Algorithm

Homework 2

Name: Shubham Balasaheb Daule **Id: 12141550**

Question 2

Note: As type of graph is not mentioned in question, we will assume it to be undirected graph.

In order to solve the given question, we will add a new property to every node for DFS. We will call this property pcolor. For the first node we will set color to red and we pass the color other then the current (i.e. blue if red is input and vise versa) while calling DFS. Whenever we encounter a back edge, if the color of this node is same, we can say that given graph is not bipartite, but if we do not encounter any such edge, we can say that the graph is bipartite.

```
DFS(G)
bipartiate = True
for each vertex u in G.V
    u.color = WHITE
    u.pi = NIL
    u.pcolor = NIL
endfor
time = 0
for each vertex u in G.V
    if u.color == WHITE
        u.pcolor = RED
        DFS-VISIT(G, u)
    endif
endfor
return bipartiate

DFS-VISIT(G, u)
time = time +1
u.d = time
u.color = GRAY
for each v in G.Adj[u]
    if v.color == WHITE
        v.pi = u
        if u.pcolor == RED
            v.pcolor = BLUE
        else
            v.pcolor = RED
        endif
        DFS-VISIT(G.v)
    else    // Back edge
        if (v.pcolor == u.pcolor)
            bipartiate = False
        endif
    endif
endfor
u.color = BLACK
time = time + 1
u.f = time
```

Question 3

Define mother vertex as a vertex from which, path to every node exists. Suppose a graph has a mother vertex. If we assume that DFS does not end at the mother vertex (u), then it should end at some other vertex v. Now as DFS terminates at v,

$$v.f > u.f$$

We now have two cases, either

$$u.d > v.d$$

There should be a path from v to u. So we can get a path from v to every other node w, as we have a path from v to u and u to w. So v is also a mother vertex.

$$u.d < v.d$$

In this case, we firstly finish visiting u and all its children before visiting v. So there are two cases, firstly, both u and v can be part of different trees in DFS forest. But this means that there are nodes which cannot be reached from u as we visited it earlier. Now the only possibility that remains is that u and v are on different branches of same DFS tree. So, $v.\pi$ is not Nil. So, there will be a node ($v.\pi$) for which finish time is more then v. So v is not the last node finishing which is against our assumption that v is the last node finishing. So we conclude that the given statement is true.

Question 4

```
Strongly-Connected-Components(G)    // Returns array of trees
    DFS(G)
    GT = Transpose(G)
    for each vertex u in GT.V
        u.color = WHITE
        u.pi = NIL
        u.pcolor = NIL
    endfor
    time = 0
    trees = {}
    for each vertex u in sort(GT.V, key=(u)=>u.f, descending = true)
        if u.color == WHITE
```

```
            curr_tree = tree.empty_new()
            trees.add(curr_tree)
            // Modified version of DFS-VISIT that will add every node
            // to curr_tree as it is visited and the edge from its parent
            // verted to it
            DFS-VISIT(GT, u)
        endif
    endfor
    return trees

Transpose(G)
    GT.V = G.V
    E = {}
    for each edge e in G.E
        // Add vertex in reverse direction
        E.add((e.v, e.u))
    return GT
```

Question 5

a)

Let us assume that root, r of DFS tree has only single child, c. Now if we remove root, the tree still remains as a single component because all other nodes of tree have node c as ancestor. So they are all connected.
If DFS tree has multiple children, say c1 and c2, and we remove the root node, then no link exists between c1 and c2. Because if such link would have existed, then c2 would have been descendent of c1.

b)

i. If there is a child, u, without backedge from descendent of child to ancestor of v. Now if we remove v from the graph, the subtree starting from u will get completely separated from the original tree. That is, the graph has an additional connected component.

ii. If there is no child without backedge from descendent of child to ancestor of v and we remove vertex v, for every subtree starting from child of v, there is an edge (the said backedge) that acts as a link between this subtree and the original tree. So no component is completely separated from the parent tree. Hence v is not an articulation point.

c)

We will use a modified version of DFS in which we will keep track of an additional property of a vertex say e. Whenever we are about to color vertex black, we will check which of its child has least e and set value of e equal to this value. If we encounter a backedge, we will update value of e to match the discovering time of the other end of this back edge.
After running DFS we will use statements proved in (a) and (b) subproblems to calculate the

articulation points. For the statement given in subproblem b, we have to check if for any vertex u $\exists$ a child v such that, $v.e \geq u.d$. If such v exists, we will say that u is articulation point.

Question 6

Let us represent these rebels by a directed graph, G. Nodes of this graph denotes rebels. We will add a directed edge from rebel u, to rebel v iff

$$contact(u) == v$$

a)

The given subproblem can alternatively described as:
"Find a node u, s.t, $\forall$ v $\in$ G.V $\exists$ a path from u to v"
This statement is equivalent to one given in question 3. So we can describe algorithm as:
Run DFS on graph G and note the vertex which has highest finish time. Now run DFS starting from this vertex. If we get single tree in DFS forest, then this is the required node. Otherwise, there is no solution.

b)

Here we have to find number of connected components of G. Every individual connected component will have a mother vertex to which we can give the message. So we will convert this graph G to undirected graph, G_2 such that for every edge (u,v) in G, we will add an edge between u and v in G_2. Now we can simply use BFS to calculate number of connected components.

c)

The solution to this subproblem is same as the one for (a). The only difference will be in how we build our graph. We will represent rebels by vertices of directed graph and for every v, we will add edge from v to u $\forall$ u $\in$ R(v).
Time complexity of this algorithm will be same as that of DFS. And as we know time complexity of DFS is $O(|G.V| + |G.E|)$. From our representation of graph we get $|G.V| = numberofrebels, n$. And $|G.E| = m$.
So time complexity of the algorithm will be O(m + n). Since, $m \geq n$, $2m \geq n$ so time complexity of the algorithm will be O(2m) which is equivalent to O(m). That is, this algorithm will have linear time complexity.

Question 7

The algorithm for this question is as follows:

```
int sum[n];
int* SpaceSum(int A[], int n){
    if(sum[n] == -1){
        if(n == 0) sum[n] = A[0]
        else if(n == 1) sum[n] = max(A[1], A[0])
        else sum[n] = max(SpaceSum(A, n-1),
                          SpaceSum(A, n-2) + A[n]);
    }
    return sum[n];
}
int SpaceSumMain(A){
    n = A.length();
    for(i in range(0,n))
        sum[n] = -1;
    return (SpaceSum(A, n-1));
}
```

For the given example, A = [3, 5, 8, 4, 9, 16, 14, 6, 13]
We have n = 9. It will make calls like so:

```
SpaceSum(A, 8): // sum = [-1, -1, -1, -1, -1, -1, -1, -1, -1]
  max(sum[7], sum[6]+13)
  SpaceSum(A, 6): // sum = [-1, -1, -1, -1, -1, -1, -1, -1, -1]
    max(sum[5], sum[4]+14)
    SpaceSum(A, 4): // sum = [-1, -1, -1, -1, -1, -1, -1, -1, -1]
      max(sum[3], sum[2]+9)
      SpaceSum(A, 2): // sum = [-1, -1, -1, -1, -1, -1, -1, -1, -1]
        max(sum[1], sum[0]+9)
        SpaceSum(A, 0): // sum = [-1, -1, -1, -1, -1, -1, -1, -1, -1]
          sum[0] = 3
        SpaceSum(A, 1): // sum = [3, -1, -1, -1, -1, -1, -1, -1, -1]
          sum[1] = max(3, 5) = 5
        sum[2] = 12
      SpaceSum(A, 3): // sum = [3, 5, 12, -1, -1, -1, -1, -1, -1]
        max(sum[1], sum[2]+4)
        sum[3] = 16
      sum[4] = 21
    SpaceSum(A, 5): // sum = [3, 5, 12, 16, 21, -1, -1, -1, -1]
      max(sum[4], sum[3]+16)
      sum[5] = 32
    sum[6] = 35
  SpaceSum(A, 7): // sum = [3, 5, 12, 16, 21, 32, 35, -1, -1]
    max(sum[6], sum[5]+6)
    sum[7] = 38
  sum[8] = 48
```

So the answer will be 48

Question 8

```
// We store max value a student can get before time t such that he is allowed to
oly use first job_index interviews in v[t].

int value[time][job_index]
job jobs[]

int get_value(int till, int job_index){
  if(v[till, job_index] == -1){
    v[till, job_index] = calc_value(till, job_index)
  }
  return v[till, job_index]
}
int calc_value(int till, int job_index){
  // Get the last job finishing before till
  for (int i = job_index; i >= 0; i--){
    job = jobs[i]
    if(job.end_time <= till){
      return(max(
             job.value + get_value(job.start_time - 1, job_index - 1),
             get_value(till, job_index - 1)
        ))
    }
  }
  // If no such job is available, we cannot attend interview
  return 0
}

// Finally calculating which one of the jobs he should select
// Initialized to Null
bool job_included[]
int get_job_list(int t, int j){
  if(value[t, j] == value[t, j-1]){
    job_included[j] = false
    get_job_list(jobs[j].start_time - 1, j - 1)
  }else{
    job_included[j] = true
    get_job_list(t, j - 1)
  }
  return job_included
}
```

Question 9

```
// We define a function that will return us number of ways in which we can pay
// amount value using at most max_curr coins.
int* deno = [] // Array containing all available denominations
int ways[value][deno.length]
int get_ways(int value, int max_curr_index){
  if(value <= 0) return 0;
  if (ways[value, max_curr_index] == -1){
    ways[value, max_curr_index] = calc_ways(value, max_curr_index);
  }
  return ways[value, max_curr_index]
}
```

```
int calc_ways(value, max_curr_index){
  curr_ways = 0
  for(i in range(0, max_curr_index+1)){
    curr_ways += get_ways(value - deno[i], i)
  }
}
```

To run this algorithm, run **calc_ways(value, deno.length)**. In order to avoid counting same change in different order twice, we will fixed one permutation of denominations. That is we always keep lower valued denominations before higher valued ones in our order. Then, we can get ways of making a change by first deciding which coin we are going to use first and then calculating ways in which the remaining change can be paid using coins with value equal or lesser then chosen one.

Question 10

a)

Consider the following example:
0.5, 1, 1.4, 1.6, 2, 2.5
using the greedy algorithm mentioned, it will select (1, 2) to get 1, 1.4, 1.6, 2 first and then 0 to 1 and 2 to 3 to get other values. But the optimal solution to this problem will be selecting (0.5, 1.5) and (1.6, 2.6). So the given greedy algorithm will not work.

b)

A working greedy algorithm to this problem will be as follows: We will find the smallest point and start our first interval from there. Next we will remove the covered points and repeat this step on remaining points till we cover all of them.