

# AuE-8930: Homework 3

---

**Author:** Tanmay Samak

## CLONE

- Clone this repository:

```
$ git clone https://github.com/Tinker-Twins/Computing-and-Simulation-for-Autonomy.git
```

- Change directory to Tanmay's HW3:

```
$ cd Computing-and-Simulation-for-Autonomy/HW3/Tanmay
```

## BUILD

- Compile the C++ program for questions Q1 to Q5 and Q7 to Q8:

```
$ g++ Q1.cpp -o Q1
$ g++ Q2.cpp -o Q2
$ g++ Q3.cpp -o Q3
$ g++ Q4.cpp -o Q4
$ g++ Q5.cpp -o Q5
$ g++ Q7.cpp -o Q7
$ g++ Q8.cpp -o Q8
```

- For Q6, the build instructions are a little different:

You will first need to install OpenCV.

Update/upgrade apt packages:

```
$ sudo apt update
$ sudo apt upgrade
```

Install core OpenCV libraries:

```
$ sudo apt install libopencv-dev
```

[Optional] Install OpenCV with Python3 bindings:

```
$ sudo apt install python3-opencv
```

Now compile the C++ program for Q6, with the necessary flags to link against OpenCV libraries:

```
$ g++ Q6.cpp -o Q6 `pkg-config opencv4 --cflags --libs`
```

## RUN

- Run the executable for questions Q1 to Q8:

```
$ ./Q1  
$ ./Q2  
$ ./Q3  
$ ./Q4  
$ ./Q5  
$ ./Q6  
$ ./Q7  
$ ./Q8
```

- For Q9 execute the respective Python scripts:

Assuming that you are already in `Computing-and-Simulation-for-Autonomy/HW3/Tanmay` directory, change directory to `map-path-search`:

```
$ cd map-path-search
```

For Q9a\_1, uncomment `a_star_occupancy` section and run:

```
$ python3 occupancy_map_8n.py
```

For Q9a\_2, uncomment `dijkstra_occupancy` section and run:

```
$ python3 occupancy_map_8n.py
```

For Q9b\_1, uncomment `a_star_quadtree` section and run:

```
$ python3 quadtree_map_8n.py
```

For Q9b\_2, uncomment `dijkstra_quadtree` section and run:

```
$ python3 quadtree_map_8n.py
```

For Q9c\_1a, change the resolution, uncomment both `a_star_occupancy` and `dijkstra_occupancy` sections and run:

```
$ python3 occupancy_map_8n.py
```

For Q9c\_1b, change the resolution, uncomment both `a_star_quadtree` and `dijkstra_quadtree` sections and run:

```
$ python3 quadtree_map_8n.py
```

For Q9c\_2a, uncomment both `a_star_occupancy` and `dijkstra_occupancy` sections and run:

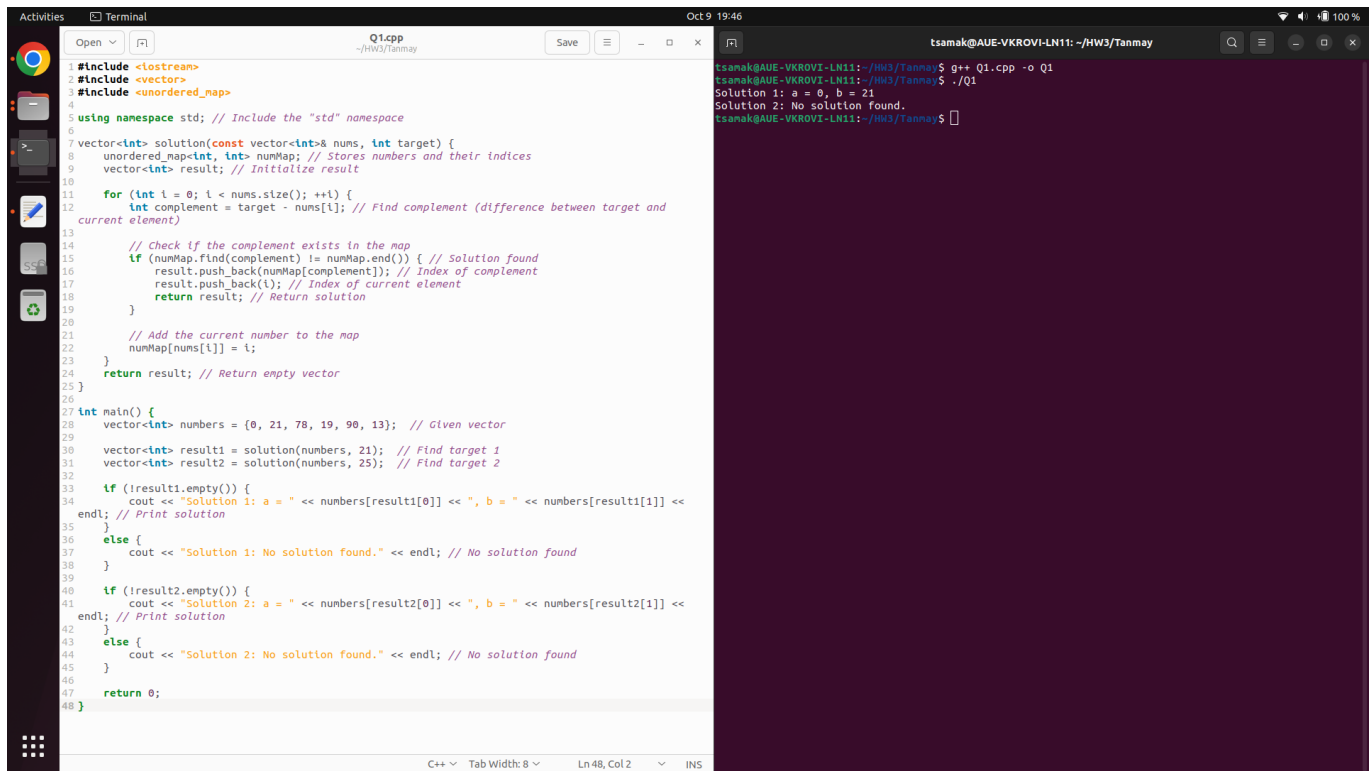
```
$ python3 occupancy_map_8n.py
```

For Q9c\_2b, uncomment both `a_star_quadtree` and `dijkstra_quadtree` sections and run:

```
$ python3 quadtree_map_8n.py
```

## RESULTS

1. A sample execution for Q1 is depicted below:



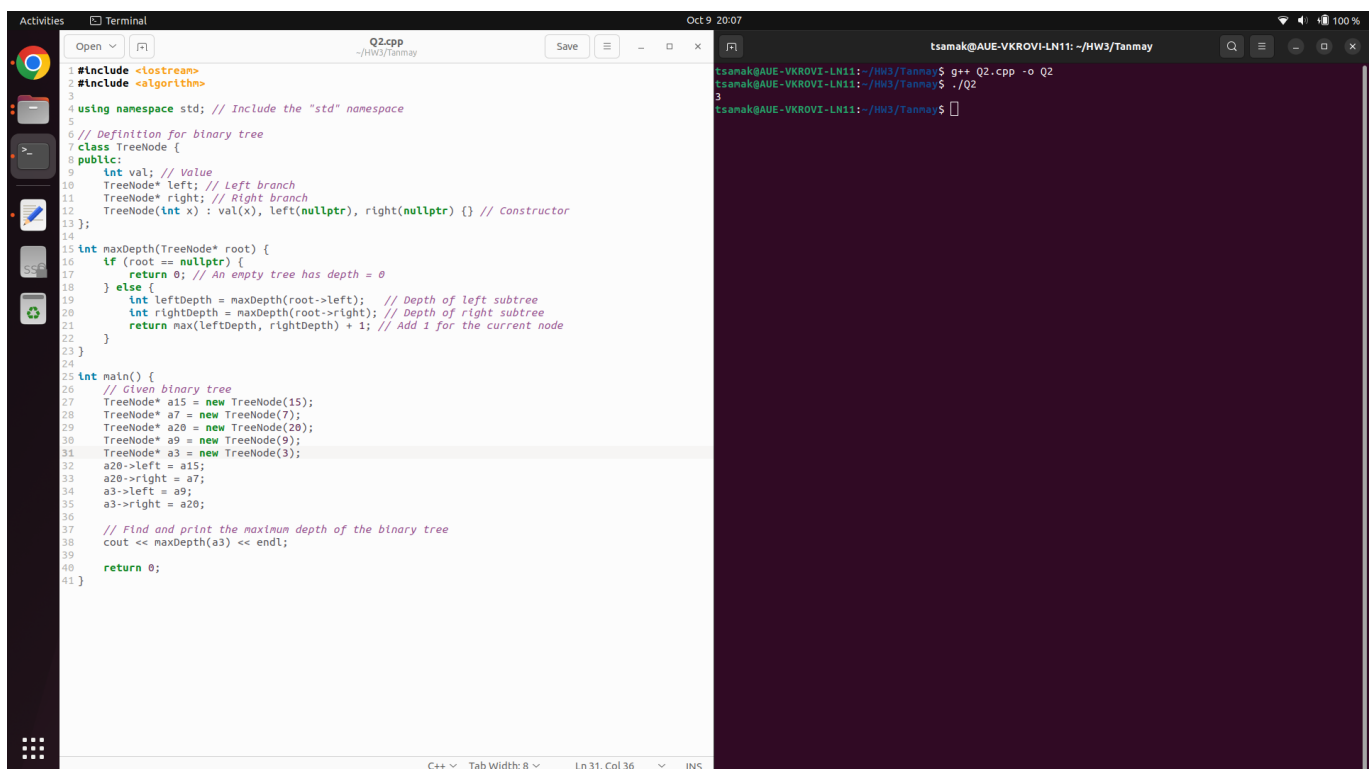
```

1 #include <iostream>
2 #include <vector>
3 #include <unordered_map>
4
5 using namespace std; // Include the "std" namespace
6
7 vector<int> solution(const vector<int>& nums, int target) {
8     unordered_map<int, int> numMap; // Stores numbers and their indices
9     vector<int> result; // Initialize result
10
11     for (int i = 0; i < nums.size(); ++i) {
12         int complement = target - nums[i]; // Find complement (difference between target and
            current element)
13
14         // Check if the complement exists in the map
15         if (numMap.find(complement) != numMap.end()) { // Solution found
16             result.push_back(numMap[complement]); // Index of complement
17             result.push_back(i); // Index of current element
18             return result; // Return solution
19         }
20
21         // Add the current number to the map
22         numMap[nums[i]] = i;
23     }
24     return result; // Return empty vector
25 }
26
27 int main() {
28     vector<int> numbers = {0, 21, 78, 19, 90, 13}; // Given vector
29
30     vector<int> result1 = solution(numbers, 21); // Find target 1
31     vector<int> result2 = solution(numbers, 25); // Find target 2
32
33     if (!result1.empty()) {
34         cout << "Solution 1: a = " << numbers[result1[0]] << ", b = " << numbers[result1[1]] <<
            endl; // Print solution
35     }
36     else {
37         cout << "Solution 1: No solution found." << endl; // No solution found
38     }
39
40     if (!result2.empty()) {
41         cout << "Solution 2: a = " << numbers[result2[0]] << ", b = " << numbers[result2[1]] <<
            endl; // Print solution
42     }
43     else {
44         cout << "Solution 2: No solution found." << endl; // No solution found
45     }
46
47     return 0;
48 }

```

The time complexity of this algorithm is  $O(n)$ , where  $n$  is the number of elements in the array, because the map operations (insertion and retrieval) are done in constant time on average.

2. A sample execution for Q2 is depicted below:



```

1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std; // Include the "std" namespace
5
6 // Definition for binary tree
7 class TreeNode {
8 public:
9     int val; // Value
10    TreeNode* left; // Left branch
11    TreeNode* right; // Right branch
12    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} // Constructor
13 };
14
15 int maxDepth(TreeNode* root) {
16     if (root == nullptr) {
17         return 0; // An empty tree has depth = 0
18     } else {
19         int leftDepth = maxDepth(root->left); // Depth of left subtree
20         int rightDepth = maxDepth(root->right); // Depth of right subtree
21         return max(leftDepth, rightDepth) + 1; // Add 1 for the current node
22     }
23 }
24
25 int main() {
26     // Given binary tree
27     TreeNode* a15 = new TreeNode(15);
28     TreeNode* a7 = new TreeNode(7);
29     TreeNode* a20 = new TreeNode(20);
30     TreeNode* a9 = new TreeNode(9);
31     TreeNode* a3 = new TreeNode(3);
32     a20->left = a15;
33     a20->right = a7;
34     a3->left = a9;
35     a3->right = a20;
36
37     // Find and print the maximum depth of the binary tree
38     cout << maxDepth(a3) << endl;
39
40     return 0;
41 }

```

The time complexity of this algorithm is  $O(n)$ , where  $n$  is the number of nodes in the binary tree, as it visits each node exactly once.

3. A sample execution for Q3 is depicted below:

```

14  ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
15      ListNode* dummy = new ListNode(0); // Create a dummy node
16      ListNode* current = dummy; // Initialize a current pointer to the dummy node
17      int carry = 0; // Initialize carry to 0 to track any carryover
18
19      while (l1 || l2) { // Continue loop until both input linked lists are processed
20          int x = (l1) ? l1->val : 0; // Get value of current node in l1, or 0 if l1 is nullptr
21          int y = (l2) ? l2->val : 0; // Get value of current node in l2, or 0 if l2 is nullptr
22
23          int sum = x + y + carry; // Calculate the sum of digits from l1, l2, and any carry
24          carry = sum / 10; // Update carry for the next iteration
25          current->next = new ListNode(sum % 10); // Create a new node for units place
26          current = current->next; // Move the current pointer to the newly created node
27
28          if (l1) l1 = l1->next; // Move to the next node in l1 if it exists
29          if (l2) l2 = l2->next; // Move to the next node in l2 if it exists
30      }
31
32      if (carry > 0) { // If there is a carry after the loop
33          current->next = new ListNode(carry); // Create a new node for carry
34      }
35
36      return dummy->next; // Return the next node of the dummy, which is the actual result
37  }
38
39  // Function to print a linked list for testing
40  void printList(ListNode* head) {
41      while (head) {
42          cout << head->val << " "; // Print current value
43          head = head->next; // Go to next link
44      }
45      cout << "nullptr" << endl; // Print 'nullptr' at the end
46  }
47
48  int main() {
49      // Given linked lists
50      ListNode* l1 = new ListNode(2);
51      l1->next = new ListNode(4);
52      l1->next->next = new ListNode(3);
53
54      ListNode* l2 = new ListNode(5);
55      l2->next = new ListNode(6);
56      l2->next->next = new ListNode(4);
57
58      // Add the two numbers
59      Solution solution;
60      ListNode* result = solution.addTwoNumbers(l1, l2);
61
62      // Print the result
63      printList(result);
64      return 0;
65  }
66
67  }

```

```

tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ g++ Q3.cpp -o Q3
tsamak@AUE-VKROVI-LN11:~/HW3/Tanmay$ ./Q3
7 -> 0 -> 8 -> nullptr
tsamak@AUE-VKROVI-LN11:~/HW3/Tanmay$

```

The time complexity of this algorithm is  $O(\max(n, m))$ , where  $n$  is the length of the first linked list **11** and  $m$  is the length of the second linked list **12** (worst case scenario). This is because the while loop runs until both linked lists (**11** and **12**) are fully traversed or until the carry becomes zero. In the worst case, it will iterate through the longer of the two input lists. Within the loop, each iteration involves constant time operations such as addition, modulo, and updating pointers, all of which take  $O(1)$  time. After the loop, if there's a carry, an additional node is added to the result, which takes  $O(1)$  time. Since the number of iterations in the while loop is determined by the length of the longer linked list among **11** and **12**, the time complexity is  $O(\max(N, M))$ .

4. A sample execution for **Q4** is depicted below:

```

1  #include <iostream>
2  #include <vector>
3  #include <unordered_map>
4
5  using namespace std; // Include the "std" namespace
6
7  class Solution {
8  public:
9      int lengthOfLongestSubstring(string s) {
10         vector<int> charIndex(128, -1); // Stores the last seen index of each character
11         int maxLength = 0; // Initialize 'maxLength'
12         int start = 0; // Starting index of the current substring
13
14         for (int end = 0; end < s.length(); ++end) {
15             // If the current character has been seen before and its last seen index
16             // is greater than or equal to the start index of the current substring
17             if (charIndex[s[end]] >= start) {
18                 // Update start index to the next position after last occurrence of the character
19                 start = charIndex[s[end]] + 1;
20             }
21
22             // Update the last seen index of the current character
23             charIndex[s[end]] = end;
24
25             int currentLength = end - start + 1; // Calculate length of the current substring
26             maxLength = max(maxLength, currentLength); // Update the maximum length
27         }
28
29         return maxLength; // Return the result
30     }
31 };
32
33 int main() {
34     // Given string
35     string s = "abcabcbb";
36     // Instantiate 'Solution' object
37     Solution solution;
38     // Find substring of maximum length
39     int result = solution.lengthOfLongestSubstring(s);
40     // Print result
41     cout << "Length of the longest substring without repeating characters: " << result << endl;
42     return 0;
43 }

```

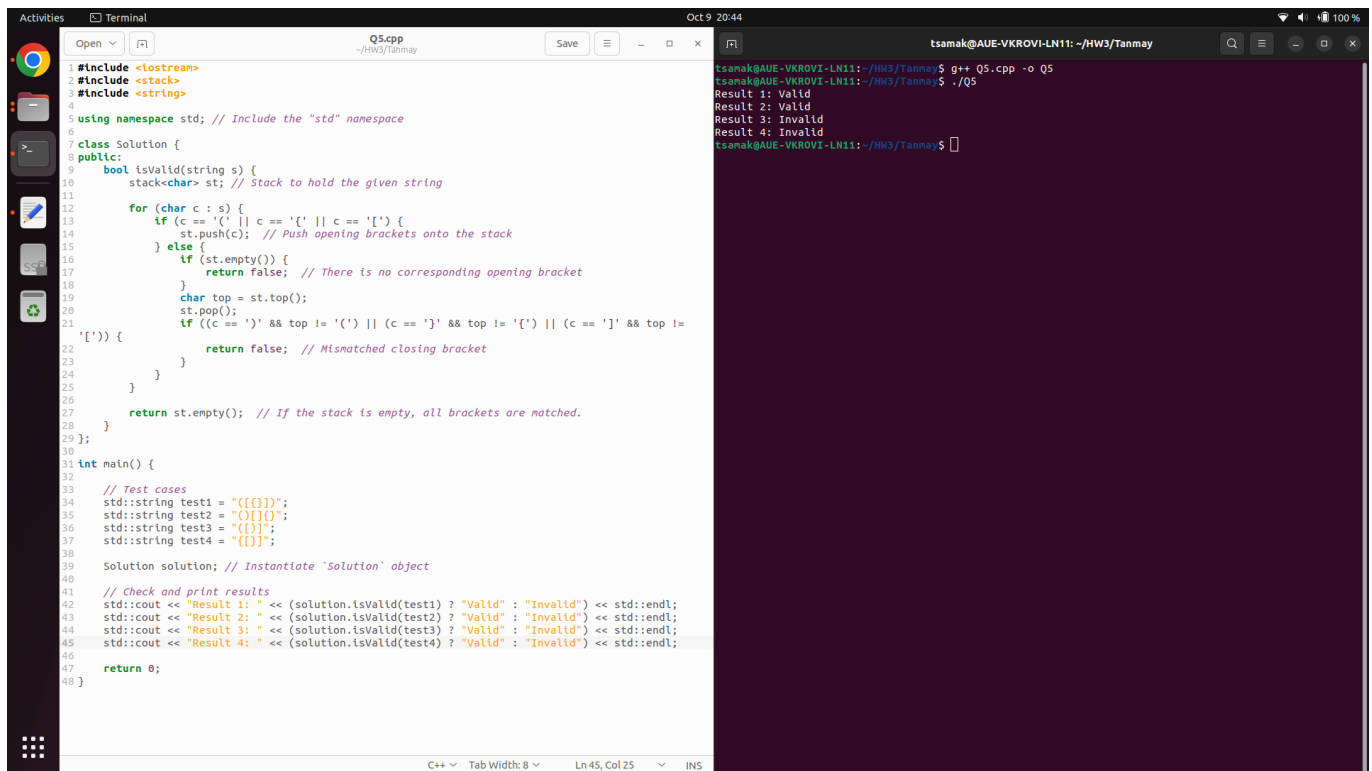
```

tsamak@AUE-VKROVI-LN11:~/HW3/Tanmay$ g++ Q4.cpp -o Q4
tsamak@AUE-VKROVI-LN11:~/HW3/Tanmay$ ./Q4
Length of the longest substring without repeating characters: 3
tsamak@AUE-VKROVI-LN11:~/HW3/Tanmay$

```

The time complexity of this algorithm is  $O(n)$ , where  $n$  is the length of the input string.

5. A sample execution for Q5 is depicted below:



```

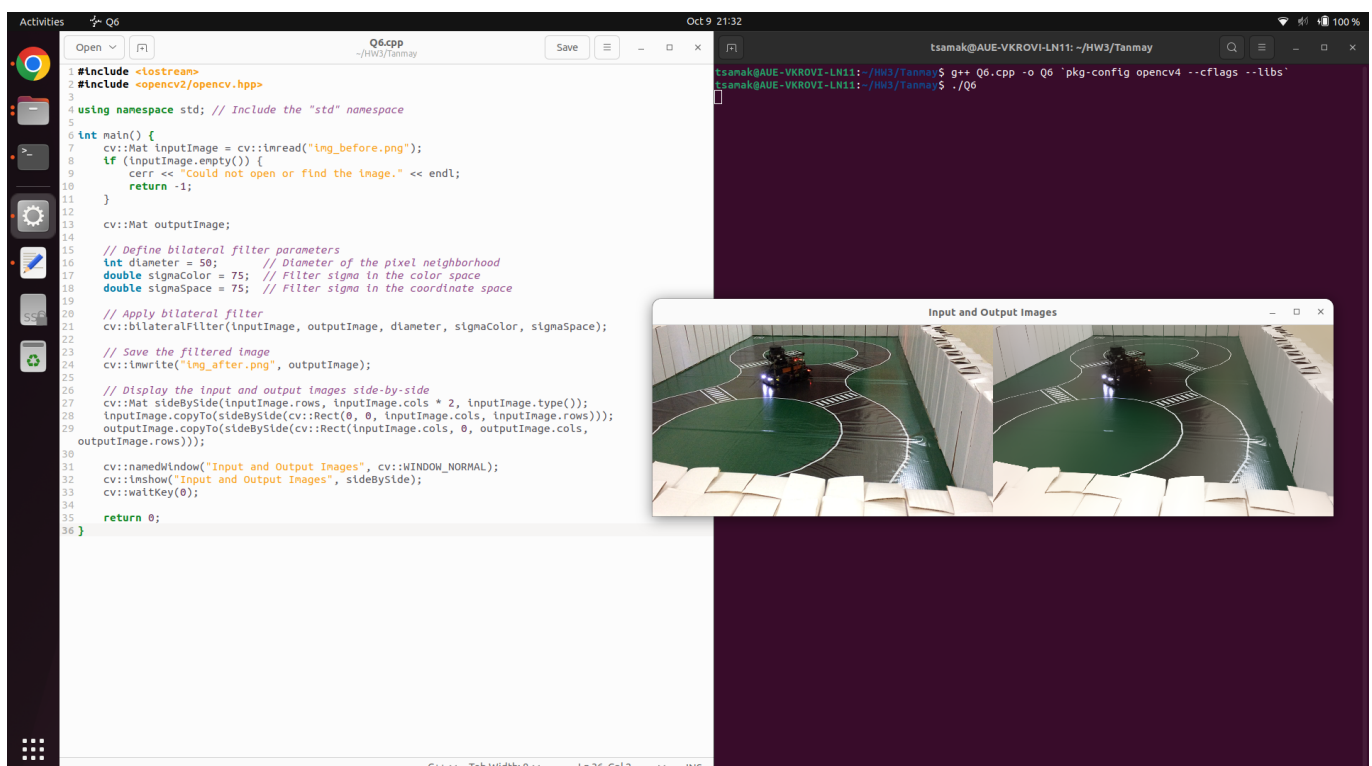
1 #include <iostream>
2 #include <stack>
3 #include <string>
4
5 using namespace std; // Include the "std" namespace
6
7 class Solution {
8 public:
9     bool isValid(string s) {
10         stack<char> st; // Stack to hold the given string
11
12         for (char c : s) {
13             if (c == '(' || c == '[' || c == '{') {
14                 st.push(c); // Push opening brackets onto the stack
15             } else {
16                 if (st.empty()) {
17                     return false; // There is no corresponding opening bracket
18                 }
19                 char top = st.top();
20                 st.pop();
21                 if ((c == ')' && top != '(') || (c == ']' && top != '[') || (c == '}' && top != '{')) {
22                     return false; // Mismatched closing bracket
23                 }
24             }
25         }
26         return st.empty(); // If the stack is empty, all brackets are matched.
27     }
28 };
29
30 int main() {
31     // Test cases
32     std::string test1 = "([{}])";
33     std::string test2 = "([)]";
34     std::string test3 = "([])";
35     std::string test4 = "([{}])";
36
37     Solution solution; // Instantiate 'Solution' object
38
39     // Check and print results
40     std::cout << "Result 1: " << (solution.isValid(test1) ? "Valid" : "Invalid") << std::endl;
41     std::cout << "Result 2: " << (solution.isValid(test2) ? "Valid" : "Invalid") << std::endl;
42     std::cout << "Result 3: " << (solution.isValid(test3) ? "Valid" : "Invalid") << std::endl;
43     std::cout << "Result 4: " << (solution.isValid(test4) ? "Valid" : "Invalid") << std::endl;
44
45     return 0;
46 }
  
```

```

tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ g++ Q5.cpp -o Q5
tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ ./Q5
Result 1: Valid
Result 2: Invalid
Result 3: Valid
Result 4: Invalid
tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$
  
```

The time complexity of this algorithm is  $O(n)$ , where  $n$  is the length of the input string, as it iterates through the string once. The space complexity is also  $O(n)$  because in the worst case, the stack can contain all opening brackets.

6. A sample execution for Q6 is depicted below:



```

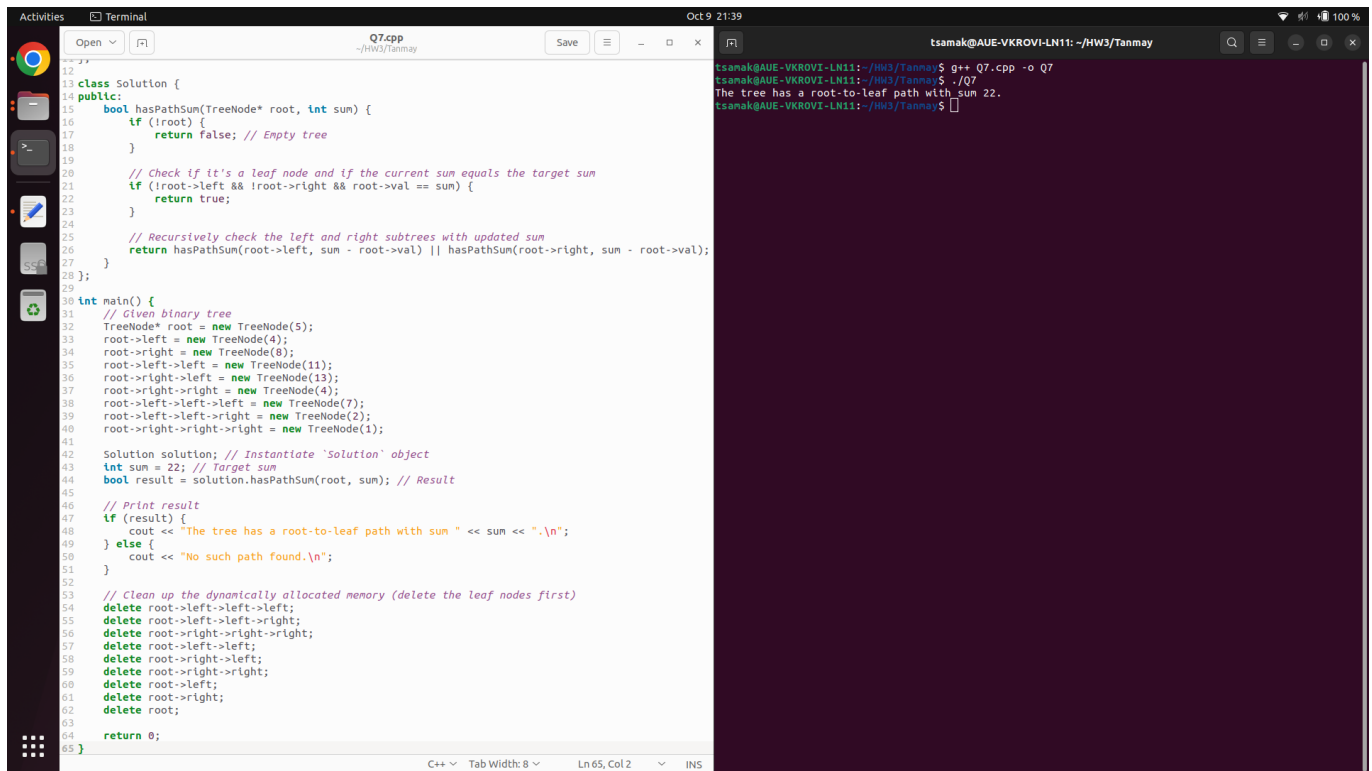
1 #include <iostream>
2 #include <opencv2/opencv.hpp>
3
4 using namespace std; // Include the "std" namespace
5
6 int main() {
7     cv::Mat inputImage = cv::imread("img_before.png");
8     if (inputImage.empty()) {
9         cerr << "Could not open or find the image." << endl;
10        return -1;
11    }
12
13    cv::Mat outputImage;
14
15    // Define bilateral filter parameters
16    int diameter = 50; // Diameter of the pixel neighborhood
17    double sigmaColor = 75; // Filter sigma in the color space
18    double sigmaSpace = 75; // Filter sigma in the coordinate space
19
20    // Apply bilateral filter
21    cv::bilateralFilter(inputImage, outputImage, diameter, sigmaColor, sigmaSpace);
22
23    // Save the filtered image
24    cv::imwrite("img_after.png", outputImage);
25
26    // Display the input and output images side-by-side
27    cv::Mat sideBySide(cv::Mat_<_C, CV_MAKETYPE(inputImage.type()) * 2, inputImage.rows, inputImage.cols * 2);
28    inputImage.copyTo(sideBySide(cv::Rect(0, 0, inputImage.cols, inputImage.rows)));
29    outputImage.copyTo(sideBySide(cv::Rect(inputImage.cols, 0, outputImage.cols, outputImage.rows)));
30
31    cv::namedWindow("Input and Output Images", cv::WINDOW_NORMAL);
32    cv::imshow("Input and Output Images", sideBySide);
33    cv::waitKey(0);
34
35    return 0;
36 }
  
```

```

tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ g++ Q6.cpp -o Q6 `pkg-config opencv4 --cflags --libs`
tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ ./Q6
  
```

The execution window titled "Input and Output Images" shows two side-by-side photographs of a basketball court. The left image is the original input, and the right image is the result of applying a bilateral filter, which appears slightly smoother.

7. A sample execution for Q7 is depicted below:



```

13 class Solution {
14 public:
15     bool hasPathSum(TreeNode* root, int sum) {
16         if (!root) {
17             return false; // Empty tree
18         }
19
20         // Check if it's a leaf node and if the current sum equals the target sum
21         if (!root->left && !root->right && root->val == sum) {
22             return true;
23         }
24
25         // Recursively check the left and right subtrees with updated sum
26         return hasPathSum(root->left, sum - root->val) || hasPathSum(root->right, sum - root->val);
27     }
28 };
29
30 int main() {
31     // Given binary tree
32     TreeNode* root = new TreeNode(5);
33     root->left = new TreeNode(4);
34     root->right = new TreeNode(8);
35     root->left->left = new TreeNode(11);
36     root->right->left = new TreeNode(13);
37     root->right->right = new TreeNode(4);
38     root->left->left->left = new TreeNode(7);
39     root->left->left->right = new TreeNode(2);
40     root->right->right->right = new TreeNode(1);
41
42     Solution solution; // Instantiate 'Solution' object
43     int sum = 22; // Target sum
44     bool result = solution.hasPathSum(root, sum); // Result
45
46     // Print result
47     if (result) {
48         cout << "The tree has a root-to-leaf path with sum " << sum << ".\n";
49     } else {
50         cout << "No such path found.\n";
51     }
52
53     // Clean up the dynamically allocated memory (delete the leaf nodes first)
54     delete root->left->left->left;
55     delete root->left->left->right;
56     delete root->right->right->right;
57     delete root->left->left;
58     delete root->right->left;
59     delete root->right->right;
60     delete root->left;
61     delete root->right;
62     delete root;
63
64     return 0;
65 }

```

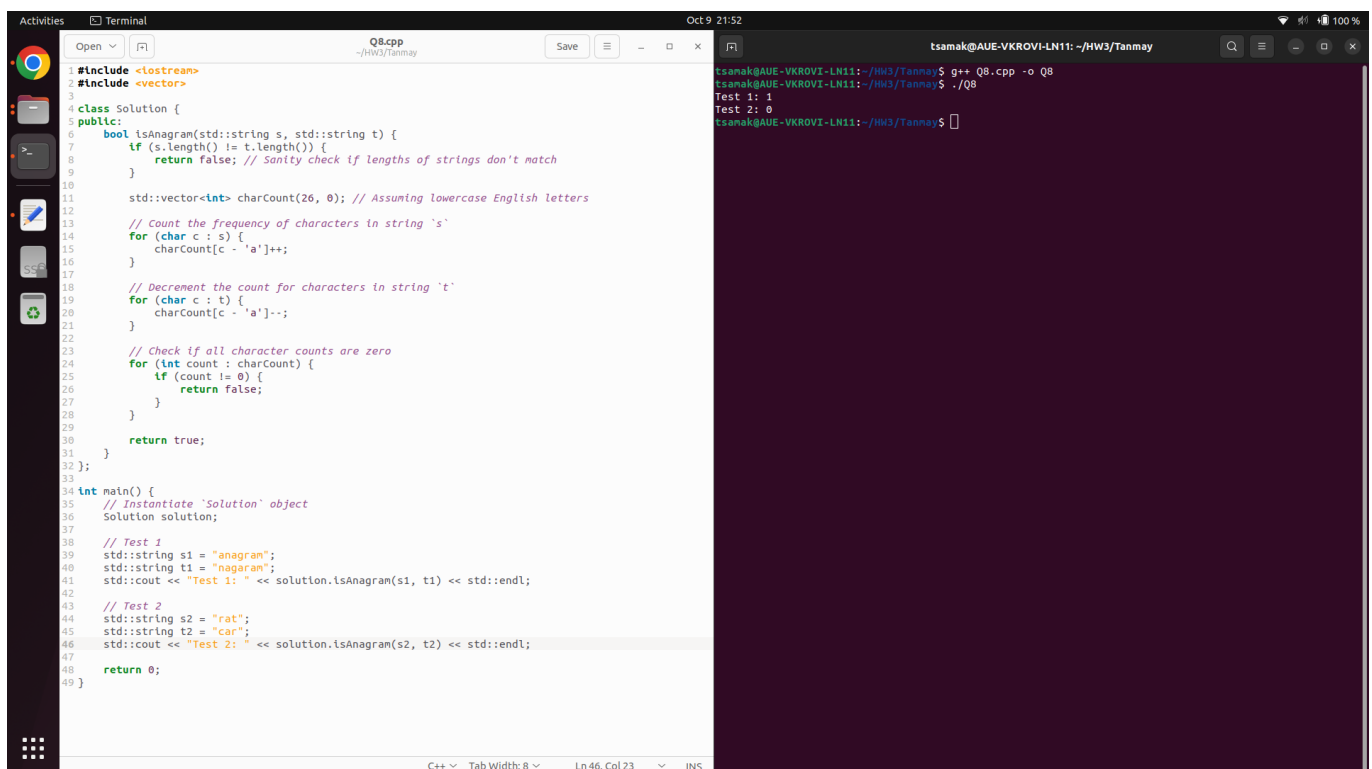
```

tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ g++ Q7.cpp -o Q7
tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ ./Q7
The tree has a root-to-leaf path with sum 22.
tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$

```

The time complexity of this algorithm is  $O(n)$ , where  $n$  is the number of nodes in the binary tree. This is because we perform a depth-first search (DFS) traversal of the binary tree, visiting each node exactly once. At each node, we perform constant-time operations to check if it's a leaf node and if the current sum equals the target sum. In the worst case, we visit all  $n$  nodes in the binary tree to explore all possible paths.

8. A sample execution for Q8 is depicted below:



```

1 #include <iostream>
2 #include <vector>
3
4 class Solution {
5 public:
6     bool isAnagram(std::string s, std::string t) {
7         if (s.length() != t.length()) {
8             return false; // Sanity check if lengths of strings don't match
9         }
10
11         std::vector<int> charCount(26, 0); // Assuming lowercase English letters
12
13         // Count the frequency of characters in string 's'
14         for (char c : s) {
15             charCount[c - 'a']++;
16         }
17
18         // Decrement the count for characters in string 't'
19         for (char c : t) {
20             charCount[c - 'a']--;
21         }
22
23         // Check if all character counts are zero
24         for (int count : charCount) {
25             if (count != 0) {
26                 return false;
27             }
28         }
29
30         return true;
31     }
32 };
33
34 int main() {
35     // Instantiate 'Solution' object
36     Solution solution;
37
38     // Test 1
39     std::string s1 = "anagram";
40     std::string t1 = "nagaran";
41     std::cout << "Test 1: " << solution.isAnagram(s1, t1) << std::endl;
42
43     // Test 2
44     std::string s2 = "rat";
45     std::string t2 = "car";
46     std::cout << "Test 2: " << solution.isAnagram(s2, t2) << std::endl;
47
48     return 0;
49 }

```

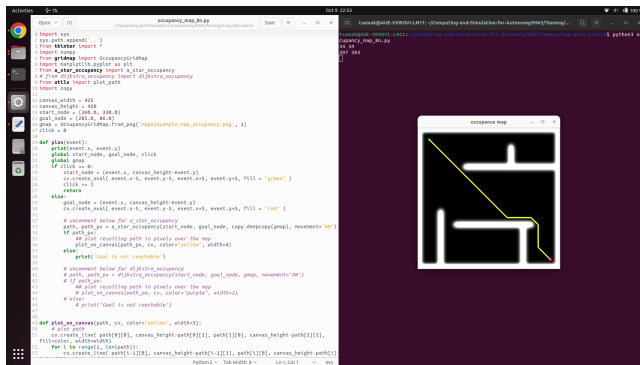
```

tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ g++ Q8.cpp -o Q8
tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$ ./Q8
Test 1: 1
Test 2: 0
tsamak@AUE-VKROVI-LN11: ~/HW3/Tanmay$

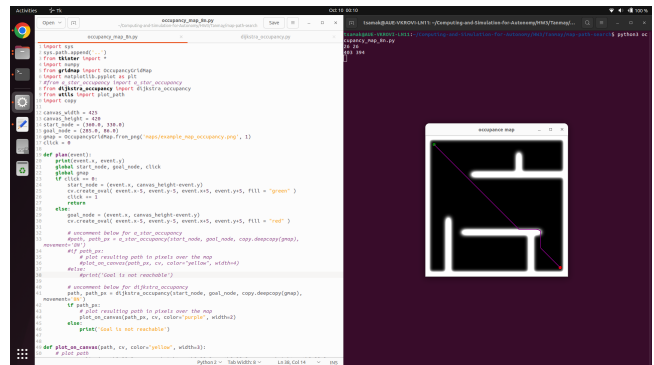
```

The time complexity of this algorithm is  $O(n)$ , where  $n$  is the length of the input strings, as it makes a single pass through both strings.

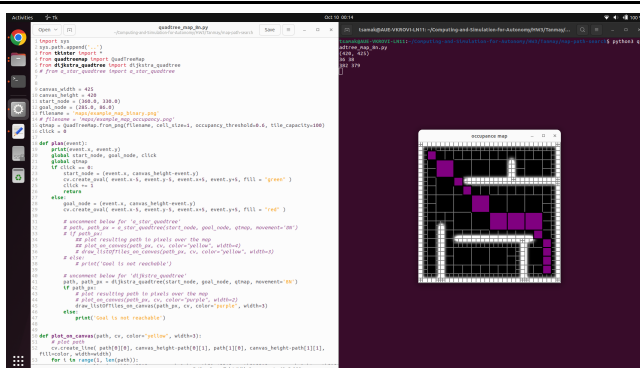
9. Map path search:



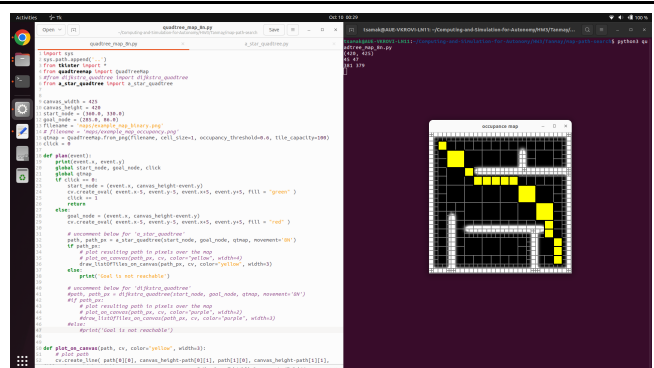
A\* Occupancy Grid



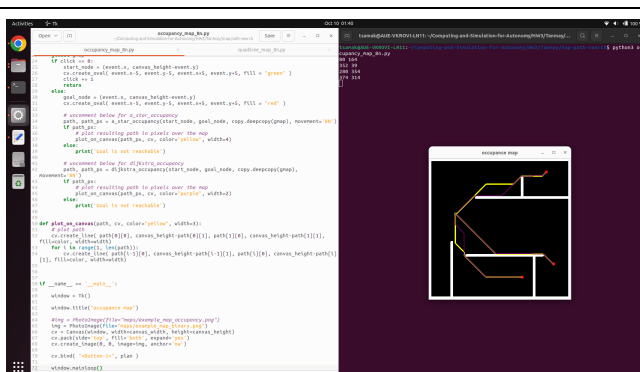
Dijkstra Occupancy Grid



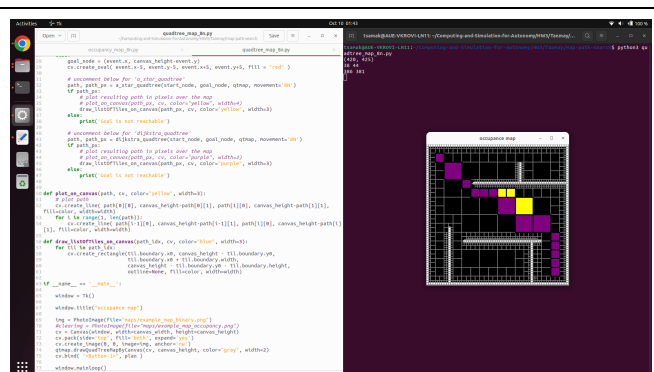
A\* Quadtree



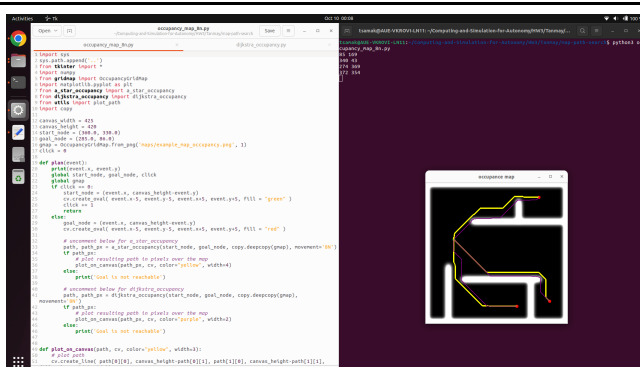
Dijkstra Quadtree



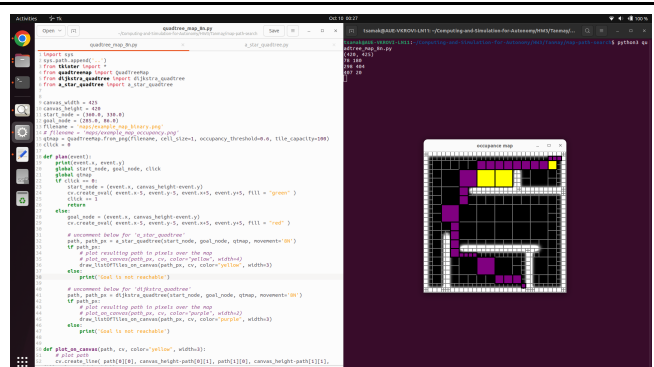
A\* Resolution



Dijkstra Resolution



A\* and Dijkstra Occupancy Grid



A\* and Dijkstra Occupancy Grid

The time complexity of pathfinding algorithms like A\* and Dijkstra's algorithm can vary depending on the data structures and heuristics used, as well as the specifics of the problem and the map representation. Here, I'll discuss the time complexity for both occupancy grid-based and quadtree-based implementations of A\* and Dijkstra's algorithm.



- **Occupancy Grid-Based Algorithms:**

- **Dijkstra's Algorithm (Grid-Based):**

- **Time Complexity:**  $O((V + E) * \log(V))$ , where  $V$  is the number of grid cells (vertices) and  $E$  is the number of grid cell connections (edges).
    - **Explanation:** Dijkstra's algorithm explores vertices in increasing order of their distance from the start node. The use of a priority queue (typically implemented as a binary heap) for selecting the next node to expand ensures efficient selection of nodes with the lowest distance.

- **A\* Algorithm (Grid-Based):**

- **Time Complexity:**  $O((V + E) * \log(V))$  with a good heuristic.
    - **Explanation:** A\* combines Dijkstra's algorithm with an admissible heuristic that guides the search towards the goal. In practice, A\* often performs better than Dijkstra's algorithm because it explores fewer nodes due to the heuristic's guidance. The time complexity is similar to Dijkstra's but with potentially fewer node expansions.

- **Quadtree-Based Algorithms:**

- **Dijkstra's Algorithm (Quadtree-Based):**

- **Time Complexity:**  $O((V + E) * \log(V))$ , where  $V$  is the number of tiles (vertices) in the quadtree and  $E$  is the number of connections between tiles.
    - **Explanation:** The time complexity for Dijkstra's algorithm remains the same as in the grid-based case when applied to a quadtree-based map representation. The data structures used for managing nodes (e.g., priority queues) are the same.

- **A\* Algorithm (Quadtree-Based):**

- **Time Complexity:**  $O((V + E) * \log(V))$  with a good heuristic.
    - **Explanation:** A\* with a quadtree-based map representation has a time complexity similar to that of Dijkstra's algorithm. The use of a good heuristic can reduce the number of node expansions, making A\* more efficient than Dijkstra's in practice.

In both occupancy grid-based and quadtree-based implementations, the primary factor influencing time complexity is the number of vertices ( $V$ ) and edges ( $E$ ) in the graph or map representation. The choice of data structures for managing nodes and the efficiency of the heuristic function also play a role. When a good heuristic is used and the search space is reduced, A\* tends to outperform Dijkstra's algorithm in terms of speed.

It's important to note that these time complexities are theoretical upper bounds and can vary in practice based on map characteristics, the quality of the heuristic, and implementation details.