National Institute of Technology
Karnataka, Surathkal

Assignment -3

CS700

Algorithms And Complexity

Submitted To:
Dr. Vani M
Associate Professor
Department Of Computer
Science And Engineering

Submitted By:
Dileep Reddy G
242CS017
S Vishnu Sai
242CS035

Problem 1

Problem Statement

Dance partners: You are pairing couples for a very conservative formal ball. There are n men and m women, and you know the height and gender of each person there. Each dancing couple must be a man and a woman, and the man must be at least as tall as, but no more than 3 inches taller than, his partner. You wish to maximize the number of dancing couples given this constraint.

Example:

- **Input:**
  - Number of males: 3
  - Number of females: 3
  - Heights of males: 12, 42, 23
  - Heights of females: 41, 19, 10
- **Output:** 2

Algorithm

The algorithm we implement consists of the following steps:

```
1. Read the number of males $n$ and the number of females $m$.
2. Read the heights of the males into an array men_heights of size $n$.
3. Read the heights of the females into an array women_heights of size $m$.
4. Sort both men_heights and women_heights in ascending order.
5. Initialize two indices $i \leftarrow 0$ and $j \leftarrow 0$, and set a counter matches $\leftarrow 0$.
6. Perform the following loop until $i \geq n$ or $j \geq m$:
   6.1. If men_heights[$i$] $<$ women_heights[$j$], then:
        6.1.1. Increment $i$ by 1 (move to the next male).
   6.2. Else if men_heights[$i$] $>$ women_heights[$j$] $+ 3$, then:
        6.2.1. Increment $j$ by 1 (move to the next female).
   6.3. Else (the male and female can form a valid pair):
        6.3.1. Increment matches by 1.
        6.3.2. Increment both $i$ and $j$ by 1 (move to the next male and female).
7. Return the value of matches as the maximum number of dancing couples.
```

C++ Implementation

```
#include <iostream>
#include <vector>
#include <algorithm>

// Function to calculate the maximum number of dancing couples
int calculate_max_pairs(std::vector<int>& men_heights, std::vector<
int>& women_heights) {
  // Sort the heights
  std::sort(men_heights.begin(), men_heights.end());
  std::sort(women_heights.begin(), women_heights.end());

  int i = 0; // Index for men
  int j = 0; // Index for women
  int matches = 0;
  int n = men_heights.size();
  int m = women_heights.size();

  // Match couples based on the height constraints
  while (i < n && j < m) {
    if (men_heights[i] < women_heights[j]) {
      // Man is shorter than the woman; move to the next man
      ++i;
    } else if (men_heights[i] > women_heights[j] + 3) {
      // Man is more than 3 inches taller; move to the next woman
      ++j;
    } else {
      // Valid couple found
      ++matches;
      ++i;
      ++j;
    }
  }

  return matches;
}

int main() {
  // Test case
  int n = 3; // Number of males
  int m = 3; // Number of females

  std::vector<int> men_heights = {12, 42, 23};
  std::vector<int> women_heights = {41, 4, 10};

  int max_pairs = calculate_max_pairs(men_heights, women_heights);

  std::cout << "Maximum number of dancing couples: " << max_pairs
<< std::endl;

  return 0;
}
```

Complexity Analysis

Assuming n is the number of males and m is the number of females, we analyze the time complexity of each step in the algorithm:

1. **Reading Input:** Reading the heights of males and females takes $O(n)$ and $O(m)$ time, respectively.

2. **Sorting Heights:** Sorting the arrays men_heights and women_heights requires $O(n \log n)$ and $O(m \log m)$ time, respectively.

3. **Matching Pairs:**

   - We use two indices i and j to traverse the sorted arrays.
   - In each iteration of the while loop, we increment either i or j, or both.
   - The maximum number of iterations of the loop is $n + m$, since each index increments up to n or m.
   - Therefore, the matching process takes $O(n + m)$ time.

4. **Total Time Complexity:**

   - The total time complexity is the sum of the time complexities of all steps:

     $$O(n) + O(m) + O(n \log n) + O(m \log m) + O(n + m) = O(n \log n + m \log m)$$

   - The $O(n \log n + m \log m)$ term dominates the linear terms, so we can simplify the total time complexity to:

     $$\boxed{O(n \log n + m \log m)}$$

Problem 2

Problem Statement

Homework Grade Maximization: In a class, there are n assignments. You have H hours to spend on all assignments, and you cannot divide an hour between assignments, but must spend each hour entirely on a single assignment. The i-th hour you spend on assignment j will improve your grade on assignment j by $B[i, j]$, where for each j, $B[1, j] \geq B[2, j] \geq \cdots \geq B[H, j] \geq 0$. In other words, if you spend h hours on assignment j, your grade will be $\sum_{i=1}^{h} B[i, j]$, and time spent on each project has diminishing returns, with the next hour being worth less than the previous one. You want to divide your H hours between the assignments to maximize your total grade on all the assignments. Give an efficient algorithm for this problem.

Example:

- **Input:**
  - Number of assignments, $n = 3$
  - Total hours available, $H = 5$
  - Grades for each assignment:
    * Assignment 1: $\{9, 6, 4, 2, 1\}$
    * Assignment 2: $\{7, 5, 3, 2, 1\}$
    * Assignment 3: $\{8, 6, 4, 2\}$
- **Output:**
  - Maximum total grade: 36
  - Selected grades: $\{9, 8, 7, 6, 6\}$

Algorithm

This algorithm uses a greedy approach with a priority queue to maximize the total grade and consists of the following steps:

1. Initialize a priority queue pq to store pairs (B_{j,h_j}, j), where B_{j,h_j} is the grade improvement for the h_j-th hour on assignment j, and j is the assignment index.
2. Initialize an array $hours_{[1...n]}$ with all elements set to 0, where $hours_j$ stores the number of hours spent on assignment j.
3. For each assignment j from 1 to n:

   3.1. If $B_{j,1} > 0$, push $(B_{j,1}, j)$ onto pq.
4. Initialize $total\_grade := 0$.
5. While $H > 0$ and pq is not empty:

   5.1. Pop the pair (a, j) from pq with the highest a.

   5.2. Add a to $total\_grade$.

```
    5.3. Decrease $H \leftarrow H - 1$.
    5.4. Increment $hours_j \leftarrow hours_j + 1$.
    5.5. If $hours_j + 1 \leq$ length of $B_j$ and $B_{j,hours_j+1} > 0$:
        5.5.1. Push $(B_{j,hours_j+1}, j)$ onto $pq$.
6. Return $total\_grade$.
```

C++ Implementation

```cpp
 #include <iostream>
 #include <vector>
 #include <queue>
 #include <algorithm>

 using namespace std;

 // Function to maximize the total grade
 // Parameters:
 // - n: number of assignments
 // - H: total hours available
 // - B: 2D vector where each subvector represents the grades for
each hour spent on an assignment
 int maximizeGrades(int n, int H, vector<vector<int>>& B) {
   // Priority queue to store grades with corresponding assignment
index
   // Higher grades have higher priority
   priority_queue<pair<int, int>, vector<pair<int, int>>> pq;

   // Initialize the priority queue with the first grade of each
assignment
   for(int i = 0; i < n; i++) {
     pq.push({B[i][0], i});
   }

   int ans = 0; // Variable to store the maximum total grade
   vector<int> v(n, 0); // Vector to track hours spent on each
assignment

   // Loop while there are hours available and grades in the
priority queue
   while(!pq.empty() && H) {
     auto [a, b] = pq.top(); // Get the highest grade available
     pq.pop(); // Remove it from the queue

     // Accumulate grade and reduce available hours
     ans += a;
     H--;

     v[b]++; // Increment the hours spent on the current assignment

     // If there are more grades available for the current
assignment, push the next grade
     if(v[b] < B[b].size()) {
       pq.push({B[b][v[b]], b});
     }
```

```
    }

    return ans; // Return the maximum total grade achievable
  }

  int main() {
    int n = 3; // Number of assignments
    int H = 5; // Total hours available for study

    // Grades available per hour of effort on each assignment (
decreasing order)
    vector<vector<int>> B = {
      {9, 6, 4, 2, 1},
      {7, 5, 3, 2, 1},
      {8, 6, 4, 2}
    };

    // Calculate and output the maximum total grade achievable
    int maxTotalGrade = maximizeGrades(n, H, B);
    cout << "Maximum total grade: " << maxTotalGrade << endl;

    return 0;
  }
```

Complexity Analysis

Let n be the number of assignments, H be the total number of hours available, and G be the total number of grades available, where $G = \sum_{j=1}^{n} h_j$, and h_j is the number of hours for which grades are available for assignment j.

We will analyze the time complexity of each step in the algorithm:

1. **Reading Input:**
   - Reading the grades for all assignments involves reading each element of the 2D array B.
   - This operation takes $O(G)$ time because there are G grades in total.
2. **Initializing the Priority Queue:**
   - For each assignment j from 1 to n, we push the first grade $B_{j,1}$ into the priority queue if $B_{j,1} > 0$.
   - Each insertion into the priority queue takes $O(\log n)$ time because the priority queue maintains a heap structure.
   - Since we perform up to n insertions, this step takes $O(n \log n)$ time.
3. **Processing the Assignments (While Loop):**
   - The while loop runs while there are hours remaining $(H > 0)$ and the priority queue is not empty.
   - Each iteration involves the following operations:

a) **Pop Operation:** Remove the highest grade from the priority queue, which takes $O(\log n)$ time.
b) **Grade Accumulation:** Add the grade to $total\_grade$, which is an $O(1)$ operation.
c) **Hour Decrement:** Decrease H by 1, which is an $O(1)$ operation.
d) **Update Hours Spent:** Increment $hours_j$ for assignment j, which is an $O(1)$ operation.
e) **Push Operation:** If more grades are available for assignment j, push the next grade $B_{j,hours_j+1}$ into the priority queue, which takes $O(\log n)$ time.

- Since each iteration involves up to two priority queue operations (pop and push), each taking $O(\log n)$ time, the total time per iteration is $O(\log n)$.
- The loop runs at most H times because we can spend at most H hours.
- Therefore, the total time for processing the assignments is $O(H \log n)$.

4. **Total Time Complexity:**

   - Summing the time complexities of all steps:

$$O(G) + O(n \log n) + O(H \log n) = O(G + (n + H) \log n)$$

   - Since $H \leq G$, the dominant terms are $O(G)$ and $O(H \log n)$.
   - Thus, the total time complexity of the algorithm is:

$$\boxed{O(G + H \log n)}$$

Space Complexity

- **Grades Storage:** Storing the 2D array B requires $O(G)$ space because there are G grades.
- **Priority Queue:** The priority queue stores at most n elements at any time (one for each assignment), requiring $O(n)$ space.
- **Auxiliary Arrays:** The array *hours* of size n tracks the hours spent on each assignment, consuming $O(n)$ space.
- **Total Space Complexity:** Combining the above, the total space complexity is:

$$\boxed{O(G + n)}$$

Problem 3

Problem Statement

An Activity-Selection Problem: Given a set $S = \{1, 2, \ldots, n\}$ of n proposed activities, with a start time s_i and a finish time f_i for each activity i, select a maximum-size set of mutually compatible activities.

Example:

- **Input:**

Activity i	**Start Time s_i**	**Finish Time f_i**
1	1	4
2	3	5
3	0	6
4	5	7
5	8	9
6	5	9

- **Output:** Activities 1, 4, 5
- **Visualization:**

```
Time --->

Activity 1: [1---4]
Activity 4:      [5---7]
Activity 5:           [8---9]
```

Algorithm

The algorithm follows a greedy approach to select the maximum number of mutually compatible activities by always choosing the next activity that finishes earliest.

The algorithm we implement consists of the following steps:

1. **Sort the activities based on their finish times.**
   1.1. Given a set of activities $S = \{1, 2, \ldots, n\}$, each with a start time s_i and finish time f_i.
   1.2. Sort the activities in non-decreasing order of their finish times f_i.
2. **Initialize the list of selected activities.**
   2.1. Select the first activity in the sorted list and add it to the list of selected activities.
   2.2. Let *last_selected* denote the index of the last selected activity.
3. **Iterate through the sorted list of activities and select compatible ones.**
   3.1. For each activity i from the second activity to the last:

3.1.1. If the start time s_i of the current activity is greater than or equal to the finish time $f_{last\_selected}$ of the last selected activity, then:
  3.1.1*. Select activity i and add it to the list of selected activities.
  3.1.1*. Update *last_selected* to i.
3.1.2. Otherwise, skip the current activity.

4. **Return the list of selected activities.**

  4.1. The list now contains the maximum number of mutually compatible activities.

C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

struct Activity {
  int id;
  int start;
  int finish;
};

// Comparator function to sort activities based on finish time
bool activityCompare(Activity a, Activity b) {
  return a.finish < b.finish;
}

int main() {
  // Example input: list of activities with start and finish times
  std::vector<Activity> activities = {
    {1, 1, 4},
    {2, 3, 5},
    {3, 0, 6},
    {4, 5, 7},
    {5, 8, 9},
    {6, 5, 9}
  };

  // Sort activities based on their finish times
  std::sort(activities.begin(), activities.end(), activityCompare);

  std::cout << "Selected Activities:\n";

  // The first activity always gets selected
  int n = activities.size();
  int lastSelectedIndex = 0;
  std::cout << "Activity " << activities[lastSelectedIndex].id
  << " (Time: " << activities[lastSelectedIndex].start
  << " to " << activities[lastSelectedIndex].finish << ")\n";

  // Iterate through the rest of the activities
  for (int i = 1; i < n; i++) {
    // If the start time is greater or equal to the finish time
    // of the last selected activity, select it
    if (activities[i].start >= activities[lastSelectedIndex].finish
) {
```

```
            std::cout << "Activity " << activities[i].id
            << " (Time: " << activities[i].start
            << " to " << activities[i].finish << ")\n";
            lastSelectedIndex = i;
        }
    }

    return 0;
}
```

Complexity Analysis

Time Complexity:

Assuming n is the number of proposed activities, the algorithm consists of the following steps:

1. **Sorting the activities based on their finish times:**
   1.1. The algorithm begins by sorting the n activities in non-decreasing order of their finish times f_i.
   1.2. Sorting can be performed using an efficient sorting algorithm like Merge Sort or Quick Sort, which has a time complexity of $O(n \log n)$.
2. **Iterating through the sorted list to select compatible activities:**
   2.1. After sorting, the algorithm iterates through the list of activities to select the maximum number of mutually compatible activities.
   2.2. This iteration involves a single pass through the sorted list, comparing the start time of the current activity with the finish time of the last selected activity.
   2.3. Since each activity is considered exactly once, this step has a time complexity of $O(n)$.

Combining both steps, the overall time complexity $T(n)$ of the algorithm can be expressed as:

$$T(n) = O(n \log n) + O(n) = \boxed{O(n \log n)}$$

Space Complexity:

- The primary space requirements are for storing the list of activities and any additional space used by the sorting algorithm.
- If Merge Sort is used for sorting, it requires $O(n)$ additional space.
- Therefore, the overall space complexity is $\boxed{O(n)}$.

Problem 4

Problem Statement

Alice wants to throw a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they **know** and five other people whom they **don't know**.
Goal: Give an efficient algorithm that takes as input the list of n people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of n.

Algorithm

Overview:
We need to find the largest possible subset of people satisfying the following conditions:

- Each person knows at least 5 other people in the subset.
- Each person does *not* know at least 5 other people in the subset.

Approach:
We can model this problem using graphs:

- Each person is represented as a node in a graph.
- An edge between two nodes indicates that the two people know each other.
- We are to find the largest induced subgraph where each node v satisfies:

$$\text{Degree}(v) \geq 5$$
$$\text{Non-degree}(v) \geq 5$$

- Since the number of people a person doesn't know in the subset is $k - 1 - \text{Degree}(v)$ (where k is the size of the subset), the second condition becomes:

$$k - 1 - \text{Degree}(v) \geq 5 \implies \text{Degree}(v) \leq k - 6$$

Algorithm Steps:
We use an iterative process to eliminate people who do not satisfy the constraints:

1. **Initialization:**
   - Let S be the set of all n people.
   - For each person v in S, compute:
     - Degree$[v]$: Number of people in S that v knows.
     - NonDegree$[v] = |S| - 1 - $ Degree$[v]$: Number of people in S that v doesn't know.

2. **Initialize** two queues:
   - Q_{low}: Contains people with Degree$[v] < 5$.
   - Q_{high}: Contains people with Degree$[v] > |S| - 6$.
3. **While** Q_{low} or Q_{high} is not empty:
   (a) **If** Q_{low} is not empty:
      i. Remove a person v from Q_{low}.
      ii. **If** v is in S:
         - Remove v from S.
         - Update $|S| = |S| - 1$.
         - **For** each neighbor u of v still in S:
           – Decrease Degree$[u]$ by 1.
           – **If** Degree$[u] < 5$ and u is not in Q_{low}, add u to Q_{low}.

   (b) **Else if** Q_{high} is not empty:
      i. Remove a person v from Q_{high}.
      ii. **If** v is in S:
         - Remove v from S.
         - Update $|S| = |S| - 1$.
         - **For** each neighbor u of v still in S:
           – Decrease Degree$[u]$ by 1.
           – **If** Degree$[u] < 5$ and u is not in Q_{low}, add u to Q_{low}.
           – **If** Degree$[u] > |S| - 6$ and u is not in Q_{high}, add u to Q_{high}.

   (c) **After each removal**, recompute NonDegree$[u] = |S| - 1 -$ Degree$[u]$ for all u in S.

   (d) **Update** Q_{high} by adding any u where Degree$[u] > |S| - 6$.
4. **Return** the set S as the best choice of party invitees.

Explanation

- **Removing People Violating Constraints:**
  - If a person v knows fewer than 5 people in S, they cannot satisfy the first constraint, so we remove them.
  - If a person v knows more than $|S| - 6$ people in S, then they do not have at least 5 people they don't know in S, violating the second constraint.
- **Updating Degrees:**
  - When we remove a person v, we need to update the degrees of their neighbors u in S.
  - We decrease Degree$[u]$ by 1 since v is no longer in S.
  - The NonDegree$[u]$ is updated as $|S| - 1 -$ Degree$[u]$.

- **Queues Management:**
  - We maintain Q_{low} and Q_{high} to efficiently find and process people violating the constraints.
  - We add people to the queues whenever their degrees fall below 5 or exceed $|S| - 6$.
- **Termination:**
  - The algorithm terminates when there are no more people violating the constraints, i.e., both queues are empty.

C++ Implementation

```
 #include <iostream>
 #include <vector>
 #include <queue>
 #include <unordered_set>
 #include <unordered_map>
 using namespace std;

 // Function to find the best choice of party invitees
 vector<int> findPartyInvitees(int n, const vector<pair<int, int>>&
acquaintanceList) {
    vector<unordered_set<int>> adjList(n); // Adjacency list
    vector<int> degree(n, 0); // Degree of each person
    unordered_set<int> S; // Set of current invitees

    // Build the adjacency list and initialize degrees
    for (const auto& pair : acquaintanceList) {
       int u = pair.first;
       int v = pair.second;
       adjList[u].insert(v);
       adjList[v].insert(u);
    }

    for (int i = 0; i < n; ++i) {
       degree[i] = adjList[i].size();
       S.insert(i);
    }

    queue<int> Q_low, Q_high;

    // Initialize Q_low and Q_high
    for (int i = 0; i < n; ++i) {
       if (S.count(i)) {
          if (degree[i] < 5) {
             Q_low.push(i);
          }
          if (degree[i] > (int)S.size() - 6) {
             Q_high.push(i);
          }
       }
    }

    while (!Q_low.empty() || !Q_high.empty()) {
        while (!Q_low.empty()) {
          int v = Q_low.front();
          Q_low.pop();
          if (!S.count(v)) continue;
          S.erase(v);
          for (int u : adjList[v]) {
            if (S.count(u)) {
              degree[u]--;
              if (degree[u] < 5) {
                Q_low.push(u);
              }
            }
          }
        }

        // Update degrees and recompute Q_high
        int S_size = S.size();
        queue<int> new_Q_high;
        for (int v : S) {
          if (degree[v] > S_size - 6) {
            new_Q_high.push(v);
          }
        }
        Q_high = new_Q_high;

        while (!Q_high.empty()) {
          int v = Q_high.front();
          Q_high.pop();
          if (!S.count(v)) continue;
          S.erase(v);
          for (int u : adjList[v]) {
            if (S.count(u)) {
              degree[u]--;
              if (degree[u] < 5) {
                Q_low.push(u);
              }
            }
          }
        }
      }

      // Return the remaining people in S
      vector<int> invitees(S.begin(), S.end());
      return invitees;
    }

    int main() {
      int n = 15; // Number of people
      vector<pair<int, int>> acquaintanceList = {
        {0, 1}, {0, 2}, {0, 3}, {0, 4}, {0, 5},
        {1, 2}, {1, 3}, {1, 4}, {1, 6}, {1, 7},
        {2, 3}, {2, 5}, {2, 8}, {2, 9},
        {3, 4}, {3, 10}, {3, 11},
        {4, 5}, {4, 12}, {4, 13},
        {5, 14}
      };

      vector<int> invitees = findPartyInvitees(n, acquaintanceList);
      cout << "Best choice of party invitees (total " << invitees.size
   () << " people):" << endl;
      for (int v : invitees) {
        cout << "Person " << v << endl;
      }

      return 0;
    }
```

Explanation of the Code

- **Adjacency List:** We use an adjacency list to represent the relationships between people.
- **Degree Array:** An array degree[v] keeps track of the number of acquaintances for each person v within the current set S.
- **Sets and Queues:**
  - S: Represents the current set of potential invitees.
  - Q_{low} and Q_{high}: Queues to process people violating the constraints.
- **Main Loop:** The main loop continues until no more people violate the constraints, updating degrees and the set S accordingly.
- **Output:** The remaining people in S are the best choice of party invitees.

Complexity Analysis

- **Time Complexity:** $O(n^2)$
  - In the worst case, we may need to process each person and their acquaintances multiple times.
  - Each edge (acquaintance relationship) is processed at most twice (once when each person is removed).
  - The total time complexity is $O(n+m)$, where m is the number of acquaintance pairs.
  - Since m can be up to $O(n^2)$ in the worst case (dense graph), the overall time complexity is $O(n^2)$.
- **Space Complexity:** $O(n^2)$
  - The adjacency list may contain up to $O(n^2)$ edges.
  - Additional space is used for the degree array, sets, and queues, all of which are $O(n)$.

Conclusion

The algorithm efficiently finds the largest possible subset of people satisfying the given constraints by iteratively removing people who violate the conditions and updating the degrees of their acquaintances. The use of queues helps to focus only on those who might affect the constraints, leading to an overall time complexity of $O(n^2)$.

Problem 5

Problem Statement

A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is $5, 15, -30, 10, -5, 40, 10$, then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:
Input: A list of numbers, $a_1, a_2, \ldots, a_n$.
Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55.

Algorithm

Kadane's Algorithm:

We can solve this problem in linear time using Kadane's algorithm. The idea is to iterate through the array, keeping track of the maximum sum ending at each position, and updating the overall maximum sum found so far.

1. **Initialize:**
   - Set max_so_far = 0 (maximum sum found so far).
   - Set max_ending_here = 0 (maximum sum ending at current position).
   - Set start = 0 (start index of the maximum sum subsequence).
   - Set end = -1 (end index of the maximum sum subsequence).
   - Set s = 0 (potential start index of a new subsequence).
2. **For** $i = 0$ to $n - 1$:
   (a) max_ending_here = max_ending_here + a_i
   (b) **If** max_ending_here < 0:
      - max_ending_here = 0
      - s = $i + 1$
   (c) **Else if** max_so_far < max_ending_here:
      - max_so_far = max_ending_here
      - start = s
      - end = i
3. **Return** the subsequence from start to end and max_so_far.

If end = -1, then all numbers are negative, and the maximum sum is 0 (empty subsequence).

C++ Implementation

```cpp
#include <iostream>
#include <vector>

using namespace std;

// Structure to hold the result
struct Subsequence {
  int start;
  int end;
  int sum;
};

// Function to find the contiguous subsequence with maximum sum
Subsequence maxSubsequence(const vector<int>& v) {
  Subsequence result;
  int max_so_far = 0;
  int max_ending_here = 0;
  int s = 0;
  result.start = 0;
  result.end = -1; // Indicates empty subsequence
  result.sum = 0;

  for(int i = 0; i < v.size(); i++) {
    max_ending_here += v[i];

    if(max_ending_here < 0) {
      max_ending_here = 0;
      s = i + 1;
    } else if(max_so_far < max_ending_here) {
      max_so_far = max_ending_here;
      result.start = s;
      result.end = i;
      result.sum = max_so_far;
    }
  }

  return result;
}

int main() {
  // Sample data
  vector<int> S = {5, 15, -30, 10, -5, 40, 10};

  // Find the maximum subsequence
  Subsequence max_seq = maxSubsequence(S);

  // Output the result
  cout << "Input Sequence: ";
  for (int num : S) {
    cout << num << " ";
  }
  cout << endl;

  if (max_seq.end == -1) {
    cout << "The maximum sum is 0 (empty subsequence)." << endl;
  } else {
    cout << "Maximum Sum: " << max_seq.sum << endl;
    cout << "Subsequence: ";
    for (int i = max_seq.start; i <= max_seq.end; ++i) {
      cout << S[i] << " ";
    }
    cout << endl;
  }

  return 0;
}
```

Explanation

In this implementation:

- We maintain two variables:
  - max_ending_here: the maximum sum of a subsequence ending at the current position.
  - max_so_far: the maximum sum found so far.
- We also keep track of the potential start index s for the new subsequence when max_ending_here resets to zero.
- If max_ending_here becomes negative, we reset it to zero and move s to the next index.
- Whenever max_ending_here exceeds max_so_far, we update max_so_far and record the start and end indices.

Complexity Analysis

Time Complexity: $O(n)$, where n is the number of elements in the list. We traverse the list exactly once.

Space Complexity: $O(1)$. We use a fixed amount of additional space regardless of the input size.

Problem 6

Problem Statement

You are going on a long trip. You start on the road at mile post 0. Along the way, there are n hotels at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties.
Goal: Give an efficient algorithm that determines the optimal sequence of hotels at which to stop to minimize the total penalty.

Algorithm

Dynamic Programming Approach:

We can model this problem using dynamic programming. The idea is to compute the minimum total penalty to reach each hotel i, using the penalties from reaching previous hotels.

1. **Initialize:**
   - Let n be the number of hotels.
   - Let $a_0 = 0$ (starting point at mile post 0).
   - Let $\text{dp}[i]$ represent the minimum total penalty to reach hotel i.
   - Initialize $\text{dp}[0] = 0$ (no penalty at the starting point).
2. **Recurrence Relation:**
   - For each hotel i from 1 to n:
     - Set $\text{dp}[i] = \infty$.
     - For each hotel j from 0 to $i - 1$:
       * Calculate the distance traveled from hotel j to hotel i: $x = a_i - a_j$.
       * Calculate the penalty for this leg: $\text{penalty} = (200 - x)^2$.
       * Update $\text{dp}[i] = \min(\text{dp}[i], \text{dp}[j] + \text{penalty})$.
3. **Reconstruct the Path (Optional):**
   - To find the actual sequence of hotels, maintain a parent array $\text{parent}[i]$ that stores the previous hotel index leading to the minimum penalty for hotel i.
   - After filling the dp array, backtrack from $\text{parent}[n]$ to reconstruct the optimal path.

C++ Implementation

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve(int n, vector<int>& v) {
  // Include the starting point at mile post 0
  v.insert(v.begin(), 0);
  n = v.size();

  vector<int> dp(n, INT_MAX);
  vector<int> parent(n, -1); // To reconstruct the path
  dp[0] = 0; // No penalty at the starting point

  for(int i = 1; i < n; i++) {
    for(int j = 0; j < i; j++) {
      int distance = v[i] - v[j];
      int penalty = (200 - distance) * (200 - distance);
      int totalPenalty = dp[j] + penalty;
      if(totalPenalty < dp[i]) {
        dp[i] = totalPenalty;
        parent[i] = j; // Store the previous hotel index
      }
    }
  }

  // Output the minimum total penalty
  cout << "Minimum Total Penalty: " << dp[n - 1] << endl;

  // Reconstruct the optimal sequence of hotels
  vector<int> path;
  int idx = n - 1;
  while(idx != -1) {
    path.push_back(idx);
    idx = parent[idx];
  }
  reverse(path.begin(), path.end());

  // Output the optimal sequence of hotels
  cout << "Optimal Sequence of Hotels (mile posts):" << endl;
  for(int i = 1; i < path.size(); i++) { // Skip the starting point
at index 0
    cout << v[path[i]] << " ";
  }
  cout << endl;
}

int main() {
  int n = 5;
  vector<int> v = {120, 290, 480, 650, 900}; // Hotel mile posts

  solve(n, v);

  return 0;
}
```

Explanation

In this implementation:

- We include the starting point 0 as the first element in the mile posts vector v to simplify calculations.
- We initialize a dynamic programming array dp of size n, where dp[i] represents the minimum total penalty to reach hotel i.
- We also maintain a parent array to reconstruct the optimal sequence of hotels.
- For each hotel i, we consider all previous hotels j and calculate the penalty to travel from j to i. We update dp[i] if we find a lower total penalty.
- After filling the dp array, we backtrack using the parent array to find the optimal path.

Sample Output

Given the sample input:

- Number of hotels: $n = 5$
- Hotel mile posts: $120, 290, 480, 650, 900$

The program outputs:

```
Minimum Total Penalty: 30000
Optimal Sequence of Hotels (mile posts):
290 480 650 900
```

Complexity Analysis

Time Complexity: $O(n^2)$

- The outer loop runs n times.
- The inner loop runs up to n times.
- Therefore, the total time complexity is $O(n^2)$.

Space Complexity: $O(n)$

- We use additional space for the dp array and the parent array, each of size n.

Optimization

If we need to optimize the time complexity, we can consider using memoization or other advanced techniques. However, since the penalty function depends on the square of the distance and there is no specific pattern to exploit, the $O(n^2)$ time complexity is acceptable for moderate values of n.

Problem 7

Problem Statement

A subsequence is palindromic if it is the same whether read left to right or right to left. For instance, the sequence $A, C, G, T, G, T, C, A, A, A, A, T, C, G$ has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is not palindromic). Devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

Algorithm

Dynamic Programming Approach using Longest Common Subsequence (LCS):
The longest palindromic subsequence problem can be transformed into finding the longest common subsequence between the original sequence and its reverse. This works because a palindromic subsequence reads the same forwards and backwards.

1. **Let** X be the original sequence and Y be the reverse of X.
2. **Compute** the length of the longest common subsequence between X and Y using dynamic programming.
3. **Return** the length obtained from the LCS computation as the length of the longest palindromic subsequence.

Dynamic Programming Table:
Let $\text{dp}[i][j]$ represent the length of the longest common subsequence between the first i characters of X and the first j characters of Y.

- **Initialization:**

$$\begin{aligned} \text{dp}[0][j] &= 0 \quad \text{for all } j \\ \text{dp}[i][0] &= 0 \quad \text{for all } i \end{aligned}$$

- **Recurrence Relation:**

$$\text{dp}[i][j] = \begin{cases} \text{dp}[i-1][j-1] + 1, & \text{if } X[i] = Y[j] \\ \max(\text{dp}[i-1][j], \text{dp}[i][j-1]), & \text{if } X[i] \neq Y[j] \end{cases}$$

C++ Implementation

```cpp
#include <bits/stdc++.h>

using namespace std;

// Function to compute the Longest Common Subsequence (LCS)
int lcs(const vector<char>& v1, const vector<char>& v2) {
  int n = v1.size();
  vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
```

```
    // Build the dp table
    for(int i = 1; i <= n; i++) {
      for(int j = 1; j <= n; j++) {
        if(v1[i - 1] == v2[j - 1]) {
          dp[i][j] = dp[i - 1][j - 1] + 1;
        } else {
          dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
      }
    }
    // Return the length of the LCS
    return dp[n][n];
  }

  // Function to find the length of the longest palindromic
 subsequence
  int longestPalindromicSubsequence(const vector<char>& v) {
    vector<char> reversed_v = v;
    reverse(reversed_v.begin(), reversed_v.end());
    return lcs(v, reversed_v);
  }

  int main() {
    vector<vector<char>> test_cases = {
      {'A', 'G', 'T', 'A', 'T', 'G', 'D'},    // Expected LPS length:
  5
      {'A', 'B', 'C', 'B', 'A'},              // Expected LPS length:
  5
      {'A', 'A', 'A', 'A'},                   // Expected LPS length:
  4
      {'B', 'A', 'B', 'B', 'A'},              // Expected LPS length:
  5
      {'C', 'A', 'C', 'A', 'C'},              // Expected LPS length:
  5
      {'D', 'E', 'F', 'E', 'D'},              // Expected LPS length:
  5
      {'M', 'A', 'D', 'A', 'M'},              // Expected LPS length:
  5
      {'R', 'A', 'C', 'E', 'C', 'A', 'R'},    // Expected LPS length:
  7
      {'N', 'O', 'O', 'N'},                   // Expected LPS length:
  4
      {'S', 'E', 'R', 'I', 'A', 'L'},         // Expected LPS length:
  1
    };

    for (size_t i = 0; i < test_cases.size(); ++i) {
      cout << "Test Case " << i + 1 << ": ";
      for (char c : test_cases[i]) {
        cout << c << ' ';
      }
      cout << "\nLongest Palindromic Subsequence Length: ";
      int length = longestPalindromicSubsequence(test_cases[i]);
      cout << length << endl;
      cout << "-----------------------------\n";
    }
    return 0;
  }
```

Explanation

In this implementation:

- The function `lcs` computes the length of the longest common subsequence between the original sequence and its reverse.
- The function `longestPalindromicSubsequence` prepares the reversed sequence and calls `lcs`.
- In the `main` function, several test cases are provided to demonstrate the algorithm.

Why Does This Work?
A palindrome reads the same forwards and backwards. By reversing the original sequence and finding the longest common subsequence between the two, we effectively find the longest sequence that appears in both the original and reversed sequences, which corresponds to the longest palindromic subsequence.

Complexity Analysis

- **Time Complexity:** $O(n^2)$
  - The dynamic programming table has $n \times n$ elements.
  - Filling each cell takes constant time.
- **Space Complexity:** $O(n^2)$
  - We use a two-dimensional array dp of size $(n + 1) \times (n + 1)$.

Sample Output

```
Test Case 1: A G T A T G D
Longest Palindromic Subsequence Length: 5
-----------------------------
Test Case 2: A B C B A
Longest Palindromic Subsequence Length: 5
-----------------------------
Test Case 3: A A A A
Longest Palindromic Subsequence Length: 4
-----------------------------
Test Case 4: B A B B A
Longest Palindromic Subsequence Length: 4
-----------------------------
Test Case 5: C A C A C
Longest Palindromic Subsequence Length: 5
-----------------------------
Test Case 6: D E F E D
Longest Palindromic Subsequence Length: 5
```

```
-----------------------------
Test Case 7: M A D A M
Longest Palindromic Subsequence Length: 5
-----------------------------
Test Case 8: R A C E C A R
Longest Palindromic Subsequence Length: 7
-----------------------------
Test Case 9: N O O N
Longest Palindromic Subsequence Length: 4
-----------------------------
Test Case 10: S E R I A L
Longest Palindromic Subsequence Length: 1
-----------------------------
```

Additional Notes

- The algorithm can be modified to reconstruct the actual longest palindromic subsequence by keeping track of the choices made during the dynamic programming table construction.
- For space optimization, the two-dimensional array can be reduced to two one-dimensional arrays since only the previous row is needed at each step.

Problem 8

Problem Statement

Given two strings $x = x_1x_2 \dots x_n$ and $y = y_1y_2 \dots y_m$, we wish to find the length of their longest common substring, that is, the largest k for which there are indices i and j with $x_ix_{i+1} \dots x_{i+k-1} = y_jy_{j+1} \dots y_{j+k-1}$. Show how to do this in time $O(mn)$.

Algorithm

Dynamic Programming Approach:

We can solve this problem using dynamic programming in $O(mn)$ time. The idea is to build a two-dimensional array dp where $\text{dp}[i][j]$ represents the length of the longest common substring ending at positions x_i and y_j.

1. **Initialize** a 2D array dp of size $(n + 1) \times (m + 1)$ with all elements initialized to zero.
2. **Initialize** a variable $\text{max\_length} = 0$ to keep track of the maximum length found so far.
3. **For** $i = 1$ to n:
    - **For** $j = 1$ to m:
        - **If** $x_i = y_j$:
            * $\text{dp}[i][j] = \text{dp}[i-1][j-1] + 1$
            * Update $\text{max\_length} = \max(\text{max\_length}, \text{dp}[i][j])$
        - **Else**:
            * $\text{dp}[i][j] = 0$
4. **Return** max_length as the length of the longest common substring.

Explanation

In this algorithm:

- The $\text{dp}[i][j]$ entry represents the length of the longest common substring ending at x_i and y_j.
- If $x_i = y_j$, we extend the previous common substring ending at x_{i-1} and y_{j-1} by one character.
- If $x_i \neq y_j$, there is no common substring ending at these positions, so $\text{dp}[i][j] = 0$.
- We keep track of the maximum value of $\text{dp}[i][j]$ throughout the computation.

C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// Function to find the length of the longest common substring
int longestCommonSubstring(const string& X, const string& Y) {
  int n = X.size();
  int m = Y.size();
  int max_length = 0;

  // Create a 2D vector initialized with zeros
  vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

  // Build the dp table
  for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= m; j++) {
      if(X[i - 1] == Y[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1] + 1;
        max_length = max(max_length, dp[i][j]);
      } else {
        dp[i][j] = 0;
      }
    }
  }

  return max_length;
}

int main() {
  vector<pair<string, string>> test_cases = {
    {"ABAB", "BABA"},              // Expected: 3 ("ABA" or "BAB")
    {"abcdxyz", "xyzabcd"},        // Expected: 4 ("abcd")
    {"zxabcdezy", "yzabcdezx"},    // Expected: 6 ("abcdez")
    {"abc", "def"},                // Expected: 0 (No common
substring)
    {"abcdef", "abfdef"},          // Expected: 3 ("def")
    {"", "abcdef"},                // Expected: 0 (Empty string)
    {"abcdef", ""},                // Expected: 0 (Empty string)
    {"aaa", "aa"},                 // Expected: 2 ("aa")
    {"The quick brown fox", "quick brown"}, // Expected: 11 ("quick
 brown")
  };

  for(size_t i = 0; i < test_cases.size(); ++i) {
    const string& X = test_cases[i].first;
    const string& Y = test_cases[i].second;
    int length = longestCommonSubstring(X, Y);
    cout << "Test Case " << i + 1 << ": " << endl;
    cout << "String X: \"" << X << "\"" << endl;
    cout << "String Y: \"" << Y << "\"" << endl;
    cout << "Length of Longest Common Substring: " << length <<
endl;
    cout << "-----------------------------\n";
    }

    return 0;
  }
```

Complexity Analysis

- **Time Complexity:** $O(nm)$
  - We fill a table of size $(n+1) \times (m+1)$.
  - Each cell takes constant time to compute.
- **Space Complexity:** $O(nm)$
  - We use a two-dimensional array dp of size $(n+1) \times (m+1)$.

Sample Output

```
Test Case 1:
String X: "ABAB"
String Y: "BABA"
Length of Longest Common Substring: 3
------------------------------
Test Case 2:
String X: "HelloThere"
String Y: "A lot"
Length of Longest Common Substring: 2
------------------------------
Test Case 3:
String X: "abcdxyz"
String Y: "xyzabcd"
Length of Longest Common Substring: 4
------------------------------
Test Case 4:
String X: "zxabcdezy"
String Y: "yzabcdezx"
Length of Longest Common Substring: 6
------------------------------
Test Case 5:
String X: "abc"
String Y: "def"
Length of Longest Common Substring: 0
------------------------------
Test Case 6:
String X: "abcdef"
String Y: "abfdef"
Length of Longest Common Substring: 3
------------------------------
```

```
Test Case 7:
String X: ""
String Y: "abcdef"
Length of Longest Common Substring: 0
-----------------------------
Test Case 8:
String X: "abcdef"
String Y: ""
Length of Longest Common Substring: 0
-----------------------------
Test Case 9:
String X: "aaa"
String Y: "aa"
Length of Longest Common Substring: 2
-----------------------------
Test Case 10:
String X: "The quick brown fox"
String Y: "quick brown"
Length of Longest Common Substring: 11
-----------------------------
```

Additional Notes

- **Reconstructing the Longest Common Substring:** If we need to reconstruct the actual substring, we can keep track of the position where the maximum length was updated and then backtrack.
- **Space Optimization:** Since we only need the previous row to compute the current row, we can optimize space to $O(\min(n, m))$ by using two one-dimensional arrays.
- **Alternative Algorithms:** For larger alphabets or when more efficient algorithms are required, advanced techniques like suffix trees or suffix arrays can be used to achieve $O(n + m)$ time complexity, but they are more complex and may not be practical for all cases.

Problem 9

Problem Statement

Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic programming algorithm for the following problem.
Input: $x_1, x_2, \ldots, x_n$; v.
Question: Is it possible to make change for v using coins of denominations $x_1, x_2, \ldots, x_n$?

Algorithm

Dynamic Programming Approach:
We can solve this problem using a one-dimensional dynamic programming array. The idea is to determine for each amount i from 0 to v whether it is possible to make change for amount i using the given coin denominations.

1. **Initialize** a boolean array dp of size $v+1$ where dp$[i]$ indicates whether it is possible to make change for amount i.
2. **Set** dp$[0] =$ true since we can make change for 0 using no coins.
3. **For** each coin denomination x_j in $x_1, x_2, \ldots, x_n$:
   - **For** $i = x_j$ to v:
     - **If** dp$[i - x_j]$ is true, then set dp$[i] =$ true.
4. **After filling the dp array**, check dp$[v]$:
   - **If** dp$[v]$ is true, then it is possible to make change for v.
   - **Else**, it is not possible.

Explanation:

- The dp array represents whether each amount i can be formed using the given coin denominations.
- We start from amount 0 and build up to amount v.
- For each coin denomination, we update the dp array by marking dp$[i]$ as true if dp$[i - x_j]$ is true.

C++ Implementation

```
#include <iostream>
#include <vector>

using namespace std;
```

```
 // Function to determine if it is possible to make change for
amount v
 bool canMakeChange(const vector<int>& coins, int v) {
   vector<bool> dp(v + 1, false);
   dp[0] = true; // Base case: we can make change for 0

   for(int coin : coins) {
     for(int i = coin; i <= v; i++) {
       if(dp[i - coin]) {
         dp[i] = true;
       }
     }
   }

   return dp[v];
 }

 int main() {
   // Sample test cases
   vector<pair<vector<int>, int>> test_cases = {
     {{5, 10}, 15},    // Expected: Possible
     {{5, 10}, 12},    // Expected: Not possible
     {{1, 2, 5}, 11}, // Expected: Possible
     {{2}, 3},         // Expected: Not possible
     {{2, 3, 7}, 14}, // Expected: Possible
     {{3, 6, 9}, 13}, // Expected: Not possible
     {{1, 5, 10, 25}, 30}, // Expected: Possible
     {{4, 5}, 7},      // Expected: Not possible
     {{1}, 0},         // Expected: Possible (amount 0)
     {{}, 5},          // Expected: Not possible (no coins)
   };

   for(size_t i = 0; i < test_cases.size(); ++i) {
     const vector<int>& coins = test_cases[i].first;
     int v = test_cases[i].second;
     bool result = canMakeChange(coins, v);

     cout << "Test Case " << i + 1 << ": " << endl;
     cout << "Coins: ";
     for(int coin : coins) {
       cout << coin << " ";
     }
     cout << "\nAmount v: " << v << endl;
     cout << "Can make change? " << (result ? "Yes" : "No") << endl;
     cout << "-----------------------------\n";
   }

   return 0;
 }
```

Explanation

In this implementation:

- We define a function `canMakeChange` that takes the list of coin denominations and the target amount v.

- We initialize a boolean array dp of size $v + 1$ with all values set to `false`, except for dp[0] which is set to `true`.
- We iterate over each coin denomination and update the dp array accordingly.
- After filling the dp array, we check dp[v] to determine if it is possible to make change for amount v.
- The `main` function includes several test cases to demonstrate the algorithm.

Complexity Analysis

- **Time Complexity:** $O(nv)$
  - The outer loop iterates over each coin denomination (n coins).
  - The inner loop iterates over the amounts from the coin value to v (up to v iterations).
  - Therefore, the total time complexity is $O(nv)$.
- **Space Complexity:** $O(v)$
  - We use a one-dimensional array dp of size $v + 1$.

Sample Output

```
Test Case 1:
Coins: 5 10
Amount v: 15
Can make change? Yes
-----------------------------
Test Case 2:
Coins: 5 10
Amount v: 12
Can make change? No
-----------------------------
Test Case 3:
Coins: 1 2 5
Amount v: 11
Can make change? Yes
-----------------------------
Test Case 4:
Coins: 2
Amount v: 3
Can make change? No
-----------------------------
Test Case 5:
Coins: 2 3 7
Amount v: 14
Can make change? Yes
-----------------------------
```

```
Test Case 6:
Coins: 3 6 9
Amount v: 13
Can make change? No
-----------------------------
Test Case 7:
Coins: 1 5 10 25
Amount v: 30
Can make change? Yes
-----------------------------
Test Case 8:
Coins: 4 5
Amount v: 7
Can make change? No
-----------------------------
Test Case 9:
Coins: 1
Amount v: 0
Can make change? Yes
-----------------------------
Test Case 10:
Coins:
Amount v: 5
Can make change? No
-----------------------------
```

Additional Notes

- **Edge Cases:**
  - If the amount v is 0, it is always possible to make change (with no coins).
  - If the list of coin denominations is empty, it is not possible to make change for any positive amount.
- **Reconstructing the Coin Combination (Optional):**
  - If we need to find the actual combination of coins used to make change for v, we can maintain an additional array to store the last coin used for each amount.
  - After filling the dp array, we can backtrack from v to 0 to retrieve the coins used.
- **Comparison with Coin Change Minimum Coins Problem:**
  - This problem asks whether it is possible to make change for v, not the minimum number of coins required.
  - The minimum coins problem also uses dynamic programming but requires keeping track of the minimum number of coins for each amount.

Problem 10

Problem Statement

Consider the following variation on the change-making problem: you are given denominations $x_1, x_2, \ldots, x_n$, and you want to make change for a value v, but you are allowed to use each denomination at most once. For instance, if the denominations are $1, 5, 10, 20$, then you can make change for $16 = 1 + 15$ and for $31 = 1 + 10 + 20$ but not for 40 (because you can't use 20 twice).
Input: Positive integers $x_1, x_2, \ldots, x_n$; another integer v.
Output: Can you make change for v, using each denomination x_i at most once?
Goal: Show how to solve this problem in time $O(nv)$.

Algorithm

Dynamic Programming Approach (Subset Sum Problem):
This problem is a variation of the classic subset sum problem. We need to determine if there exists a subset of the coin denominations whose sum equals v.
We can use dynamic programming to solve this problem in $O(nv)$ time.

1. **Initialize** a two-dimensional boolean array $\mathrm{dp}[i][j]$ of size $(n+1) \times (v+1)$, where:
   - $\mathrm{dp}[i][j] = \text{true}$ if it is possible to achieve sum j using the first i coins.
   - $\mathrm{dp}[0][0] = \text{true}$, since a sum of 0 can be achieved with zero coins.
2. **Fill** the dp table using the following recurrence:

$$\mathrm{dp}[i][j] = \begin{cases} \mathrm{dp}[i-1][j] & \text{(exclude the } i\text{-th coin)} \\ \mathrm{dp}[i-1][j] \lor \mathrm{dp}[i-1][j-x_i] & \text{if } j \geq x_i \text{ (include the } i\text{-th coin)} \end{cases}$$

3. **Final Answer:** After filling the table, $\mathrm{dp}[n][v]$ will be true if it is possible to make change for v using each coin at most once.

Explanation

- **dp Table:** The table $\mathrm{dp}[i][j]$ represents whether it is possible to achieve a sum of j using the first i coins.
- **Include or Exclude:** For each coin, we have two choices:
  - **Exclude** the coin: We carry over the value from $\mathrm{dp}[i-1][j]$.
  - **Include** the coin (if $j \geq x_i$): We check if we can achieve the remaining sum $j - x_i$ using the first $i - 1$ coins.
- **Initialization:**
  - $\mathrm{dp}[0][0] = \text{true}$: Zero sum is possible with zero coins.
  - $\mathrm{dp}[0][j] = \text{false}$ for $j > 0$: Non-zero sums are not possible with zero coins.

C++ Implementation

```
 #include <iostream>
 #include <vector>

 using namespace std;

 // Function to determine if it is possible to make change for
amount v
 bool canMakeChangeOnce(const vector<int>& coins, int v) {
   int n = coins.size();
   vector<vector<bool>> dp(n + 1, vector<bool>(v + 1, false));
   dp[0][0] = true; // Base case: sum of 0 is possible with 0 coins

   for(int i = 1; i <= n; i++) {
     int coin = coins[i - 1];
     for(int j = 0; j <= v; j++) {
       // Exclude the current coin
       dp[i][j] = dp[i - 1][j];

       // Include the current coin if possible
       if(j >= coin && dp[i - 1][j - coin]) {
         dp[i][j] = true;
       }
     }
   }

   return dp[n][v];
 }

 int main() {
   // Sample test cases
   vector<pair<vector<int>, int>> test_cases = {
     {{1, 5, 10, 20}, 16}, // Expected: Yes (1 + 15)
     {{1, 5, 10, 20}, 31}, // Expected: Yes (1 + 10 + 20)
     {{1, 5, 10, 20}, 40}, // Expected: No
     {{2, 4, 6, 8}, 14},   // Expected: Yes (6 + 8)
     {{2, 4, 6, 8}, 15},   // Expected: No
     {{5, 10, 15}, 25},    // Expected: Yes (10 + 15)
     {{5, 10, 15}, 30},    // Expected: Yes (5 + 10 + 15)
     {{1, 2, 3, 4, 5}, 10},// Expected: Yes (1+2+3+4)
     {{}, 0},              // Expected: Yes (sum 0)
     {{}, 5},              // Expected: No (no coins)
   };

   for(size_t i = 0; i < test_cases.size(); ++i) {
     const vector<int>& coins = test_cases[i].first;
     int v = test_cases[i].second;
     bool result = canMakeChangeOnce(coins, v);

     cout << "Test Case " << i + 1 << ": " << endl;
     cout << "Coins: ";
     for(int coin : coins) {
       cout << coin << " ";
     }
     cout << "\nAmount v: " << v << endl;
     cout << "Can make change using each coin at most once? " << (
result ? "Yes" : "No") << endl;
```

```
        cout << "-----------------------------\n";
    }

    return 0;
}
```

Explanation

In this implementation:

- We define a function `canMakeChangeOnce` that takes the list of coin denominations and the target amount v.
- We initialize a two-dimensional boolean array dp of size $(n + 1) \times (v + 1)$.
- For each coin, we update the dp table based on whether we include or exclude the coin.
- The `main` function includes several test cases to demonstrate the algorithm.

Complexity Analysis

- **Time Complexity:** $O(nv)$
  - The outer loop runs n times (for each coin).
  - The inner loop runs $v + 1$ times (for each possible sum from 0 to v).
- **Space Complexity:** $O(nv)$
  - We use a two-dimensional array dp of size $(n + 1) \times (v + 1)$.

Sample Output

```
Test Case 1:
Coins: 1 5 10 20
Amount v: 16
Can make change using each coin at most once? Yes
-----------------------------
Test Case 2:
Coins: 1 5 10 20
Amount v: 31
Can make change using each coin at most once? Yes
-----------------------------
Test Case 3:
Coins: 1 5 10 20
Amount v: 40
Can make change using each coin at most once? No
-----------------------------
Test Case 4:
Coins: 2 4 6 8
```

```
Amount v: 14
Can make change using each coin at most once? Yes
-----------------------------
Test Case 5:
Coins: 2 4 6 8
Amount v: 15
Can make change using each coin at most once? No
-----------------------------
Test Case 6:
Coins: 5 10 15
Amount v: 25
Can make change using each coin at most once? Yes
-----------------------------
Test Case 7:
Coins: 5 10 15
Amount v: 30
Can make change using each coin at most once? No
-----------------------------
Test Case 8:
Coins: 1 2 3 4 5
Amount v: 10
Can make change using each coin at most once? Yes
-----------------------------
Test Case 9:
Coins:
Amount v: 0
Can make change using each coin at most once? Yes
-----------------------------
Test Case 10:
Coins:
Amount v: 5
Can make change using each coin at most once? No
-----------------------------
```

Additional Notes

- **Edge Cases:**
  - If the amount v is 0, it is always possible to make change (using no coins).
  - If the list of coin denominations is empty and $v > 0$, it is not possible to make change.
- **Space Optimization:**
  - The space complexity can be reduced to $O(v)$ by using a one-dimensional array and iterating over the coins in reverse order.
  - However, since we need to avoid using the same coin multiple times, we must process the coins in such a way to prevent reuse.

- **Reconstructing the Coin Combination (Optional):**
  - If we need to find the actual set of coins used to make change for v, we can maintain a parent array or backtracking mechanism.
  - After filling the dp table, we can backtrack from dp$[n][v]$ to determine which coins were included.

Problem 11

Problem Statement

Given a convex polygon $P = (v_0, v_1, \ldots, v_{n-1})$ with n vertices, and a weight function $w(i, j, k)$ that assigns a weight to each possible triangle formed by the vertices v_i, v_j, and v_k, find a triangulation of the polygon that minimizes the total weight. The total weight is defined as the sum of the weights of the triangles in the triangulation.

Algorithm

Dynamic Programming Approach:

We can solve this problem using dynamic programming. The idea is to consider all possible sub-polygons and compute the minimum total weight triangulation for each sub-polygon.

1. **Define** a 2D array $\text{dp}[i][j]$ to represent the minimum total weight required to triangulate the sub-polygon starting at vertex v_i and ending at vertex v_j.
2. **Initialize** $\text{dp}[i][i+1] = 0$ for all i, since a sub-polygon with less than 3 vertices cannot be triangulated.
3. **Fill** the dp table using the following recurrence relation:

$$\text{dp}[i][j] = \min_{i<k<j} (\text{dp}[i][k] + \text{dp}[k][j] + w(i, k, j))$$

4. **Compute** $\text{dp}[i][j]$ for all $j - i \geq 2$ (i.e., for sub-polygons with at least 3 vertices).
5. **The minimum total weight** of the entire polygon is $\text{dp}[0][n-1]$.

Explanation of the Recurrence Relation

- For each sub-polygon from v_i to v_j, we consider all possible vertices v_k where $i < k < j$ to split the sub-polygon into two smaller sub-polygons: from v_i to v_k, and from v_k to v_j. - The cost of triangulating the sub-polygon from v_i to v_j is then the sum of: - The minimum cost of triangulating the sub-polygon from v_i to v_k ($\text{dp}[i][k]$), - The minimum cost of triangulating the sub-polygon from v_k to v_j ($\text{dp}[k][j]$), - The weight of the triangle formed by v_i, v_k, and v_j ($w(i, k, j)$).

C++ Implementation

```
 #include <iostream>
 #include <vector>
 #include <cmath>
 #include <cfloat>
 #include <iomanip>

 using namespace std;

 // Function to calculate the weight of the triangle formed by
vertices i, j, k
```

```
 double weightFunction(const vector<pair<double, double>>& points,
int i, int j, int k) {
   // Example weight function: Perimeter of the triangle
   double dx1 = points[i].first - points[j].first;
   double dy1 = points[i].second - points[j].second;
   double side1 = sqrt(dx1 * dx1 + dy1 * dy1);

   double dx2 = points[j].first - points[k].first;
   double dy2 = points[j].second - points[k].second;
   double side2 = sqrt(dx2 * dx2 + dy2 * dy2);

   double dx3 = points[k].first - points[i].first;
   double dy3 = points[k].second - points[i].second;
   double side3 = sqrt(dx3 * dx3 + dy3 * dy3);

   return side1 + side2 + side3; // Sum of the sides (Perimeter)
 }

 // Function to find the minimum total weight triangulation
 double minWeightTriangulation(const vector<pair<double, double>>&
points) {
   int n = points.size();
   vector<vector<double>> dp(n, vector<double>(n, 0));

   // Fill dp table
   for (int gap = 2; gap < n; gap++) {
     for (int i = 0, j = i + gap; j < n; i++, j++) {
       dp[i][j] = DBL_MAX;
       for (int k = i + 1; k < j; k++) {
         double cost = dp[i][k] + dp[k][j] + weightFunction(points,
i, k, j);
         if (cost < dp[i][j]) {
           dp[i][j] = cost;
         }
       }
     }
   }

   return dp[0][n - 1];
 }

 int main() {
   // Example convex polygon (vertices in order)
   vector<pair<double, double>> points = {
     {0, 0},
     {2, 0},
     {4, 0},
     {4, 2},
     {4, 4},
     {2, 4},
     {0, 4},
     {0, 2}
   };

   double minTotalWeight = minWeightTriangulation(points);
   cout << fixed << setprecision(2);
   cout << "Minimum Total Weight of Triangulation: " <<
minTotalWeight << endl;
```

```
    return 0;
}
```

Explanation

In this implementation:

- The function `weightFunction` calculates the weight of a triangle formed by three vertices. In this example, we use the perimeter of the triangle as the weight.
- The function `minWeightTriangulation` computes the minimum total weight triangulation using dynamic programming.
- We use a 2D array `dp` to store the minimum total weight for sub-polygons.
- We fill the `dp` table by increasing the gap between the starting and ending vertices.
- For each sub-polygon, we consider all possible points k to split the sub-polygon and update the minimum cost.
- The main function defines an example convex polygon and computes the minimum total weight triangulation.

Complexity Analysis

- **Time Complexity:** $O(n^3)$
  - The outer loop runs n times for the gap.
  - The middle loop runs n times for i.
  - The inner loop runs up to n times for k.
  - Therefore, the total time complexity is $O(n^3)$.
- **Space Complexity:** $O(n^2)$
  - We use a 2D array `dp` of size $n \times n$.

Optimization

For certain specific weight functions and properties of the polygon, the problem can sometimes be optimized to $O(n^2)$ time complexity. However, for arbitrary weight functions and general convex polygons, the dynamic programming approach with $O(n^3)$ time complexity is standard.

Sample Output

```
Minimum Total Weight of Triangulation: 34.97
```

Additional Notes

- **Weight Function Customization:** The weight function $w(i, j, k)$ can be customized according to the problem's requirements. Common choices include perimeter, area, or the sum of the lengths of the diagonals.
- **Reconstruction of Triangulation:** If we need to reconstruct the actual triangulation, we can maintain an additional table to keep track of the index k that gave the minimum cost for each sub-polygon $[i][j]$.
- **Convexity Requirement:** The algorithm assumes that the polygon is convex. For non-convex polygons, the problem becomes more complex and may require different algorithms.

Problem 12

Problem Statement

In the art gallery guarding problem, we are given a line L that represents a long hallway in an art gallery. We are also given a set $X = \{x_1, x_2, \ldots, x_n\}$ of real numbers that specify the positions of the paintings along this hallway, where $x_i \leq x_{i+1}$ for all i. Suppose that a single guard can protect all the paintings within a distance of at most 1 of his or her position (on both sides). Design an algorithm for finding a placement of guards that uses the minimum number of guards to guard all the paintings with positions in X.

Algorithm

Greedy Algorithm for Interval Covering:
We can solve this problem optimally using a greedy algorithm. The idea is to always place a guard as far to the right as possible, while still covering the earliest uncovered painting.

1. **Sort** the positions of the paintings in increasing order (if not already sorted).
2. **Initialize** an empty list to store the positions of the guards.
3. **Initialize** an index $i = 0$.
4. **While** $i < n$:
   (a) Let x_i be the position of the leftmost uncovered painting.
   (b) **Place a guard** at position $p = x_i + 1$. This guard will cover all paintings from x_i to $x_i + 2$.
   (c) **Add** p to the list of guard positions.
   (d) **Advance** i to the smallest index such that $x_i > p + 1$ (i.e., find the first painting not covered by this guard).
5. **Return** the list of guard positions.

Explanation

- The guard placed at position $p = x_i + 1$ covers all paintings in the interval $[p - 1, p + 1] = [x_i, x_i + 2]$.
- By placing the guard at $x_i + 1$, we maximize the coverage to the right while still covering x_i.
- We then skip all paintings covered by this guard and proceed to the next uncovered painting.
- This greedy strategy ensures that we use the minimum number of guards necessary to cover all paintings.

C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Function to find the minimum number of guards and their
positions
vector<double> findGuardPositions(const vector<double>& paintings)
{
  int n = paintings.size();
  vector<double> guards;

  // Sort the positions of the paintings
  vector<double> sorted_paintings = paintings;
  sort(sorted_paintings.begin(), sorted_paintings.end());

  int i = 0;
  while (i < n) {
    // Position of the leftmost uncovered painting
    double x = sorted_paintings[i];

    // Place a guard at position p = x + 1
    double guard_position = x + 1;
    guards.push_back(guard_position);

    // Advance i to skip all paintings covered by this guard
    i++;
    while (i < n && sorted_paintings[i] <= guard_position + 1) {
      i++;
    }
  }

  return guards;
}

int main() {
  // Example positions of paintings
  vector<double> paintings = {0.5, 1.2, 2.8, 3.0, 4.1, 5.5, 6.0,
7.2, 8.5, 9.1};

  vector<double> guard_positions = findGuardPositions(paintings);

  // Output the results
  cout << "Minimum number of guards required: " << guard_positions.
size() << endl;
  cout << "Guard positions:" << endl;
  for (double pos : guard_positions) {
    cout << pos << " ";
  }
  cout << endl;

  return 0;
}
```

Explanation of the Code

- The function `findGuardPositions` takes a vector of painting positions and returns a vector of guard positions that cover all the paintings.
- We first sort the painting positions to ensure they are in increasing order.
- We initialize an index $i = 0$ and an empty vector to store guard positions.
- While $i < n$, we:
  - Place a guard at position $x_i + 1$.
  - Increment i to skip all paintings covered by this guard (i.e., while $x_i \leq \text{guard\_position} + 1$).
- In the `main` function, we provide an example set of painting positions and output the minimum number of guards and their positions.

Sample Output

```
Minimum number of guards required: 4
Guard positions:
1.5 4.0 6.5 9.5
```

Explanation of the Sample Output

Given the painting positions:

$$\{0.5,\quad 1.2,\quad 2.8,\quad 3.0,\quad 4.1,\quad 5.5,\quad 6.0,\quad 7.2,\quad 8.5,\quad 9.1\}$$

The algorithm places guards at positions:

- $0.5 + 1 = 1.5$: Covers paintings at positions 0.5 and 1.2 (from 0.5 to 2.5).
- Next uncovered painting is at 2.8.
- $2.8 + 1 = 3.8$: Place guard at 3.8, covering paintings at 2.8, 3.0, and 4.1 (from 2.8 to 4.8).
- Next uncovered painting is at 5.5.
- $5.5 + 1 = 6.5$: Place guard at 6.5, covering paintings at 5.5, 6.0, and 7.2 (from 5.5 to 7.5).
- Next uncovered painting is at 8.5.
- $8.5 + 1 = 9.5$: Place guard at 9.5, covering paintings at 8.5 and 9.1 (from 8.5 to 10.5).

Thus, we need a minimum of 4 guards to cover all the paintings.

Complexity Analysis

- **Time Complexity:** $O(n \log n)$
  - Sorting the painting positions takes $O(n \log n)$ time.
  - The while loop runs $O(n)$ times, as each painting is considered at most once.
  - Therefore, the overall time complexity is dominated by the sorting step.
- **Space Complexity:** $O(n)$
  - We use additional space to store the sorted painting positions and the guard positions.
  - Both require $O(n)$ space.

Proof of Optimality

The greedy algorithm described above produces an optimal solution. Here's why:

- By always placing the guard at $x_i + 1$, we maximize the coverage to the right while still covering the leftmost uncovered painting x_i.
- This ensures that we cover as many paintings as possible with each guard placement.
- If we were to place the guard at any position less than $x_i + 1$, we would cover fewer paintings to the right, potentially requiring more guards.
- Therefore, the greedy choice at each step leads to an optimal overall solution.

Additional Notes

- **Extensions:** The algorithm can be easily adapted if the coverage distance of the guards changes. For a coverage radius of r, we place the guard at $x_i + r$ and adjust the coverage interval accordingly.
- **Applications:** This type of problem is common in scheduling, network coverage, and resource allocation where intervals need to be covered optimally.