```python
from typing import List, Optional
from collections import deque
class TreeNode:
   def __init__(self, x):
      self.val = x
      self.left = None
      self.right = None
def build_tree(values):
   if not values:
      return None
   root = TreeNode(values[0])
   queue = deque([root])
   i = 1
   while i < len(values):
      current = queue.popleft()
      if values[i] is not None:
         current.left = TreeNode(values[i])
         queue.append(current.left)
      i += 1
      if i < len(values) and values[i] is not None:
         current.right = TreeNode(values[i])
         queue.append(current.right)
      i += 1
   return root
def height_of_tree(node):
   if not node:
      return -1
   left_height = height_of_tree(node.left)
   right_height = height_of_tree(node.right)
   return 1 + max(left_height, right_height)
```

```python
def remove_subtree(root, val):
    if not root:
        return None
    if root.val == val:
        return None
    root.left = remove_subtree(root.left, val)
    root.right = remove_subtree(root.right, val)
    return root
def height_after_removal(root: List[int], queries: List[int]) -> List[int]:
    root_node = build_tree(root)
    result = []
    for query in queries:
        new_root = remove_subtree(build_tree(root), query)
        result.append(height_of_tree(new_root))
    return result
root = [5,8,9,2,1,3,7,4,6]
queries = [3,2,4,8]
print(height_after_removal(root, queries))
```

```python
from typing import List
def min_operations_to_sort(nums: List[int]) -> int:
    n = len(nums)
    empty_pos = nums.index(0)
    target = list(range(n))
    operations = 0
    while nums != target:
        if empty_pos != 0:
            zero_to_index = nums.index(empty_pos)
            nums[empty_pos], nums[zero_to_index] = nums[zero_to_index], nums[empty_pos]
```

```
            empty_pos = zero_to_index
        else:
            for i in range(n):
                if nums[i] != target[i]:
                    nums[0], nums[i] = nums[i], nums[0]
                    empty_pos = i
                    break
        operations += 1
    return operations
nums = [4, 2, 0, 3, 1]
print(min_operations_to_sort(nums))
```

```
from typing import List
def apply_operations(nums: List[int]) -> List[int]:
    n = len(nums)
    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]
    result.extend([0] * (n - len(result)))
    return result
nums = [1, 2, 2, 1, 1, 0]
print(apply_operations(nums))
```

```
from typing import List
def apply_operations(nums: List[int]) -> List[int]:
    n = len(nums)
```

```
    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]
    result.extend([0] * (n - len(result)))
    return result
nums = [1, 2, 2, 1, 1, 0]
print(apply_operations(nums))



def total_cost_to_hire(costs: List[int], k: int, candidates: int) -> int:
    from heapq import heappush, heappop

    n = len(costs)
    left = costs[:candidates]
    right = costs[max(candidates, n - candidates):]

    heap = []
    for i in left:
        heappush(heap, (i, 'L'))
    for i in right:
        heappush(heap, (i, 'R'))

    total_cost = 0
    left_ptr = candidates
    right_ptr = max(candidates, n - candidates)

    for _ in range(k):
        cost, side = heappop(heap)
        total_cost += cost
```

```
        if side == 'L' and left_ptr < right_ptr:
            heappush(heap, (costs[left_ptr], 'L'))
            left_ptr += 1
        elif side == 'R' and right_ptr < n:
            heappush(heap, (costs[right_ptr], 'R'))
            right_ptr += 1

    return total_cost
costs = [17, 12, 10, 2, 7, 2, 11, 20, 8]
k = 3
candidates = 4
print(total_cost_to_hire(costs, k, candidates))
```

```
def minimum_total_distance_traveled(robot: List[int], factory: List[List[int]]) -> int:
    robot.sort()
    factory.sort()
    dp = [0] * (len(factory) + 1)

    for r in robot:
        new_dp = [float('inf')] * (len(factory) + 1)
        for j, (pos, limit) in enumerate(factory):
            if j > 0:
                new_dp[j] = min(new_dp[j], dp[j])
            if limit > 0:
                new_dp[j] = min(new_dp[j], dp[j] + abs(pos - r))
                if j + 1 < len(factory):
                    factory[j][1] -= 1
                    factory[j + 1][1] += 1
```

```
        dp = new_dp

    return min(dp)

robot = [0, 4, 6]
factory = [[2, 2], [6, 2]]
print(minimum_total_distance_traveled(robot, factory))
```

```
from math import gcd
from functools import lru_cache
def min_subarrays_in_valid_split(nums: List[int]) -> int:
    n = len(nums)
    @lru_cache(None)
    def valid(start, end):
        return gcd(nums[start], nums[end]) > 1
    dp = [float('inf')] * n
    dp[0] = 1
    for i in range(1, n):
        for j in range(i):
            if valid(j, i):
                dp[i] = min(dp[i], dp[j] + 1)
    return dp[-1] if dp[-1] != float('inf') else -1
nums = [2, 6, 3, 4, 3]
print(min_subarrays_in_valid_split(nums))
```

```
def number_of_distinct_averages(nums: List[int]) -> int:
    nums.sort()
    averages = set()
```

```
    while nums:
        min_num = nums.pop(0)
        max_num = nums.pop(-1)
        averages.add((min_num + max_num) / 2)
    return len(averages)
nums = [4, 1, 4, 0, 3, 5]
print(number_of_distinct_averages(nums))
```

```
from typing import List
def number_of_distinct_averages(nums: List[int]) -> int:
    nums.sort()
    averages = set()
    while nums:
        min_num = nums.pop(0)
        max_num = nums.pop(-1)
        averages.add((min_num + max_num) / 2)
    return len(averages)
nums = [4, 1, 4, 0, 3, 5]
print(number_of_distinct_averages(nums))
```

```
from collections import defaultdict, deque
from typing import List
def most_profitable_path(edges: List[List[int]], bob: int, amount: List[int]) -> int:
    n = len(amount)
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
```

```
    def bfs_bob(start: int):
        times = [-1] * n
        queue = deque([(start, 0)])
        times[start] = 0
        while queue:
            node, time = queue.popleft()
            for neighbor in graph[node]:
                if times[neighbor] == -1:
                    times[neighbor] = time + 1
                    queue.append((neighbor, time + 1))
        return times
    bob_times = bfs_bob(bob)
    def dfs_alice(node: int, parent: int, time: int) -> int:
        income = 0
        if time < bob_times[node]:
            income += amount[node]
        elif time == bob_times[node]:
            income += amount[node] // 2
        max_income = income
        for neighbor in graph[node]:
            if neighbor != parent:
                max_income = max(max_income, income + dfs_alice(neighbor, node, time + 1))
        return max_income
    return dfs_alice(0, -1, 0)
edges = [[0, 1], [1, 2], [1, 3], [3, 4]]
bob = 3
amount = [-2, 4, 2, -4, 6]
print(most_profitable_path(edges, bob, amount))
```