PROGRAMMING ASSIGNMENT 3 REPORT

Name: Bhavesh Khatri
Roll No: M23CSE011

Guided By: Dr. Pallavi Jain
Assistant Professor (CSE Department)

<u>Question 1:</u> Minimum Registers required for program variables

Bikash is working on a crucial aspect of compiler optimization: register allocation. In compiler optimization, register allocation involves efficiently assigning many program variables to a limited number of CPU registers.
Bikash needs your help determining the minimum number of registers required to allocate these program variables, considering their dependencies and conflicts.

Input Format
N = 5
E = 8
U = 1 1 2 2 3 3 4 5
V = 2 3 1 4 1 5 2 3

Constraints
An integer N (1 <= N <= 1000) represents the number of program variables.

An integer E (1 <= E <= 5000) represents the dependencies or conflicts between these variables.

U -> A list of source program variables.
V -> A list of destination program variables.

Dependency -> U to V
Output -> number of minimum registers required.
N lines follow, each containing the program variables with dependencies or conflicts with the corresponding variable.

Your program should find and return the minimum number of registers needed for an optimized allocation of these program variables, considering their dependencies and conflicts.

Sample Input
5
8
1 1 2 2 3 3 4 5
2 3 1 4 1 5 2 3
Sample Output
2

Solution:

Discussion on the function working:-

Function isSafe:

This function checks if it's safe to assign a specific color to a node. It takes three arguments: node (the node to be colored), col (the color to be assigned), and N (the total number of nodes).
It iterates through all nodes (i) and checks if there is an edge between node and i. If there is, it checks if the color of node i is the same as col. If yes, it means it's not safe to assign col to node.

Function assignColors:

This is a recursive function that tries to assign colors to nodes.
It takes two arguments: node (the current node) and N (the total number of nodes).
If the node is greater than N, all nodes have been colored. It then finds the maximum color used and returns it.
Otherwise, it iterates through colors (c) and checks if it's safe to assign c to the current node. If yes, it assigns the color and recursively calls the function for the next node. If the recursive call returns a positive value (indicating successful coloring), it returns that value.

Function findMinimumRegisters:

This function is responsible for setting up the adjacency matrix and calling assignColors.
It takes N (number of nodes), E (number of edges), and two arrays, edgesU and edgesV (representing the edges).
It populates the adjacency matrix based on the provided edges.
It then calls assignColors starting from the first node.

Main Function:

It reads the input values for N and E.
It reads the edges and adjusts them to 1-based indexing.
It calls findMinimumRegisters and prints the result.

Time Complexity Analysis:

isSafe Function:

This function has a time complexity of O(N), where N is the total number of nodes. This is because it iterates through all nodes.

assignColors Function:

In the worst case, this function is called recursively for each node. Therefore, the time complexity is O(N^N), which is exponential.

findMinimumRegisters Function:

This function sets up the adjacency matrix, which takes O(E) time, where E is the number of edges.
It then calls assignColors, which can take up to O(N^N) time.
So, the overall time complexity is O(E + N^N).

Space Complexity Analysis:

Adjacency Matrix (adjacency[MAX_NODES][MAX_NODES]):

This 2D array is used to represent the adjacency matrix of the graph.
The space required for this matrix is O(N^2) because it has N rows and N columns.
In terms of space complexity, this is the most significant contributor.

Color Array (color[MAX_NODES]):

This array is used to store the color assigned to each node.
It has a space complexity of O(N) because it stores one color value for each of the N nodes.

Local Variables:

The isSafe and assignColors functions use some local variables, but these contribute little to the overall space complexity. These variables have constant space requirements.

Function Call Stack:

The assignColors function is implemented recursively. Each recursive call to this function consumes some space on the call stack.
In the worst case, where all nodes need to be colored, the maximum depth of the call stack can be N. Therefore, the space complexity due to the call stack is O(N).
Overall, the space complexity of the provided code can be summarized as follows:

Space Complexity: O(N^2 + N) = O(N^2) (dominated by the adjacency matrix)

<u>Question 2:</u> Question 1 with other than the table data structure

Bikash is working on a crucial aspect of compiler optimization: register allocation. In compiler optimization, register allocation involves efficiently assigning many program variables to a limited number of CPU registers.
Bikash needs your help determining the minimum number of registers required to allocate these program variables, considering their dependencies and conflicts.

Input Format
N = 5
E = 8
U = 1 1 2 2 3 3 4 5
V = 2 3 1 4 1 5 2 3

Constraints
An integer N (1 <= N <= 1000) represents the number of program variables.

An integer E (1 <= E <= 5000) represents the dependencies or conflicts between these variables.

U -> A list of source program variables.
V -> A list of destination program variables.

Dependency -> U to V

Output -> number of minimum registers required.
N lines follow, each containing the program variables with dependencies or conflicts with the corresponding variable.

Your program should find and return the minimum number of registers needed for an optimized allocation of these program variables, considering their dependencies and conflicts.

Sample Input
5
8
1 1 2 2 3 3 4 5
2 3 1 4 1 5 2 3
Sample Output
2

Solution:

Code Explanation:

The code can be divided into two main functions: Coloring and main. We will analyze each of them separately.

Coloring Function:

This function performs the actual graph coloring using a greedy algorithm.
It initializes an array of colors to store the color assigned to each vertex and another array available to keep track of available colors for each vertex.
It starts by assigning color 0 to the first vertex and iteratively colors the remaining vertices.
For each vertex, it checks the adjacent vertices and marks the colors already used.
It then assigns the lowest available color to the current vertex, ensuring that adjacent vertices have different colors.
Finally, it calculates the number of colors used and returns it.

main Function:

The main function serves as the entry point of the program.
It reads the number of vertices (N) and edges (E) from the input.
It initializes an adjacency matrix adjacencyList to represent the graph and sets all elements to 0.
It reads the edges from the input and populates the adjacency matrix accordingly.
It calls the Coloring function to color the graph.
It prints the minimum number of colors required to color the graph.

Time Complexity Analysis:

Let's analyze the time complexity of each part of the code:
Initialization of colors and available arrays: O(N)

Coloring Loop (nested):

1. The outer loop runs N times (for each vertex).
2. The inner loops iterate over adjacent vertices, which can be at most N.
3. Overall, the inner loops take O(N^2) in the worst case.

Finding an available color: O(N)

Calculating the number of colors: O(N)

Total Time Complexity of Coloring function: O(N^3)

main Function:

Initializing the adjacency matrix: O(N^2)
Reading edges: O(E)
Calling Coloring function: O(N^3)
Printing the result: O(1)
Total Time Complexity of the main function: O(N^3 + E)

Space Complexity Analysis:

Input Variables (N, E, U, V):

These variables store information about the graph (number of vertices, edges, and edge connections). They have a constant space requirement, so their contribution is considered O(1).

Adjacency Matrix (adjacencyList):

This is a 2D array used to represent the graph. It consumes space proportional to the square of the number of vertices (N^2) since it needs to store connections between all pairs of vertices.
Therefore, the space complexity of the adjacency matrix is O(N^2).

Additional Arrays (colors, available):

These arrays are used in the Coloring function to keep track of the colors assigned to vertices and the availability of colors. They have a space requirement proportional to the number of vertices (N).
Thus, the space complexity of these arrays is O(N).

Local Variables (loop counters, temporary variables):

These variables are used for loop control and temporary storage. They have a constant space requirement, so their contribution is considered O(1).

Total Space Complexity:

Adding up the space requirements of all components, we get a total space complexity of O(N^2 + N).

References

1. Skiena, S. S. (2008). The Algorithm Design Manual. Springer.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
3. Tardos, É., & Kleinberg, J. (2005). Algorithm Design. Addison-Wesley.
4. GeeksforGeeks
5. HackerRank