Name: Vivek Vikram Pundkar

Roll no: 77

GR no: 11910860

Division: C

Batch: 3

--------------------------------------------------------------------------------------------------------------

**N-Queens Problem**

**Introduction:**

The N Queen Problem is one of the best problem used to teach backtracking and of course recursion. Backtracking is a general algorithm which finds all complete solutions to a problem by building over partial solutions. In this process, the problem might reach to a partial solution which may not result into a complete solution. Such partial solutions are effectively rejected by the backtracking algorithm.

The classic case of N Queen problem

In this problem, you are given an empty chess board of dimensions N \* N. The task is to safely place N queens, one in each row such that they cause no harm to each other. The basic idea is mentioned below:

* Start from the first row
* Select a column and if it is empty, well within the limits of the board and in no immediate danger, then place the queen in the cell
* Proceed to the next row
* Two conditions might occur,
* all the rows are safely filled
* somewhere down the line, one of the queens cant be placed in a row. Here we need to backtrack, because a decision was wrongly made in the past

**Analysis for N Queen Problem**

**Space complexity**

For this algorithm it is O(N). The algorithm uses an auxiliary array of length N to store just N positions.

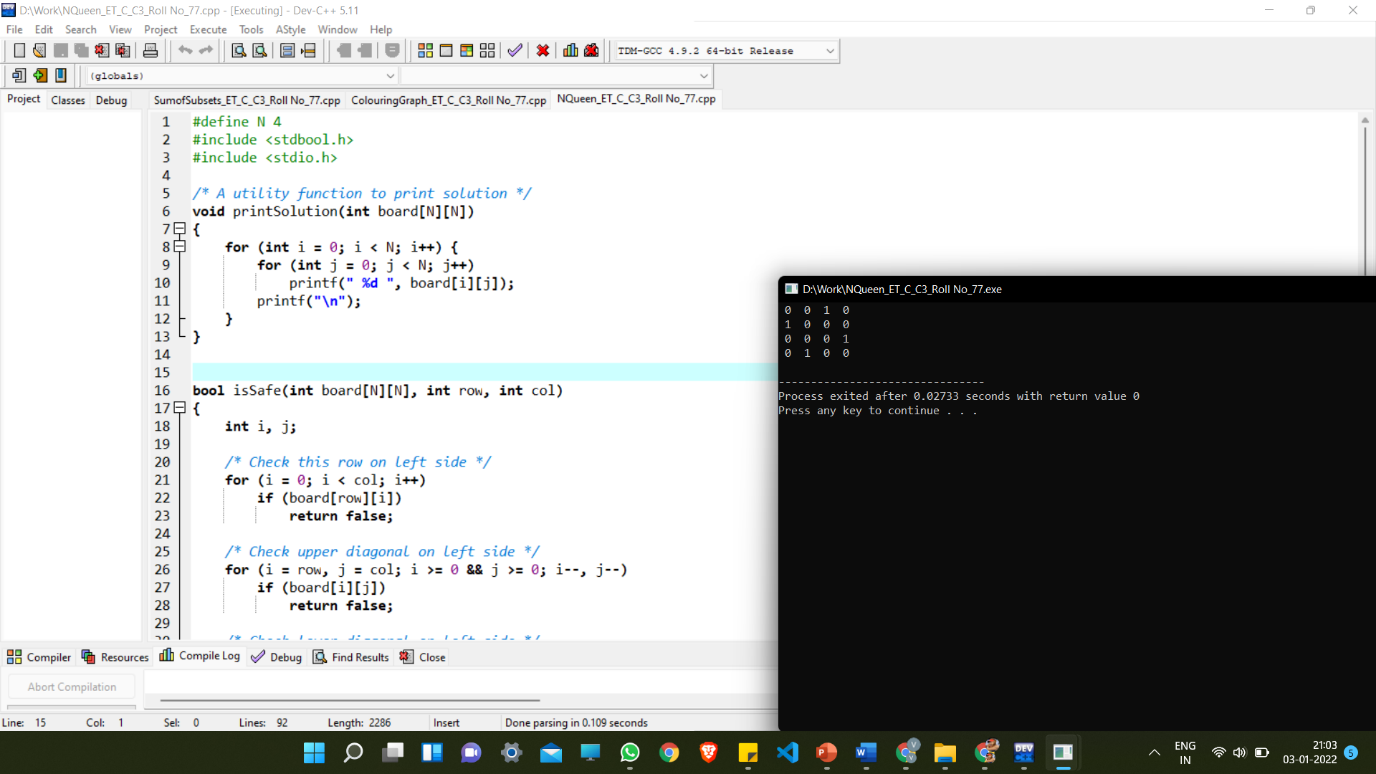
**Time complexity**

* The isSafe method takes O(N) time as it iterates through our array every time.
* For each invocation of the placeQueen method, there is a loop which runs for O(N) time.
* In each iteration of this loop, there is isSafe invocation which is O(N) and a recursive call with a smaller argument.

If we add all this up and define the run time as T(N). Then T(N) = O(N2) + N\*T(N-1). If you draw a recursion tree using this recurrence, the final term will be something like n3+ n!O(1). By the definition of Big O, this can be reduced to O(n!) running time.

Let me know in comments if you are not able to derive the n! from the recurrence, I will try to draw the recursion tree.

**OUTPUT:**



**Sum of Subset**

**Algorithm:**

* Start with an empty set
* Add the next element from the list to the set
* If the subset is having sum M, then stop with that subset as solution.
* If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
* If the subset is feasible (sum of seubset < M) then go to step 2.
* If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution

**Example:** For following list of integers : {1,3,2 }We want to find if there is a subset with sum 3.

Note that there are two such subsets **{1, 2}** and **{3}**. We will follow our backtracking approach.

Consider our empty set **{}**

We add 1 to it **{1}** (sum = 1, 1 < 3)

We add 2 to it **{1, 3}** (sum = 3, 3 == 3, found)

We remove 3 from it **{1}** (sum = 1, 1 < 3)

We add 2 to it **{1, 2}** (sum = 3, 3 == 3, found)

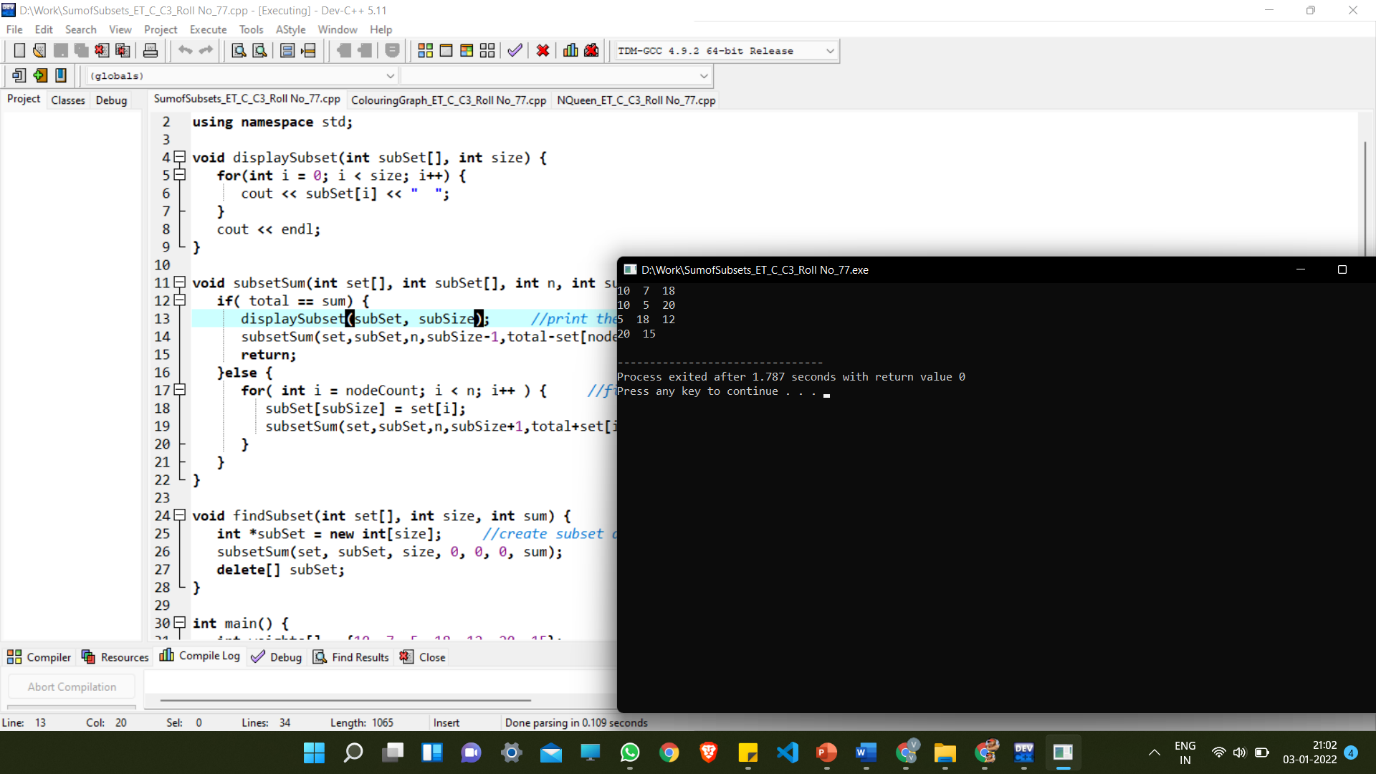
We remove 2 and see that all elements have been considered.

**Analysis:**

**Time Complexity:**

Worst case time complexity: **Θ(2^n)**

**Space Complexity: O(1)**



**Graph Colouring**

**Introduction:**

Graph coloring is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color. The objective is to minimize the number of colors while coloring a graph. The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NP Complete problem.

Method to Color a Graph

The steps required to color a graph G with n number of vertices are as follows −

Step 1 − Arrange the vertices of the graph in some order.

Step 2 − Choose the first vertex and color it with the first color.

Step 3 − Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

Input:

graph = {0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

Output:

Solution Exists:

Following are the assigned colors

1 2 3 2

Explanation: By coloring the vertices with following colors, adjacent vertices does not have same colors

**Analysis:**

**Time Complexity:** O(m^V).

There are total O(m^V) combination of colors. So time complexity is O(m^V). The upperbound time complexity remains the same but the average time taken will be less.

**Space Complexity:** O(V).

Recursive Stack of graphColoring(…) function will require O(V) space.

