Suraj Nepali
BESE 2017 (45)
ADA.

Q1 (a) Pseudocode to reverse a string

String

G	a	n	d	a	k	i
0	1	2	3	4	5	6

start = 0 ↑ ... ↑ end

1) Start
2) Take 2 variables 'start' and 'end'.
3) 'Start' is positioned of on first character.
4) 'End' is positioned on last character
5) String [start] is interchanged with string [end]
6) Increment 'start' and decrement 'end'.
7) If 'start' < 'end' than go to step 5.
8) Stop

```
Algorithm Reverse (String, n)
 // string is an array of size n; string [0: n-1]
 // n is number of character in given string
{
    start := 0;
    end := n-1;
    while (start < end)
    {
         temp := string [start];
         string [start] := string [end];
         string [end] := temp;
```

```
        start := start + 1 ;
        end := end - 1 ;
    }
    return ;
}
```

Q3 Ans

(a)

(b)

Q5
(a) Ans

~~Sumbdot sentence problem~~ is ~~related~~

Q2.
Ans

Maximum Subarray Problem:

In maximum subarray problem we are given an array of positive and negative integers and asked to find the subarray whose element have their largest sum.

Given,

$A = [a_1, a_2, \ldots, a_n]$

To find the ~~indit~~ indices j and k that maximize the sum. $S_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i$

If we use $A[j:k]$ to denote the subarray of A from index j to index k, then the maximum subarray problem is to find the subarray $A[j:k]$ that maximizes the sum of its value.

→ Simply, the maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1, \ldots, n]$ of numbers. Formally, the task is to find indices i & j with $1 \le i \le j \le n$ such that the sum

$\sum_{i=j}^{k} A[i]$ is as large as possible.

Each number in the input array A could be positive, negative or zero.
For example: for the array of values [-2, 1, -3, 4, -1, 2, 1, -5, 4], the contiguous subarray with the largest sum is [4, -1, 2, 1] with sum 6.

Some properties of the problem are

1) If the array contains all non-negative numbers, then the problem is trivial, a maximum subarray is the entire array.

2) If the array contains all non-positive numbers then a solution is any subarray of size 1 containing the maximum value of the array.

3) This problem can be solved using several different algorithms techniques, including
- Brute Force
- Divide and conquer
- Dynamic Programming
  - → using auxillary array
  - → using Kadane's algorithm
- Reduction to the shortest path

$\sum_{i=j}^{k} A[i]$ is as large as possible.

Let's discuss five solutions of the problem
1) Bruteforce approach I : Using 3 nested loops.
2) Bruteforce approach II : Using 2 nested loops.
3 Divide and Conquer approach : Similar to merge sort
4) Dynamic Programming Approach I : Using an auxilliary array
5) Dynamic Programming Approach II : Kadane's Algorithm

1) Brute For approach I : Using nested loop
- Use nested loops to determine all the possible sub array sums & return the maximum among them
- Generate all $\boxed{(i,j) : i <= j}$ pairs and calculate the sum between

Pseudo-code

```
Algorithm maxSubarraySum (A [], n)
// user array A of the n
{
    maxSum := 0
    for i := 0 to n-1 do
    {
        for j := 0 to n-1 do
        {
            sum := 0
            for k := 0 to j do
            {
```

```
                sum = sum + A[k];
            }
            if (sum > maxSum)
                maxSum = sum;
        }
    }
    return maxSum;
}
```

Complexity analysis
Time Complexity : $O(n^3)$
Space Complexity : $O(1)$

2) Brute Force Approach II : Using 2 nested loops.

- Optimized version of above approach
- The idea is to start ~~at~~ all positions in the array and calculate running sum.
- The outer loops pick the beginning element, the inner loop finds the maximum possible sum with the first element picked by the outerloop and compares the maximum with the overall minimum

Algorithm maxSubarraySum (A[], n)
// uses an array A of size n.

```
{
    maxSum := 0;
    for i := 0 to n-1 do
    {
        sum := 0;
        for j := 1 to n-1
        {
            sum := sum + A[j];
            if (sum > maxSum)
                maxSum = sum;
        }
    } return maxSum;
}
```

Complexity Analysis:

Time Complexity $= O(n^2)$

Space Complexity $= O(1)$

3) Divide and Conquer: Similar to merge sort

- Divide the array into two equal parts and then recursively find the maximum subarray sum of the left part and right part.
- The subarray we're looking for can be in any of three places

- on the left part of the array (between 0 & mid)
- or the right part of the array (between mid + 1 and end)
- Somewhere crossing the midpoint

Pseudocode

// INT_MIN specifies that an integer cannot store value below limit

Algorithm maxCrossingSum (A[], l, mid, r)
// uses array A, l represent left side of array, r is
// right side of array and represent the mid value

```
{
    sum := 0 ;
    lsum := INT_MIN ;
    for i := mid to l
    {
        sum := sum + A [i] ;
        if (sum > lsum)
            lsum := sum ;
    }
    sum := 0 ;
    rsum := INT_MIN ;
    for i := mid + 1 to r
    {
        sum := sum + A [i] ;
        if (sum > rsum) {
            rsum := sum ;
        }
    }
```

```
    return (lsum+ rsum);
}

// below limit of INT_MIN is 214.7483648

Algorithm maxSubarraySum (A[], low, high)
// original values would be low=0 & high = n-1
{
    if (low := high)
        return A [low];
    else {
        mid := low+ (high-low) /2 ;
        leftSum = maxSubarraySum (A, low, mid);
        rightSum = maxSubarraySum (A, mid+1, high);
        crossingSum = maxSubarraySum (A, low, mid, high);

        return max( leftSum, rightSum, crossing-
                    Sum);
    }
}
```

Complexity Analysis:

Time Complexity: The recurrence relation formed for this Divide and conquer approach is similar to the recurrence relation of merge sort

$T(n) = 2T(n/2) + O(n) = O(n\log n)$

Space Complexity = $O(\log n)$

4) Dynamic Programming approach I : Using an auxilliary array.

- Store the maximum subarray sum ending at particular index in an auxilliary array and then traverse the auxilliary array to find the maximum subarray sum.

Pseudo Code

```
Algorithm maxSubarraySum (A[], n)
{
  // uses array A of size n
   maxEnding_here [0]= A[0]
   for i:=1 to n-1 do
   {
      if (A [i] + maxEnding_here [i-1]>0)
         maxEnding_here [i]= A [i] + maxEnding_here [i-1];
      else
         maxEnding_here [i]= A[i];
   }
   ans := 0;
   for i:= 0 to n-1 do
       ans := max (ans, maxEnding_here [i]);
   return ans;
}
```

Time Complexity : Traversing array A + Traversing auxilliary array
=> $O(n) + O(n) = O(n)$.

and

Space complexity = O(n)

5) Dynamic Programming Approach II : Kadane's Algorithm

- The idea is to maintain the maximum possible sum of a subarray ending at an index without needing to store the numbers in an auxilliary array.

- It's an improvement in the previous dynamic programming approach optimizing the space complexity.

Pseudo code

```
Algorithm maxSubarraySum (A[], n)
//uses array A of size n
{
    max_so_far := 0;
    maxEnding_here := 0;
    for i := 0 to n-1 do
    {
        maxEnding_here := maxEnding_here + A[i];
        if (maxEnding_here < 0)
            maxEnding_here := 0;
        max_so_far = max( max_so_far, max-
                          Ending_here);
    } return max_so_far;
}
```

Note : Each element has visited only once

∴ Time complexity = $O(n)$

∴ Space complexity = $O(1)$

Q3
Ans

(a) Straightforward method:

```
// A is an array with index from 1 to n; A [1:n]
// max is the maximum element and min is the
// minimum element of A [1:n]
{
    max := min := A [1];

    for i := 2 to n do
    {
        if (A [i] > max) then
            max := A [i];
        else if (A [i] < min) then
            min := A [i];
    }
}
```

(b) Recursive method.
Divide and conquer method is a recursive approach to solve a problem. So we will use divide and conquer in this case.

```
Algorithm Maxmin (i, j, max, min)
// A [1:n] is a global array. Parameters i & j
// are integers. Set max to largest values & min
// to smallest value
{
    if (i=j) then
        max := min := A[i];
    else if (i=j-1) then
    {
        if (A[i] < A[j]) then
        {
            max := A[j];
            min := A[i];
        } else {
            max := A[i];
            min := A[j];
        }
    } else {
        mid := [(i+j)/2];
        MaxMin ( i, mid, max, min);
        Maxmin ( mid + 1, max1, min1)
        if (max < max 1) then
            max := max 1;
        if (min > min1) then
            min := min 1;
    }
}
```

c. Discuss complexity of both approach

With straight forward method :

Best case : When elements are in increasing order

max →, min →

10	20	30	40

↑ min ↑ max ↑ max

Total number of comparisons $= n-1$

∴ Time complexity $= O(n)$

Worst case : when elements are in decreasing order

max →, min →

50	40	30	10

↑ min ↑ min ↑ min

∴ Total number of comparison $= 2(n-1)$

∴ Time complexity $= O(n)$

GUPTA 215X270_SINGLE LINE 55X44_2COL_CK/DL 28-03-2018

With Recursive approach :

let Maxmin ( ) takes $T(n)$ time to execute.

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ T(n/2)+T(n/2)+2 & \text{if } n>2 \end{cases}$$

$$= 2 \cdot T\left(n/2\right)+2$$

Using : Master theorem for dividing function.

$$T(n) = 2T\left(\frac{n}{2}\right)+2 \left[\text{Form} : aT\left(\frac{n}{b}\right)+f(n)\right]$$

$a=2$ $b=2$ $f(n) = O(1)$

$= O(n^0 \log^0 n)$

$\Rightarrow k=0,\ p=0$ in $O(n^k \log^p n)$

Here $\log_b a$

$= \log_2 2$

$= 1 > k$

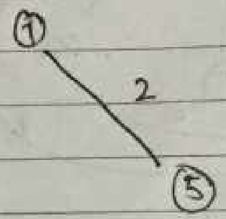
$\therefore T(n) = \Theta(n^1) = O(n)$ which is the best, worst and average case complexity.

In terms of storage, straight forward algorithm is much better than recursive as requires stack space for i, j, max, min, max1 & min1.

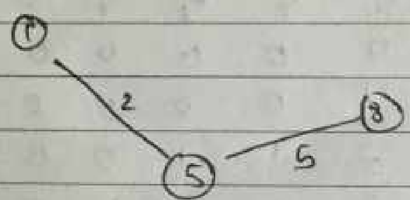
Q4 (a)
Ans

(a) Prim's Algorithm

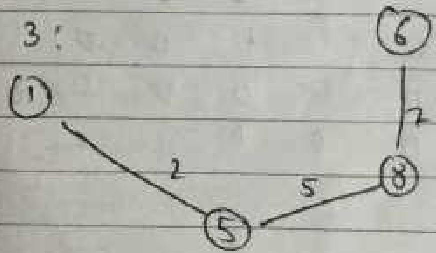
Step 1:



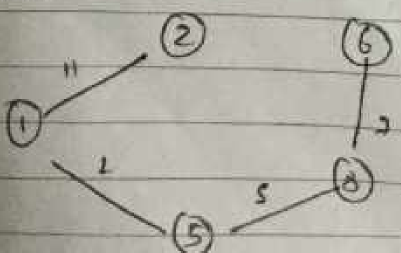
Step 2:



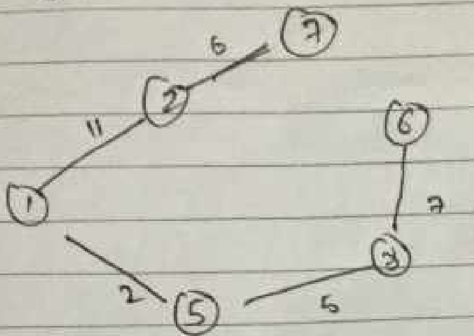
Step 3:



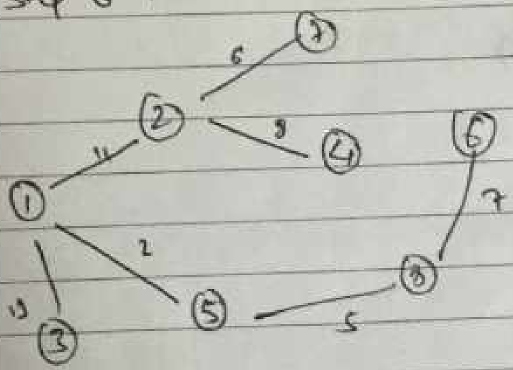
Step 4:



Step 5 :



Step 6 :

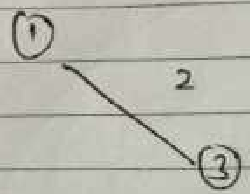


$\therefore$ Cost = 11 + 13 + 2 + 5 + 7 + 8 + 6

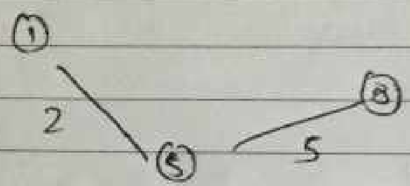
= 52

(b) Kruskal's Algorithm:

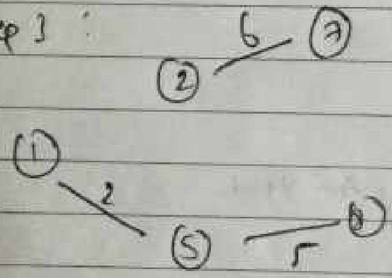
Step 1:



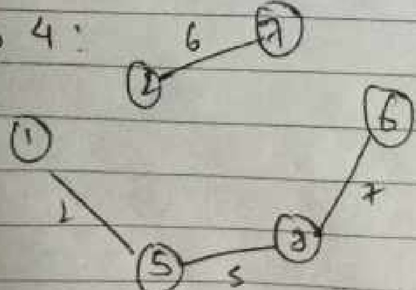
Step 2:



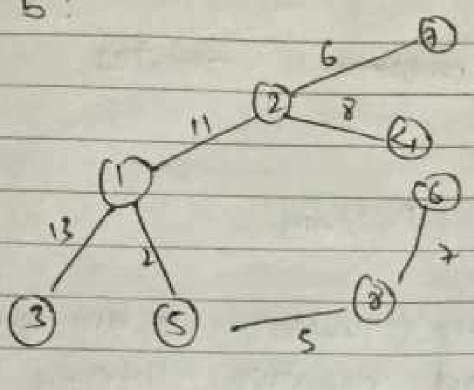
Step 3:



Step 4:



Step 5:



$\therefore$ Cost $= 2+5+6+7+8+11+13$
$= 52$

Q5
(a)
Ans

This problem can be related to the string editing problem which is to identify by a minimum cost sequence of edit operations that will transform X into Y.

Edit operation : insert, delete, change (a symbol of X into another), there is cost associated with the performance each.

The cost of a sequence of operations is the sum of the costs of the individual ~~problem~~ operations in the sequence

We are given two strings ~~X~~ $= x_1, x_2, x_3, \ldots, x_n$. $Y = y_1, y_2, \ldots, y_m$, where x_i, $1 \le i \le n$ and y_j, $1 \le j \le m$ are members of a finite set a symbols (alphabets) We want to transform X into Y using a sequence of edit operations.

Let $D(x_i)$ be cost of deleting symbol x_i from X
$I(y_j)$ be cost of inserting symbol y_j to X.
$C(x_i, y_j)$ be cost of changing symbol x_i of X into y_j of Y.

Let, In given sentences [consider case insensitive]
[i.e. a = A & Brush = Brush]

a = a
b = day
c = twice
d = brush

e = tooth
f = your
g = clear

We get X = abcdefg
Y = dfecab

Now to find minimum cost to transform X to Y.

operation: $\longrightarrow$ insert, $\downarrow$ delete, $\searrow$ change

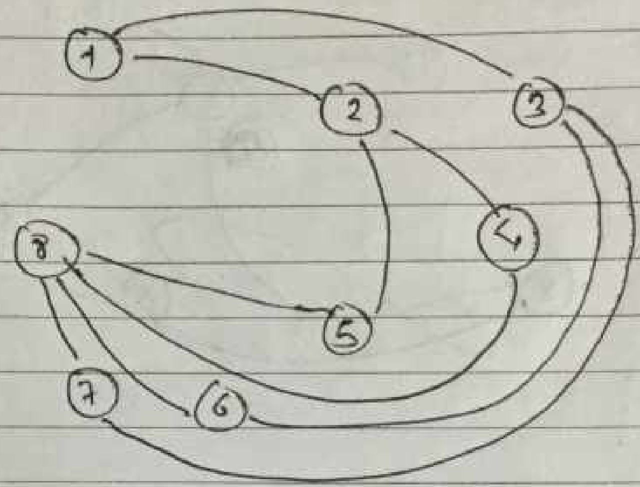
Using Dynamic Programming
Take Memorization Matrix:

X \ Y	null	d	f	e	c	a	b
null	0	1	2	3	4	5	6
a	1	1	2	3	4	4	5
b	2	2	2	3	4	5	4
c	3	3	3	3	3	4	5
d	4	3	4	4	4	4	5
e	5	4	4	4	5	5	5
f	6	5	4	5	6	6	6
g	7	6	5	5	6	6	7

= 7

So 7 minimum operations are required to convert X into Y.

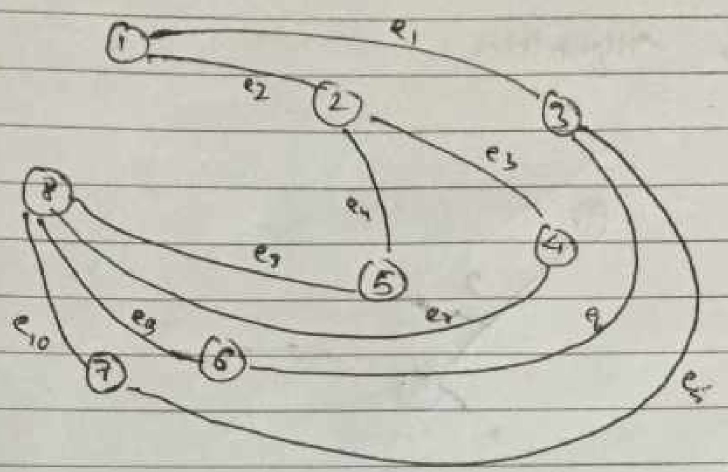
Q6 (a)
Ans



Adjacent list :

1 [ ] → [2 |] → [3 | /]
2 [ ] → [1 |] → [4 |] → [3 | /]
3 [ ] → [1 |] → [6 |] → [7 | /]
4 [ ] → [2 |] → [8 | /]
5 [ ] → [2 |] → [8 | /]
6 [ ] → [3 |] → [8 | /]
7 [ ] → [3 |] → [8 | /]
8 [ ] → [4 |] → [5 |] → [6 |] → [7 | /]

incidence matrix



$$
M = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 & e_{10} \end{array} \\
\begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{bmatrix}
\end{array}
$$

Q6 (b)

Ans Breath first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored notes. The algorithm follow the same process for each of the nearest node until it finds the goal.

Algorithm:

Step 1: SET STATUS = 1 (ready state)
for each node in G

Step 2: Enqueue the starting node A
and set its STATUS = 2
(waiting state)

Step 3: Repeat steps 4 and 5 until
QUEUE is empty

Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state)

Step 5: Enqueue all the neighbours of
N that are the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]

Step 6 : EXIT

⇒ Time and space complexity

The time complexity can be expressed as $O(|V|+|E|)$ since every vertex and vertex edge will be explored in the worst case. $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Note that $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$, depending on how sparse the input graph is.

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$, where $|V|$ is the number of vertices. This is in addition to the space required for the graph itself, which may vary depending on the graph representation used by an implementation of the algorithm.

When working with graphs that are too large to store explicity (or infinite). it is more practical to describe the complexity of breath- first search in different terms: to find the nodes that are at distance d from the start node (measured in number of edge traversals), BFS takes $O(b^{d+1})$ time and memory, where b is the "branching factor" of the graph (the average out-degree).

```
// Breadth first search of Graph G is carried at beginning at
// vertex V. For any node i, visited [i] := 1 if i has been
// visited already, G and array visited [] are global.
// visited [] is initialized to 0.
{
   u := v;   // q is a queue of unexplored vertices.
   visited [v] := 1;
   repeat {
        for all vertices w adjacent from u do
        {
             if (visited [w] = 0) then {
                  Add w to q;  // w is unexplored (enqueue)
                  visited [w] := 1;
             }
        }
        if q is empty then return;  // no explored vertex
        Delete the u from q;  // get 1st unexplored vertex
                              // (dequeue)
   } until (false);
}
```

Q6 (C)

Ans Graph colouring problem is to assign colors to certain element of a graph subject to certain constraints.

Vertex coloring is the most common graph colouring problem. The ~~given is~~ problem is, given m colors, find a way of coloring the vertres of a graph such that no two adjacent vertices are coloured using same color. The other graph coloring problems like Edge coloring (No vertex is ~~adjan~~ incident to two edges of same color) and Face Coloring (Geographical Map coloring) can be transformed into vertex coloring.

Let G be a graph and m be a given positive integer we want to discover whether the nodes of G can be coloured in such a way that no two adjacent nodes (vertices) have the same colour yet only m colours are used. If d is the degree of the given graph, then it can be coloured with d+1 colors. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be coloured. This integer is referred to as the chromatic number of the graph.

For example: the graph shown below can be colored with three colors: 1, 2 & 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color it and hence this graph's chromatic number is 3.

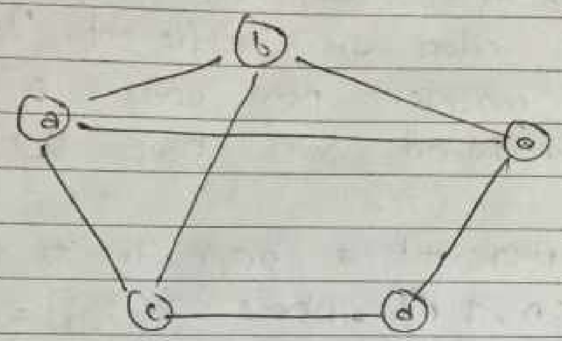


Fig: An example of graph and its coloring

A graph is said to be planar if it can be drawn in such a way that no two edges cross each other. A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs. This problem asks the following question:

Given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed? This turns out to be a problem for which graphs are very useful, because a map can easily be transferred into graph.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ~~been~~ ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient. In this section, we consider not only graphics that are produced from maps but all graphs.

Suppose we represent a graph by its adjacency matrix G [1:n, 1:n], where a[i, j] = 1 if (i, j) is an edge of G and G [i, j] = 0 otherwise. The colors are represented by the integers 1, 2, ..., m and the solutions are given by the n-tuple $(x_1, ..., x_n)$, where x_i is the color of node i. Using the recursive backtracking formulation, the resulting algorithm is mColouring.

Function mColoring is began by first assigning the graph to its adjacency matrix, setting the array x[] to zero, and then involving the statement mColoring (1);

Algorithm mColoring (k)
// The algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjace-
// ncy matrix a[1:n, 1:n]. All assignments of 1,2,...,m
// to the vertices of the graph such that adjacent
// vertices are assigned distinct integers are printed. k
// is the index of the next vertex of color

```
{
  repeat
  { // Generate all legal assignments for x[k].
        NextValue (k);      // Assign to x[k] a legal color
        if (x[k] = 0) then return;    // no new color possible

        if (k = n) then  // At most m colors have been used
                         // to color the n vertices
             write (x[1:n]);
        else mColoring (k+1);
  } until (false);
}
```

Algorithm NextValue (k)
// x[1], ... x[k-1] have been assigned integer values in the
// range [1, m] such that adjacent vertices have distinct
// integers. A value of x[k] is determined in the
// range [0, m] x[k] is assigned the next highest
// numbered color while maintaining distinctness from
// the adjacent vertices of vertex k if no such
// color exists then x[k] is 0.

```
{
  repeat
  {
     x[k] := (x[k]+1) mod (m+1) ; // next highest color
     If (x[k] = 0) then return;      // All colors
                                      // have been used
     for j := 1 to n do
     {
        //check for this color is distinct from adjacent
        //colors
        if (G[k, j] ≠ 0) and (x[k] = x[j])
          // if (k, j) is and edge and if adj
          // vertices have the same color
        then break;
     }
     if (j = n+1) then return;  //new color formed
  } until (false);  // otherwise try to find another color
}
```