Distance of nearest cell having 1

$m * n$ binary grid

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

For (0,0) nearest 1 is at distance 0, as it itself is a 1

[0,1] " " " " " 1

[0,2] → 0

[1,2] → 1

[2,2] → 2

So our answer will be →

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

We only calculate distance horizontally or vertically, diagonal distance is not considered

Approach - Here, we can use any graph traversal algo, we'll be using BFS. We'll start from cells having 1. We'll traverse all its 4 directions. Whenever we find a 0, we'll put a step count = 1, at that index.

So, this is similar to Level order traversal, wherever level increases, we increase step count

e.g. - 2 2 2 → 2 1 2
1 0 1 → 1 1 1
1 1 1 → 0 1 0

We'll use visited & resultant arrays & a queue for BFS traversal. And in queue we will store index & min steps.

So, initially

(2,2,0)
(2,0,0)
(1,1,0)

Now, node = (1,1) min steps = 0

visit its neighbours and if neighbours = 0.

Push them in queue & mark them visited. While pushing in queue add min steps + 1

TC - $O(n * m * 4)$ SC - $O(N \times N) + O(N \times N) + O(N) * 3$

Surrounded Regions

Given a matrix of size N x M, containing 'O' & 'X'. Replace all 'O' with 'X' that are surrounded by 'X'.

Eg - n = 5, m = 4

X	X	X	X
X	O	X	X
X	O	O	X
X	O	X	X
X	X	O	O

convert to X,

If someone is on the boundary, that is where it cannot be replaced,

Approach - So, the main observation is that the group of O's should not be converted to boundary.

Suppose, if matrix is

X	X	X	X
X	X	X	X
O	O	O	X
X	O	O	X
X	X	X	X

Then none of the 'O' can be converted to 'X'.

Start from the Boundary O's & perform DFS traversal to nearby O's. And mark them that they cannot be converted.

Traverse all the boundaries first, if O is found perform DFS on it, & mark them visited, in vis array.

Now, we'll again traverse the matrix. And we will compare mat[i][j] == 'O' && vis[i][j] = 0

@ It means vis[i][j] has not been visited by any boundary.

$T.C. - O(N) + O(N \times M) \times 4 \approx O(N \times M)$

$S.C. - O(N \times M)$

Number of Enclaves

n x m binary matrix grid, 0 represents sea
1 represents land

A move consists of moving from one land cell to another

Return the no. of land cells in grid for which we cannot walk off the boundary.

e.g

0 0 0 1 →
0 1 1 0
0 1 1 0
0 0 0 1 →
0 1 1 0
↓ ↓

So answer = 4
rest all 1's if we try to move will move us out of boundary.

So, count no of lands which are not connected to boundary.

Approach- Lands that is connected to boundary can never be answer
Figure all boundary 1's & then connect all 1's adjacent to it and mark them all visited,

This time we will use BFS,
Initially put all binary boundary 1's in the queue - And then start traversing,
Then perform BFS & mark them visited

(4,2)
(4,1)
(3,3)
(0,3)

TC - O(N x m) x 4 for 4 directions
SC - O(N x m)

Number of Distinct Islands

Given a binary 2d matrix of size n x m,
Find no. of distinct islands where a group of connected 1's forms an island. Two islands are distinct if they are not equal. (not rotated)

1 1 0 1 1
1 0 0 0 0
0 0 0 0 1
1 1 0 1 1

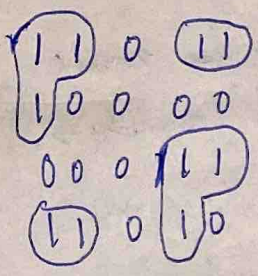
These 2 islands look identical. But one is rotated.
So, they are distinct,

So, in this matrix there are 3 distinct islands.

Approach -

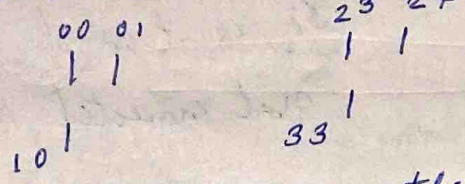
What if we store the shape in set data structure

How do we store shape?



There are 2 distinct islands in this grid.

Let's take 2 islands



They are identical, but if we store them in set as it's set will consider them as distinct.

{{0,0} {0,1}, {1,0}} {{2,3} {2,4} {3,3}}

Here we can see that 0,0 & 2,3 are base of both islands.
So what if before inserting the pair in set, we'll subtract the base.

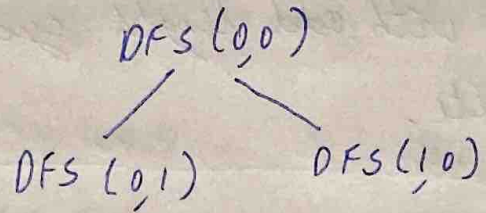
(2,3) - (2,3) → (0,0)
(2,4) - (2,3) → (0,1)
(3,3) - (2,3) → (1,0)

(0,0) - (0,0) → (0,0)
(0,1) - (0,0) → (0,1)
(1,0) - (0,0) → (1,0)

Now they can be stored as unique, Meaning set will only store one entry.

We have to follow the same order while storing indices. That means if we went right first then we will go right while storing second island.

Follow the same approach of no. of islands, and store the indices in set there only.



{(0,0), (0,1), (1,0)}

Store this in set

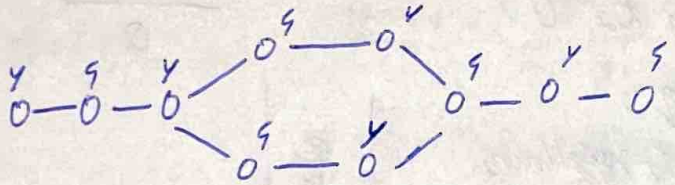
The only difference is we'll be using set to store shapes.

TC - (N x M) x 4 + (N x M) x log(set length)

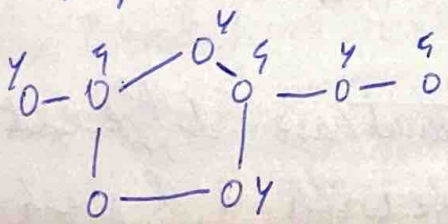
Bipartite Graph

Color the graph with 2 colors such that no adjacent nodes have same color.

Y - Yellow
G - Green



This graph can be called as a bipartite graph



↑ This node cannot be colored, so this graph is not a Bipartite graph.

Key Observation - We can always color a linear graph, i.e. a graph which doesn't have cycle.

So, Linear ~~cycles~~ graphs with no cycles is always a Bipartite graph.

Observation 2 - Any graph with even cycle length can also be bipartite

Observation 3 - Any graph with odd length cycle can never be bipartite.

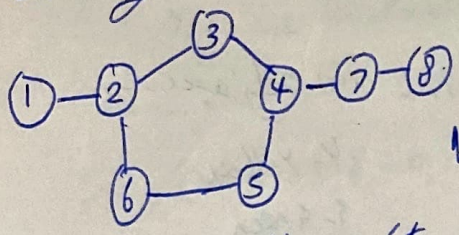
Approach -

BFS algo - We will use a queue ds

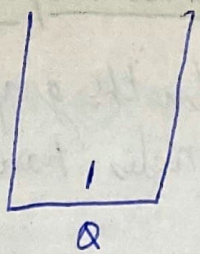
& a color array

-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

& every node is non colored initially.



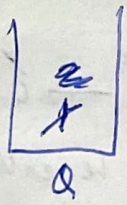
We start with node (1)



Push it in q. Change its color to 0

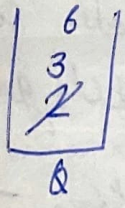
Now start iterating the q.

Node → 1 . Get its neighbours



0	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

Now, node (1) had color 0, node (2) should have different color i.e. 1 . So, color this with 1 & push in queue



0	1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

Now take out 2, check its neighbours

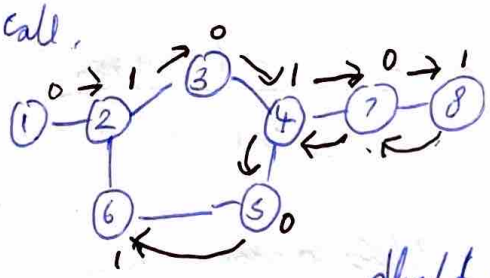
Color (3) & (6) - Put in queue

Now, this way we'll proceed and node ④ & ⑤ will eventually end up with same color. So, we straight away return false.

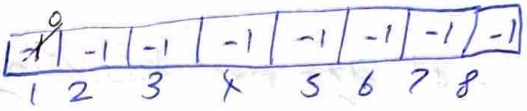
Using DFS →

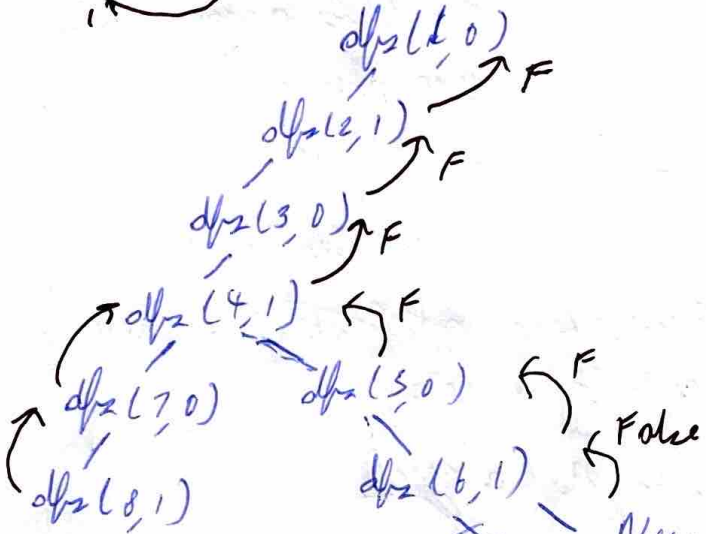
Use color array to update colors/store colors.

When we make a call, we use dfs(node, color) to make call.



color





Now here while traversing adjacency list of 6.

We will notice that color[2] has already been set, and color[node] == color[adjacent node]

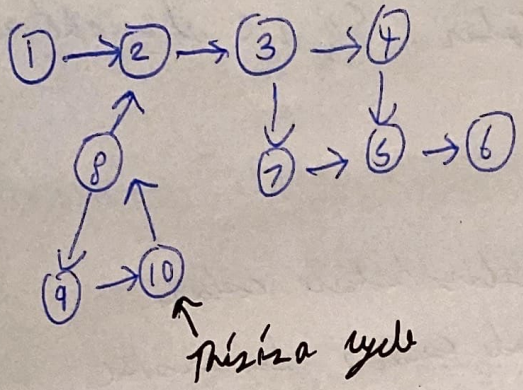
This tells that graph is not bipartite.

So, this graph returns false.

TC - O(V + 2E)

SC - O(V)

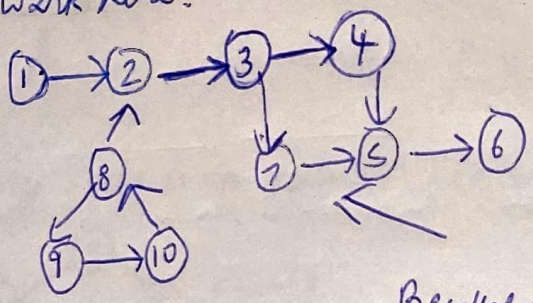
Detect Cycle in a directed graph using DFS



Approach:

On the same path node has to be visited again.

Our old algo for detecting cycle in undirected graph will not work here.



In this graph, if we use previous old algo it will detect a cycle at This position in the graph.

Because here ~~adjacent~~ vis(node) == vis(adj)

It will return true here,

But in reality this is not a cycle.

Approach -

We will take 2 visited arrays, vis & pathvisited.

vis →

0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0

pathvis →

0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0

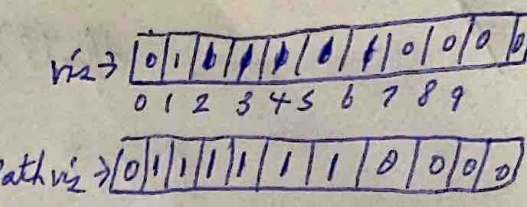
Now, ~~node=1, mark it~~ We'll solve this component wise.

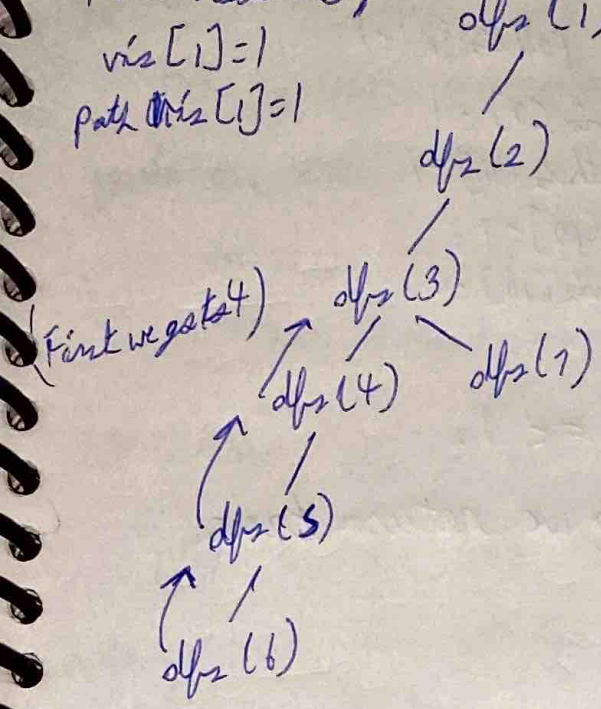
i.e. for (i → v)

~~dfs~~ if (! vis[i])

dfs(i)

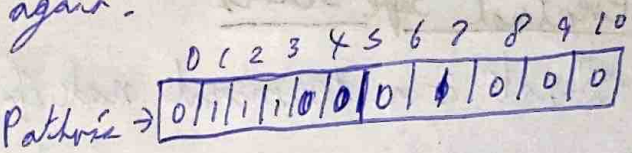
For node = ①,
vis[1] = 1
pathvis[1] = 1





Now when we go back from 6, we omit the 6 from path visited, i.e. we make 0 again.

Make pathvis[6] = 0 again



Now when we reach 3, it will not become 0, because its other call is yet to be made.

Now, dfs(3)
dfs(7)

We see that adjacent node of 5 is 7, and 5 is already visited.

So, is this a cycle?

No, because its path visited = 0
So, no cycle, we'll go back. And all path visited array become 0.

Now, we can see the component ⑧ ⑨ ⑩ can only be visited

If we solve this component wise, so, it now call will be made for dfs(8)

dfs(8)
/
dfs(9)
/
dfs(10)

vis[8] = 1
pathvis[8] = 1
vis[9] = 1
pathvis[9] = 1
vis[10] = 1
pathvis[10] = 1

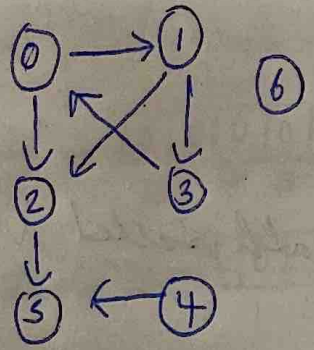
Now, when both vis & path vis == 1.
We say a cycle is detected. So, we return true.

TC - O(V+E)
SC - O(2N)

Find Eventual Safe States

A node is a terminal node if there are no outgoing edges.
A node is a safe node if every possible path starting from that node leads to a terminal node.

Return the array containing all the safe nodes of the graph. Ascending order.



2 is a safe node
4 is also a safe node
5 itself is a terminal node, so it is a safe node.
6 is also a safe node

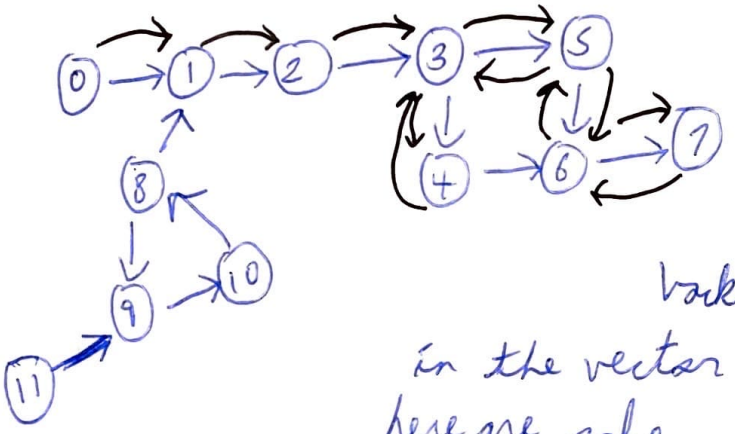
ans → [2 4 5 6]

Approach - We can use the cycle detection technique using DFS. If there is a cycle then all the nodes in that cycle are not safe.

→ Anyone who is part of a cycle cannot be safe.
→ Anyone that leads to a cycle " " " .

We will use vis & pathvis arr to detect cycle.

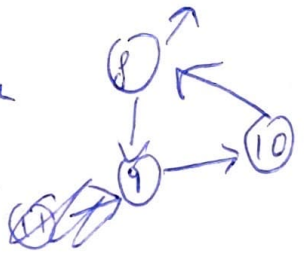
When the dfs calls comes back & detects no cycle in a component, then we can say that ~~if~~ there is a safe node in this component, or all nodes are safe in this component.



Now till 7, no cycle detected, & 7 is a terminal, So while going back/ backtracking, we'll push values in the vector, because all nodes till here are safe.

vec → [7 6 5 4 3 2 1 0]

Now, call dfs for



component.

This is already vis, so no call

dfs(8)
dfs(1)
dfs(9)

Here, we will detect a cycle and no node in this component is safe.

Idea of KMP is that is there any string in pattern where prefix == suffix

We observe "abc" is a prefix which is also a suffix

abc d abc

This is the observation KMP algo will do to avoid no. of comparisons.

In KMP algo, for a pattern we generate a

PI Table /(Longest Prefix/Suffix) LPS Table

Now, to generate a PI table,

	1	2	3	4	5	6	7	8	9	10	← 1 based index
P_1:	a	b	c	d	a	b	e	a	b	f	

ab is occurring 3 times, so we'll create a table. In this table, wherever ab is occurring for 2nd/3rd time, below that occurrence we'll write indexes of their first occurrence.

P_1:	a	b	c	d	a	b	e	a	b	f
	0	0	0	0	1	2	0	1	2	0

0 denotes that char is occurring first time

1 2 tells first occurrence of ab

P_2:	a	b	c	d	e	a	b	f	a	b	c
	0	0	0	0	0	1	2	0	1	2	3

P_3:	a	a	b	c	a	d	a	a	b	c
	0	1	0	0	1	0	1	2	3	0

P_4:	a	a	a	a	b	a	a	c	d
	0	1	2	3	0	1	2	0	0

Working of KMP algo -

Make a PI table / LPS for pattern;

	0	1	2	3	4	5
Pattern:		a	b	a	b	d
	0	0	1	2	0	

← indexes

← occurrence

We start from $j = 0$ & $i = 1$

String: a b a b a d c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

if (pat[j+1] == txt[i]) i++; j++;

a b a b d
j @0

a b a b c a b c a b a b a b d
i

At this point, txt[i] != pat[j+1]

When there is a mismatch, we'll make j to index 2, i.e. first occurrence where it occurred, we'll determine this using PI table.

j is at index 2

Now, a b a b d
j

a b a b c a b c a b a b a b d
i

Again mismatch, we'll again move j to index 0

Now, a b a b d
j @0

Now, pat[j+1] == txt[i] move both

Now, a b a b c a b c a b a b a b d
i

a b a b d
j

Again mismatch, move j to 0

i

a b a b c a b c a b a b a b d

j a b a b d

pat[j+1] == str[i] , move both.

a b a b c a b c a b a b a b d

i i i i

a b a b d

j j j j

At this point j+1 ≠ i

So we move j back to index 2

i

a b a b c a b c a b a b a b d

a b a b d

j

Now both will match till the end.

This is advantage of this algo, as we have already compared a b, earlier then we don't need to compare it again.

TC - O(n+m)

Algorithm to Create LPS → (Prefix array)

pat = "AAACAAAA"

0	1	2	3	4	5	6	7
0							

len = 0, i = 0

lps[0] is always 0

increment i

len = 0, i = 1

pat[len] == pat[i], do len++

Store in lps[i] and do i++, len = 1 i = 2

lps[1] = 1

len = 1, i = 2

pat[len] == pat[i] ,

len = 2, lps[2] = 2, i = 3

Now, len = 2, i = 3

pat[len] != pat[i] & len > 0

len = lps[len - 1] = lps[1] = 1

This way we proceed

Code →

```
computeLPS array {
  int len = 0;  lps[0] = 0;
  int i = 1;
  while (i < M) {
    if (pat[i] == pat[len]) {
      len++; lps[i] = len; i++;
    }
    else {
      if (len != 0) len = lps[len-1];
      else {
        lps[i] = 0;
        i++;
      }
    }
  }
}
```

KMP Code →

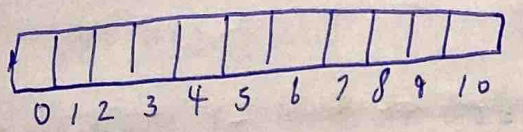
```
int i, j = 0
while (N - i >= M - j) {
  if (pat[j] == txt[i]) i++, j++;
  if (j == M) { Pattern found at index i - j
                j = lps[j-1]; }
  else if (i < N && pat[j] != txt[i]) {
    if (j != 0) j = lps[j-1];
    else i = i+1;
  }
}
```

Z-algorithm Pattern Matching -

Z-function is a function which calculates z-array

z[i] denotes max length of prefix array which is same as substring originating from i

e.g. 1 2 3 4 5 6 7 8 9 10 11
a b c d a b c d a b c

z → 

To compute z[i], we will compare str[i] with str[j] where j is the starting of string. We will keep incrementing both i & j until they match. At last we'll store the length that matched in the z-array.

This Z-algorithm finds all occurrences of a pattern in a text in linear time.

T.C - O(n+x)

What is Z array? → An element Z[i] stores length of the largest substring starting from str[i] which is also a prefix of str[0...x-1].

First entry of z array is meaningless as complete string is always a prefix of itself

Eg- str = "a a a a a a a"
z[] = {X, 5, 4, 3, 2, 1}

str = "a a b a a c d"
z[] = {X, 1, 0, 2, 1, 0, 0}

How to construct z array?

Approach 1- Run 2 nested loops, outer loop goes to every index and inner loop finds length of the longest prefix that matches the substring starting at the current index.

TC - $O(n^2)$

z-array explained →
Our main task is to tell what the z-array is.
Let's understand it

z array stores the length of longest substring, starting from current index, that is also its prefix.

This means that if each index stores the no. of characters matching the starting character, comparing from this index.

This ~~was~~ implies that if z[i] = k
Then k characters after index i match first k characters of the string.

E.g - a a b a a c d

z[0] = X, whole array is a prefix

z[1] = 1, it means 1 character from index 1, matches first 1 elements, which is true

z[2] = 0, it means 0 character from index 2, matches first 0 elements

z[3] = 2, it means 2 characters from index 3, matches first 2 elements

z[4] = 1, it means 1 characters from index 4, matches first 1 element

z[5] = z[6] = 0

Better approach

pat = "aab"
str = "caab x aaaab"

[L, R] initially [0, 0]

Before generating z-array concat 2 strings & name it as txt

```
          0 1 2 3 4 5 6 7 8 9 10 11 12
txt = "aab $ caab x aaaab"
z[] = [-1 0 0 0 3 1 0 0 2 3 1 0]
```

z[0] is always empty. So, we start from index 1

z[1] = 1, as only 1 character 'a' matches with starting character 'a', Here interval - [0,0]

z[2] = 0, no match

z[3] = 0 & z[4] = 0, no match

z[5] = 3, as 3 characters match till index 5.
Here window becomes [5, 7]

This interval [L,R] is prefix substring.

Steps for maintaining this interval -

1) if (i > R), there is no prefix substring that starts before i and ends after i, so reset [L, R] & z[i] = R-L+1

2) if (i <= R), then K = i - L, now z[i] = min(z[K], R-i+1)
because str[i] matches with str[K] for atleast R-i+1 characters

Two sub-cases :-

a) If (z[k] < R-i+1), then there is no prefix substring starting at str[i] so z[i] = z[k] and interval remains same

b) if (z[k] >= R-i+1), then it is possible to extend [L, R] interval, thus set L = i and start comparing str[R] onwards and get new R.

Now, at z[6] = 1 , here interval is [5, 7]

i < R , i.e. 6 < 7

K = 6 - 5 = 1

z[1] = 1 7 - 6 + 1 = 2

min(1, 2) = 1

And in this case, we'll also update interval just like discussed earlier.

This way we'll create z-array

T.C - O(m + n) to generate z array

Then to search for pattern in text, it takes O(1) time

Code →

```
int L=0, R=0, k;
for (i=1 → len)
{
 if (i > R) {
     L=R=i;
     while(R<len && str[R-L] == str[R]) R++;
     z[i] = R-L;
     R--;
  }
 else {
   k = i-L;
   if (z[k] < R-i+1) { z[i] = z[k]; }
   else {
     L=i;
     while (R<len && str[R-L] == str[R]) R++;
     z[i] = R-L;
     R--;
   }
 }
}
search (pat, txt) {
getZarr (pat + "$" + txt, z);
for(i = 0 → len) if (z[i] == pat.length()) Found at i - pat.length()
```

Minimum Characters required to make a String palindrome

We are only allowed to insert characters at the beginning of the string.

str = "ABC"

O/P → 2

"CBABC" Underlined chars added

str = "AACECAAAA"

O/P → 2

"AAAACECAAAA"

Approach-

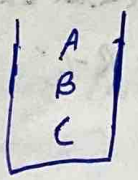
The first approach which I am thinking of

Is to use a stack,

We will take a pointer i which points at last index of the str.

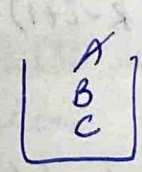
We'll push char in stack & do i--

"ABC"



Now, when i reaches mid point, we'll start popping items from stack.

"ABC"



We'll start popping, when i reaches start of str

or str[i-1] == str[i+1]

Brute Force - Check each time whether the string is a palindrome or not, if not then delete last character and check again.

TC - $O(n^2)$

Efficient approach - We can use lps array of KMP algo.

First concat the given string, $, reverse of string

Then generate lps array for generated string.

E.g - AACECAAAA

Concat - AACECAAAA$AAAACECAA

LPS[] = {0 1 0 0 3 1 2 2 2 0 1 2 2 2 3 4 5 6 7}

Last value of lps array shows the longest suffix of the reversed string that matches the prefix of the original string.

So, min no. of characters = length of input - LPS[mn] (last)

Generate lps code →

```
int len = 0;
lps[0] = 0;
for(i = 1 → end) {
    if(str[len] == str[i]) { len++; lps[i] = len; i++; }
    else { if(len != 0) len = lps[len - 1];
           else { lps[i] = 0; i++; }
    }
}
```

TC - $O(N)$ SC - $O(N)$

2 pointer approach -

- Initialize 2 pointers start & end to beginning and end of the string, respectively.
- While (start < end) - if s[st] == s[end] { st++; end--; }

If characters not equal res++; & reinitialize both pointers to start = 0 & end = n - res - 1;

TC - $O(N)$ SC - $O(1)$

~~Check for anagrams~~ →

Count and Say →

First write count then the number

3 3 | 2 2 2 | 5 | 1

2 3 | 3 2 | 1 5 | 1 1 ← Ans

In the question they will give n, and we have to solve it for n^{th} value sequence.

If n = 1, then "1" → one 1

If n = 2, then "11" → two 1

If n = 3, then "21" → one 2 one 1

If n = 4, then "1211" →

Approach-

Start with "1", build it N-1 times

build means: 1) break string into same 'num' subarrays, push freq(num), 'num' in result

We'll break string into subarrays of same number
Then in result string, push frequency then push number

e.g. - "21"

sub 1 | sub 2

One 2 in this subarray | one 1 in this subarray

"1211" → "111221"

again subarrays

Pseudo code -

```
string initial = "1";

string build (string num) {
  int freq = 1;
  char cur = num[0];
  for (int i = 1; i < num.length(); i++) {
      if (cur == num[i]) freq++;
      else {
      char frequency = '0' + freq;
      result.push_back (frequency);
      result.push_back (cur);

      cur = num[i];
      freq = 1;
    }
  }
return res;
}

  for (int i = 0 → N-1) {
    initial = build (initial);
  }
return initial;
```

TC - O(N * M)
SC - O(N * M)

Compare Version Numbers →

version 1 = "1.01"
version 2 = "1.001"

If v1 == v2 return 0
v1 > v2 return 1
v1 < v2 return -1

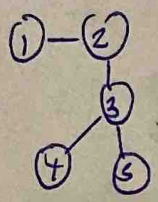
We can convert these chars into numbers and then compare.

Graphs

Undirected graph

Directed graph

Path - A node cannot appear twice in a path



1 2 3 5 ✓ Valid Path

1 2 3 2 1 X

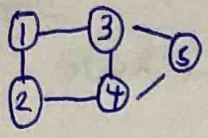
Degrees on Graph -

Degree of an undirected graph - No. of edges attached to it

Property → Total degree of Graph = 2 × Edges (Undirected graph)

Directed graph - Indegree & Outdegree

Representation of Graph



n=5

Adjacency Matrix →

We'll make a matrix of n+1 × n+1

adj[n+1][n+1]

Space → (N × N)

(costly in space)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	1	0	0
2	0	1	0	0	1	0
3	0	1	0	0	1	1
4	0	0	1	1	0	1
5	0	0	0	1	1	0

~~adj[u].push_back(v);~~
~~adj[v].push_back(u);~~

OR adj[u][v] = 1
adj[v][u] = 1

Adjacency List →

Create an array of n+1 size

. This array will contain list

adj[u].push_back(v); which means it will contain
adj[v].push_back(u); another array

0
1 → {2, 3}
2 → {1, 4, 5}
3 → {1, 4, 5}
4 → {2, 5, 3}
5 → {3, 4}

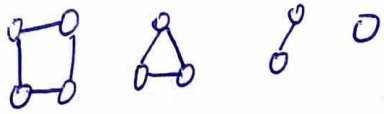
vector<vector<int>> adj(n+1);

Space complexity - O(2E)

(where E is no. of edges)

2E because this is a undirected graph

Connected Components



These are 4 disconnected components of a graph.

These are 4 different components of a single graph.

$N = 10$ $E = 8$

So, to traverse these disconnected components, we will always use a visited array.

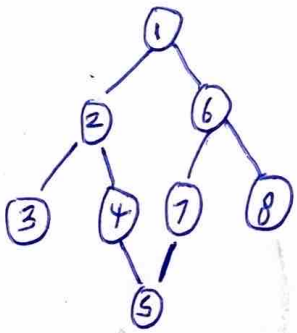
Here, we have 10 nodes, we'll create a visited array of size n+1, i.e. 11

And we will always run our loop from

```
for (i = 1 → 10) {
    if (!vis[i]) {
        traversal(i); }
}
```

This traversal algo will mark everyone as visited

BFS of a Graph → (Breadth First Search)



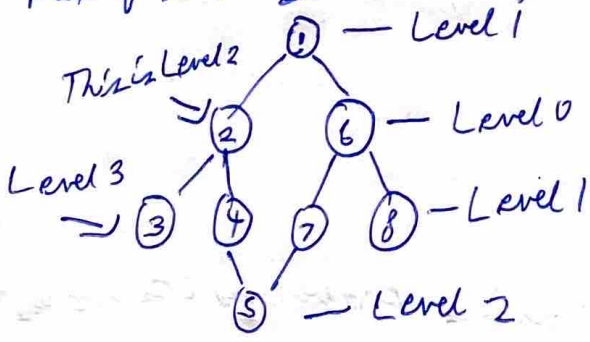
1-based graph

In BFS, we visit Level-Wise

If starting node = (1), then BFS will be level wise

1 | 2 6 | 3 4 7 8 | 5

But if starting node = (6) then that will level 0



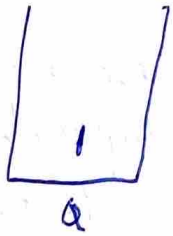
So, BFS will be

6
1 7 8
2 5
3 4

We take a Queue to perform BFS traversal and take a visited array

Starting node = 1, push this in queue & mark it as visited

vis array of size 10



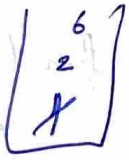
vis: 0 1 0 0 0 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9

Now, take out top from queue

node = 1, now we will check adjacency list of

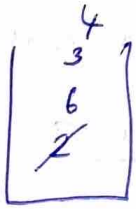
1. 1 → {2, 6}



Push 2, 6 in queue

When we push them in queue, we also mark them as visited

vis: 0 1 1 0 0 0 1 0 0 0



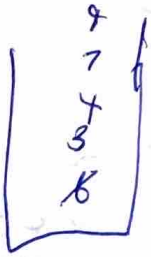
Now node = 2

check their neighbours and push in graph.

2 → {3, 4}

Mark visited

0 1 1 1 1 0 1 0 0 0



Now node = 6

6 → {7, 8}

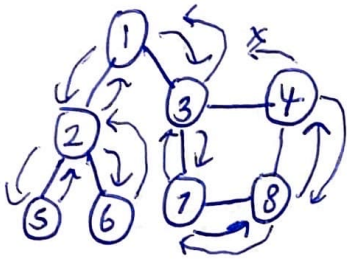
Push them in queue and mark as visited

SC- $O(3*N) \approx O(N)$

TC- $O(N) * O(2E)$

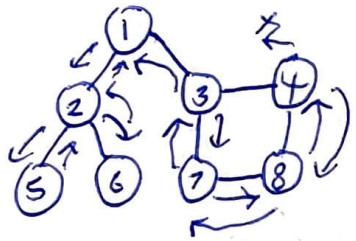
This because each node is traversing its edges

DFS Traversal in a Graph (Depth First search)



Starting node = 1

DFS → 1 2 5 6 3 7 8 4



starting node = 3

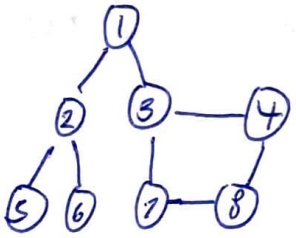
We can go on any node from 3, its our choice

We'll go to 7, then 8, then 4

3 7 8 4 , now it will come back, we'll go to 1

3 7 8 4 1 2 5 6

As we know, if we are going till depth of a node, we'll be using recursion.



Adjacency List →

1 → {2, 3}
2 → {1, 5, 6}
3 → {1, 4, 7}
4 → {3, 8}
5 → {2}
6 → {2}
7 → {3, 8}
8 → {4, 7}

Now, starting node = 1,
Create a vis array of n+1 size

vis	0	1	0	0	0	0	0	0	0	
	0	1	2	3	4	5	6	7	8	9

Mark starting node as vis = 1

Call a recursive dfs function from starting node

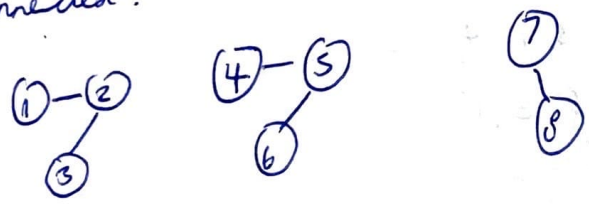
```
dfs (node){
    vis[node] = 1;
    ans.add(node)
    for (auto it : adj[node]) {
        if (!vis[it])
            dfs(it);
    }
}
```

SC - O(N) + O(N) + O(N)

TC - O(N) + O(2 × E) ← This is only for a undirected graph. This specifies that for each node we are visiting its degree.

Number of Provinces

A graph is given, we have to count the number of provinces. A province is if its components are / nodes are connected.



O/P → 3

But Input is given in a matrix format i.e.

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

⇒ n × n matrix

isConnected[i][j] == 1, if ith city & jth city are interconnected

otherwise isConnected[i][j] = 0.

Approach:

First we will create an adjacency list by traversing the matrix.

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Whenever it is 1, we'll push i & j in the list.
We'll push $i+1$ & $j+1$.
A node is obviously connected by itself so we don't need to check those cases where $i == j$.

At $i=0, j=0$, it is 1, skip this because $i == j$
At $i=0, j=1$, it is 1, map → adj[1].push_back(2)
At $i=0, j=2$, it is 0, move on
At $i=1, j=0$, it is 1, adj[2].push_back(1)
~~Now in 3rd row we have no one, so push adj[3].push_back(0)~~

Then after creating adjList, we'll simply do any traversal, to find no. of components of a graph.

TC - O(N) + O(V+E)
SC - O(N)

Number of Islands -

A $n \times n$ matrix is given, where 1 represents land 0 represents water.
Return the no. of islands
An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.
All 4 edges of the grid are ~~sup~~ surrounded by water.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

O/P → 1

Approach -

This ques is similar to prev question in which we had to find connected components using BFS or DFS.

Assume all 1's which are connected as a single component, and everytime we do BFS on it we increment the count by 1.

Here we will not create an adjacency List separately and use this matrix for determining connected nodes.

We will use BFS traversal.

Create a 2-D visited array

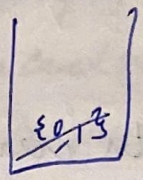
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

matrix

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Starting node → {0, 1}

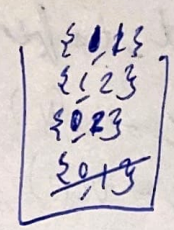
Push in queue



Now go in all 8 directions of this index in the matrix, if we find 1, push it in queue

{0,2} = 1
{1,2} = 1
{1,1} = 1

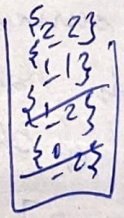
Push these 3 in queue



Now, pick {0,2}, check all its neighbours and push in queue. Here we find {1,2} & {1,1}

But they are already visited, so no need to push them again.

Now, {1,2}



Check its neighbours

{2,2} = 1. Push it

Now, {1,1} & {2,2} don't have any further neighbours

We pop them one by one.

Now queue becomes empty. This means in 1 traversal we were able to traverse all adjacent lands.

Now, starting node = {4, 0}. Follow the same process.

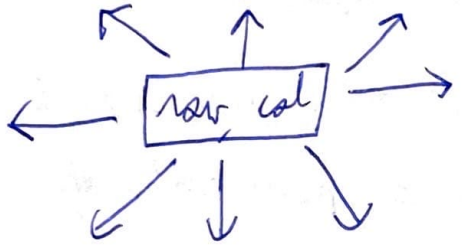
So, apparently we got 3 starting points, i.e. we made the queue empty 3 times.

So, Total Islands = 3

Q. How to decide starting point?

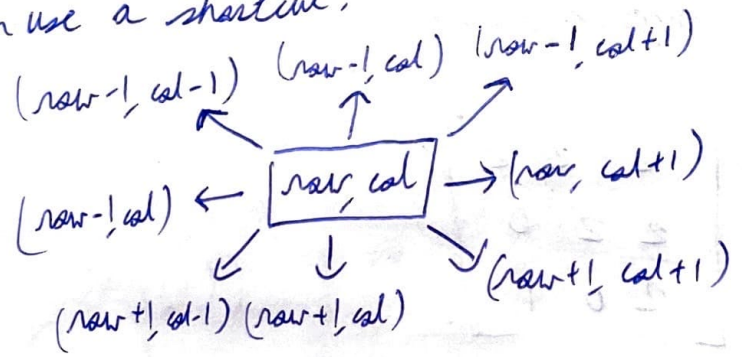
Ans - Start traversing the matrix, if it is a land and it is marked as unvisited. We perform BFS traversal.

Now, how to traverse neighbours?



Either we can compare all 8 neighbours by comparing 8 different lines.

Or, we can use a shortcut.



So, everything varies from -1 to +1,

This is the best way to traverse neighbours

```
for (delta row → -1 to +1)
    for (delta col → -1 to +1)

        neighbour row = row + delta row
        neighbour col = col + delta col;

    }
}
```

If we want to traverse 4 directions, then we can make a vector<pair<int,int>> directions = {{-1,0},{1,0},{0,-1},{0,1}};

Now, while traversing

```
new_row = row + directions[i].first;
new_col = row + direction[i].second;
```

$SC - O(N^2) + O(N^2)$

vis queue

$TC - O(N^2) \times 8$

Flood Fill Algorithm

A image is represented by m x n integer grid

Where image[i][j] represents pixel value of the image.

Also given 3 integers sr, sc, and color.

Perform a flood fill on the image starting from the pixel image[sr][sc]

E.g -

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix}$$

sr=1, sc=1, color=2

Approach - Question says whatever is connected to 1 and has the same color. Color.

Change their color.

Initially change grid[sr][sc] color to 2,

Then check with 4 direction. whatever has same color

go in that direction & color those nodes too.

As we notice that if we are going to one node which has same color, we'll further check for nodes who has same colors after this node.

Meaning we are going to depth.

So, we'll use DFS or BFS.

E.g -

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 0 \\ 2 & 2 & 2 \end{bmatrix}$$

Starting here

newColor = 3 sr = 2 & sc = 0

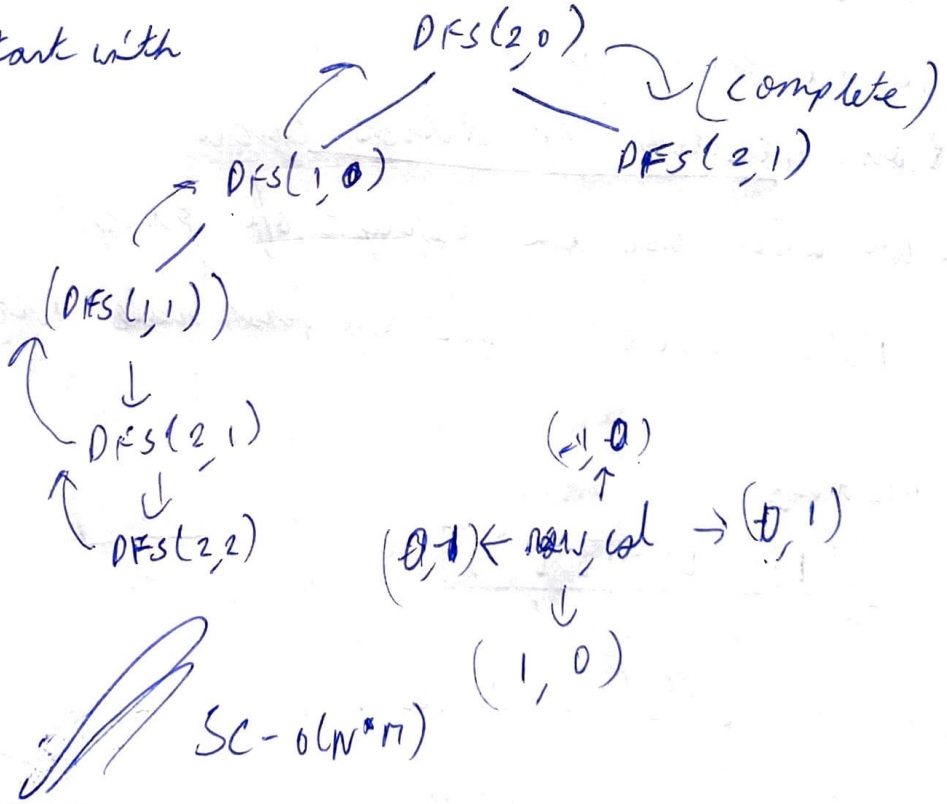
We'll make grid[sr][sc] = 3

Go to adjacents which have color = 2.

Make them 3.

We can either do this task in place, or we can copy this data in some other matrix.

Start with

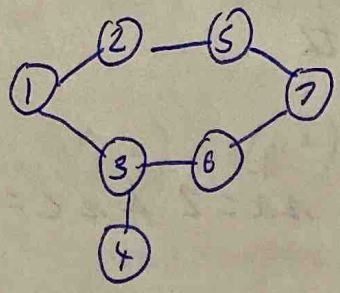


SC - O(N*M)

Detect Cycle in an Undirected Graph

Using BFS →

If we come back to the same node while we traverse then we can say that there is a cycle



adj List

1 → {2, 3}
2 → {1, 5}
3 → {1, 4, 6, 7}
4 → {3}
5 → {2, 7}
6 → {3, 7}
7 → {5, 6, 3}

Node ① is at Level 0

In BFS, from node ①, we go to ② & ③ which are at Level 1

From ②, we go to ⑤
From ③, we go to ⑥, ④] These are in Level 2

From ② & ③, we go to node ⑦, which is at Level 3.

As we can see, we diverged from ①, and here at ⑦, we are converging again.

This is our intuition to detect cycle.

So, here we will use the concept of parent node.

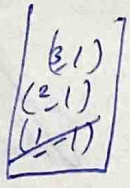
Approach → Make a queue & insert the first node with the parent.

Create a vis array

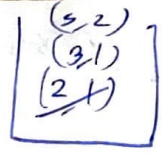
0	1	0	0	0	0	0	0
0	1	2	3	4	5	6	7

Take 1 out, check its neighbours i.e. 2 & 3

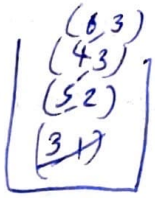
Push (2,1) & (3,1) in queue



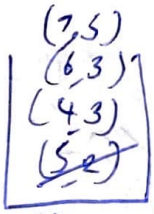
Now, node = 2, insert its neighbours along with parent as 2



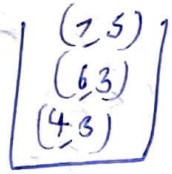
Now, node = 3, insert its neighbours



Now, node = 5, perform same procedure



Now, node = 4 & then node = 6



Now at node 6, its neighbour is 7, & and we have already marked 7 as visited in the array.

So, yes now we can say that it has a cycle.

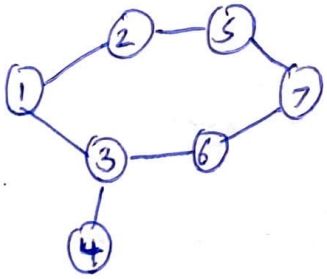
Write same code as we wrote for traversing disconnected components using BFS, then, add a else if condition if node is visited if (parent != neighbour) return true. It means a cycle exists.

TC - Σ(degrees) + O(N) nodes

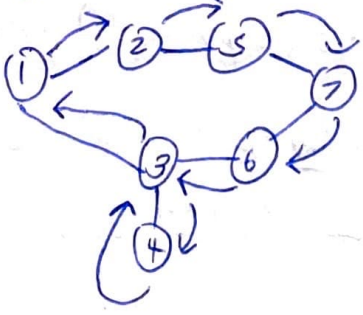
$\approx$ O(2E + N)

SC - O(N)

Using DFS -

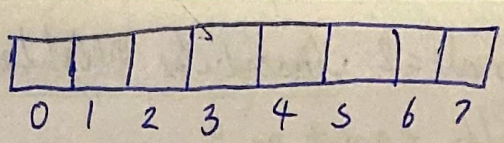


The flow of execution of a DFS traversal will be different than BFS. BFS travels Level wise, whereas DFS travels till depth



Now, if it reaches back to 1 we can say a cycle exists.

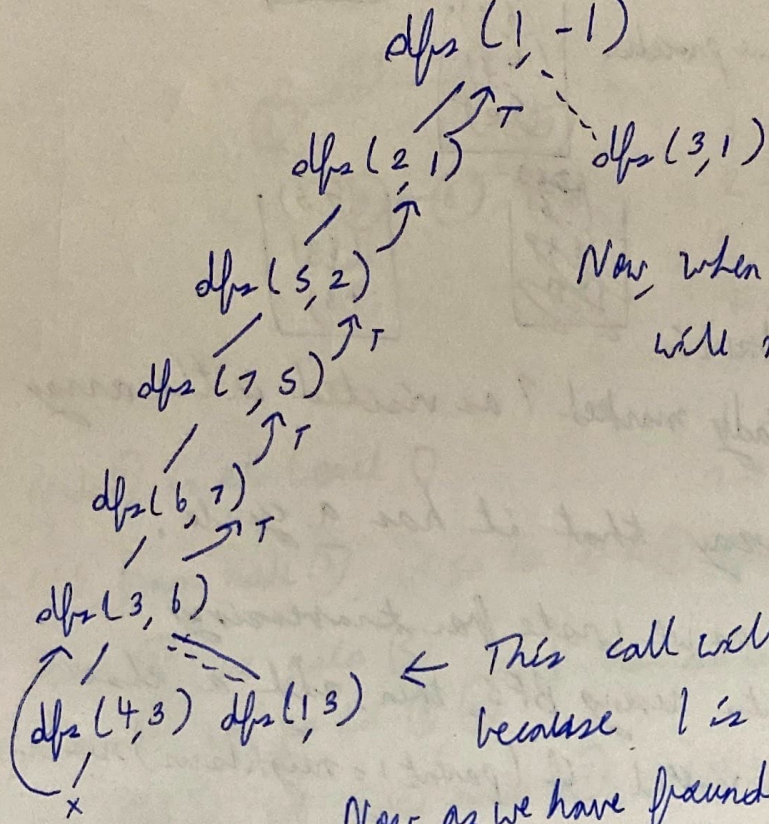
Make a visited array.



Initially, node = 1, parent = -1

We will also use a parent array.

We will also update parent while making recursion call



Now, when it gets True, it will not make further calls

← This call will not be made because 1 is already visited

Now, as we have found someone who is already visited and it is not equivalent to parent, then

We will return true

```
dfs(node, parent){
  vis[node] = 1,
  for(auto it: adj[node])
    {
    if(!vis){
      if(dfs(it, node) == true) return True; }

    else if(it != parent) return true
```

$T(n) = O(N) + O(E)$

$\$ - O(N) + O(N) \approx O(N)$