ALGORITHMS ANALYSIS AND DESIGN

‘GETTING STARTED TO SOLVE PROBLEMS ON ALGORITHMS’

Contents

DIVIDE AND CONQUER

(great strategy to solve real life problems too :)

One important thing that helps us to solve these kinds of problems is the ability to break problem into smaller subproblems. And by solving multiple smaller subproblems, you'll be able to find the solution to the given problem.

PROBLEM-1

Let's solve a basic problem that requires divide and conquer strategy :

Problem: Given a sorted array arr[ ] of size N. Find the element that appears only once in the array. All other elements appear exactly twice.
Expected Time Complexity: O(log N)
Expected Auxiliary Space: O(1)

Approach:

- we would usually think of traversing the whole array and keeping the count of each element. When count==2 for an element, we would return the element. But this takes O(n) time complexity because we're traversing the whole array once. But the expected time complexity is O(N log N) which means we're not allowed to traverse the entire array, we need to reduce the number of steps for getting the answer.
- On observing, we see that the given array is a sorted array. Another thing is that we'll find the first occurrence of elements in odd positions till the element which occurs only once comes to the picture. After then, we'll find that the first occurrence of elements will have even indexed positions. This can be visualized as follows:

x1	x1	x2	x2	x3	x3	X4	x5	x5	x6	x6	x7	x7	….
1	2	3	4	5	6	7	8	9	10	11	12	13	14

We would observe that the first occurrences were at positions 1,3,5,7 till the element which occurred once came to the picture. Then, the first occurrences are at positions 8,10,12,14. If we find an element whose 1st occurrence has an even index, then our answer lies to the left of it. If we find an element whose 1st occurrence has an odd index, then our answer lies to the right of it.

Since the array is already sorted, and the answer either lies left or right based on condition, we can use binary searching.

- First look at mid, if the mid element has an even index as the first occurrence, then it means we need to search in the left part of the array. Else if it has an odd index for the first occurrence, we need to search in the right part of the array. Else if the mid element has no immediate duplicates that mean it occurred once. (i.e, arr[mid-1]!=arr[mid] && arr[mid]!=arr[mid+1])

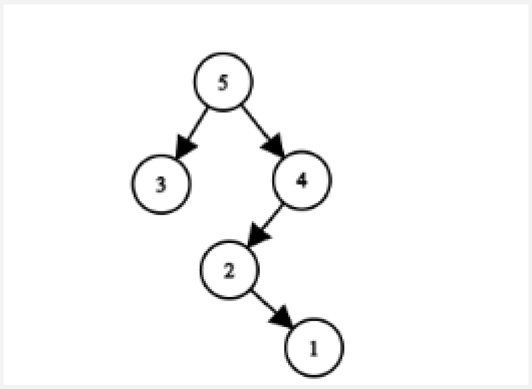
Link for the solution of the problem:
solution

PROBLEM-2

A permutation is a sequence of length 'n', integers from 1 to n in which all numbers occur exactly once. Ex: [1,2,3], [5,4,3,2,1],[1,3,2] but not [1,1,2] [4,3,1],[0]
Given a permutation a[1.....n] i.e, of length n, you need to transform permutation into a rooted 'binary tree'.
Rules for transforming into a binary tree:

- Max element of the array becomes the root of the tree
- All elements to the left of maximum, form a left subtree (built according to the same rules but applied to the left part of the array)
- All elements to the left of maximum, form a right subtree (built according to the same rules but applied to the right part of the array)

Example:
a=[3,5,2,1,4]
Then root will be a[2]=5, left subtree will be built for a[1...1]=[3] and the right subtree will be built for a[3....5]=[2,1,4]. the tree would be as follows:
Depth of vertex is defined as the distance from the vertex, depth of vertex=0. Depth of '3' = 1, similarly depth of 4 = 1... so, you need to print depth of each element of given permutation.
So, depth for given permutation = [1,0,2,3,1]



Approach:

The problem seems a little difficult at first glance but it can easily be solved using divide and conquer. here, we divide the problem into simpler subproblems for which solution can easily be found, and then, by solving all independent subproblems, we'll be left with the final answer,

- We would write a function depth(l,r,d), where l is the left index of the array to be considered and r s the right index (i.e, till where we need to consider), d is depth.
- In this function, we would find the maximum in the part of array a[l] to a[r] by iterating over these elements. I.w, we get max_element_index, we would assign its depth as the parameter passed (i.e, depth in argument). If its possible to perform depth() in the left part of the array (i.e, a[l], a[max_element_index-1]), then we would call the function depth and pass appropriate indices, depth as depth+1
- Similarly, if it's possible, we would also call depth() for the right part of the array. After calling both, we return back.
- Consider Array[] to be declared globally, and there is some array depth[] which contains depth of the corresponding index which is also declared globally. So, when we assign the depth of ith element, it gets updated because it's declared globally.
- When the function is called initially, it looks as depth(0,n-1,0) because the first max_element needs to be assigned depth=0.

Link to solution:

PROBLEM - 3

You are given a string [1...n] consisting of lowercase letters, whose length is $n = 2^k for\ k >= 0$. the string s[1..n] is called c-good string if at least one of the following conditions is satisfied.

- The length of s is 1 and it consists of character c
- The length of s>1, the first half of string consists of only character c $i.e, s_1 = s_2 =... s_{\frac{n}{2}} = c$ and the second half of string is (c+1)-good string
- The length of s>1, the second half of string consists of only character c $i.e, s_{\frac{n}{2}+1} = s_{\frac{n}{2}+2} =... s_{\frac{n}{2}+3} =... s_n = c$, and the first half of string is (c+1)-good string

Example: aabc is 'a'-good string and ffgheeee is 'e' good string.
In one move, you can choose index i from 1 to n and replace it with si (i.e, any lower case letter) we've to find the number of moves required to obtain a-good string from s.
Problem link: problem

Approach:

This problem is solved using the divide and conquer strategy.

- We've to find the minimum cost of transforming n length string to a-good string.

- This can be performed in 2 ways
- Either the first half be equal to ‘a’(i.e, all letters in 1st half) and the other half is ‘b-good string’ or
- The first half is b-good string and 2nd half be equal to all a’s.
- We would take cost which is minimum from the above 2 ways. I.e, calculate the cost to transform into a-good string in the 1st method and 2nd method. Take minimum of them.
- transformation of the half strings (i.e, previous step) into b-good strings, can also be done in similar steps.

Pseudo code looks as follows:

```
int getMin(int l,int r, char ch){
        //base case
        if(l==r){
                if(s[l]==ch) return 0;
                Else return 1;
        }
        Mid = (l+r)/2
        X = cost(l,mid,ch)+getMin(mid+1,r,ch+1);
        Y = cost(mid+1,r,ch)+getMin(l,mid,ch+1);
        Return min(x,y);
}
```

Here, getMin function will return minimum number of moves required to transform s[l] to s[r] into character ch-good string (ch=a / b/..)
Here, the cost function will simply iterate over the given indices and return the number of letters that are not equal to character ch. So, indirectly, this is number of moves to change part of the string to character ch.
In this way, we get minimum number of moves to transform the given string into c-good string.
Link to solution,

Dynamic Programming

In this type of approach, we usually try all possibilities but in a clever way. Its like careful bruteforce, which involves remembering solution to smaller subproblems.

We would also divide the problem into smaller subproblems, and then use these solutions in finding solution to the bigger problem.
Let's look at basic (easy) problem.

PROBLEM-1

There are *N* stones, numbered 1,2,…,*N*. For each *i* (1≤*i*≤*N*), the height of Stone *i* is h_i. There is a frog which is initially on stone-1. He will repeat following until he reaches stone N.

- If frog is on stone i, jump to stone i+1 or i+2. Here, cost of |h_i - h_j | is incurred, where j is the stone to land on.

Find minimum possible total cost incurred before the frog reaches stone N.
INPUT:
N
h_1 h_2 h_3 ….. H_n

OUTPUT: print minimum possible total cost incurred.
Problem link: problem

APPROACH:
We'll divide the problem into smaller subproblem. Here, lets calculate minimum cost to reach stone 'i' for all 1 <= i <= N. its easier to calculate for stone 1. If we know answer for stone 1, we can calculate for stone 2, 3… by just considering minimum of $min(\ |h_{i} - h_{i-1}|, \ |h_{i} - h_{i-2}| \ )$this gives us minimum cost for reaching stone i (because we're considering minimum in every case, we get right answer).

For implementation, declare array min_cost[N], here min_cost[i] represents min_cost taken to reach stone i.
Pseudo code:

```
min_cost[0]=0;
Min_cost[1] = abs ( h[0] - h[1] )
for( i = 2; i→ n)
      Min_cost[i] = min ( abs( h[i-1] - h[i],abs ( h[i-2] - h[i] ) );
```

Solution link: solution

PROBLEM-1b

The above problem can be modified and asked as following
 There is a frog which is initially on stone-1. He will repeat following until he reaches stone N.

- If frog is on stone i, jump to stone i+1 or i+2 or i+3 or i+4…. i+k. Here, cost of |h_i - h_j | is incurred, where j is the stone to land on.

INPUT:
N K
h_1 h_2 h_3 ….. H_n
OUTPUT: print minimum possible total cost incurred.
Problem link: problem
Here, we'll initially calculate min_cost[0],min_cost[1]. Then from i=2, we'll calculate in as follows:

- Stone 2 can be reached from 1(2-1), 0(2-2) [when k>=2 i.e, we can jump to k+2 or greater]
- Stone 3 can be reached from 3-1,3-2,3-3 (when k>=3)
  - - - - - - - - -
  - - - - - - - - -
  - - - - - - - - -
- Stone 's' can be reached from s-1, s-2, s-3, s-4,......s-k. (provided k>=s) now, $min\ cost[s]\ = min\ (\ |h_{s-1} - h_s|\ +\ min\ cost[h_{s-1}],\ |h_{s-2} - h_s|\ +\ mincost[h_{s-2}],$ $|h_{s-3} - h_s|\ +\ mincost[h_{s-3}]$...... $|h_{s-k} - h_s|\ +\ mincost[h_{s-k}]\ )$
- Similarly, we'll calculate till N.

We'll have to take care of conditions like, s < k. We'll have to take minimum accordingly.
Solution link: solution

PROBLEM-3

Taro's summer vacation starts tomorrow., and he decides to make plans for it.
There are N holidays. For each i, 1<= i <=N , he chooses one of the activities to do it on ith day.

- A: swim in sea and gain a_i points of happiness
- B: catch bugs and gain b_i points of happiness
- C: Do homework at home and gain c_i points of happiness

As taro gets bored easily, he cannot do same activities for 2 or more consecutive days.
Find max_possible total points of happiness that taro gains.

INPUT:

N
A1 b1 c1
A2 b2 c2
………..
………..
………..
An bn cn
APPROACH:
The solution includes a very simple method to get an answer. We use dynamic programming to solve this. We divide problem into smaller subproblems and use their solution in finding solution to the final bigger problem.
here, smaller subproblem is 'for the ith day, find max_points earned by doing activity x on that day. So, find max_points that can be earned by doing activity 'a', activity b, activity c. if you are calculating max_points by doing activity 'a', then we'll take max (points earned by doing activity b, points earned by doing activity c) on the previous day because he cannot do the same activity for 2 consecutive days.
For implementation, use N x 3 array, where arr[i][0] represents max_points earned by doing 'a' activity on i th day.
Solution link: solution

PROBLEM 4

There are N items 1,2…..N. for each i, (1 <= i <= N), item i has weight of w_i and value v_i. Taro has decided to choose some of N items carry home in a knapsack. The capacity of knapsack is W, which means the sum of weights of items taken must be atmost W.
Find max_possible sum of values of items that taro takes home.

APPROACH: (considered 1-indexing)
We can easily solve this by using dynamic programming approach. So, the subproblem we would consider is that we'll calculate max_value that can be carried when weight w, items 1,2..i are considered. We're going to solution for each and every weight. I.e, weights from w = 1 ...to w = W. simultaneously, we're going to increase the items allowed.
We'll be considering a 2-D array of size N x W. dp[i][j] represents max_value that can be carried by considering items 1,2..i and weight j.
We can start filling array[][] row-wise.
For first row and first column,

- Dp[1][1] -if weight of item-1 <=1, then we can carray item-1 which implies max_value=value[1]. If weight[1] > 1, value=0.

- Similarly, we can fill the rest of the column in 1st row.

For general case, the ans would be as follows:
Dp[i][j] = max(dp[i - 1][j - weight[i]] + value[i], dp[i - 1][j])
This means that, max_value which can be taken by considering 1,2...i and weight j is either -- equal to max_value that can be taken by considering only 1,2..i-1 items or equal to max_value that can be taken by considering i-1 items and weight w-weight[j] + value[j] . i.e, we've considered item j in 2nd case whereas we didn't in 1st case.

- We'll use this logic to fill out the rest of the table.
- dp[N][W] gives us the actual answer needed (which is obvious)

Solution link: solution

PROBLEM-5

There is a grid with horizontal rows 'H' and 'V' vertical columns. Let (i,j) denote square at i-th row from top and j-th column from the left.
For each i and j, square (i,j) is described by character $a_{i,j}$. if $a_{i,j}$ is '.', then its empty square.
If its '#' its a wall square. Its guaranteed that (1,1) and (H,W) are empty squares.
Taro will start from (1,1) and reach (H,W) by repeatedly moving right or down to an adjacent empty square.
Find number of taro's paths from initial to final.
Problem link: problem

APPROACH :
This might look a little complicated but still dynamic programming makes it fairly simple. Again, we try to find the right kind of subproblems and then use their solutions for final bigger problem.
Here, we consider a 2-D array of size H x W d[H][W] . (1-indexing)
Dp[i][j] represents the number of ways from (1,1) to (i,j). This can be done as follows:

- For row-1, any cell is of form (1, j). It can be reached only from its left cell (because we can only move right or down) so, d[1][j] =dp[1][j-1]
- For row-i (i > 1), the case looks as follows:
  → if column > 1, the it can be reached from its left cell else it can only reached from its top cell.

  If (j > 1) dp[i][j] = dp[i][j] + dp[i-1][j]
  If (i > 1) dp[i][j] = dp[i-1][j] + dp[i][j]

I.e, we add number of ways possible to reach its upper cell to the number of ways to reach the current cell. Similarly we add number of ways to reach its left cell. (try to make a table and fill table using this method, you'll get visualisation of the answer and the reason for doing this way)

So, we fill this table row wise. Since we're traversing through 2D matrix, time complexity becomes O(H x W)
Solution link: solution

LONGEST_INCREASING_SUBSEQUENCE

the problem is we're given a sequence of numbers, we need to find the longest increasing subsequence in the given sequence.

input : a1, a2, a3......an.

a subsequence ai1, ai2, ai3....aik is increasing if 1<= i1< i2 < i3....<ik <=n (numbers are getting strictly larger)

- we need to convert the problem into a dag of subproblems. so, we'll create a DAG where all numbers are vertices and 2 vertices are have a edge (u1 -> u2) iff u2> u1.

- by creating edges in such a way, we've captured all possible (& permissible) sequences.

- now, we need to find longest path in the dag. the only issue is that we donot know source and destination. so, we've find all possible paths and take maximum from them.

- for making it simpler, we divide problem into subproblems where each subproblem is longest increasing subsequence ending with i.

L(i) is longest path susequence ending with 'i'.

collection of subproblems : { L(i): 1<= i <= n }

L(i)= 1+ max{L(j);(i,j) is an edge) { here i<j because (i,j) is an edge }

here we're simply counting the nodes in the path and adding 1 (to include present vertex).

hence the pseudo code looks like this:

```
for j = 1, 2, . . . , n:

$L(j) = 1 + \max\{L(i) : (i, j) \in E\}$

return $\max_j L(j)$
```

Problem link: [problem]
Solution link: [solution]

(medium level question)

PROBLEM-6

Given two strings ‘text1’ and ‘text2’, return length of longest common subsequence.
A subsequence of a string is a new string generated from original string wih some characters removed or not removed without changing relative order.
A common subsequence of two strings is the subsequence that is common to both strings.
Example:
Input: text1 = "abcde", text2 = "ace"
Output: 3
Solution link : [solution]

Edit-distance

the problem is to find edit distance when we're given 2 strings of lengths m,n. i.e, minimum number of insert/delete/overwrite operations performed to convert one string
to another.

example: we need to convert 'sunny' to 'snowy', we came up with following alignment

```
s u n _ _ n y
_ s o w n y _
```

the cost in this case is 6, because no two characters in respective column are matching. the first operation is delete s, next overwrite u with 's' , n with o, insert o,w, overwrite n with y and delete y.

cost of above operation is '7' because number of mismatched characters =7.

similarly, we can come up with cost = 3

the edit distance between 2 strings is cost of their best possible alignment.so, we need to find minimum cost.lets break this into subproblems. subproblems would be to find edit distance between prefix of length m (for string 1) , prefix of length n(string 2). lets consider the right most part of the alignment.

there would be only 3 possibilities . deleting x[i] / inserting y[j] / overwriting x[i] with y[j].

in case 1, the problem would be converted to E(i,j) = 1+ E(i-1,j). we're deleting x[i], so we add '1' and we consider edit distance of i-1 chars from x, j chars from y.

in case 2, E(i,j) = 1+ E(i,j-1) . we're inserting y[j]. so we add 1 and we'll consider edit dist of i chars from x, j-1 chars from y.

in case 3, E(i,j)= diff(x[i],y[j])+E(i-1,j-1) , diff(x[i],y[j])=0 if x[i]=y[j] and 1 if not equal. (this is obvious)

now we choose minimum over these 3.E(i,j) = min { 1+ E(i-1,j) , 1+E(i,j-1) , diff(x[i],y[j])+E(i-1,j-1) } where diff(x[i],y[j])=0 if x[i]=y[j] and 1 if not equal.

to find E(m,n) we'll construct 2D array, where M[i][j] represents E(i,j). example, M(0,1) represents edit distance of 0 chars from x, 1 char from y. obviuosly, this is 1 because we've to insert y[0] .

we'll fill all entries of mxn matrix in specific order. we first fill 1st row and column. then, we'll start filling rowwise/columnwise (any order is fine) because

if we fill row wise, we'll have M[i-1][j-1], M[i-1][j], M[j-1][i]. so, its fine.. the answer would be M[m-1][n-1]., becuase this means E(m,n) the time complexity for above approach would be O(m * n) because, it takes O(1) to compute E(i,j) for 0<=i<=m, 0<=j<=n, and we need to compute all values of matrix M[m][n]. so the time complexity becomes O(m * n). The following code takes 2 strings ans outputs edit_distance.

```
#include<bits/stdc++.h>
using namespace std;
int main(){
   char s1[100],s2[100];
   cin>>s1>>s2;
   int l1=strlen(s1);
   int l2=strlen(s2);
   int M[l1+1][l2+1];
   int diff=1;
   for(int i=0;i<=l1;i++){
      M[i][0]=i;
   }
   for(int i=0;i<=l1;i++){
      for(int j=0;j<=l2;j++){
         if(i==0||j==0){
            M[i][j]=max(i,j);
         }
         else{
            if(s1[i]==s2[j]){
               diff=0;
            }
            else
            diff=1;

            M[i][j]=min({1+M[i-1][j],M[i-1][j-1]+diff,1+M[i][j-1]});
            diff=1;
         }
      }
   }
   cout<<M[l1-1][l2-1]<<"\n";

}
```

PROBLEMS ON GRAPHS

The following problems are purely based on depth first search and breadth first search algorithms.

PROBLEM-1

You are given a map of a building, and your task is to count the number of its rooms. The size of the map is n×m squares, and each square is either floor or wall. You can walk left, right, up, and down through the floor squares.

First line of input contains n,m : height and width of map.
Then, there are n lines of m characters describing the map.each character is either '.' (floor) or '#' (wall)

Output: number of rooms

Example:
```
5 8
# # # # # # # #
# . . # . . . #
# # # # . # . #
# . . # . . . #
# # # # # # # #
```
Here there are 3 rooms . (because 3-connected components)

APPROACH:
We just have to count number of connected components and he number of connected components becomes answer. To find number of connected components, we've to perform depth-first search till all nodes are visited.

- We'll start performing dfs on all points on given 2-d graph.
- We'll perform dfs only if its not visited. (we maintain 2d bool array to check visited or not)
- After finishing dfs on some node, we'll increment number of connected_components
- dfs(): we'll first mark this node as true.
  we've freedom to move right, left up, down. Declare a vector of pairs which contains the coordinates coorecponding to left,right,down,up moves
  For example: {-1,0},{1,0},{0,1},{0,-1}

For each node, we check whether it can be moved left/right/up/down by adding these (above mentioned moves) . if it turns out to be valid, then we'll start dfs on that node too.
Problem link: problem
Solution link:solution

PROBLEM-2

You are given a map of a labyrinth, and your task is to find a path from start to end. You can walk left, right, up and down.

The first input line has two integers n and m: the height and width of the map.Then there are n lines of m characters describing the labyrinth. Each character is . (floor), # (wall), A (start), or B (end). There is exactly one A and one B in the input.

First print "YES", if there is a path, and "NO" otherwise.
If there is a path, print the length of the shortest such path and its description as a string consisting of characters L (left), R (right), U (up), and D (down). You can print any valid solution.
link

APPROACH:

We'll be using breadth first search on grids. BFS for graphs is little different from normal one.
Lets say we're on some box in a 2-D grid, then the left,right, up, down cells can be reached in one step when we use bfs.

			3	2	3				
		3	2	1	2	3			
	3	2	1	start	1	2	3		
		3	2	1	2	3			
			3	2	3				

- For BFS(), we'll; already store starting and ending coordinates of the nodes.
- Now we'll push-starting coordinate into a queue

- We'll check all possible moves (i.e, up, right, left, down) if some node is not visited/ the coordinates are not out-of-bound, then they are valid coordinates. If they are valid, we'll mark them as visited, then We'll push them into the queue.
- Now, queue is again not empty, we'll push all possible nodes onto queue and mark them as visited.
- Here, we'll be marking parent of each node.
  Parent: if i reach (1,2) from (1,1), then parent[1][2]={1,1}, but instead of nodes, we'll store moves i.e, {0,-1} [think carefully]
- In this way, we'll finish BFS.
- If the ending node (i.e, 'B') is visited, then we'll have path otherwise its unreachable
- Now, if its visited, we've to trace back parents from the ending node.
- The way we do it is -- we'll first have ending_x,ending_y. We'll however be able to know its parent.[this is because we've stored the moves coordinates in par[i][j], if we subtract moves' coordinates from original coordinates, we get coordinates of the cell from which we reached to the current cell] We've to continue this finding parent till the starting_x, starting_y are encountered.
- According to moves, we'll be displaying R [right] / L[left] / U [up] / D [down]
  Solution: link

PROBLEM- 3

Byteland has 'n' cities, 'm' roads between them. The goal is to construct new roads so that there is a route between any two cites.
Find the minimum number of roads required and determine which roads to be built.
Problem: link
Here,

$1 <= n <= 10^5$; $1 <= m <= 2x10^5$; $1 <= city\_1, city\_2 ... city\_n - 1 <= n$

INPUT FORMAT
m n
City_1 city_2
City_3 city_4
....
M roads

OUTPUT :

<number of roads>
<city_i> <city_j>
…………………

APPROACH:
This is most simple problem because logic is straight. They said to construct roads such that there is route between all the vertices. We can just find number of connected components:

- If num_of_connected_components = 1, then there is a route between all vertices
- But if it's > 1, that means there are few disconnected vertices.
- Its enough to construct (num_of_conn_compo)-1 number of roads.
- If we're constructing a road between 2 connected components, one vertex needs to be in one set and other from another connected component.

Solution: link

PROBLEM-4

Syrjälä's network has n computers and m connections. Your task is to find out if Uolevi can send a message to Maija, and if it is possible, what is the minimum number of computers on such a route.

INPUT:
The first input line has two integers n and m: the number of computers and connections. The computers are numbered 1,2,…,n. Uolevi's computer is 1 and Maija's computer is n.
OUTPUT:
If it is possible to send a message, first print k: the minimum number of computers on a valid route. After this, print an example of such a route. You can print any valid solution.

Problem link: link

APPROACH: Breadth-first search gives us minimum distance.so, bfs() on index 1 would be enough to see whether there is a connection possible or not.

- In bfs(), we'll be traversing each level and we'll find the level in which our required vertex is present.
- we'll also maintain 1-D array parent[1,2,3…..n], which tells us parent of vertex_i. This is useful in retrieving path from vertex n to vertex 1.

- We'll also maintain 1-D array, distance[1,2….n] which tells us number_of_computers_visited from vertex_1 to vertex_i. On the first iteration, (i.e, neighbors of vertex_1)distances of all would be equal to 1.
- I've used a queue in implementing bfs(). It starts from vertex '1' and then traverses over its neighbors (i.e, connected vertices list) , mark them visited, push them into queue.
- Now we'll pop an element from the queue and traverse the list of its neighbors, if they are not visited, mark them visited and push them into the queue.
- Here, while traversing over its neighbors, we'll mark parent[neighbor]=curr_vertex. This is because we've reached the neighbour_vertex from current_vertex.
- We'll continue this process till the queue becomes empty.
- It becomes empty when vertices in the connected component are visited.

Pseudo code would look like:

```
Void bfs(){
        queue<int> que;
        que.push(1);
        dist[1]=1;
        while(!que.empty()){
                x=que.front();
                que.pop();
                for( auto it: arr[x]){
                        if(!visited[it]){
                                visited[it]=true;
                                distance[it]=dist[x]+1;
                                parent[it]=x;
                                que.push(it);
                        }
                }
        }
```

Link: link

PROBLEM-5

Byteland has n cities, m roads between them. Your task is to design a round trip that begins in a city, goes through two or more other cities, and finally returns to the starting city. Every intermediate city on the route has to be distinct.

Input

The first input line has two integers n and m: the number of cities and roads. The cities are numbered 1,2,...,n.Then, there are m lines describing the roads. Each line has two integers
a and b: there is a road between those cities.

Every road is between two different cities, and there is at most one road between any two cities.

Output
K: the number of cities on the route. Then print k cities in the order they will be visited.
Link: problem

APPROACH:

- I've used DFS to detect a cycle and an array parent[] which is used to retrieve that cycle (if exists).
- We'll perform DFS() such that all connected components are covered.
- We'll start DFS() on vertex 1 . we'll perform following steps in DFS()
- We'll mark visited[1] = true
- We'll traverse over its neighbors and if they are not visited, perform DFS() on them. At the same time, mark parent[neighbor] = current_vertex.
- If neighbor is visited, and parent[neighbor] !=current_vertex, then cycle exists.
- Then we'll retrieve cycle using parent array.

Solution link: link

Link to google doc: AAD_PROJECT