LI. Yuntong
20944800

1. (25 points) You are given the locations of n buildings $B_1, \ldots, B_n$ on the real line. Each location is a coordinate (i.e., a real number). Each building B_i has a WIFI signal receiver with a range of $r_i > 0$. That is, if we place a signal tower at distance r_i or less from B_i, then B_i gets WIFI. Note that the signal receivers of different buildings may have different ranges.

   Describe a greedy algorithm that places the smallest number of signal towers so that every building gets WIFI. Explain the correctness of your algorithm. Derive the running time of your algorithm.

```
PLACE-WIFI-TOWERS(buildings)
    creat a new empty array call intervals
    for each building B_i:
        append (x_i - r_i, x_i + r_i) in to the intervals
    sort intervals by the second element (right endpoint)

    current_tower_position ← -∞
    tower_count ← 0

    for each interval (left, right) in intervals:
        if left > current_tower_position then
            current_tower_position ← right
            tower_count ← tower_count + 1

    return tower_count
```

[Explain]: By placing the tower at the rightmost point of the first uncovered interval, we maximize the coverage to the right. Then, we only place a new tower when we encounter an interval that is not covered by the last placed tower, ensuring that we are using the minimum number of towers necessary.

[Running Time]: ① creating intervals: $O(n)$
② sorting intervals: $O(n \log n)$
③ placing Towers: $O(n)$
So overall, the running time is $O(n \log n)$

2. (25 pts) Given an array $A[1..n]$ of positive integers and an integer $m \leq n$, we want to split A into at most m non-empty contiguous subarrays so that the largest sum among these subarrays is minimized. Design an algorithm based on a greedy strategy to solve this problem. Analyze the running time of your algorithm. Explain the correctness of your algorithm.

   For example, if the array is $A = [7, 2, 5, 10, 8]$ and $m = 2$, there are several ways to split A: $([7, 2, 4, 10, 8], \emptyset)$, $([7], [2, 5, 10, 8])$, $([7, 2], [5, 10, 8])$, $([7, 2, 5], [10, 8])$, and $([7, 2, 5, 10], [8])$. The best way is $([7, 2, 5], [10, 8])$ because the maximum sum is $10 + 8 = 18$ which is the minimum among all possible ways of splitting.

   Hint: You need to invoke a greedy algorithm multiple times. How many times?

```
minimizeLargestesum(A, m):
    Left ← max element in A
    Right ← sum of A

    while Left < Right:
        mid ← (Left + Right)/2
        if canSplit(A, m, mid) = true then
            right ← mid
        else    left ← mid+1
    return left
```

[Running Time]: $O(n \log(\text{sum}(A)))$
① For the binary Search part $O(\log(\text{sum}(A)))$
② For the canSplit part: $O(n)$
Total running time: $O(n \log(\text{sum}(A)))$

[Explain]: ① Binary Search: Because we need searching for the smallest possible value of the largest sum among subarrays. The binary search ensures that we efficiently converge on this minimum value.

② Greedy : it ensures that we form the minimum number of subarrays needed for a given maximum sum

```
canSplit (A, m, maxSum)
    count ← 1
    currentSum ← 0
    for element in A
        if currentSum + element > maxSum then
            count ← count + 1
            currentSum ← element
            if count > m then
                return false
            else currentSum ← currentSum + element
    return true
```

3. (25 points) Consider a two-dimensional array $A[1..n, 1..n]$ of distinct integers. We want to find the *longest increasing path* in A. A sequence of entries $A[i_1, j_1], A[i_2, j_2], \ldots, A[i_k, j_k], A[i_{k+1}, j_{k+1}], \ldots$ is a path in A if and only if every two consecutive entries share a common index and the other indices differ by 1, that is, for all k,

   - either $i_k = i_{k+1}$ and $j_{k+1} \in \{j_k - 1, j_k + 1\}$, or
   - $j_k = j_{k+1}$ and $i_{k+1} \in \{i_k - 1, i_k + 1\}$.

   A path in A is increasing $A[i_1, j_1], A[i_2, j_2], \ldots, A[i_k, j_k], A[i_{k+1}, j_{k+1}], \ldots$ if and only if $A[i_k, j_k] < A[i_{i+1}, j_{k+1}]$. The length of a path is the number of entries in it.

   Design a dynamic programming algorithm to find the longest increasing path in A. Your algorithm needs to output the maximum length as well as the indices of the array entries in the path. Note that there is no restriction on where the longest increasing path may start or end. Define and explain your notations. Define and explain your recurrence and boundary conditions. Write your algorithm in pseudo-code. Derive the running time of your algorithm.

```
LongestIncreasingPath(A)
    n1 ← number of rows in A
    n2 ← number of columns in A
    dp ← array of size [n1][n2] initialized to -1
    previous ← array of size [n1][n2] initialized to None   // store the path
    maxLength ← 0
    startPosition ← None

    function dfs(i, j):
        if dp[i][j] ≠ -1 then return dp[i][j]   // Boundary condition : if dp[i][j] has already been computed , return it
        dp[i][j] ← 1
        directions ← [(0,1), (1,0), (0,-1), (-1,0)]
        for (dx, dy) in directions
            ni ← i + dx
            nj ← j + dy
            if 0 ≤ ni < n1 and 0 ≤ nj < n2 and A[ni][nj] > A[i][j] then   // only consider neighboring elements greater than A[i][j]
                length ← 1 + dfs(ni, nj)   // recursively calculate the longest path length from the neighboring position
```

```
            if length > dp[i][j] then
                previous[i][j] ← (ni, nj)
    return dp[i][j]

for i from 0 to n1-1
    for j from 0 to n2-1
        if dp[i][j] = -1 then dfs(i, j)
        if dp[i][j] > maxLength then
            maxLength ← dp[i][j]
            startPosition ← (i, j)

create an empty array called path
current ← startPosition
while current is not None
    append current into the end of path
    current ← previous[current[0]][current[1]]

return maxLength , path
```

[Running Time]: ① For dp[][]: O(m×n)

② For previous[][]: O(m×n)

③ For path: worst case O(m×n)

So overall: O(m×n)

4. (25 points) Let T be a rooted full binary tree of n nodes (not necessarily balanced). Each node v of T is given a positive weight $w(v)$. The depth $d(v)$ of v is the number of edges between v and the root of T. An *ancestor* of v is either v itself or a node u that can be reached from v by following parent pointers.

   Let $k \leq n$ be a given positive integer. You are asked to mark exactly k nodes of T as depots such that:

   - Every node v in T has a depot as its ancestor. Among the depots that are ancestors of v, the one with the largest depth is the *nearest depot* of v. We denote it by t_v. It is possible that $t_v = v$.
   - The sum $\sum_{v \in T} w(v) \cdot (d(v) - d(t_v))$ is minimized.

   Design a dynamic programming algorithm for this problem. You only need to output the minimized sum. You need to derive and explain your recurrence and boundary conditions. Analyze the running time of your algorithm.

   Hint: Depending on the tree height, k, and whether the subtree root is a depot, you may need many subproblems for a subtree.

Definition:
- T_v denote the subtree rooted at node v
- $w(v)$ be the weight of node v
- $d(v)$ be the depth of node v
- $f(v, m, c)$ be the minimum sum of weighted distances for the subtree T_v when we mark exactly m depots in T_v, and c means whether node v is marked as a depot (if yes. 1 if no. 0)

Recurrence Relation: if v is a leaf node, then: $f(v, m, 1) = 0$ (if $m = 1$)

$f(v, m, c)$

$f(v, m, 0) = \infty$ (we can't have 0 depots in a non-empty subtree)

For non-leaf node with left and right children u and w:

if v is marked as a depot ($c=1$): $f(v, m, 1) = \min_{m_1 + m_2 = m-1} (f(u, m_1, 0) + f(w, m_2, 0))$

if v is not marked as a depot ($c=0$): $f(v, m, 0) = \min_{m_1 + m_2 = m} \left(f(u, m_1, 1) + f(w, m_2, 1) + \sum_{x \in T_v} w(x) \cdot (d(x) - d(v)) \right)$

Boundary Condition: ① if $m=0$ and $c=0$, $f(v, m, c) = 0$. (no depots are needed)

② if $m > 0$ and the number of nodes in T_v is less than m, then $f(v, m, c) = \infty$ because it's impossible to mark more depots than there are nodes.

Running Time: $O(n \times k^2)$

for each node v and each possible number of depots m, compute the minimum over $O(k)$ combinations of m_1 and m_2.