Assignment - 9

Student Name: Nikhil Kumar Tiwari
UID:22BCS10471
Branch :BE-CSE
Section/Group:22BCS-IOT-FL-601 A
Semester: 6 th
Subject Code: 22CSP-351
Subject Name: Advanced Programming Lab- 2

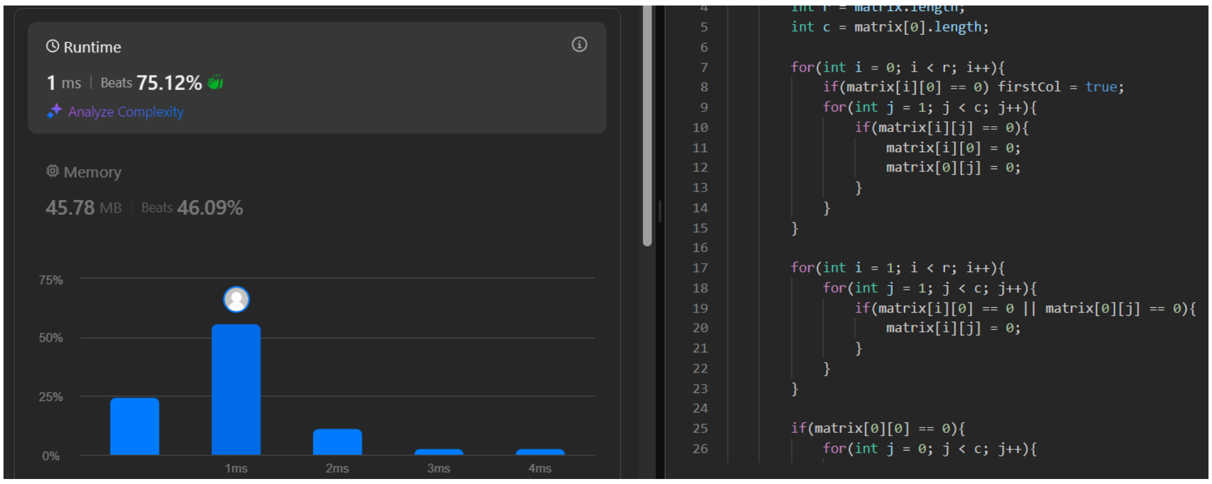
Fast Learners Assignemnt Solutions:-

1. **Set Matrix Zeroes:** Given an m x n matrix, if an element is 0, set its entire row and column to 0.

```
class Solution {
   public void setZeroes(int[][] matrix) {
      boolean firstCol = false;
      int r = matrix.length;
      int c = matrix[0].length;

      for(int i = 0; i < r; i++){
         if(matrix[i][0] == 0) firstCol = true;
         for(int j = 1; j < c; j++){
            if(matrix[i][j] == 0){
               matrix[i][0] = 0;
               matrix[0][j] = 0;
            }
         }
      }

      for(int i = 1; i < r; i++){
         for(int j = 1; j < c; j++){
            if(matrix[i][0] == 0 || matrix[0][j] == 0){
               matrix[i][j] = 0;
            }
         }
      }

      if(matrix[0][0] == 0){
         for(int j = 0; j < c; j++){
            matrix[0][j] = 0;
         }
      }
```

```
        if(firstCol){
            for(int i = 0; i < r; i++){
                matrix[i][0] = 0;
            }
        }
    }
}
```

Output:-


Runtime

1 ms | Beats 75.12%

Analyze Complexity

Memory

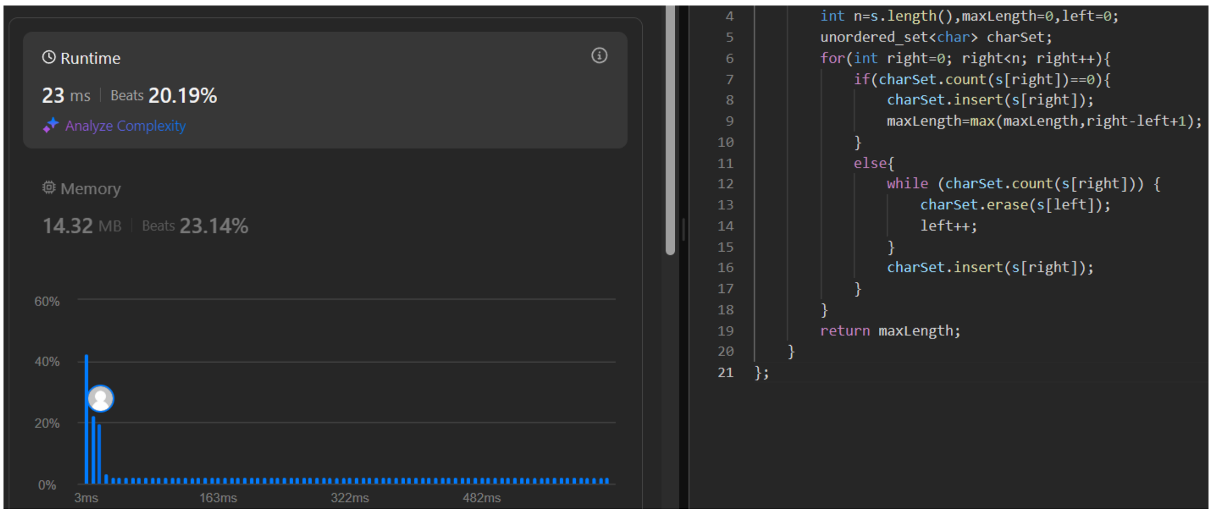
45.78 MB | Beats 46.09%

```
        int r = matrix.length;
        int c = matrix[0].length;

        for(int i = 0; i < r; i++){
            if(matrix[i][0] == 0) firstCol = true;
            for(int j = 1; j < c; j++){
                if(matrix[i][j] == 0){
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        }

        for(int i = 1; i < r; i++){
            for(int j = 1; j < c; j++){
                if(matrix[i][0] == 0 || matrix[0][j] == 0){
                    matrix[i][j] = 0;
                }
            }
        }

        if(matrix[0][0] == 0){
            for(int j = 0; j < c; j++){
```

2. **Longest Substring Without Repeating Characters:** Given a string s, find the length of longest substring that does not contain any repeating characters.

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int n=s.length(),maxLength=0,left=0;
        unordered_set<char> charSet;
        for(int right=0; right<n; right++){
            if(charSet.count(s[right])==0){
                charSet.insert(s[right]);
                maxLength=max(maxLength,right-left+1);
            }
            else{
                while (charSet.count(s[right])) {
                    charSet.erase(s[left]);
                    left++;
                }
                charSet.insert(s[right]);
            }
        }
        return maxLength;
    }
};
```

<u>Result:-</u>


Runtime

23 ms | Beats 20.19%

Analyze Complexity

Memory

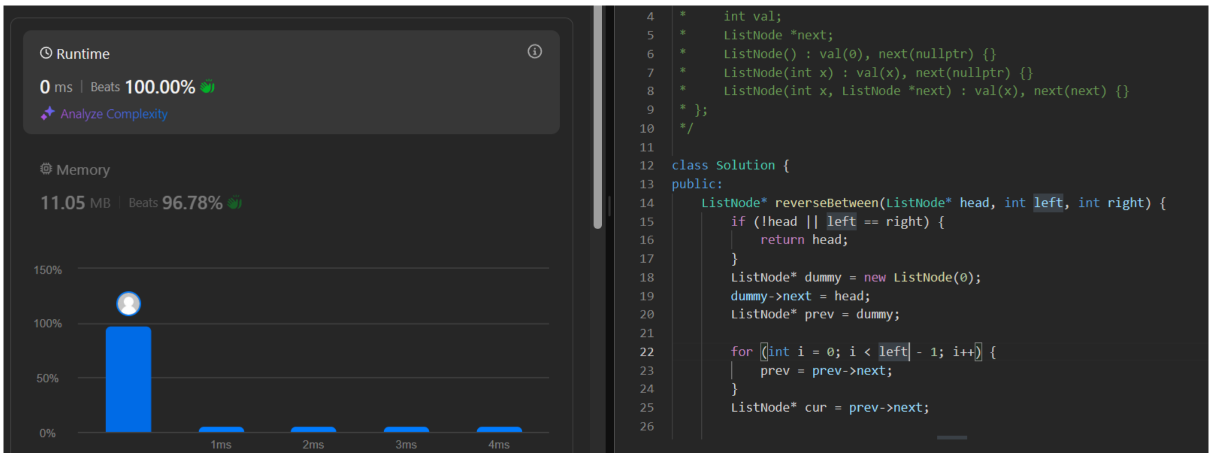
14.32 MB | Beats 23.14%

```
        int n=s.length(),maxLength=0,left=0;
        unordered_set<char> charSet;
        for(int right=0; right<n; right++){
            if(charSet.count(s[right])==0){
                charSet.insert(s[right]);
                maxLength=max(maxLength,right-left+1);
            }
            else{
                while (charSet.count(s[right])) {
                    charSet.erase(s[left]);
                    left++;
                }
                charSet.insert(s[right]);
            }
        }
        return maxLength;
    }
};
```

3. **<u>Reverse Linked List II:</u>** Given the head of a singly linked list and two integers left and right, reverse the nodes of the list from position left to right.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * }; */
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        if (!head || left == right) {
            return head;
        }
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;

        for (int i = 0; i < left - 1; i++) {
            prev = prev->next;
        }
        ListNode* cur = prev->next;

        for (int i = 0; i < right - left; i++) {
            ListNode* temp = cur->next;
            cur->next = temp->next;
            temp->next = prev->next;
            prev->next = temp;
        }

        return dummy->next;
    }
};
```

<u>Output:-</u>

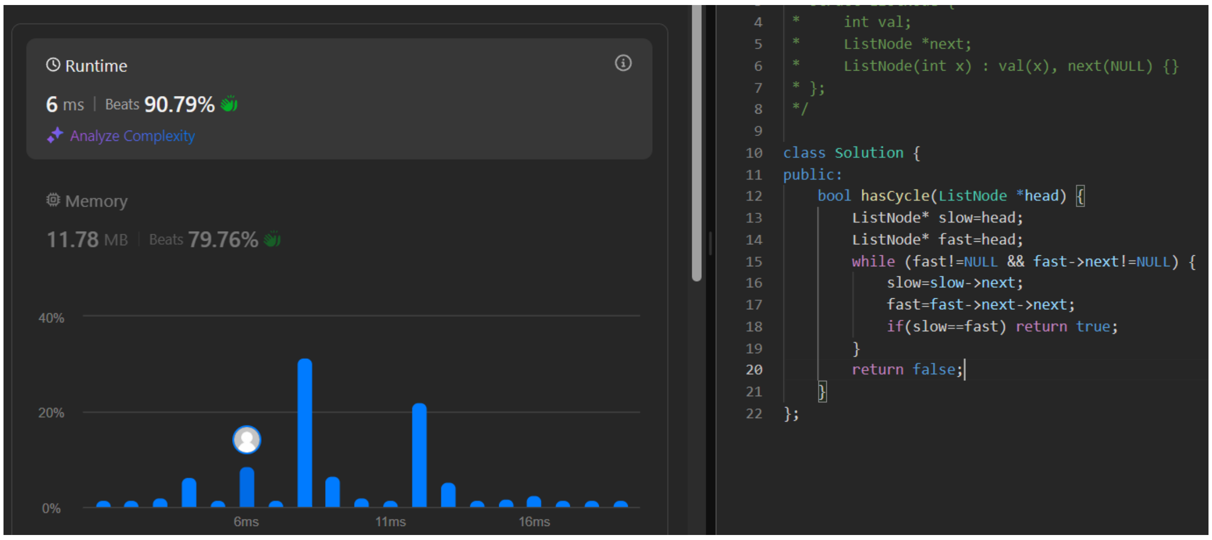


```
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */

class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        if (!head || left == right) {
            return head;
        }
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;

        for (int i = 0; i < left - 1; i++) {
            prev = prev->next;
        }
        ListNode* cur = prev->next;

```

4. **Detect a Cycle in a Linked List:** Given the head of a linked list, determine whether the linked list contains a cycle. A cycle occurs if a node's next pointer points to a previous node in the list.

```
/* Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * }; */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* slow=head;
        ListNode* fast=head;
        while (fast!=NULL && fast->next!=NULL) {
            slow=slow->next;
            fast=fast->next->next;
            if(slow==fast) return true;
        }
        return false;
    }
};
```

Output:-



Runtime

6 ms | Beats 90.79%

Analyze Complexity

Memory

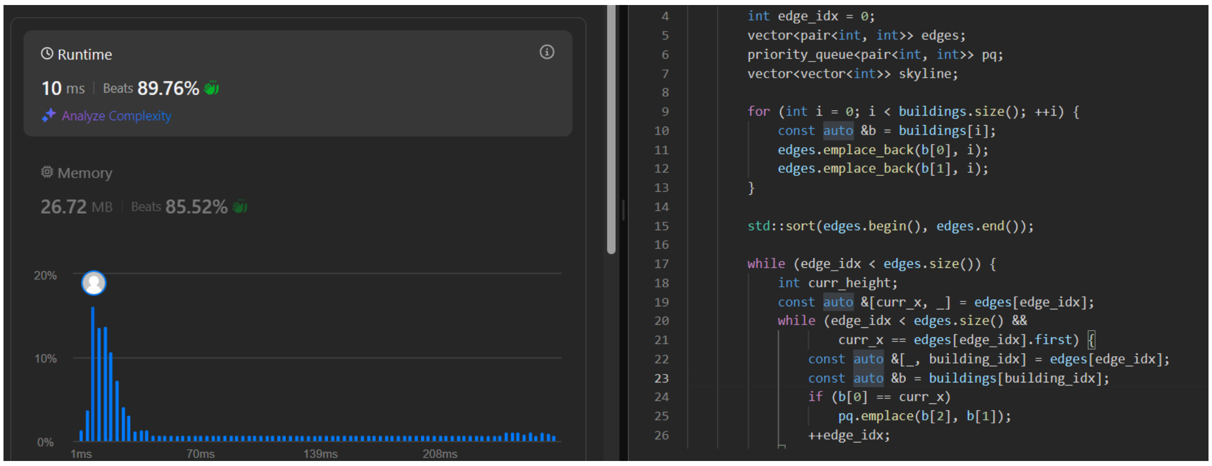
11.78 MB | Beats 79.76%

```
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */

class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* slow=head;
        ListNode* fast=head;
        while (fast!=NULL && fast->next!=NULL) {
            slow=slow->next;
            fast=fast->next->next;
            if(slow==fast) return true;
        }
        return false;
    }
};
```

5. **The Skyline Problem:** Given a list of buildings represented as [left, right, height], where each building is a rectangle, return the key points of the skyline. A key point is represented as [x, y], where x is the x-coordinate where the height changes to y.

```
class Solution {
public:
  vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
    int edge_idx = 0;
    vector<pair<int, int>> edges;
    priority_queue<pair<int, int>> pq;
    vector<vector<int>> skyline;

    for (int i = 0; i < buildings.size(); ++i) {
      const auto &b = buildings[i];
      edges.emplace_back(b[0], i);
      edges.emplace_back(b[1], i);
    }

    std::sort(edges.begin(), edges.end());

    while (edge_idx < edges.size()) {
      int curr_height;
      const auto &[curr_x, _] = edges[edge_idx];
      while (edge_idx < edges.size() &&
          curr_x == edges[edge_idx].first) {
        const auto &[_, building_idx] = edges[edge_idx];
        const auto &b = buildings[building_idx];
        if (b[0] == curr_x)
          pq.emplace(b[2], b[1]);
        ++edge_idx;
      }
      while (!pq.empty() && pq.top().second <= curr_x)
        pq.pop();
      curr_height = pq.empty() ? 0 : pq.top().first;
      if (skyline.empty() || skyline.back()[1] != curr_height)
        skyline.push_back({curr_x, curr_height});
        }
        return skyline;
    }
};
```

Output:-



Runtime
10 ms | Beats 89.76%
Analyze Complexity

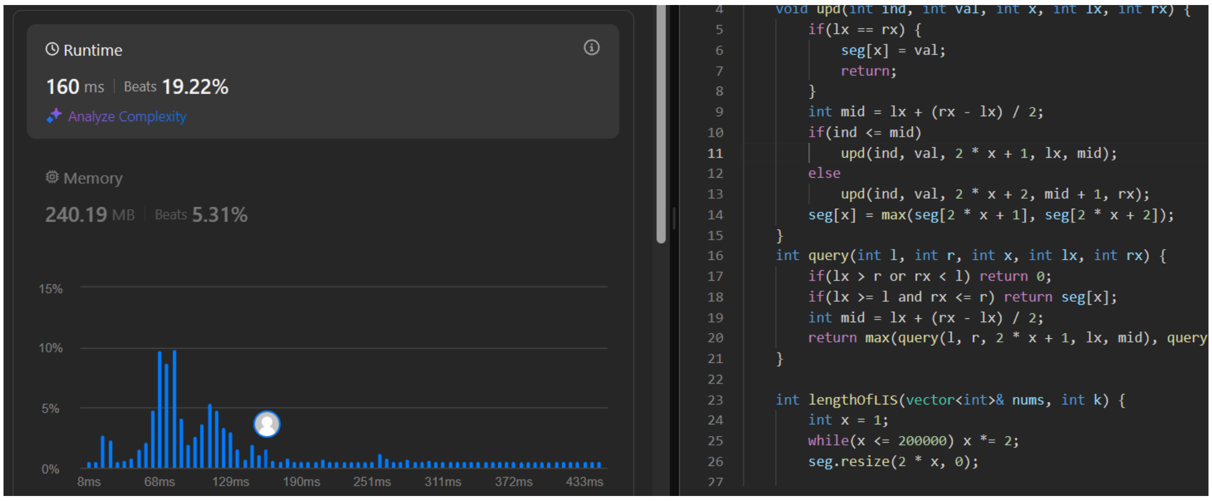
Memory
26.72 MB | Beats 85.52%

```
int edge_idx = 0;
vector<pair<int, int>> edges;
priority_queue<pair<int, int>> pq;
vector<vector<int>> skyline;

for (int i = 0; i < buildings.size(); ++i) {
    const auto &b = buildings[i];
    edges.emplace_back(b[0], i);
    edges.emplace_back(b[1], i);
}

std::sort(edges.begin(), edges.end());

while (edge_idx < edges.size()) {
    int curr_height;
    const auto &[curr_x, _] = edges[edge_idx];
    while (edge_idx < edges.size() &&
            curr_x == edges[edge_idx].first) {
        const auto &[_, building_idx] = edges[edge_idx];
        const auto &b = buildings[building_idx];
        if (b[0] == curr_x)
            pq.emplace(b[2], b[1]);
        ++edge_idx;
```

6. **Longest Increasing Subsequence II:** Given an integer array nums, find the length of the longest strictly increasing subsequence. A subsequence is derived from the array by deleting some or no elements without changing the order of the remaining elements.

```
class Solution {
public:
    vector<int> seg;
    void upd(int ind, int val, int x, int lx, int rx) {
        if(lx == rx) {
            seg[x] = val;
            return;
        }
        int mid = lx + (rx - lx) / 2;
        if(ind <= mid)
            upd(ind, val, 2 * x + 1, lx, mid);
        else
            upd(ind, val, 2 * x + 2, mid + 1, rx);
        seg[x] = max(seg[2 * x + 1], seg[2 * x + 2]);
    }
    int query(int l, int r, int x, int lx, int rx) {
        if(lx > r or rx < l) return 0;
        if(lx >= l and rx <= r) return seg[x];
        int mid = lx + (rx - lx) / 2;
        return max(query(l, r, 2 * x + 1, lx, mid), query(l, r, 2 * x + 2, mid + 1,
rx));
    }

    int lengthOfLIS(vector<int>& nums, int k) {
        int x = 1;
        while(x <= 200000) x *= 2;
        seg.resize(2 * x, 0);

        int res = 1;
        for(int i = 0; i < nums.size(); ++i) {
            int left = max(1, nums[i] - k), right = nums[i] - 1;
            int q = query(left, right, 0, 0, x - 1);
            res = max(res, q + 1);
            upd(nums[i], q + 1, 0, 0, x - 1);
        }
        return res;
    }
};
```

<u>Output:-</u>

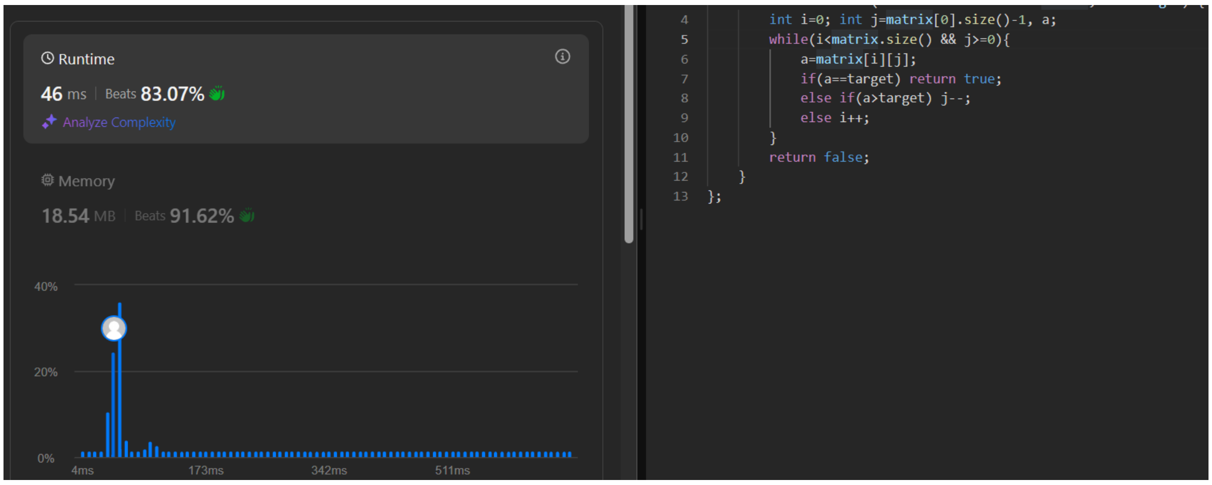


```
    void upd(int ind, int val, int x, int lx, int rx) {
        if(lx == rx) {
            seg[x] = val;
            return;
        }
        int mid = lx + (rx - lx) / 2;
        if(ind <= mid)
            upd(ind, val, 2 * x + 1, lx, mid);
        else
            upd(ind, val, 2 * x + 2, mid + 1, rx);
        seg[x] = max(seg[2 * x + 1], seg[2 * x + 2]);
    }
    int query(int l, int r, int x, int lx, int rx) {
        if(lx > r or rx < l) return 0;
        if(lx >= l and rx <= r) return seg[x];
        int mid = lx + (rx - lx) / 2;
        return max(query(l, r, 2 * x + 1, lx, mid), query
    }

    int lengthOfLIS(vector<int>& nums, int k) {
        int x = 1;
        while(x <= 200000) x *= 2;
        seg.resize(2 * x, 0);

```

7. **<u>Search a 2D Matrix II:</u>** Given an m x n matrix where each row is sorted in ascending order from left to right and each column is sorted in ascending order from top to bottom, and an integer target, determine if the target exists in the matrix.

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int i=0; int j=matrix[0].size()-1, a;
        while(i<matrix.size() && j>=0){
            a=matrix[i][j];
            if(a==target) return true;
            else if(a>target) j--;
            else i++;
        }
        return false;
    }
};
```

<u>Output:-</u>



Runtime

46 ms | Beats 83.07%

Analyze Complexity

Memory

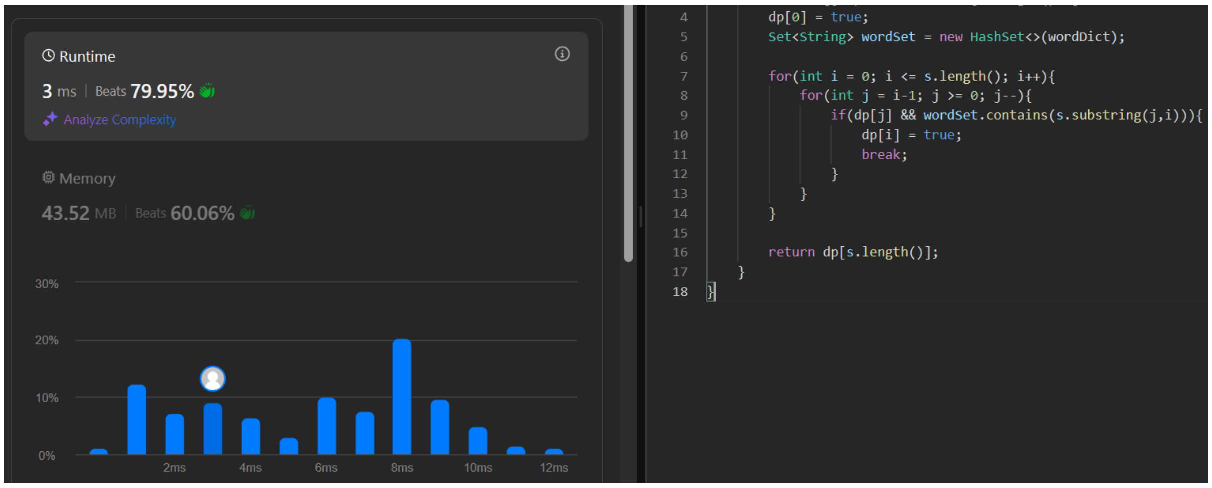
18.54 MB | Beats 91.62%

```
        int i=0; int j=matrix[0].size()-1, a;
        while(i<matrix.size() && j>=0){
            a=matrix[i][j];
            if(a==target) return true;
            else if(a>target) j--;
            else i++;
        }
        return false;
    }
};
```

8. **<u>Word Break:</u>** Given a string s and a dictionary wordDict containing a list of words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. The same word can be reused multiple times.

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        boolean[] dp = new boolean[s.length()+1];
        dp[0] = true;
        Set<String> wordSet = new HashSet<>(wordDict);

        for(int i = 0; i <= s.length(); i++){
            for(int j = i-1; j >= 0; j--){
                if(dp[j] && wordSet.contains(s.substring(j,i))){
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[s.length()];
    }
}
```

<u>Output:-</u>



3 ms | Beats 79.95%

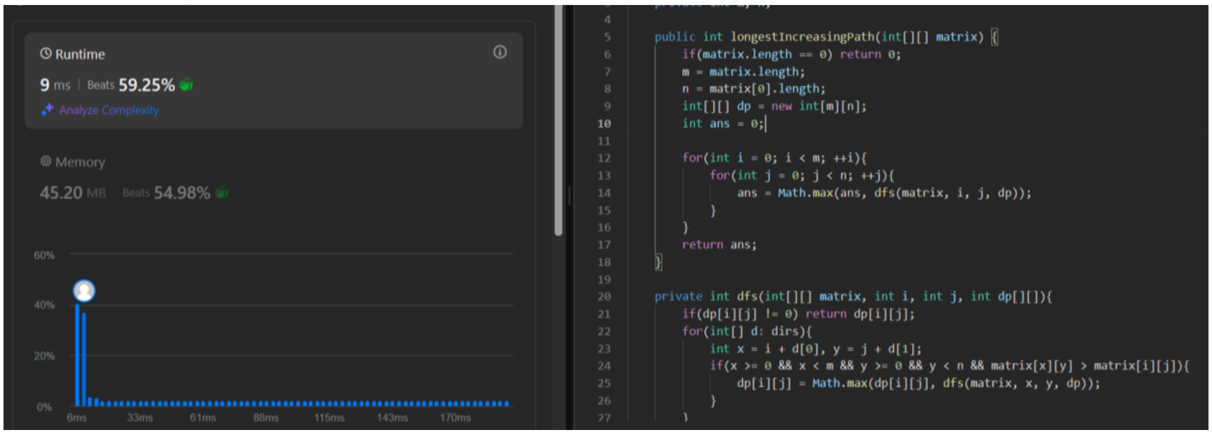
43.52 MB | Beats 60.06%

```
        dp[0] = true;
        Set<String> wordSet = new HashSet<>(wordDict);

        for(int i = 0; i <= s.length(); i++){
            for(int j = i-1; j >= 0; j--){
                if(dp[j] && wordSet.contains(s.substring(j,i))){
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[s.length()];
    }
}
```

9. **Longest Increasing Path in a Matrix:** Given an m x n integer matrix, find the length of the longest strictly increasing path. You can move up, down, left, or right from each cell. Diagonal moves and moves outside the boundaries are not allowed.

```
class Solution {
   private static int[][] dirs = {{0,1}, {0,-1}, {-1,0}, {1,0}};
   private int m, n;

   public int longestIncreasingPath(int[][] matrix) {
      if(matrix.length == 0) return 0;
      m = matrix.length;
      n = matrix[0].length;
      int[][] dp = new int[m][n];
      int ans = 0;

      for(int i = 0; i < m; ++i){
         for(int j = 0; j < n; ++j){
            ans = Math.max(ans, dfs(matrix, i, j, dp));
         }
      }
      return ans;
   }

   private int dfs(int[][] matrix, int i, int j, int dp[][]){
      if(dp[i][j] != 0) return dp[i][j];
      for(int[] d: dirs){
         int x = i + d[0], y = j + d[1];
         if(x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] >
matrix[i][j]){
            dp[i][j] = Math.max(dp[i][j], dfs(matrix, x, y, dp));
         }
      }
      return ++dp[i][j];
   }
}
```

Output:-



Runtime

9 ms | Beats 59.25%

Analyze Complexity

Memory

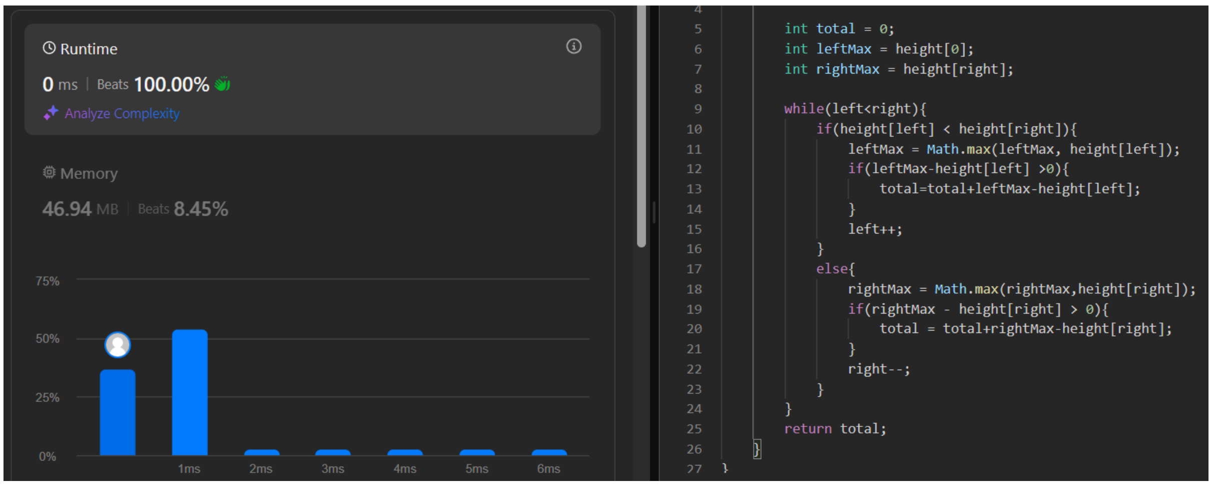
45.20 MB | Beats 54.98%

```
    public int longestIncreasingPath(int[][] matrix) {
        if(matrix.length == 0) return 0;
        m = matrix.length;
        n = matrix[0].length;
        int[][] dp = new int[m][n];
        int ans = 0;

        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                ans = Math.max(ans, dfs(matrix, i, j, dp));
            }
        }
        return ans;
    }

    private int dfs(int[][] matrix, int i, int j, int dp[][]){
        if(dp[i][j] != 0) return dp[i][j];
        for(int[] d: dirs){
            int x = i + d[0], y = j + d[1];
            if(x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] > matrix[i][j]){
                dp[i][j] = Math.max(dp[i][j], dfs(matrix, x, y, dp));
            }
        }
```

10. **Trapping Rain Water:** Given n non-negative integers representing an elevation map where the width of each bar is 1, compute the total amount of water that can be trapped after raining.

```
class Solution {
    public int trap(int[] height) {
        int left = 0;
        int right = height.length - 1;

        int total = 0;
        int leftMax = height[0];
        int rightMax = height[right];

        while(left<right){
            if(height[left] < height[right]){
                leftMax = Math.max(leftMax, height[left]);
                if(leftMax-height[left] >0){
                    total=total+leftMax-height[left];
                }
                left++;
            }
            else{
                rightMax = Math.max(rightMax,height[right]);
                if(rightMax - height[right] > 0){
                    total = total+rightMax-height[right];
                }
                right--;
            }
        }
        return total;
    }
}
```

Output:-



Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

46.94 MB | Beats 8.45%

```
        int total = 0;
        int leftMax = height[0];
        int rightMax = height[right];

        while(left<right){
            if(height[left] < height[right]){
                leftMax = Math.max(leftMax, height[left]);
                if(leftMax-height[left] >0){
                    total=total+leftMax-height[left];
                }
                left++;
            }
            else{
                rightMax = Math.max(rightMax,height[right]);
                if(rightMax - height[right] > 0){
                    total = total+rightMax-height[right];
                }
                right--;
            }
        }
        return total;
    }
}
```