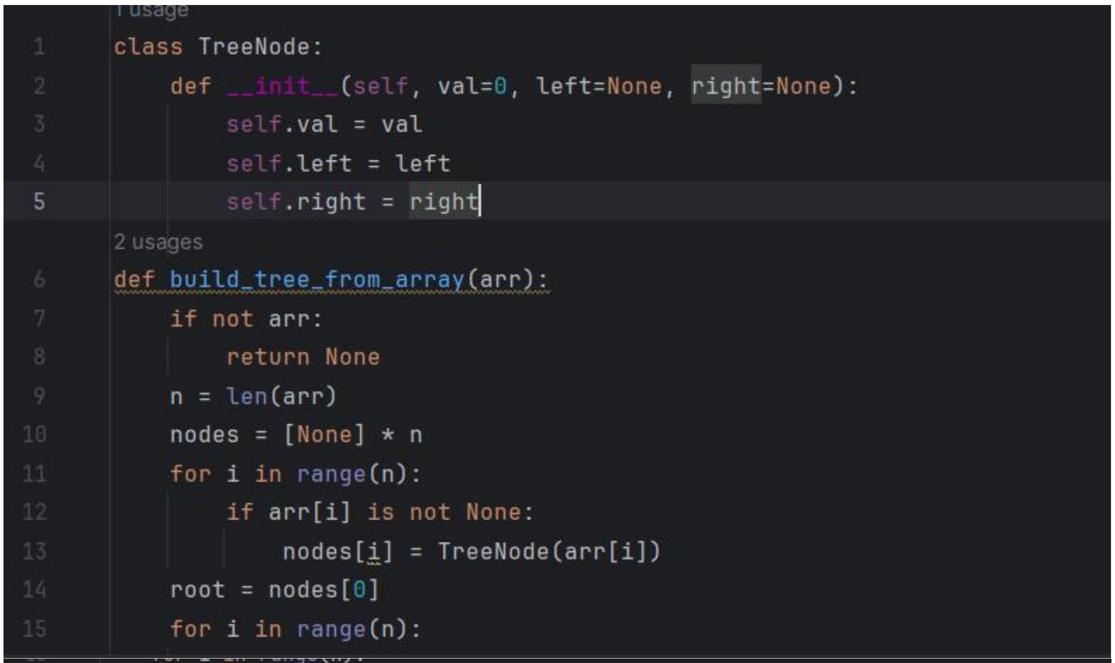
Example 1:



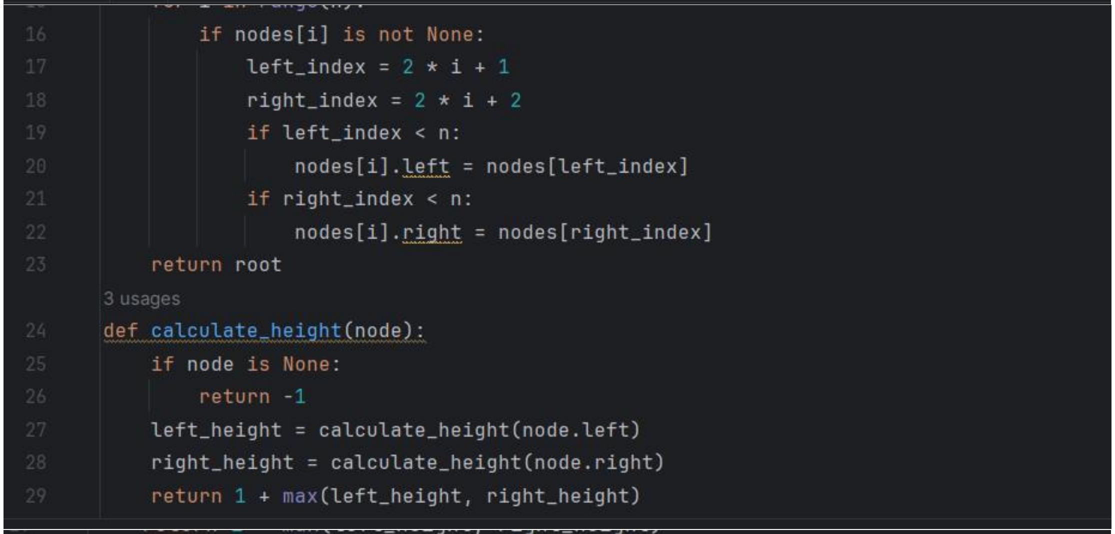
Input: root = [1,3,4,2,null,6,5,null,null,null,null,null,7], queries = [4]
Output: [2]
Explanation: The diagram above shows the tree after removing the subtree rooted at node
with value 4.
The height of the tree is 2 (The path 1 -> 3 -> 2).
Example 2:


Input: root = [5,8,9,2,1,3,7,4,6], queries = [3,2,4,8]
Output: [3,2,3,2]
Explanation: We have the following queries:
- Removing the subtree rooted at node with value 3. The height of the tree becomes 3
(The
path 5 -> 8 -> 2 -> 4).
- Removing the subtree rooted at node with value 2. The height of the tree becomes 2
(The
path 5 -> 8 -> 1).
- Removing the subtree rooted at node with value 4. The height of the tree becomes 3
(The
path 5 -> 8 -> 2 -> 6).
- Removing the subtree rooted at node with value 8. The height of the tree becomes 2
(The
path 5 -> 9 -> 3).
Constraints:
- The number of nodes in the tree is n.
- 2 <= n <= 105
- 1 <= Node.val <= n
- All the values in the tree are unique.
- m == queries.length
- 1 <= m <= min(n, 104)
- 1 <= queries[i] <= n
- queries[i] != root.val

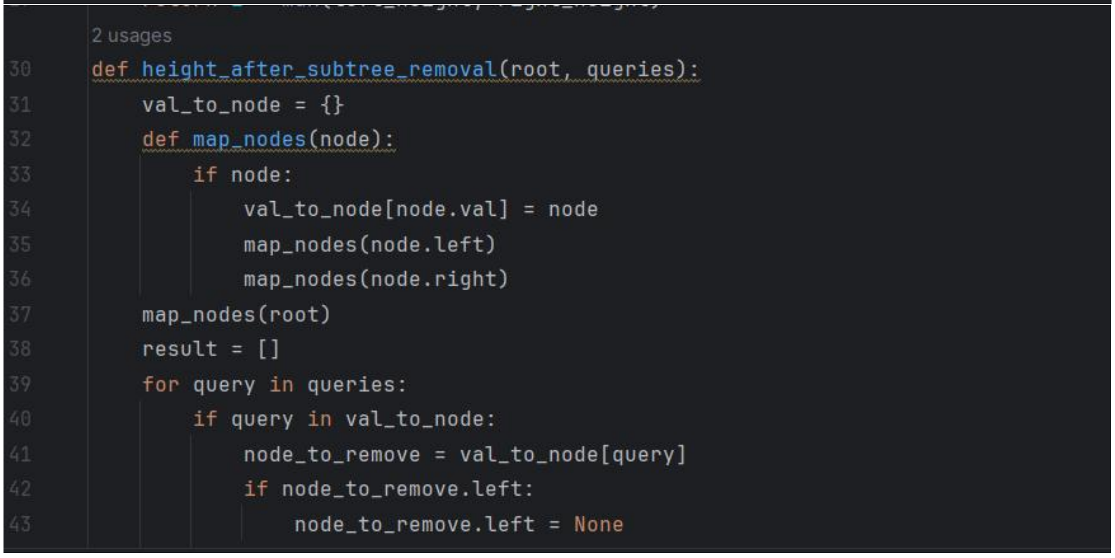
CODING:



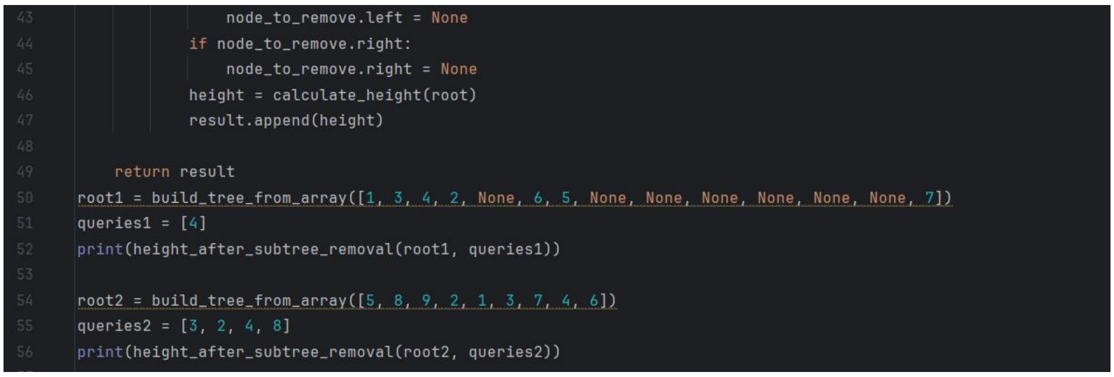
```
1 usage
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
2 usages
def build_tree_from_array(arr):
    if not arr:
        return None
    n = len(arr)
    nodes = [None] * n
    for i in range(n):
        if arr[i] is not None:
            nodes[i] = TreeNode(arr[i])
    root = nodes[0]
    for i in range(n):
```



```
        if nodes[i] is not None:
            left_index = 2 * i + 1
            right_index = 2 * i + 2
            if left_index < n:
                nodes[i].left = nodes[left_index]
            if right_index < n:
                nodes[i].right = nodes[right_index]
    return root
3 usages
def calculate_height(node):
    if node is None:
        return -1
    left_height = calculate_height(node.left)
    right_height = calculate_height(node.right)
    return 1 + max(left_height, right_height)
```

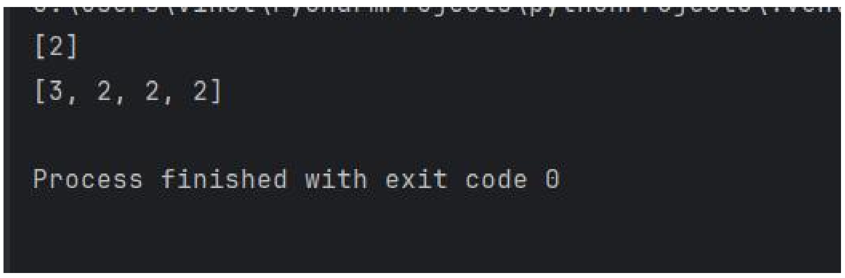


```
2 usages
def height_after_subtree_removal(root, queries):
    val_to_node = {}
    def map_nodes(node):
        if node:
            val_to_node[node.val] = node
            map_nodes(node.left)
            map_nodes(node.right)
    map_nodes(root)
    result = []
    for query in queries:
        if query in val_to_node:
            node_to_remove = val_to_node[query]
            if node_to_remove.left:
                node_to_remove.left = None
```



```python
                node_to_remove.left = None
            if node_to_remove.right:
                node_to_remove.right = None
            height = calculate_height(root)
            result.append(height)

    return result
root1 = build_tree_from_array([1, 3, 4, 2, None, 6, 5, None, None, None, None, None, None, 7])
queries1 = [4]
print(height_after_subtree_removal(root1, queries1))

root2 = build_tree_from_array([5, 8, 9, 2, 1, 3, 7, 4, 6])
queries2 = [3, 2, 4, 8]
print(height_after_subtree_removal(root2, queries2))
```

OUTPUT:



```
[2]
[3, 2, 2, 2]

Process finished with exit code 0
```

2. Sort Array by Moving Items to Empty Space

You are given an integer array nums of size n containing each element from 0 to n - 1 (inclusive). Each of the elements from 1 to n - 1 represents an item, and the element 0 represents an empty space.
In one operation, you can move any item to the empty space. nums is considered to be sorted
if the numbers of all the items are in ascending order and the empty space is either at the
beginning or at the end of the array.
For example, if n = 4, nums is sorted if:

- nums = [0,1,2,3] or
- nums = [1,2,3,0]

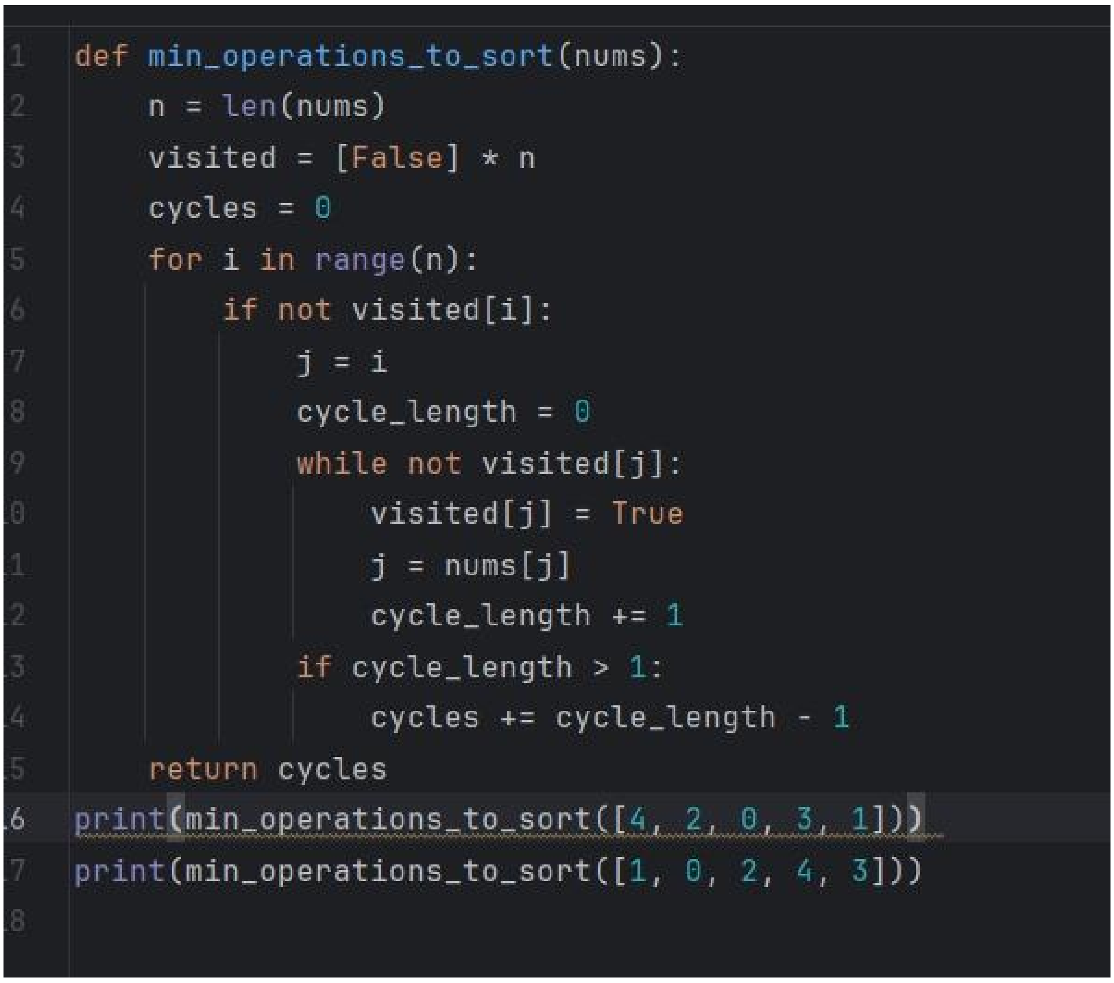
...and considered to be unsorted otherwise.Return *the minimum number of operations needed*
to sort nums.
Example 1:
Input: nums = [4,2,0,3,1]
Output: 3
Explanation:
- Move item 2 to the empty space. Now, nums = [4,0,2,3,1].
- Move item 1 to the empty space. Now, nums = [4,1,2,3,0].
- Move item 4 to the empty space. Now, nums = [0,1,2,3,4].

It can be proven that 3 is the minimum number of operations needed.
Example 2:
Input: nums = [1,2,3,4,0]
Output: 0
Explanation: nums is already sorted so return 0.Example 3:

Input: nums = [1,0,2,4,3]
Output: 2
Explanation:
- Move item 2 to the empty space. Now, nums = [1,2,0,4,3].
- Move item 3 to the empty space. Now, nums = [1,2,3,4,0].
It can be proven that 2 is the minimum number of operations needed.
Constraints:

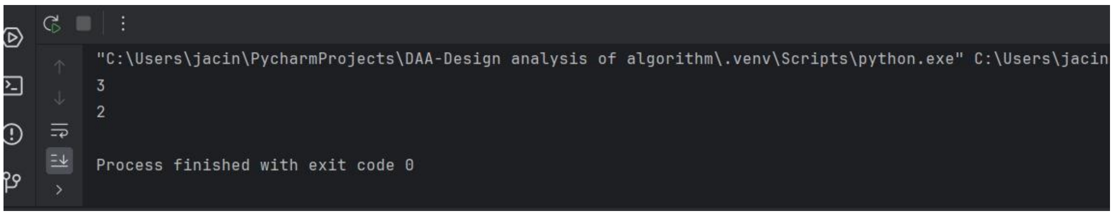
- n == nums.length
- 2 <= n <= 105
- 0 <= nums[i] < n
- All the values of nums are unique.\

CODING:



```python
def min_operations_to_sort(nums):
    n = len(nums)
    visited = [False] * n
    cycles = 0
    for i in range(n):
        if not visited[i]:
            j = i
            cycle_length = 0
            while not visited[j]:
                visited[j] = True
                j = nums[j]
                cycle_length += 1
            if cycle_length > 1:
                cycles += cycle_length - 1
    return cycles
print(min_operations_to_sort([4, 2, 0, 3, 1]))
print(min_operations_to_sort([1, 0, 2, 4, 3]))
```

OUTPUT:



```
"C:\Users\jacin\PycharmProjects\DAA-Design analysis of algorithm\.venv\Scripts\python.exe" C:\Users\jacin
3
2

Process finished with exit code 0
```

3. Apply Operations to an Array

You are given a 0-indexed array nums of size n consisting of non-negative integers.You need
to apply n - 1 operations to this array where, in the ith operation (0-indexed), you will apply
the following on the ith element of nums:

- If nums[i] == nums[i + 1], then multiply nums[i] by 2 and set nums[i + 1] to 0.

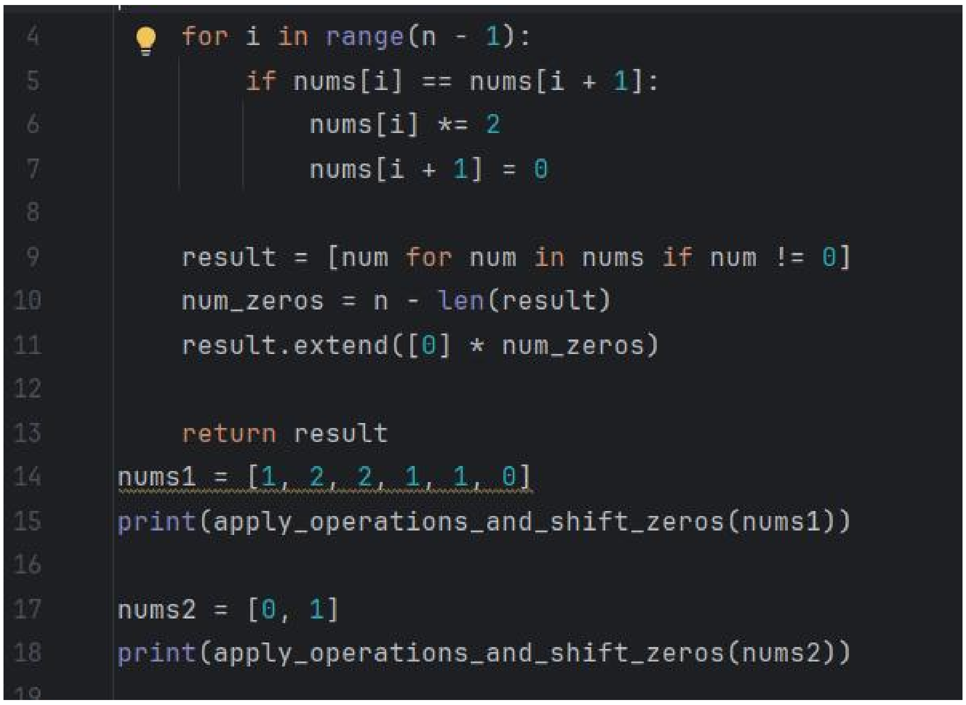
Otherwise, you skip this operation.
After performing all the operations, shift all the 0's to the end of the array.

- For example, the array [1,0,2,0,0,1] after shifting all its 0's to the end, is [1,2,1,0,0,0].

Return *the resulting array*.Note that the operations are applied sequentially, not all at once.
Example 1:
Input: nums = [1,2,2,1,1,0]
Output: [1,4,2,0,0,0]
Explanation: We do the following operations:
- i = 0: nums[0] and nums[1] are not equal, so we skip this operation.
- i = 1: nums[1] and nums[2] are equal, we multiply nums[1] by 2 and change nums[2] to 0.
The array becomes [1,4,0,1,1,0].
- i = 2: nums[2] and nums[3] are not equal, so we skip this operation.
- i = 3: nums[3] and nums[4] are equal, we multiply nums[3] by 2 and change nums[4] to 0.
The array becomes [1,4,0,2,0,0].
- i = 4: nums[4] and nums[5] are equal, we multiply nums[4] by 2 and change nums[5] to 0.
The array becomes [1,4,0,2,0,0].
After that, we shift the 0's to the end, which gives the array [1,4,2,0,0,0].
Example 2:
Input: nums = [0,1]
Output: [1,0]
Explanation: No operation can be applied, we just shift the 0 to the end.
Constraints:

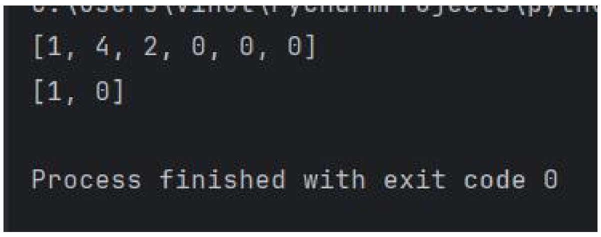
- 2 <= nums.length <= 2000
- 0 <= nums[i] <= 1000

CODING:



```python
    for i in range(n - 1):
        if nums[i] == nums[i + 1]:
            nums[i] *= 2
            nums[i + 1] = 0

    result = [num for num in nums if num != 0]
    num_zeros = n - len(result)
    result.extend([0] * num_zeros)

    return result
nums1 = [1, 2, 2, 1, 1, 0]
print(apply_operations_and_shift_zeros(nums1))

nums2 = [0, 1]
print(apply_operations_and_shift_zeros(nums2))
```

OUTPUT:



```
[1, 4, 2, 0, 0, 0]
[1, 0]

Process finished with exit code 0
```

4. Maximum Sum of Distinct Subarrays With Length K

You are given an integer array nums and an integer k. Find the maximum subarray sum of all
the subarrays of nums that meet the following conditions:

- The length of the subarray is k, and
- All the elements of the subarray are distinct.

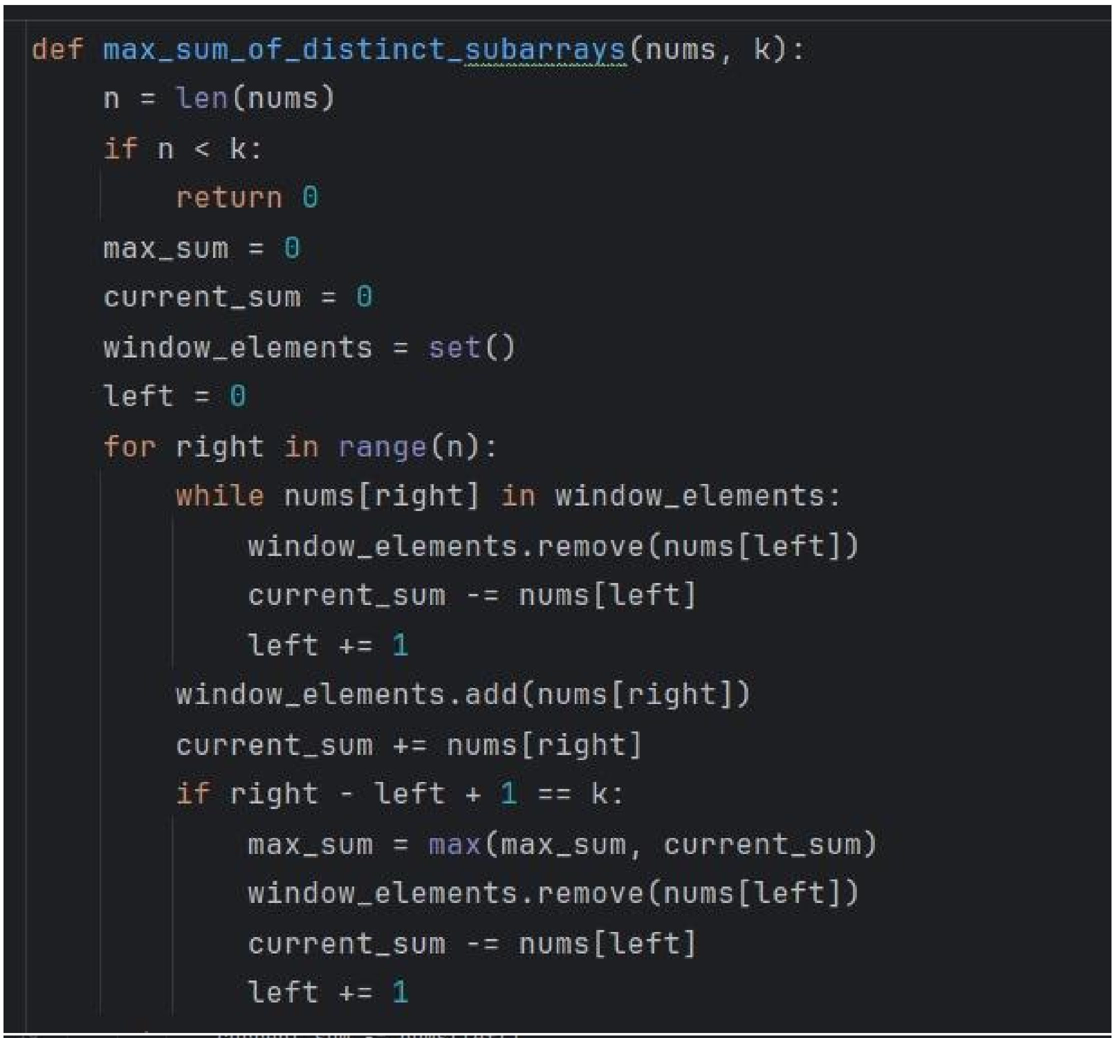
Return *the maximum subarray sum of all the subarrays that meet the conditions.* If no subarray meets the conditions, return 0. *A subarray is a contiguous non-empty sequence of*
elements within an array.

Example 1:
Input: nums = [1,5,4,2,9,9,9], k = 3
Output: 15
Explanation: The subarrays of nums with length 3 are:
- [1,5,4] which meets the requirements and has a sum of 10.
- [5,4,2] which meets the requirements and has a sum of 11.
- [4,2,9] which meets the requirements and has a sum of 15.
- [2,9,9] which does not meet the requirements because the element 9 is repeated.

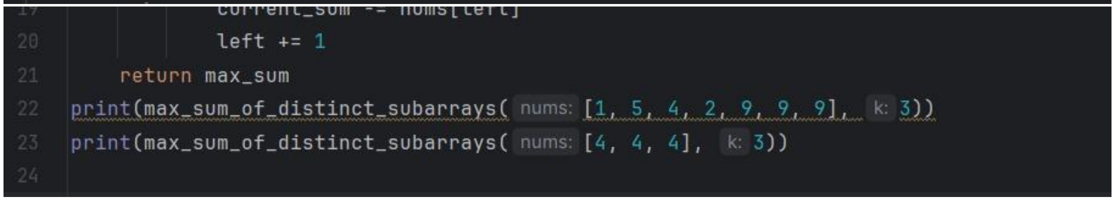
- [9,9,9] which does not meet the requirements because the element 9 is repeated.
We return 15 because it is the maximum subarray sum of all the subarrays that meet the
conditions
Example 2:
Input: nums = [4,4,4], k = 3
Output: 0
Explanation: The subarrays of nums with length 3 are:
- [4,4,4] which does not meet the requirements because the element 4 is repeated.
We return 0 because no subarrays meet the conditions.
Constraints:

- 1 <= k <= nums.length <= 105
- 1 <= nums[i] <= 105

CODING:

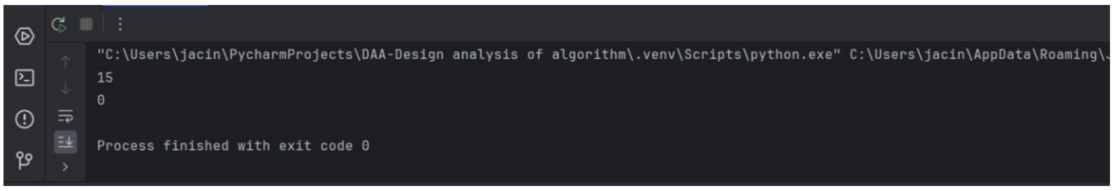


```python
def max_sum_of_distinct_subarrays(nums, k):
    n = len(nums)
    if n < k:
        return 0
    max_sum = 0
    current_sum = 0
    window_elements = set()
    left = 0
    for right in range(n):
        while nums[right] in window_elements:
            window_elements.remove(nums[left])
            current_sum -= nums[left]
            left += 1
        window_elements.add(nums[right])
        current_sum += nums[right]
        if right - left + 1 == k:
            max_sum = max(max_sum, current_sum)
            window_elements.remove(nums[left])
            current_sum -= nums[left]
            left += 1
```



```python
            left += 1
    return max_sum
print(max_sum_of_distinct_subarrays( nums: [1, 5, 4, 2, 9, 9, 9],  k: 3))
print(max_sum_of_distinct_subarrays( nums: [4, 4, 4],  k: 3))
```

OUTPUT:



```
"C:\Users\jacin\PycharmProjects\DAA-Design analysis of algorithm\.venv\Scripts\python.exe" C:\Users\jacin\AppData\Roaming\
15
0

Process finished with exit code 0
```

5. Total Cost to Hire K Workers

You are given a 0-indexed integer array costs where costs[i] is the cost of hiring the ith
worker.You are also given two integers k and candidates. We want to hire exactly k workers
according to the following rules:

● You will run k sessions and hire exactly one worker in each session.

● In each hiring session, choose the worker with the lowest cost from either the first candidates workers or the last candidates workers. Break the tie by the smallest index.

○ For example, if costs = [3,2,7,7,1,2] and candidates = 2, then in the first hiring session, we will choose the 4th worker because they have the lowest cost [3,2,7,7,1,2].

○ In the second hiring session, we will choose 1st worker because they have the same lowest cost as 4th worker but they have the smallest index [3,2,7,7,2].
Please note that the indexing may be changed in the process.

● If there are fewer than candidates workers remaining, choose the worker with the lowest cost among them. Break the tie by the smallest index.

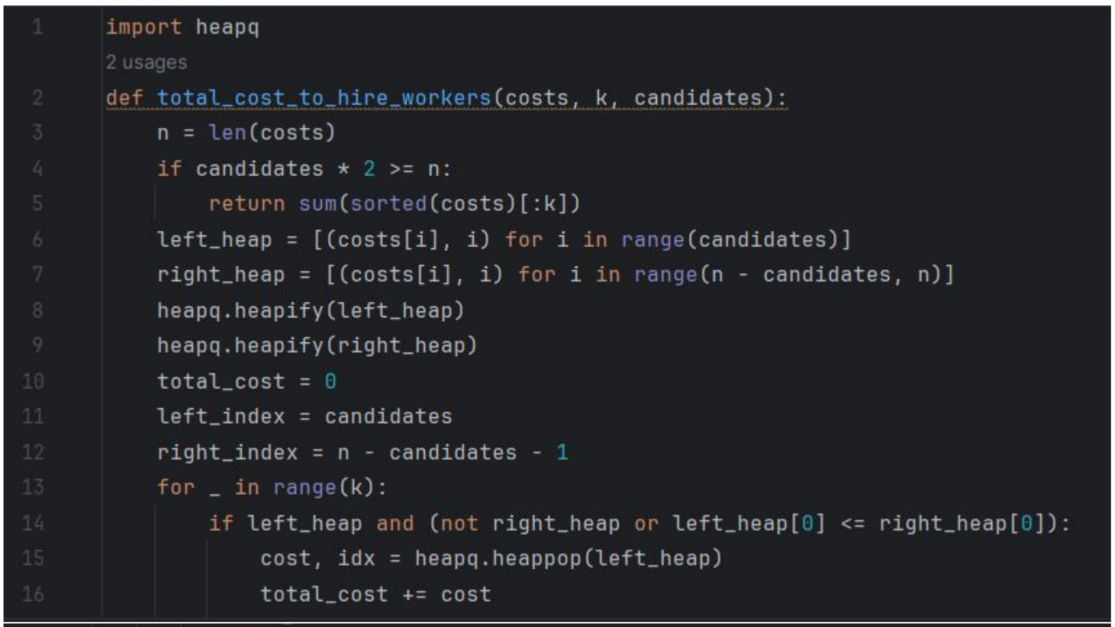
● A worker can only be chosen once.

Return *the total cost to hire exactly* k *workers.*Example 1:
Input: costs = [17,12,10,2,7,2,11,20,8], k = 3, candidates = 4
Output: 11
Explanation: We hire 3 workers in total. The total cost is initially 0.
- In the first hiring round we choose the worker from [17,12,10,2,7,2,11,20,8]. The lowest
cost is 2, and we break the tie by the smallest index, which is 3. The total cost = 0 + 2 = 2.
- In the second hiring round we choose the worker from [17,12,10,7,2,11,20,8]. The lowest
cost is 2 (index 4). The total cost = 2 + 2 = 4.
- In the third hiring round we choose the worker from [17,12,10,7,11,20,8]. The lowest cost is
7 (index 3). The total cost = 4 + 7 = 11. Notice that the worker with index 3 was common in
the first and last four workers.
The total hiring cost is 11.
Example 2:
Input: costs = [1,2,4,1], k = 3, candidates = 3
Output: 4
Explanation: We hire 3 workers in total. The total cost is initially 0.

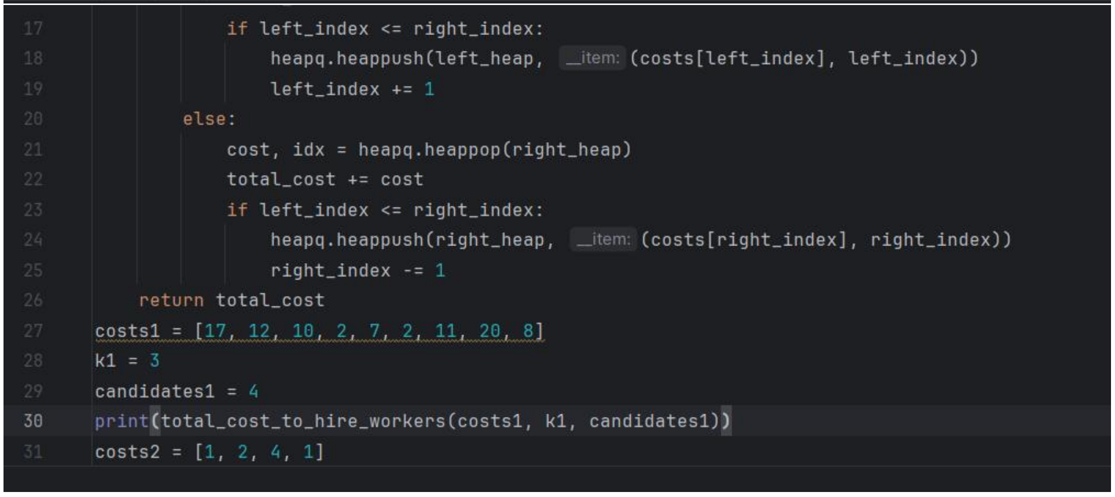
- In the first hiring round we choose the worker from [1,2,4,1]. The lowest cost is 1, and we
break the tie by the smallest index, which is 0. The total cost = 0 + 1 = 1. Notice that workers
with index 1 and 2 are common in the first and last 3 workers.
- In the second hiring round we choose the worker from [2,4,1]. The lowest cost is 1 (index 2).
The total cost = 1 + 1 = 2.
- In the third hiring round there are less than three candidates. We choose the worker from the
remaining workers [2,4]. The lowest cost is 2 (index 0). The total cost = 2 + 2 = 4.
The total hiring cost is 4.
Constraints:

- 1 <= costs.length <= 105
- 1 <= costs[i] <= 105
- 1 <= k, candidates <= costs.length

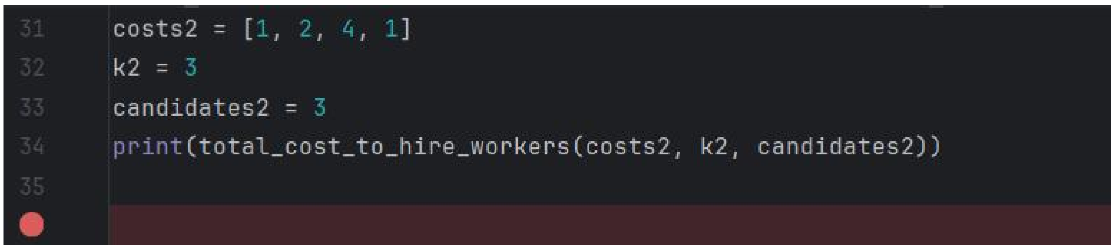
CODING:



```python
import heapq
2 usages
def total_cost_to_hire_workers(costs, k, candidates):
    n = len(costs)
    if candidates * 2 >= n:
        return sum(sorted(costs)[:k])
    left_heap = [(costs[i], i) for i in range(candidates)]
    right_heap = [(costs[i], i) for i in range(n - candidates, n)]
    heapq.heapify(left_heap)
    heapq.heapify(right_heap)
    total_cost = 0
    left_index = candidates
    right_index = n - candidates - 1
    for _ in range(k):
        if left_heap and (not right_heap or left_heap[0] <= right_heap[0]):
            cost, idx = heapq.heappop(left_heap)
            total_cost += cost
```

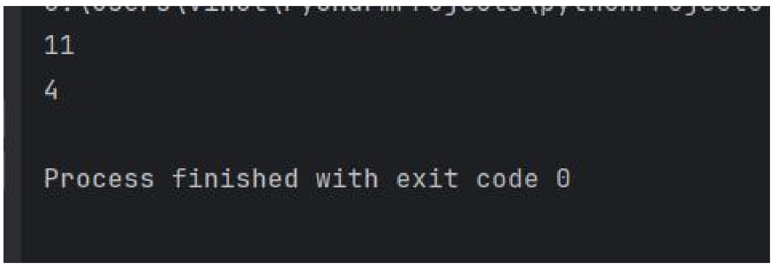


```python
            if left_index <= right_index:
                heapq.heappush(left_heap, (costs[left_index], left_index))
                left_index += 1
        else:
            cost, idx = heapq.heappop(right_heap)
            total_cost += cost
            if left_index <= right_index:
                heapq.heappush(right_heap, (costs[right_index], right_index))
                right_index -= 1
    return total_cost
costs1 = [17, 12, 10, 2, 7, 2, 11, 20, 8]
k1 = 3
candidates1 = 4
print(total_cost_to_hire_workers(costs1, k1, candidates1))
costs2 = [1, 2, 4, 1]
```



```
costs2 = [1, 2, 4, 1]
k2 = 3
candidates2 = 3
print(total_cost_to_hire_workers(costs2, k2, candidates2))
```

OUTPUT:



```
11
4

Process finished with exit code 0
```

6. Minimum Total Distance Traveled

There are some robots and factories on the X-axis. You are given an integer array robot
where robot[i] is the position of the ith robot. You are also given a 2D integer array factory
where factory[j] = [positionj, limitj] indicates that positionj is the position of the jth factory
and that the jth factory can repair at most limitj robots.
The positions of each robot are unique. The positions of each factory are also unique. Note
that a robot can be in the same position as a factory initially.
All the robots are initially broken; they keep moving in one direction. The direction could be
the negative or the positive direction of the X-axis. When a robot reaches a factory that did
not reach its limit, the factory repairs the robot, and it stops moving.
At any moment, you can set the initial direction of moving for some robot. Your target is to
minimize the total distance traveled by all the robots.
Return *the minimum total distance traveled by all the robots*. The test cases are generated
such that all the robots can be repaired.
Note that● All robots move at the same speed.

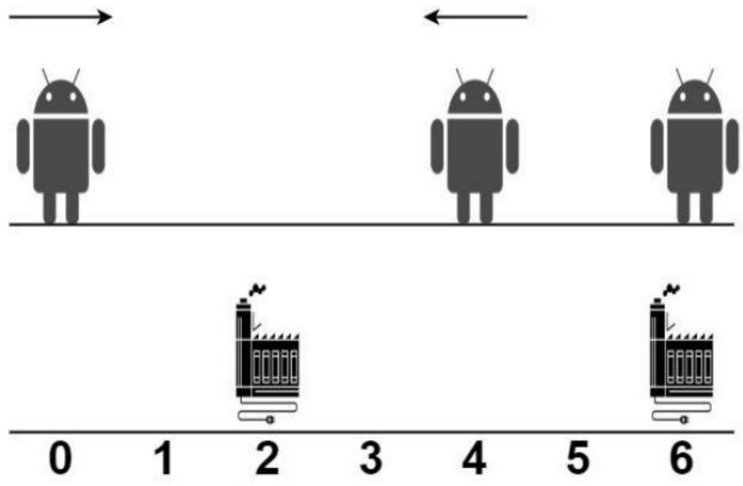
● If two robots move in the same direction, they will never collide.

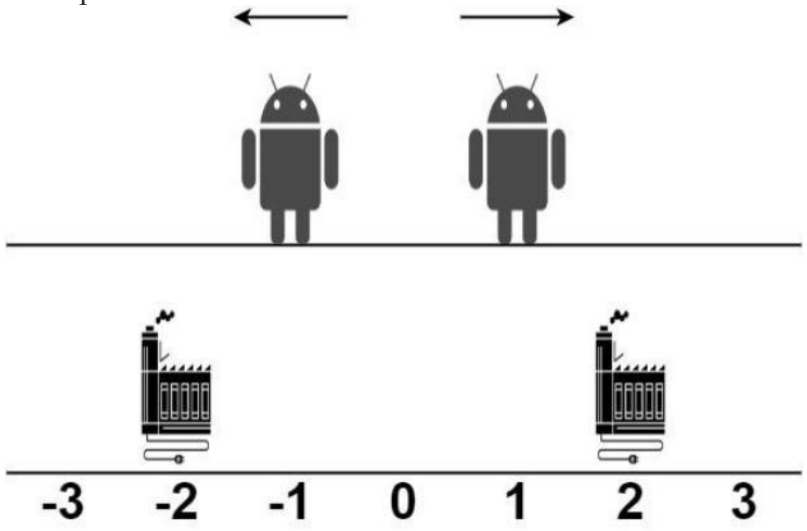
● If two robots move in opposite directions and they meet at some point, they do not collide. They cross each other.

● If a robot passes by a factory that reached its limits, it crosses it as if it does not exist.

● If the robot moved from a position x to a position y, the distance it moved is |y - x|.

Example 1:



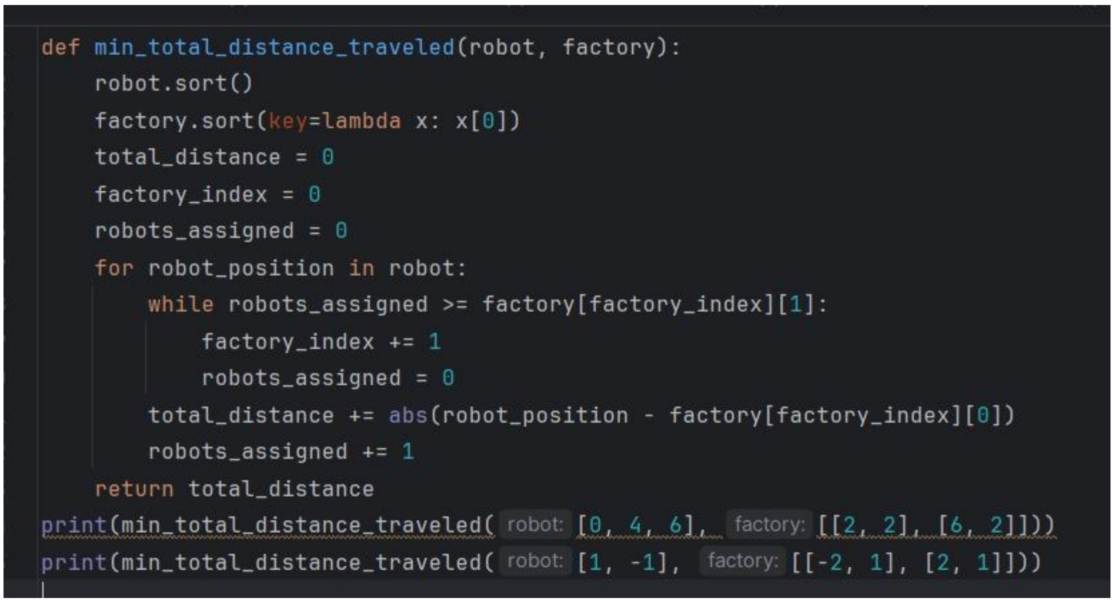
Input: robot = [0,4,6], factory = [[2,2],[6,2]]
Output: 4
Explanation: As shown in the figure:
- The first robot at position 0 moves in the positive direction. It will be repaired at the first
factory.
- The second robot at position 4 moves in the negative direction. It will be repaired at the first
factory.
- The third robot at position 6 will be repaired at the second factory. It does not need to move.
The limit of the first factory is 2, and it fixed 2 robots.
The limit of the second factory is 2, and it fixed 1 robot.
The total distance is |2 - 0| + |2 - 4| + |6 - 6| = 4. It can be shown that we cannot achieve a
better total distance than 4.
Example 2:


Input: robot = [1,-1], factory = [[-2,1],[2,1]]
Output: 2Explanation: As shown in the figure:
- The first robot at position 1 moves in the positive direction. It will be repaired at the second
factory.

- The second robot at position -1 moves in the negative direction. It will be repaired at the
first factory.
The limit of the first factory is 1, and it fixed 1 robot.
The limit of the second factory is 1, and it fixed 1 robot.
The total distance is |2 - 1| + |(-2) - (-1)| = 2. It can be shown that we cannot achieve a better
total distance than 2.
Constraints:

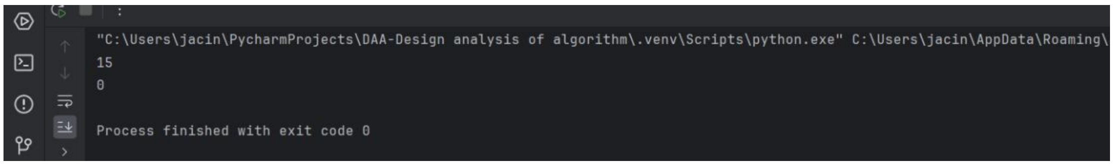
● 1 <= robot.length, factory.length <= 100
● factory[j].length == 2
● -109 <= robot[i], positionj <= 109
● 0 <= limitj <= robot.length
● The input will be generated such that it is always possible to repair every robot.

CODING:



```
def min_total_distance_traveled(robot, factory):
    robot.sort()
    factory.sort(key=lambda x: x[0])
    total_distance = 0
    factory_index = 0
    robots_assigned = 0
    for robot_position in robot:
        while robots_assigned >= factory[factory_index][1]:
            factory_index += 1
            robots_assigned = 0
        total_distance += abs(robot_position - factory[factory_index][0])
        robots_assigned += 1
    return total_distance
print(min_total_distance_traveled( robot: [0, 4, 6],  factory: [[2, 2], [6, 2]]))
print(min_total_distance_traveled( robot: [1, -1],  factory: [[-2, 1], [2, 1]]))
```

OUTPUT:



```
"C:\Users\jacin\PycharmProjects\DAA-Design analysis of algorithm\.venv\Scripts\python.exe" C:\Users\jacin\AppData\Roaming\
15
0

Process finished with exit code 0
```

7. Minimum Subarrays in a Valid Split

You are given an integer array nums.Splitting of an integer array nums into subarrays is valid
if:

● the *greatest common divisor* of the first and last elements of each subarray is greater than 1, and
● each element of nums belongs to exactly one subarray.

Return *the minimum number of subarrays in a valid subarray splitting of* nums. If a valid
subarray splitting is not possible, return -1.

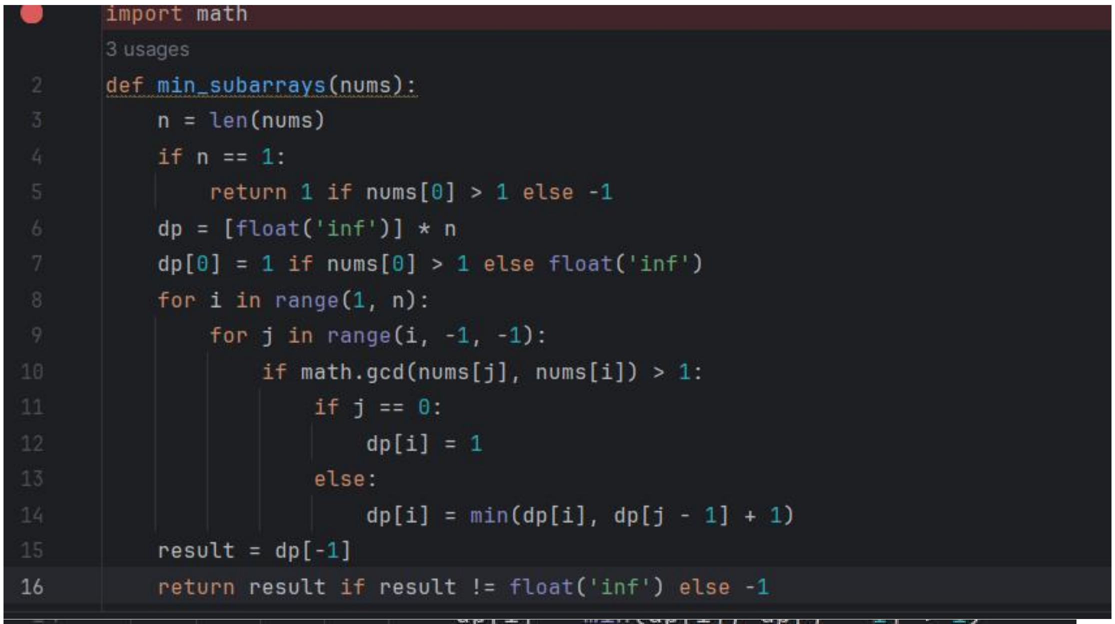
Note that:

- The greatest common divisor of two numbers is the largest positive integer that evenly divides both numbers.
- A subarray is a contiguous non-empty part of an array.

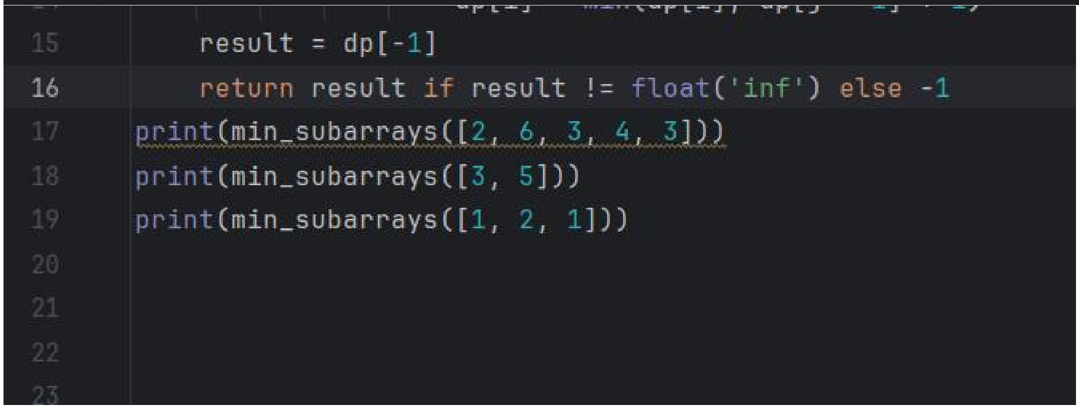
Example 1:
Input: nums = [2,6,3,4,3]
Output: 2
Explanation: We can create a valid split in the following way: [2,6] | [3,4,3].
- The starting element of the 1st subarray is 2 and the ending is 6. Their greatest common
divisor is 2, which is greater than 1.
- The starting element of the 2nd subarray is 3 and the ending is 3. Their greatest common
divisor is 3, which is greater than 1.
It can be proved that 2 is the minimum number of subarrays that we can obtain in a valid split.
Example 2:
Input: nums = [3,5]
Output: 2
Explanation: We can create a valid split in the following way: [3] | [5].
- The starting element of the 1st subarray is 3 and the ending is 3. Their greatest common
divisor is 3, which is greater than 1.
- The starting element of the 2nd subarray is 5 and the ending is 5. Their greatest common
divisor is 5, which is greater than 1.
It can be proved that 2 is the minimum number of subarrays that we can obtain in a valid split.Example 3:
Input: nums = [1,2,1]
Output: -1
Explanation: It is impossible to create valid split.
Constraints:

- 1 <= nums.length <= 1000
- 1 <= nums[i] <= 105

CODING:



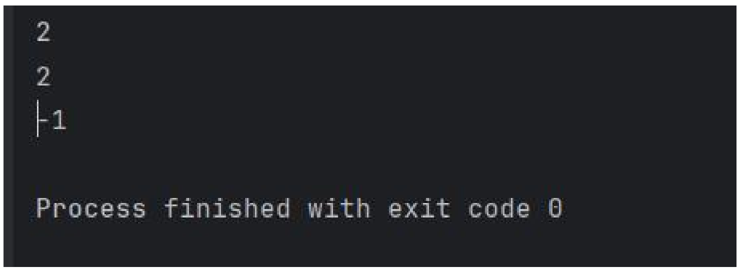
```python
import math
3 usages
def min_subarrays(nums):
    n = len(nums)
    if n == 1:
        return 1 if nums[0] > 1 else -1
    dp = [float('inf')] * n
    dp[0] = 1 if nums[0] > 1 else float('inf')
    for i in range(1, n):
        for j in range(i, -1, -1):
            if math.gcd(nums[j], nums[i]) > 1:
                if j == 0:
                    dp[i] = 1
                else:
                    dp[i] = min(dp[i], dp[j - 1] + 1)
    result = dp[-1]
    return result if result != float('inf') else -1
```



```python
    result = dp[-1]
    return result if result != float('inf') else -1
```

```python
print(min_subarrays([2, 6, 3, 4, 3]))
print(min_subarrays([3, 5]))
print(min_subarrays([1, 2, 1]))
```

OUTPUT:



```
2
2
-1

Process finished with exit code 0
```

8. Number of Distinct Averages

You are given a 0-indexed integer array nums of even length.
As long as nums is not empty, you must repetitively:

- Find the minimum number in nums and remove it.
- Find the maximum number in nums and remove it.
- Calculate the average of the two removed numbers.

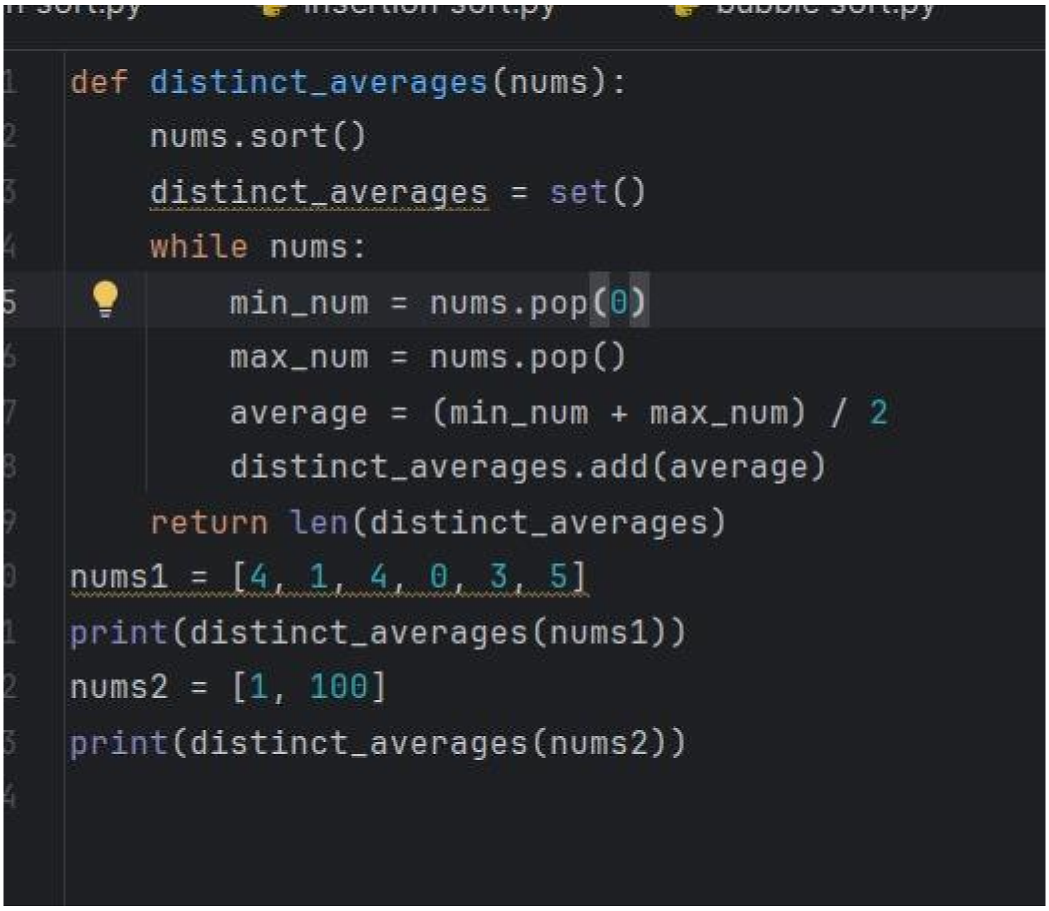
The average of two numbers a and b is (a + b) / 2.

- For example, the average of 2 and 3 is (2 + 3) / 2 = 2.5.

Return *the number of distinct averages calculated using the above process*.Note that when
there is a tie for a minimum or maximum number, any can be removed.
Example 1:
Input: nums = [4,1,4,0,3,5]
Output: 2
Explanation:
1. Remove 0 and 5, and the average is (0 + 5) / 2 = 2.5. Now, nums = [4,1,4,3].
2. Remove 1 and 4. The average is (1 + 4) / 2 = 2.5, and nums = [4,3].
3. Remove 3 and 4, and the average is (3 + 4) / 2 = 3.5.
Since there are 2 distinct numbers among 2.5, 2.5, and 3.5, we return 2.
Example 2:
Input: nums = [1,100]
Output: 1
Explanation:
There is only one average to be calculated after removing 1 and 100, so we return 1.
Constraints:

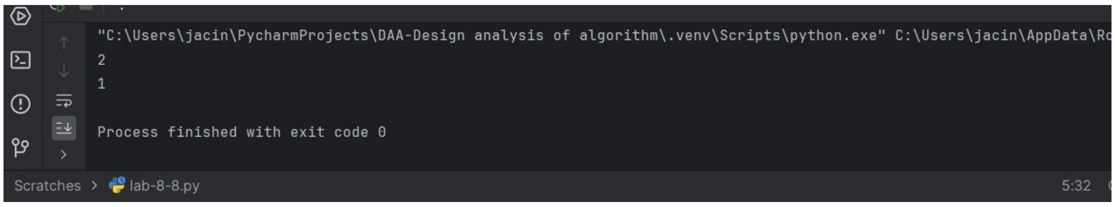
- 2 <= nums.length <= 100
- nums.length is even.
- 0 <= nums[i] <= 100

CODING:



```python
def distinct_averages(nums):
    nums.sort()
    distinct_averages = set()
    while nums:
        min_num = nums.pop(0)
        max_num = nums.pop()
        average = (min_num + max_num) / 2
        distinct_averages.add(average)
    return len(distinct_averages)
nums1 = [4, 1, 4, 0, 3, 5]
print(distinct_averages(nums1))
nums2 = [1, 100]
print(distinct_averages(nums2))
```

OUTPUT:



```
"C:\Users\jacin\PycharmProjects\DAA-Design analysis of algorithm\.venv\Scripts\python.exe" C:\Users\jacin\AppData\Ro
2
1

Process finished with exit code 0
```

Scratches > lab-8-8.py 5:32

9. Count Ways To Build Good Strings

Given the integers zero, one, low, and high, we can construct a string by starting with an
empty string, and then at each step perform either of the following:

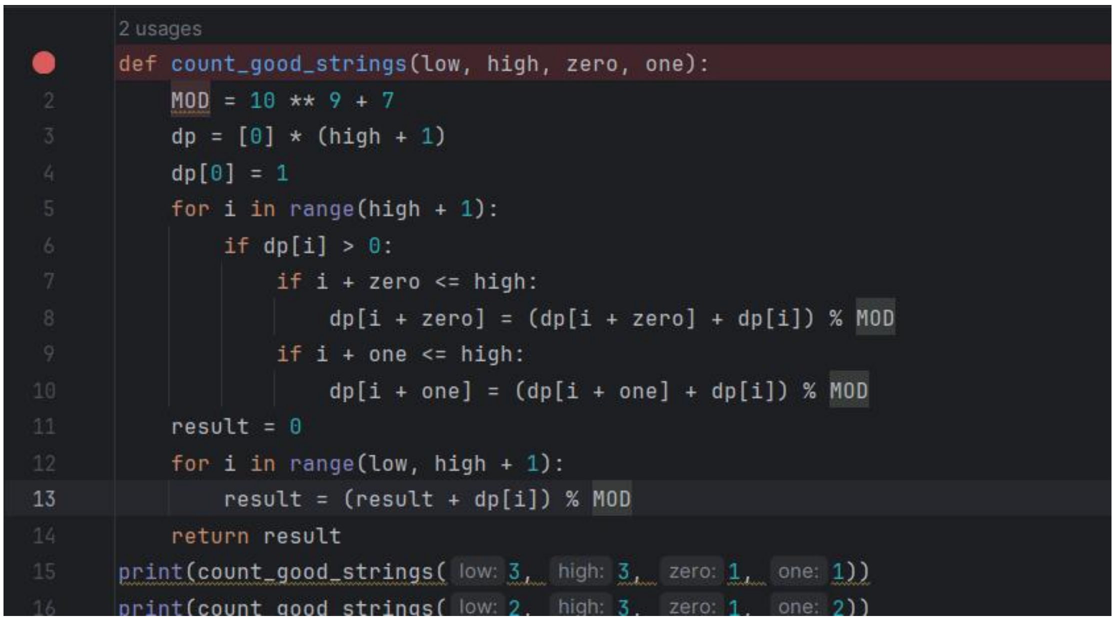
- Append the character '0' zero times.
- Append the character '1' one times.This can be performed any number of times.A good string is a string constructed by the
above process having a length between low and high (inclusive).

Return *the number of different good strings that can be constructed satisfying these properties.* Since the answer can be large, return it modulo 109 + 7.

Example 1:
Input: low = 3, high = 3, zero = 1, one = 1
Output: 8
Explanation:
One possible valid good string is "011".
It can be constructed as follows: "" -> "0" -> "01" -> "011".
All binary strings from "000" to "111" are good strings in this example.
Example 2:
Input: low = 2, high = 3, zero = 1, one = 2
Output: 5
Explanation: The good strings are "00", "11", "000", "110", and "011".
Constraints:

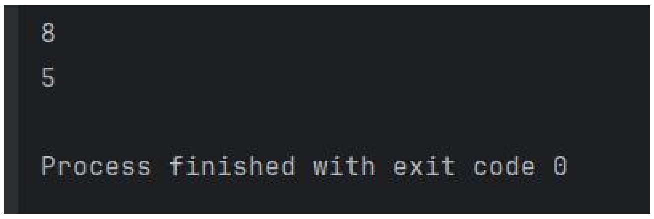
- 1 <= low <= high <= 105
- 1 <= zero, one <= low

CODING:



```
2 usages
def count_good_strings(low, high, zero, one):
    MOD = 10 ** 9 + 7
    dp = [0] * (high + 1)
    dp[0] = 1
    for i in range(high + 1):
        if dp[i] > 0:
            if i + zero <= high:
                dp[i + zero] = (dp[i + zero] + dp[i]) % MOD
            if i + one <= high:
                dp[i + one] = (dp[i + one] + dp[i]) % MOD
    result = 0
    for i in range(low, high + 1):
        result = (result + dp[i]) % MOD
    return result
print(count_good_strings( low: 3, high: 3, zero: 1, one: 1))
print(count_good_strings( low: 2, high: 3, zero: 1, one: 2))
```

OUTPUT:



```
8
5

Process finished with exit code 0
```

10. Most Profitable Path in a Tree

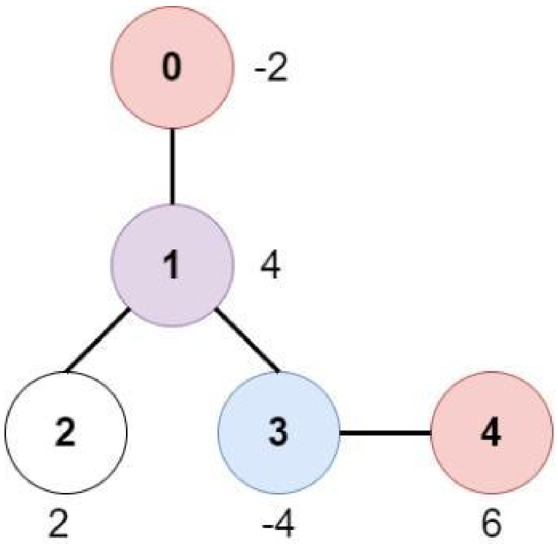
There is an undirected tree with n nodes labeled from 0 to n - 1, rooted at node 0. You are
given a 2D integer array edges of length n - 1 where edges[i] = [ai, bi] indicates that there is
an edge between nodes ai and bi in the tree.
At every node i, there is a gate. You are also given an array of even integers amount, where
amount[i] represents:

- the price needed to open the gate at node i, if amount[i] is negative, or,
- the cash reward obtained on opening the gate at node i, otherwise.

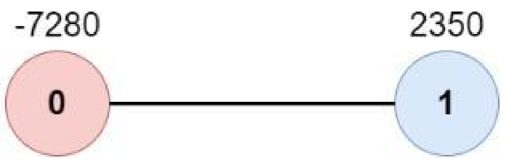
The game goes on as follows:

- Initially, Alice is at node 0 and Bob is at node bob.
- At every second, Alice and Bob each move to an adjacent node. Alice moves towards
some leaf node, while Bob moves towards node 0.
- For every node along their path, Alice and Bob either spend money to open the gate at
that node, or accept the reward. Note that:
  - If the gate is already open, no price will be required, nor will there be any cash reward.
  - If Alice and Bob reach the node simultaneously, they share the price/reward
for opening the gate there. In other words, if the price to open the gate is c,
then both Alice and Bob pay c / 2 each. Similarly, if the reward at the gate is c,
both of them receive c / 2 each.
- If Alice reaches a leaf node, she stops moving. Similarly, if Bob reaches node 0, he stops moving. Note that these events are independent of each other.

Return *the maximum net income Alice can have if she travels towards the optimal leaf node.*Example 1:



Input: edges = [[0,1],[1,2],[1,3],[3,4]], bob = 3, amount = [-2,4,2,-4,6]
Output: 6
Explanation:
The above diagram represents the given tree. The game goes as follows:
- Alice is initially on node 0, Bob on node 3. They open the gates of their respective nodes.
Alice's net income is now -2.
- Both Alice and Bob move to node 1.
Since they reach here simultaneously, they open the gate together and share the reward.
Alice's net income becomes -2 + (4 / 2) = 0.
- Alice moves on to node 3. Since Bob already opened its gate, Alice's income remains
unchanged.
Bob moves on to node 0, and stops moving.
- Alice moves on to node 4 and opens the gate there. Her net income becomes 0 + 6 = 6.
Now, neither Alice nor Bob can make any further moves, and the game ends.
It is not possible for Alice to get a higher net income.
Example 2:

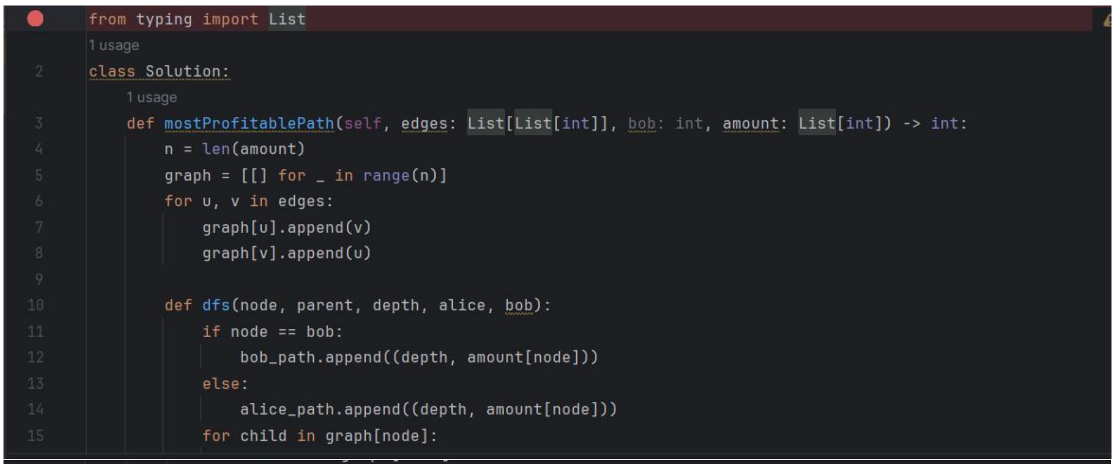


Input: edges = [[0,1]], bob = 1, amount = [-7280,2350]
Output: -7280
Explanation:
Alice follows the path 0->1 whereas Bob follows the path 1->0.
Thus, Alice opens the gate at node 0 only. Hence, her net income is -7280.
Constraints:

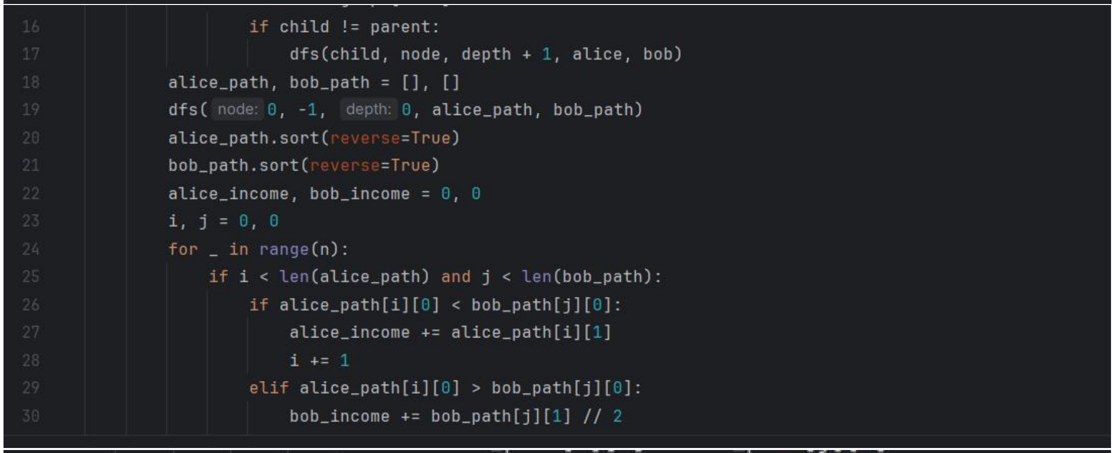
- 2 <= n <= 105
- edges.length == n - 1
- edges[i].length == 2
- 0 <= ai, bi < n

- ai != bi
- edges represents a valid tree.
- 1 <= bob < n
- amount.length == n
- amount[i] is an even integer in the range [-104, 104].

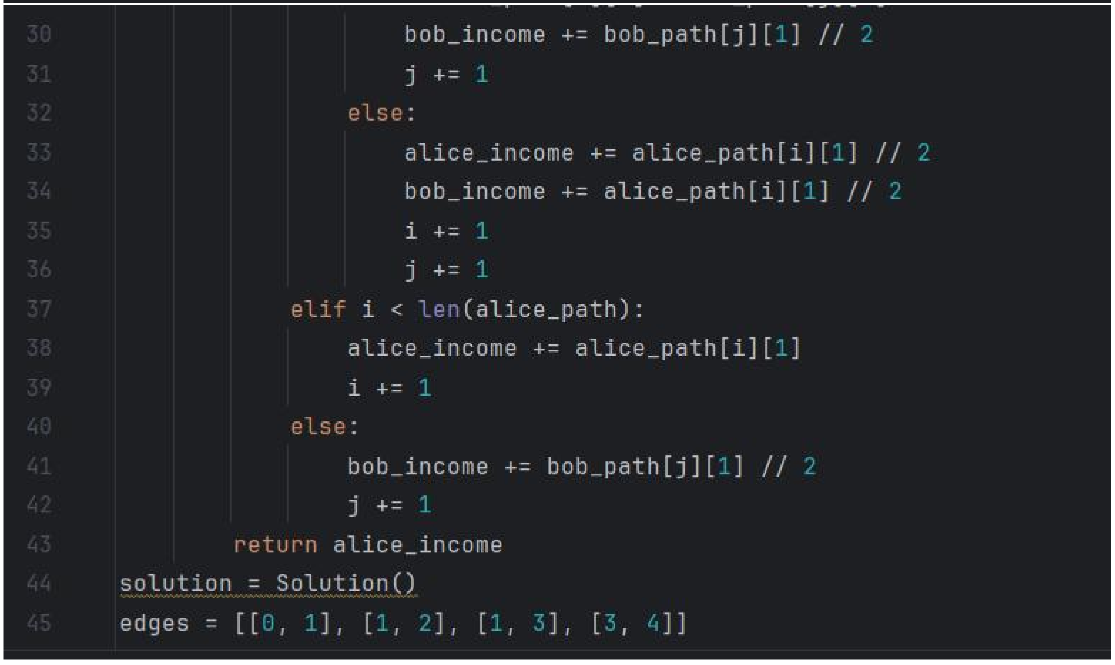
CODING:



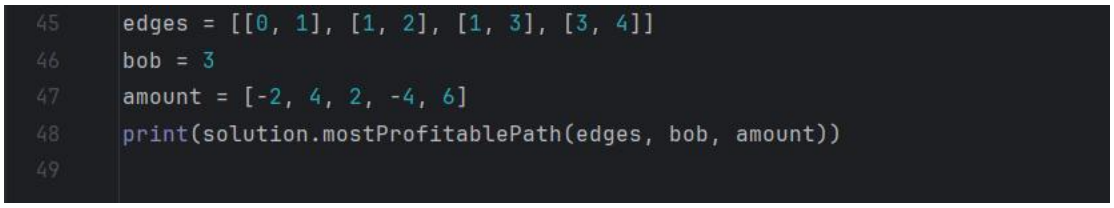
```python
from typing import List
1 usage
class Solution:
    1 usage
    def mostProfitablePath(self, edges: List[List[int]], bob: int, amount: List[int]) -> int:
        n = len(amount)
        graph = [[] for _ in range(n)]
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        def dfs(node, parent, depth, alice, bob):
            if node == bob:
                bob_path.append((depth, amount[node]))
            else:
                alice_path.append((depth, amount[node]))
            for child in graph[node]:
```



```python
                if child != parent:
                    dfs(child, node, depth + 1, alice, bob)
        alice_path, bob_path = [], []
        dfs( node: 0, -1,  depth: 0, alice_path, bob_path)
        alice_path.sort(reverse=True)
        bob_path.sort(reverse=True)
        alice_income, bob_income = 0, 0
        i, j = 0, 0
        for _ in range(n):
            if i < len(alice_path) and j < len(bob_path):
                if alice_path[i][0] < bob_path[j][0]:
                    alice_income += alice_path[i][1]
                    i += 1
                elif alice_path[i][0] > bob_path[j][0]:
                    bob_income += bob_path[j][1] // 2
```

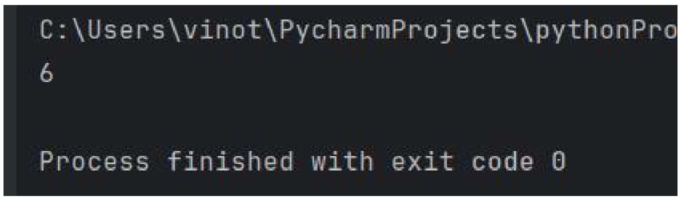


```python
                    bob_income += bob_path[j][1] // 2
                    j += 1
                else:
                    alice_income += alice_path[i][1] // 2
                    bob_income += alice_path[i][1] // 2
                    i += 1
                    j += 1
            elif i < len(alice_path):
                alice_income += alice_path[i][1]
                i += 1
            else:
                bob_income += bob_path[j][1] // 2
                j += 1
        return alice_income
solution = Solution()
edges = [[0, 1], [1, 2], [1, 3], [3, 4]]
```



```
edges = [[0, 1], [1, 2], [1, 3], [3, 4]]
bob = 3
amount = [-2, 4, 2, -4, 6]
print(solution.mostProfitablePath(edges, bob, amount))

```

OUTPUT:



```
C:\Users\vinot\PycharmProjects\pythonPro
6

Process finished with exit code 0
```