INTERVIEW PROBLEMS ON DP - 1

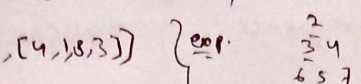
Q) TRIANGLE

Given a triangle array, return the minimum path sum from top to bottom.

For each step, you may move to an adjacent no. of the row below. More formally, if you are on index 'i' on the current row, you may move to either index i / i+1 on next row.

Ex: tri = [[2],[3,4],[6,5,7],[4,1,8,3]]

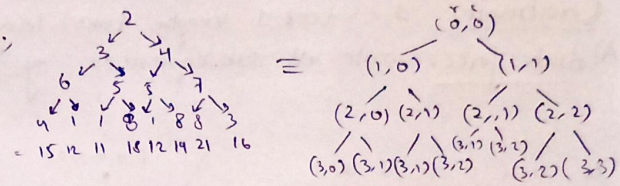
o/p = 11



* Constraints: 1 <= △.length <= 200

A) Approach: row → i^{th} col.

row+1 → i^{th} & $(i+1)^{th}$



$f(r, c)$ = grid[r][c] + min { f(r+1, c), f(r+1, c+1) }

(curr row, curr col)

↳ returns min cost to reach the last row from r^{th} row & c^{th} col

, base case → if (r == n-1)

repetetive

Eg to store values, we need 2D array!

Top-down:

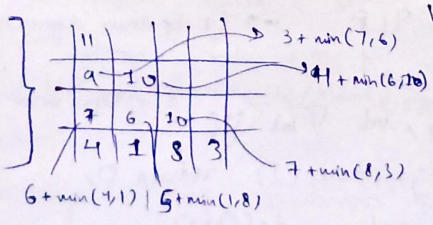
```
vector<vector<int>> grid;
vector<vector<int>> dp;
int f(int r, int c){
    if(r == grid.size()-1) return grid[r][c];
    if(dp[r][c] != -1) return dp[r][c];
    return dp[r][c] = grid[r][c] + min(f(r+1,c), f(r+1,c+1));
}
```

```
int minTotal(vector<vector<int>>& triangle){
    dp.resize(205, vector<int>(205,-1));
    grid = triangle;
    return f(0,0);
}
```

↳ resize to 205 rows & cols {constraint ingive.} → '-1'

Bottom-up:

Eg: 0 → 2
1 → 3 4
2 → 6 5 7
3 → 4 1 8 3



Logic:

dp[r,c] = grid[r][c] + min(dp[r+1][c], dp[r+1][c+1])

r = n-1 → n-2 → 0

(base case)

```
int minTotal(vector<vector<int>>& triangle){
    dp.resize(205, vector<int>(205,-1));
    grid = triangle;
    int rows = grid.size();
    for(int i=0; i<grid[rows-1].size(); i++) dp[rows-1][i] = grid[rows-1][i];   // base case
    for(int r=rows-2; r>=0; r--){
        for(int c=0; c<grid[r].size(); c++){
        }; dp[r][c] = grid[r][c] + min(dp[r+1][c], dp[r+1][c+1]);
    }
    return dp[0][0];
}
```

Q) LONGEST COMMON SUBSEQUENCE

Given 2 string text1 & text2, return length of their longest common subsequence. If there is no common subsequence, return 0.

{ subsequence of a string is a new string generated from original string with some characters (can be none) deleted without changing the relative order of the remaining characters }

ex: "ace" is subsequence of "abcde".

eg: t1 = "abc", t2 = "abc" → o/p = 3

t1 = "abc", t2 = "def" → o/p = 0

Constraints: 1 <= text1.length, text2.length <= 1000

A) Brute force: write all subsequences of 'a' & 'b' (strings) & compare

ex: a c c a b a c / b b a b c

f(a,b,0,0)

f(a,b,i,j) → longest common subsequence btw a[i,n-1] & b[j,n-1]

= f(a,b,i+1,j), f(a,b,i,j+1) } max ; if a[i] != b[j]

~~f(a,b,i+1,j+1)~~

~~f(a,b,i,j)~~ → will lead to ∞ loop

1 + f(a,b,i+1,j+1) → else

f(a,b,i,j): a[i] ✓ / b[j] ✓

f(a,b,i+1,j+1) / f(a,b,i+1,j)

a[i] ✓ / b[j+1] ✓ / a[i+1] / b[j] ✓

f(a,b,i+2,j+2) / f(a,b,i+1,j+1) / f(a,b,i+1,j+1) / f(a,b,i+2,j+1)

(already covered)

base case: when 1 string runs out of size!

Code → Using dp → to avoid TLE → (i top down approach)

```
vector<vector<int>> dp;   // if pass by value making copy takes time → TLE
int f(string &a, string &b, int i, int j){
    if(i==a.size() or j==b.size()) return 0;
    if(dp[i][j] != -1) return dp[i][j];
    if(a[i]!=b[j]) return dp[i][j] = max(f(a,b,i+1,j), f(a,b,i,j+1));
    else return dp[i][j] = 1 + f(a,b,i+1,j+1);
}
int subsequence(string a, string b){
    dp.resize(1005, vector<int>(1005,-1));
    return f(a,b,0,0);
}
```

dp[i,j]: longest common subsequence of a[i,n-1] & b[j,n-1]

Bottom-up approach → we need order of execution!

ex: A = a b c, B = a b c

dp[i,j] → a[i,n-1] / b[j,n-1]

represents 'a' there in t1 but in 't2' → its exhausted so we are writing 0

a[1,n-1] → a(4,4)) (c,e) → (c,) / (,e) ; b(2,n-1) → b(2,2))

(4,3)

max((4,3), (5,2))

f(a,b,4,2) → f(a,b,4,3) / f(a,b,5,2)

a[4,4) / b[1,2] → e,bc() : f(a,b,4,1)

max(f(a,b,4,2), f(a,b,5,1))

∴ logic same as prev. one!

Code:

```
vector<vector<int>> dp;
int subsequence(string a, string b){
    dp.resize(1005, vector<int>(1005,0));
    int n = a.size();
    int m = b.size();
    for(int i=n-1; i>=0; i--){
        for(int j=m-1; j>=0; j--){
            if(a[i] != b[j]){
                dp[i][j] = max(dp[i+1][j], dp[i][j+1]);
            }else{
                dp[i][j] = 1 + dp[i+1][j+1];
            }
        }
    }
    return dp[0][0];
}
```

Q) LONGEST INCREASING SUBSEQUENCE

Given an array int -> nums, return length of longest strictly increasing subsequence.

ex: nums = [10, 9, 2, 5, 3, 7, 101, 18] → o/p: [2,3,7,101] is longest subsequence.

o/p: 4

Constraints: 1 <= nums.length <= 2500, -10^4 <= nums[i] <= 10^4

A) ex: [10, 9, 2, 5, 3, 7, 101, 18] (i) (j)

0 1 2 3 4 5 6 7

f(i) = 1 + max(f(j)) : i: (0, n-1); (j: (0, i-1), a[j] < a[i])

length of longest subsequence ending at index i

Code: (top-down)

```
vector<int> arr;
vector<int> dp;
int f(int i){
    if(i==0) return 1;
    int mx = INT_MIN;
    if(dp[i] != -1) return dp[i];
    for(int j=0; j<=i-1; j++){
        if(arr[j] < arr[i]){
            mx = max(mx, f(j));
        }
    }
    if(mx == INT_MIN) return dp[i] = 1;
    return dp[i] = 1 + mx;
}
int lengthOfLIS(vector<int> &nums){
    arr = nums;
    dp.resize(2505, -1);
    dp[0] = 1;
    int ans = INT_MIN;
    for(int i=0; i<nums.size(); i++){
        ans = max(ans, f(i));
    }
    return ans;
}
```

Code: (bottom-up)

```
int lengthOfLIS(vector<int> &nums){
    arr = nums;
    dp.resize(2505, -1);
    dp[0] = 1;
    int ans = INT_MIN;
    for(int i=0; i<nums.size(); i++){
        for(int j=0; j<=i-1; j++){
            if(arr[j] < arr[i]){
                dp[i] = max(dp[i], 1+dp[j]);
            }
        }
        if(dp[i] == -1) dp[i] = 1;
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

2) MATRIX CHAIN MULTIPLICATION

Given the dimension of a sequence of matrices in array arr[], where the dimension of the i^{th} matrix is (arr[i-1] * arr[i]), the task is to find the most efficient way to multiply these matrices together such that the total no. of element multiplications is minimum.

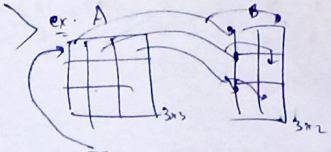
ex: arr[] = {40, 20, 30, 10, 30}
o/p: 26000

ex. 4 matrices = 40×20 (A), 20×30 (B), 30×10 (C), 10×30 (D)

Min. no. of multiplications = (A(BC))D

∴ Minimum is: 20*30*10 + 40*20*10 + 40*10*30

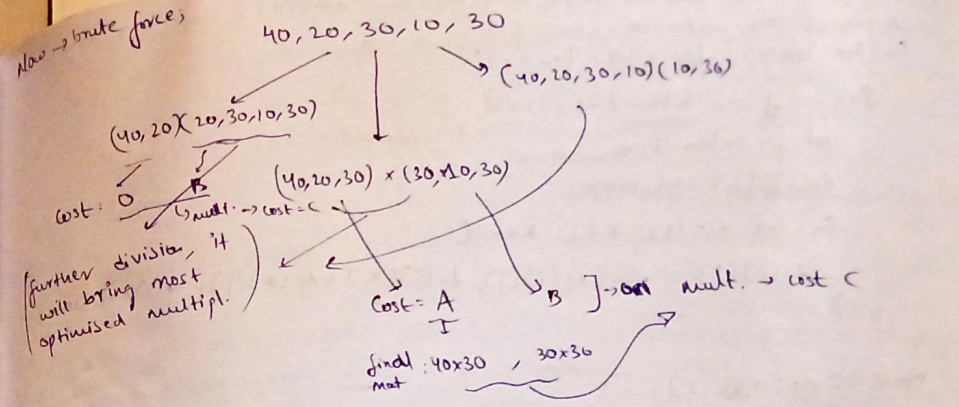
A) We will find no. of multiplication btw 2 matrices



∴ Ans = 3+3 + 3+3 + 3+3 = 18 = 3×3×2
(for 1st row) (2nd) (3rd)

∴ $A_{n\times m} \cdot B_{m\times k}$ ⟶ ∴ total no. of multiplications = n×m×k

Now → brute force:

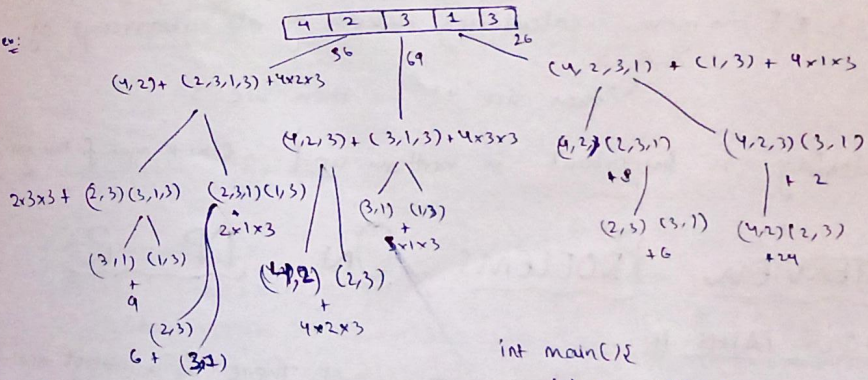


$\therefore f(arr,i,j) = \min(f(arr,i,k) + f(arr,k,j) + arr[i]*arr[j]*arr[k])$ $\forall\ k \in [i+1, j-1]$

↓ min. cost to multiply all matrices in the array [i,j]

↘ $f(arr, 0, n-1)$

ex:



Code: (TD)

```
vector<vector<int>> dp;
int f(int i, int j, vector<int> &arr){
    if(i==0 or i+1==j) return 0;   // matrix doesn't exist / only 1 matrix
    if(dp[i][j] != -1) return dp[i][j];
    int ans = INT_MAX;
    for(int k=i+1; k<j; k++){
        ans = min(ans, f(i,k,arr) + f(k,j,arr) + arr[i]*arr[k]*arr[j]);
    }
    return dp[i][j] = ans;
}
int main(){
    int n; cin>>n;
    vector<int> v(n);
    dp.resize(1005, vector<int>(1005,-1));
    for(int i=0; i<n; i++) cin>>v[i];
    cout<<f(0, n-1, v);
    return 0;
}
```

Code (BU) → Inside int main → (prev. one written!)

```
dp.resize(100, vector<int>(100,0));
for(int len=3; len<=n; len++){
  for(int i=0; i+len-1<n; i++){
    int j = i+len-1;
    dp[i][j] = INT_MAX;
    for(int k=i+1; k<j; k++){
      dp[i][j] = min(dp[i][j], dp[i][k]+dp[k][j]+ v[i]*v[k]*v[j]);
    }
  }
}
cout<<dp[0][n-1];
```

storing → subarray size

⇒ 40, 20, 30, 10, 30

to calculate via 'BU', we need all of its small subarrays answer calculated beforehand. So len=5 (then it must have '2, 3, 4' length subarrays answer)

len = 3, 4, 5 → means calculating answer of all subarrays of size '3' (then size '4' → then size '5')

* Ordering is important in bottom-up! → (check again for this ques)

INTERVIEW PROBLEMS ON DP - 2

① UNIQUE PATHS II

You are given an m x n integer array grid. There is a robot located at top-left corner: grid[0][0]. It tries to move to bottom-right corner (grid[m-1][n-1]). It can only move either down or right at any point of time.

An obstacle as space marked as 1 / 0 → respectively in grid. A path that robot takes cannot include any square in the obstacle.

Return the no. of paths that robot can take to reach bottom-right corner.

{1 ≤ m, n ≤ 100}

ex:  , [[0,0,0], [0,1,0], [0,0,0]]

o/p: 2 → {↓↓→→}, {→→↓↓}

A) → , ↓ → f(i,j) = f(i,j+1) + f(i+1,j)

(no. of way to reach bottom right from i,j by either going right / down)

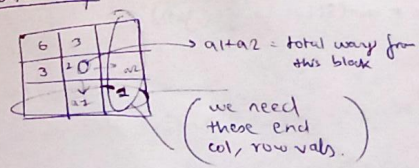
base case → out of bounds / route found / reached destination

Code: (TD)

```
int n, m;
vector<vector<int>> grid, dp;
int f(int i, int j){
  if(i==n-1 and j==m-1) return 1;
  if(i>=n or j>=m or i<0 or j<0) return 0;
  if(grid[i][j]==1) return 0;
  if(dp[i][j]!=-1) return dp[i][j];
  return dp[i][j] = f(i,j+1) + f(i+1,j);
}
```

```
int uniquePaths(vector<vector<int>>& obsGrid)
{
  grid = obsGrid;
  n = grid.size();
  m = grid[0].size();
  dp.resize(100, vector<int>(100,-1));
  if(grid[n-1][m-1]==1){
    return 0;
  }
  return f(0,0);
}
```

For bottom-up:

ex: 

Code

```
int uniquePath(...){
  dp[n-1][m-1] = 1;
  for(int i=n-2; i>=0; i--){
    if(obsGrid[i][m-1]==1) dp[i][m-1]=0;
    else dp[i][m-1] = dp[i+1][m-1];
  }
  for(int i=m-2; i>=0; i--){
    if(obsGrid[n-1][i]==1) dp[n-1][i]=0;
    else dp[n-1][i] = dp[n-1][i+1];
  }
  for(int i=n-2; i>=0; i--){
    for(int j=m-2; j>=0; j--){
      if(obsGrid[i][j]==1) dp[i][j]=0;
      else dp[i][j] = dp[i+1][j] + dp[i][j+1];
    }
  }
  return dp[0][0];
}
```

obstacle

② MINIMUM PATH SUM

Given m x n grid filled with non-ve int., find a path from top left to bottom right - which minimizes the sum of all no.'s along its path.

{Movement = down (or) right only}

ex: $\begin{bmatrix} 1 & 3 & 1 \\ 1 & 5 & 1 \\ 4 & 2 & 1 \end{bmatrix}$ → [[1,3,1], [1,5,1], [4,2,1]]

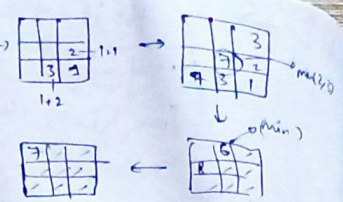
↳ o/p = 7

{1 ≤ m, n ≤ 200}

A)

ex: | 1 | 3 | 1 |
| --- | --- | --- |
| 1 | 5 | 1 |
| 4 | 2 | 1 |

→ I can calculate from bottom cell to top, each cost as I move a block



$\therefore f(i,j) = grid[i][j] + \min(f(i+1,j), f(i,j+1))$

(min cost to reach BR from TL

Code:

```
vector<vector<int>> dp,arr;
int n,m;
int f(int i, int j){
    if (i==n-1 and j==m-1) return arr[n-1][m-1];
    if(i>=n or j>=m or i<0 or j<0) return INT_MAX;
    If(dp[i][j]!=-1) return dp[i][j];
    return dp[i][j] = arr[i][j] + min(f(i+1,j), f(i,j+1));
}
```

```
int minPathSum(vector<vector<int>>&grid){
    arr=grid;
    n=arr.size();
    m=arr[0].size();
    dp.resize(205, vector<int>(205,-1));
    return f(0,0);
```

Code: (BU)

```
int minPathSum(--){
    dp[n-1][m-1] = arr[n-1][m-1];
    for(int i=m-2; i>=0; i--){
        dp[i][m-1] = arr[i][m-1] + dp[i+1][m-1];
    }
    for(int i=m-2; i>=0; i--){
        dp[n-1][i] = arr[n-1][i] + dp[n-1][i+1];
    }
    for(int i=n-2; i>=0; i--){
        for(int j=m-2; j>=0; j--){
            dp[i][j] = arr[i][j] + min(dp[i+1][j], dp[i][j+1]);
        }
    }
    return dp[0][0];
}
```

Q) Grid m×n → you're at top-left & reach b-r. How many paths possible. {1≤m,n≤100}

A) Code:

```
int m,n;
vector<vector<int>>dp;
int f(int i, int j){
    if(i==m-1 and j==n-1) return 1;
    if(i<0 || j<0 || i>=m || j>=n) return 0;
    if(dp[i][j]!=-1) return dp[i][j];
    return dp[i][j] = f(i,j+1) + f(i+1,j);
}
```

```
int uniquePaths(int M, int N){
    m=M;
    n=N;
    dp.resize(105, vector<int>(105,-1));
    return f(0,0);
}
```

Q) PARTITION EQUAL SUBSET SUM

Given an integer array 'nums', return true if you can partition the array into 2 subsets such that sum of elements in both subsets is equal or false otherwise. {1≤nums[i]≤100, 1≤nums.length≤200}

eg: ip: [1,5,11,5] op: True eg.→ Partion as: [1,5,5], [11]

A) $f(arr,i,k) = \begin{cases} f(arr,i+1,k-arr[i]) & \text{→ (include) if } (arr[i]\le k) \\ f(arr,i+1,k) & \text{→ (exclude)} \end{cases}$

update target

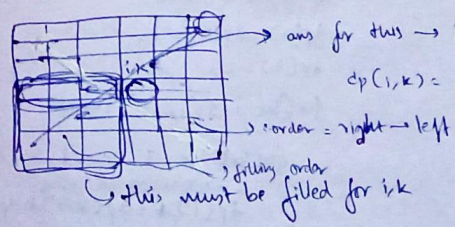
do we have a subset in the array [i..n-1] where sum = k

Code:

```
vector<vector<int>> dp;
bool f(vector<int> &arr, int i, int k){
    if(k==0) return true;
    if(i==arr.size()) return false;
    if(dp[i][k]!=-1) return dp[i][k];
    if(arr[i]<=k){
        return dp[i][k] = f(arr,i+1,k-arr[i])
                          or
                          f(arr,i+1,k);
    }else{
        return dp[i][k] = f(arr,i+1,k);
    }
}
```

```
bool part(vector<int>&nums){
    int s=0;
    for(int i=0;i<nums.size();i++){
        s+=nums[i];
    }
    if(s%2!=0) return false;
    dp.resize(205,vector<int>(20005,-1));
    return f(nums,0,s/2);
}
```

Bottom up:



Code:

```
bool part(--){
    int k=s/2;
    for(int i=0;i<n;i++) dp[i][0] = true; → base case
    for(int i=n-1;i>=0;i--){
        for(int j=1;j<=k;j++){
            if(nums[i]<=j){
                dp[i][j] = dp[i+1][j-nums[i]] or dp[i+1][j];
            }else{
                dp[i][j] = dp[i+1][j];
            }
        }
    }
    return dp[0][k];
}
```

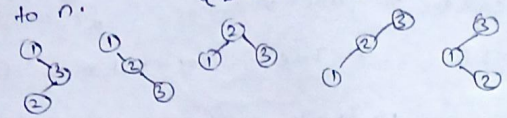
Q) UNIQUE BST

Given integer 'n', return the no. of structurally unique BST (binary search trees) which has exactly 'n' nodes of unique values from 1 to n.

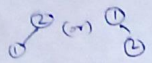
{1 <= n <= 19}

ex: if n = 3

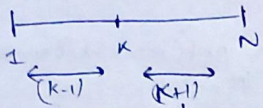
ans 5



A) n = 2 , n = 1 } → in n = 3 →  } it is similar to n = 2



∴ We have give every node a chance to become root.

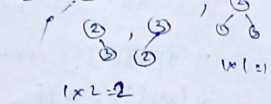


∴ $f(i)$ (i ∈ [1,N]) = $f(i-1) * f(n-i)$

$f(i-1)$ → no. of ways to make lst

$f(n-i)$ → no. of ways to make rst

∴ in how many ways we can get a bst with 'i' as the root & [1, i-1] → lst, [i+1, N] → rst

ex: 

∴ $f(n) = \sum (f(k-1) * f(n-k))$ * k ∈ [1,N]

↓ no. of unique bst with 'n' nodes having val [1→n]

n == 0 → 1
n == 1 → 1
n == 2 → 2

∴ Code:

```
vector<int> dp;
int f(int i){
    if(i==0 or i==1) return 1;
    if(i==2) return 2;
    if(dp[i]!=-1) return dp[i];
    int sum=0;
    for(int k=1; k<=i; k++){
        ↳ potential solution
        sum+= f(k-1)*f(i-k);
    }
    return dp[i]=sum;
}

int numTrees(int n){
    dp.resize(30,-1);
    return f(n);
}
```

Code:

```
vector<int> dp;
int numTrees(int n){
    dp.resize(30,n);
    dp[0]=dp[1]=1;
    dp[2]=2;
    for(int i=3; i<=n; i++){
        for(int k=1; k<=i; k++){
            k → potential root
            dp[i]+=dp[k-1]*dp[i-k];
        }
    }
    return dp[n];
}
```

* This recurrence is also called "Catalan Number"

INTERVIEW PROBLEMS ON DP-3

Q) REGULAR EXPRESSION MATCHING

Given an input string s & a pattern p, implement regular expression matching with support for '.' and '*' where:

→ '.' matches any single character

→ '*' matches zero or more of the preceding element

The matching should cover the entire input string (not partial)

ex: ip1: s = "aa", p = "a" } exp. "a" doesn't match the entire string "aa".
op1: false

ip2: s = "aa", p = "a*" } exp. '*' → zero/more of preceding ele. ∴ Repeating 'a' → "aa"
op2: true

ip3: s = "ab", p = ".*" } exp. ".*" mean "zero or more (*) of any character (.)"
op3: true

{1 <= s.length <= 20, 1 <= p.length <= 20}

A) 1st approach (right → left): $f(s,p,i,j)$ = return whether s[0,i] matches ← pattern from p[0,j]

$$f(s,p,i,j) = \begin{cases} f(s,p,i-1,j-1) & ; p[j]==\text{'.'} \ (\text{or}) \ p[j]==s[i] \\ f(s,p,i,j-2) & \\ (j-1==\text{'.'} \ (\text{or}) \ s[i]==p[j-1]) \ ? \ f(s,p,i-1,j) : \text{false} & \end{cases}$$

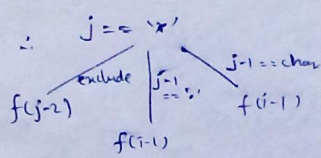
f(s, p, s.size()-1, p.size()-1)

ex: s → acbc (i), p → a*cb.. (j) (. can match with anyone) (or) s → acbc (i), p → a*b(c) (j)

; ex: c* → means c>=0, s = acbc (i), p → a*cbd* (j) } * means d>=0, (j-1)th doesn't match 's', so we can eliminate '*'

ex: s → acbc (i), p → a*cbc* (j), (j-1 == i)

∴ j == '*'



Base case: (i == -1 and j == -1) → true

(i < 0 or j < 0) → false

here '.' → can replicate any character

'*' → prev. character can occur >= 0 times!

2nd approach (left → right)

$f(s,p,i,j)$ = check if pattern from $s[i..n-1]$ & $p[j..n-1]$ matches or not

$$f(s,p,i,j) = \begin{cases} f(s,p,i+1,j+1) & \text{; if } s[i]==p[j] \text{ or } p[j]=='.' \\ f(s,p,i,j+2) & \text{; else if } (p[j+1]=='*') \text{ (skipping both chars)} \\ (s[i]==p[j] \text{ or } p[j]=='.')\ ?\ f(s,p,i+1,j) : \text{false} & \\ \text{false} & \leftarrow \text{else} \end{cases}$$

base case:

if (i==size() & j==size()) → true

if (j == p.size()) → false

ex: s = "a" (i) ; p = "b*" (j) (skip+2)

more than 1 iter: s = "aa" (i) ; p = "b*" (j) → (don't go to next iter. as i+1 might be same as 'i', so '.*' is helpful)

Code: (top-down)

```
vector<vector<int>>dp;
bool f(string s, string p, int i, int j){
    if (j==p.size()) return i==s.size();
    if(dp[i][j]!=-1) return dp[i][j];
    if(j+1<p.size() and p[j+1]=='*'){
        return dp[i][j]=f(s,p,i,j+2) or((i<s.size() and (s[i]==p[j] or p[j]=='.'))
                ? f(s,p,i+1,j) : false);
    }
    else if(i<s.size() and (p[j]=='.' or p[j]==s[i])){
        return dp[i][j]=f(s,p,i+1,j+1);
    }
    else{
        return dp[i][j]=false;
    }
}
bool isMatch(string s, string p){
    dp.resize(25,vector<int>(25,-1));
    return f(s,p,0,0);
}
```

Code: (Bottom-up)

```
bool isMatch(string s, string p){
    dp.resize(25, vector<int>(25,0));
    dp[s.size()][p.size()]=1;
    for(int i=s.size(); i>=0; i--){
        for(int j=p.size()-1; j>=0; j--){
            if(j+1<p.size() and p[j+1]=='*'){
                dp[i][j]=dp[i][j+2] or((i<s.size() and (s[i]==p[j] or p[j]=='.'))?
                        dp[i+1][j]:false);
            }else if(i<s.size() and (p[j]=='.' or p[j]==s[i])){
                dp[i][j]=dp[i+1][j+1];
            }else dp[i][j]=false;
        }
    }
    return dp[0][0];
}
```

if s = "" but 'p' still has characters, so to cover that case we are writing like this!

Q) HOUSE ROBBER III {337}

The thief has found himself a new place for his thievery again. There is only 1 entrance to this area called 'root'. Besides the root, each house has only 1 parent house. After a tour, the thief realized that all houses in this place form a binary tree. It will automatically contact the police if 2 directly linked houses were broken into on the same night.

Given root of binary tree, return max amount of money the thief can rob without alerting the police.

ex: i/p: [3,2,3,null,3,null,1] o/p: 7 } exp: 3+3+1 = 7

A) A node is robbed when it's parent is not robbed!

∴ f(node, isParentRobbed) = (returns max profit by robbing subtree rooted at node.)

- node == NULL → 0
- isParentRobbed = true → f(node→left, false) + f(node→right, false)
- max(node→val + f(node→left→left, true) + f(node→right, true) → else ; f(node→left, false) + f(node→right, false))

Code:

```
unordered_map<TreeNode*, int> robbed;
unordered_map<TreeNode*, int> notrobbed;
int f(TreeNode* node, bool isParentRobbed){
    if(node==NULL) return 0;
    if(isParentRobbed){
        if(robbed.count(node)>0){
            return robbed[node];
        }
        int ans = f(node->left, false) + f(node->right, false);
        return robbed[node]=ans;
    }
    else{
        if(notrobbed.count(node)>0){
            return notrobbed[node];
        }
        int c1 = node->val + f(node->left, true) + f(node->right, true);
        int c2 = f(node->left, false) + f(node->right, false);
        return notrobbed[node]=max(c1,c2);
    }
}
int rob(TreeNode* root){
    return f(root, false);
}
```

Q) MAXIMUM RECTANGLE

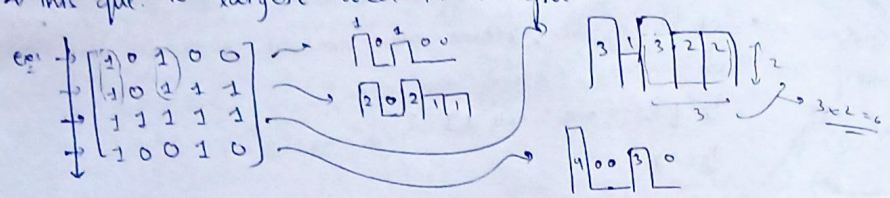
Given rows x cols binary matrix filled with 0's & 1's, find the largest rectangle containing only 1's & return its area.

Ex: matrix = [['1','0','1','0','0'], ['1','0','1','1','1'], ['1','1','1','1','1'], ['1','0','0','1','0']]

Op: 6

A) This que. ~ "largest area in histogram"

eg:



A) we go row wise, we can calculate max area in hist. & store it. Then compare it with next row !!

Code:

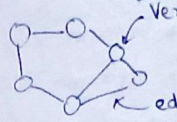
```
int histogram(vector<int> &arr){
    int n = arr.size();
    stack<int> st;
    int ans = INT_MIN;
    st.push(0);
    for(int i = 1; i<n; i++){
        while(!st.empty() and arr[i] < arr[st.top()]){
            int ele = arr[st.top()];
            st.pop();
            int nsi = i;
            int psi = (st.empty()) ? -1 : st.top();
            ans = max(ans, ele*(nsi-psi-1));
        }
        st.push(i);
    }
    while(!st.empty()){
        int ele = arr[st.top()];
        st.pop();
        int nsi = n;
        int psi = st.empty() ? -1 : st.top();
        ans = max(ans, ele*(nsi-psi-1));
    }
    return ans;
}
int largestRectangleArea(vector<int> &heights){
    return histogram(heights);
}
```

↳ Same code of "Largest Rectangle in Histogram"
↳ Done in "Stacks - Important Problems" !

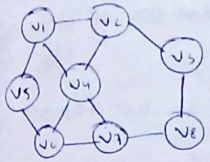
```
int maximalRectangle(vector<vector<char>> &matrix){
    int maximal = INT_MIN;
    vector<int> hist(matrix[0].size(),0);
    for(int i=0; i<matrix.size(); i++){
        to create histogram array
        for(int j=0; j<hist.size(); j++){
            if(matrix[i][j]=='1') hist[j]++;
            else hist[j]=0;
        }
        maximal = max(maximal,
                      largestRectangleArea(hist));
    }
    return maximal;
}
```

GRAPH INTRODUCTION

- A graph is a collection of nodes where each node might (or) might not point to other nodes.
- The nodes represent real life entities & are connected by edges representing relationship btw the node/entities.

Ex: 

Mathematical Defn. of Graphs.



Graph is defined as a pair: $G = \{V, E\}$
↓ (Ordering matters)
an ordered pair of set V & set E representing vertices & edges respectively.

⇒ $V = \{v_1, v_2, \ldots, v_8\}$; $E = \{\{v_1, v_2\}, \{v_2, v_4\}, \ldots\}$

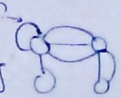
Types of Graph:

⇒ Direction → undirected / directed

⇒ Edge weights → weighted / unweighted

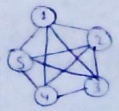
Graph Terminologies:

→ MULTI GRAPH: an undirected graph in which multiple edges are allowed btw 2 nodes (loop?)

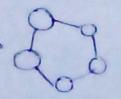


→ SIMPLE GRAPH: an undirected graph in which both multiple edges & loops are not allowed.

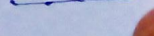
→ COMPLETE GRAPH:
- a complete graph is the one in which every node is directly connected to every other node.
- a direct edge exists for every possible node pair



→ CONNECTED GRAPH: In this graph we have a direct/indirect path for every possible pair of vertices



→ PATH: (P_n) Path is a graph whose vertices can be arranged in a sequence



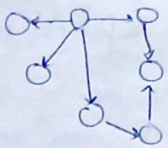
- $V = \{v_1, v_2, v_3, \ldots v_n\}$, $E = \{v_i v_{i+1}\}$ $\forall$ $i \in [1, n-1]$

→ CYCLE: - (C_n) a cycle C_n is also a graph whose vertices can be arranged in cycle sequence

- $V = \{v_1, v_2, \ldots v_n\}$, $E = \{v_i v_{i+1}\} \cup \{v_n v_1\}$ $\forall$ $i \in [1, n-1]$

→ DAG {Directed acyclic graph} : ex:

{No cycle must be present}



(DAG ✓)

(DAG ✗)

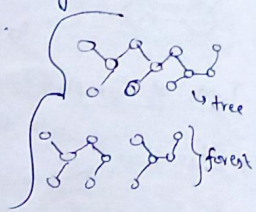
→ Degree :
- Degree of a vertex in a graph is the total no. of edges incident to it (or) away from it.
- In directed graph, we have 2 types → indegree, → outdegree

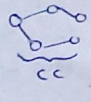
ex:  : indegree = 3, outdegree = 2

ex:  : no indegree or outdegree

⇒ In directed graph, outdegree of a vertex is total no. of outgoing edges & indegree is total no. of incoming edges.

→ Trees :
- It's a connected graph with no cycles. If we remove all cycles from a graph, we get a tree.
- Tree ≡ connected acyclic graph
- If we remove an edge from a tree, it no more remains connected & should be called 'forest'



- ex:  cc  cc

} If there is a disconnected graph then the set of vertices which are connected forms a 'connected component'

→ Facts:

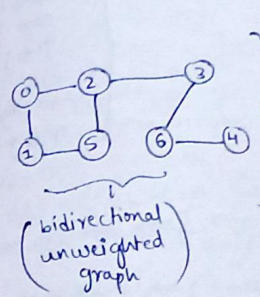
{E = edges, V = vertices}

- Tree : $|E| = |V| - 1$
- Forest : $|E| = |V| - 1$ } (max)
- Connected : $|E| = |V| - 1$ } (min)
- Undirected Complete : $|E| = {}^{|V|}C_2$ } (max)
- Directed Complete : $|E| = |V| \times (|V| - 1)$ } (max)

GRAPHS IMPLEMENTATION

Adjacency List

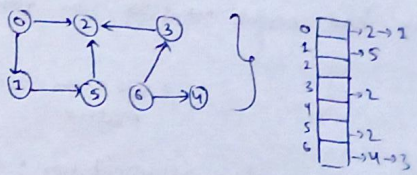
→ In this approach we represent graph as a array of linked list (represents vertex)

ex: 

⇒ i^th index represents neighbour of i^th vertex

⇒ this neighbour info. is stored in a linked list

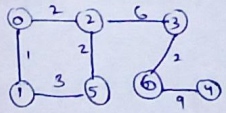
⇒ Neighbour : any node having direct edge from a vertex

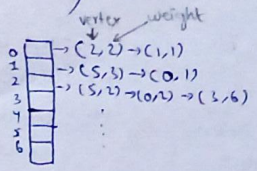
If it was directional → ex: 

⇒ What about directed weighted & undirected weighted graph?

⇒ Instead of storing an integer in the LL, we can store a pair

(first → neighbour vector
second → weight)

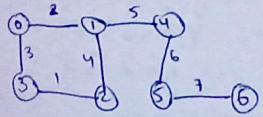
ex: 

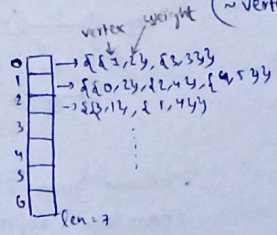


Adjacency Map :

→ Here we represent graph in the form of array of hashmap/unordered_map

(~ vertex) (~ neighbour)

ex: 



[{key, val}
neighbour, wt]

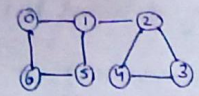
⇒ any i^th index of the array represent data of i^th vertex.

Similar for directed graph too!

Adjacency Matrix

→ In this approach we represent the graph in a 2D array (or) matrix of V×V dimensions where V = no. of vertices

ex:



bidirectional graph!

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

↳ let "adj"

adj[i][j] = { 1 ; there is an edge from 'i' to 'j' vertex
0 ; no edge from 'i' to 'j' vertex

If we have directed graph! → ① — ① → 0→1, 1→0 } both shown in undirected
② → ① → 0→1 } only 1 will be shown in matrix.

If we had weights → • we can either store (1 (or) 0 / edge, wt) pair.
• We can directly (or) store weight of edge in matrix

But prob. with this representation : the nodes which are not connected directly, their info. is also being stored!

Here graph can be of 2 types → Dense Graph: • Edges > vertices

ex:

↳ Sparse Graph
• Vertices > Edges

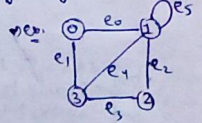
ex:

Incidence Matrix

→ In this approach we prepare a V×E matrix where 'V' is the no. of vertices & 'E' is no. of edges.

→ m[i][j] = { 1 ; if ith vertex belongs to the jth edge
0

ex:



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

Incidence Matrix

→ Column Sum = 2
→ Row Sum : represents degree.

→ If we have weighted graphs, we can store it's weight instead of '1'
→ If we have directed graphs : ex: ⓪ → ① , then we consider edge belongs to '1'. Though this method is useful only for undirected graph.

Edge List:

```
class Node{ >(vertex)
    int data;
}
```

```
class Edge {
    Node src;
    Node dest;
    int wt;
    bool dir;
};
```

```
class Graph{
    vector<Edge> edges;
    vector<Node> vertices;
}
```

Code: {Adjacency List}

```
vector< list<int> > graph;
int v; → no. of vertices
void add_edge(int src, int dest, bool bi_dir=true){
    graph[src].push_back(dest);
    if(bi_dir){
        graph[dest].push_back(src);
    }
}
void display(){
    for(int i=0; i<graph.size(); i++){
        cout<<i<<" -> ";
        for(auto el: graph[i]){
            cout<<el<<", ";
        }
        cout<<endl;
    }
```

```
int main(){
    cin>>v;
    graph.resize(v, list<int>());
    int e;
    cin>>e;
    while(e--){
        int s,d;
        cin>>s>>d;
        add_edge(s,d);
    }
    display();
    return 0;
}
```

*If we want directed graph
↳ add_edge(s,d,false);

If we have 'weights' also;

Code:

```
vector< list<pair<int,int>> > graph;
int v;
void add_edge(int src, int dest, int wt, bool bi_dir=true){
    graph[src].push_back({dest,wt});
    if(bi_dir) graph[dest].push_back({src,wt});
}
void display(){
    for(int i=0; i<graph.size(); i++){
        cout<<i<<" -> ";
        for(auto el: graph[i]){
            cout<<"("<<el.first<<" "<<el.second<<"), ";
        }
        cout<<endl;
    }
}
```

```
int main(){
    cin>>v;
    graph.resize(v,list<pair<int,int>>());
    int e;
    cin>>e;
    while(e--){
        int s,d,wt;
        cin>>s>>d>>wt;
        add_edge(s,d,wt);
    }
    display();
    return 0;
}
```

Code {Adjacency Map}: → (here it is for weighted graph)

```
vector<unordered_map<int,int>> graph;
int v;
void add_edge(int src, int dest, int wt, bool bi_dir=true){
    graph[src][dest] = wt;
    if(bi_dir) graph[dest][src] = wt;
}
void display(){
    for(int i=0; i<graph.size(); i++){
        cout<<i<<" -> ";
        for(auto el: graph[i]){
            cout<<"("<<el.first<<" "<<el.second<<"), ";
        }
        cout<<endl;
    }
}
int main(){
    cin>>v;
    graph.resize(v, unordered_map<int,int>());
    int e;
    cin>>e;
    while(e--){
        int s,d,wt;
        cin>>s>>d>>wt;
        add_edge(s,d,wt);
    }
    display();
    return 0;
}
```

Now we will write code for adjacency map for unweighted graph using unordered set.

Code:

```
vector<unordered_set<int>> graph;
int v;
void add_edge(int src, int dest, bool bi_dir=true){
    graph[src].insert(dest);
    if(bi_dir) graph[dest].insert(src);
}
void display(){
    for(int i=0; i<graph.size(); i++){
        cout<<i<<" -> ";
        for(auto el: graph[i]){
            cout<<el<<", ";
        }
        cout<<endl;
    }
}
int main(){
    cin>>v;
    graph.resize(v, unordered_set<int>());
    int e; cin>>e;
    while(e--){
        int s,d;
        cin>>s>>d;
        add_edge(s,d);
    }
    display();
    return 0;
}
```

GRAPHS BFS & DFS

We are given the adj. list representation of graph, to read any graph we have 2 major techniques → Depth first traversal/search
→ Breadth first traversal/search

Depth First Traversal (recursive):

Let's take a motivation problem; → (Q1) Given a graph, calculate all paths btw 2 vertices
(Q2) Given a graph, check whether there is a path btw any 2 vertices (or not)

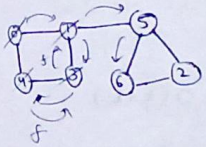
Simple soln.: If there's a path from neighbours to 'dest', then there will be a path from src to dest via the neighbour.

$$f(u,v) = \begin{cases} f(n_1,v) \\ \text{or} \\ f(n_2,v) \\ \text{or} \\ f(n_3,v) \\ \vdots \\ f(n_x,v) \end{cases}$$

; $n_1, n_2, n_3, \dots n_x$ are 'immediate neighbour of 'u' & all neighbours are unvisited

$f(u,v)$ → whether there's a path from u to v or not

ex:



; we need (0-6)

visited set array: → 0, 1, 3, 4, 5

↳ traversal: 0→1→3→4→5→6

Code: → (Q2)

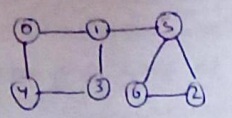
```
unordered_set<int> visited;
        +
bool dfs(int curr, int end){
    if(curr == end) return true;
    visited.insert(curr); ← mark visited
    for(auto neighbour : graph[curr]){
        if(not visited.count(neighbour)){
            bool result = dfs(neighbour, end);
            if(result){ return true; }
        }
    }
    return false;
}
bool anyPath(int src, int dest){
    return dfs(src, dest);
}
```

```
{
    *
    int x,y;
    cin>>x>>y;
    cout<<anyPath(x,y)<<endl;
    return 0;
}
```

$T_C = O(V+E)$

(no. of vertex) (no. of edges)

For Q1.



→ we can create 2D array, to store our answers

↳ dfs(0,6) → dfs(1,6) → dfs(5,6) → ✓

↳ dfs(4,6) → dfs(3,6) → dfs(1,6) → dfs(5,6) ↓ ✓

Code:

```
unordered_set<int> visited;
vector<vector<int>> result;

⊕

void dfs(int curr, int end, vector<int>&path){
    if(curr==end){
        path.push_back(curr);
        result.push_back(path);
        path.pop_back();   → doing because we might have reached end from 1st path but we can also reach from path 2!
        return;
    }
    visited.insert(curr);
    path.push_back(curr);
    for(auto neighbour : graph[curr]){
        if(not visited.count(neighbour)){
            dfs(neighbour, end, path);
        }
    }
    path.pop_back();
    visited.erase(curr);
    return;
}
```

```
void allPath(int src, int dest){
    vector<int> v;
    dfs(src, dest, v);

{
    ⊛
    int x,y;
    cin>>x>>y;
    allPath(x,y);
    for(auto path: result){
        for(auto el: path){
            cout<<el<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

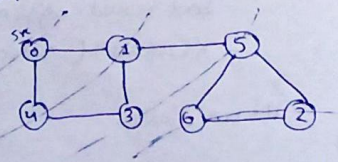
$T_c = O(E+E)$

DFS in graphs ~ Pre, post, in Order traversal in trees

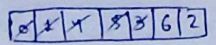
Breadth First Traversal : ($T_c = O(V+E)$, $S_c = O(V)$)

↳ (can be used for → unweighted graph → shortest path)

In BFS, we travel the immediate neighbour first together.



we use queue

↳ 

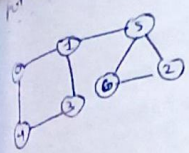
visited = {0, 1, 4, 5, 3, 6, 2}

lvl 0 | lvl 1 | lvl 2 | lvl 3

we also check if neighbours of node belong to visited or not!

↳ ex: 0 → 4, also 4 → 0 (since '0' already visited we ignore it)

for distance:



vis = {0, 1, 4, 5, 3}

queue = | 0 | 1 | 4 | 5 | 3 | 6 | 2 |

dist = | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
0 1 2 3 4 5 6
1 3 2 2 2 3

(6,2) → (6−5)+1

↳ {0→1, 1→3} dist=2

Code:

```
unordered_set<int> visited;
vector<vector<int>> result;

⊕

void bfs(int src, vector<int>&dist){
    queue<int> qu;
    dist.resize(V, INT_MAX);
    dist[src]=0;
    visited.insert(src);
    qu.push(src);
    while(!qu.empty()){
        int curr = qu.front();
        cout<<curr<<" ";  → to show the level
        qu.pop();
        for(auto neighbour :graph[curr]){
            if(not visited.count(neighbour)){
                qu.push(neighbour);
                visited.insert(neighbour);
                dist[neighbour]= dist[curr]+1;
            }
        }
    }
}
```

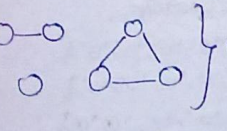
```
{
    ⊛
    int x;
    cin>>x;   → we are finding closest distance of all nodes from 'src', so only 1 input.
    vector<int> dist;
    bfs(x,dist);
    cout<<endl;
    for(int i=0;i<dist.size();i++){
        cout<<dist[i]<<" ";
    }
    return 0;
}
```

PROBLEMS BASED ON BFS & DFS

Connected Component → It's a subset of the given graph that has vertices between which there is always a path

(let's say 'cc')

ex:

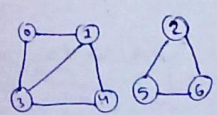


CC = 3

→ vertices present in 2 diff C.C do not have a path.

Now we can find no. of CC using DFS

↳ ex:



- Start DFS from 0: 0 → 1 → 4 → 3
- Start DFS from 1: all vertices read
  - 4 "
  - 3 "
- Start DFS from 2: 2 → 6 → 5
  - 6: all vertices read
  - 5 "

No. of times DFS/BFS is called is the CC.

Code:

```
① current node being visited        → keeping track of visited nodes.
void dfs(int node, unordered_set<int> &visited){
    visited.insert(node);  → marks the current node as visited by inserting it into the set
    for(auto neighbour : graph[node]){  → Iterates over all neighbours of the current node
        if(not visited.count(neighbour)){  → If the neighbour has not been visited, recursively call 'dfs' on that neighbour.
            dfs(neighbour, visited);
        }
    }
}

int connected_components(){
    int result = 0;  → stores no. of connected_components
    unordered_set<int> visited;  → to keep track of visited nodes
    for(int i=0; i<v; i++){  → go to every vertex
        if(visited.count(i)==0){  → if ith node is not part of 'visited' yet
            result++;
            dfs(i, visited);  → For each vertex, if it has not been visited ('visited.count(i)==0'), it increments 'result' & calls 'dfs(i, visited)'
        }
    }
    return result;
}

{
    {
        (*)
    }
    add_edge(s,d,false);
}
cout<< connected_components <<endl;
return 0;
}
```

Q) NUMBER OF ISLANDS ($1 <= m,n <= 300$)

Given an $m \times n$ 2D binary grid which represents a map of 1's (land) & 0's (water), return the number of islands.

An island is surrounded by water & is formed by connecting adjacent lands horizontally or vertically. You may assume all 4 edges of the grid are all surrounded by water.

Ex 1:
```
[['1','1','1','1','0'],
 ['1','1','0','1','0'],
 ['1','1','0','0','0'],
 ['0','0','0','0','0']]
```
o/p: 1

Ex 2:
```
[['1','1','0','0','0'],
 ['1','1','0','0','0'],
 ['0','0','1','0','0'],
 ['0','0','0','1','1']]
```
o/p: 3

A)

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

$\approx$ 

Connected components → : we can apply BFS/DFS

↳ Go to every cell of the grid, skip the 0's cell, only consider '1' valued cell.

Now, at a {row, col} → we will check it's neighbouring & mark it as visited.

Code fixing BFS → (doesn't use recursion, so advantage of space is there)

```
int numIslands(vector<vector<char>>&grid){
    int rows = grid.size();
    int cols = grid[0].size();
    int cc=0;  → stores the no. of connected components
    for(int r=0; r<rows; r++){
        for(int c=0; c<cols; c++){
            if(grid[r][c]=='0'){
                //it's a water body / landed body
                continue;
            }
            cc++; //new unvisited land piece found i.e. new connected comp.
            grid[r][c]='0'; //mark it as visited
            queue<pair<int,int>> qu;
            qu.push({r,c}); //store the src code
            while(!qu.empty()){
                auto curr = qu.front(); //get one node from queue
                qu.pop();
                int currRow = curr.first;  → go to all unvisited neighbours of curr node
                int currCol = curr.second;
check up        if(currRow-1>=0 and grid[currRow-1][currCol]=='1'){
                    qu.push({currRow-1, curr(col)};
                    grid[currRow-1][currCol]='0';
                }
check down      if(currRow+1<rows and grid[currRow+1][currCol]=='1'){
                    qu.push({currRow+1; curr(col)};
                    grid[currRow+1][curr(col]='0';
                }
check left      if(currCol-1>=0 and grid[currRow][currCol-1]=='1'){
                    qu.push({currRow, curr(col-1};
                    grid[currRow][currCol-1]='0';
                }
check right     if(currCol+1<cols and grid[currRow][currCol+1]=='1'){
                    qu.push({currRow, curr(col+1};
                    grid[currRow][currCol+1]='0';
                }
            }
        }
    }
    return cc;
}
```

Code (using DFS)

```
void dfs(vector<vector<char>>& grid, int r, int c){
    if(r<0 or c<0 or r>=grid.size() or c>=grid.size() or grid[r][c]=='0')
        return;
}
```

Mark the current cell as visited

```
grid[r][c] = '0';
dfs(grid, r+1, c); → check down
dfs(grid, r-1, c); → check up
dfs(grid, r, c+1); → check right
dfs(grid, r, c-1); → check left
}
```

```
int numIslands(vector<vector<char>>& grid){
    int row = grid.size();
    int col = grid[0].size();
    int cc = 0;
    for(int r=0; r<row; r++){
        for(int c=0; c<col; c++){
            if(grid[r][c]=='1'){ ← found unvisited island
                cc++;
                dfs(grid, r, c); → explore the island
            }
        }
    }
    return cc;
}
```

② PACIFIC ATLANTIC WATER FLOW {1 ≤ m, n ≤ 200}

There is an m×n rectangular island that borders both the Pacific Ocean & Atlantic Ocean. The Pacific Ocean touches the island's left & top edges, and the Atlantic Ocean touches the island's right & bottom edges.

The island is partitioned into a grid of square cells, You are given an 'm×n' integer matrix heights where heights[r][c] represents the height above sea level of the cell at co-ordinate (r, c).

The island receives a lot of rain, and the rain water can flow to neighbouring cells directly north, south, east & west if the neighbouring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return a 2D list of grid co-ordinates 'result' where result[i] = [ri, ci] denotes that rain water can flow from cell (ri, ci) to both the Pacific & Atlantic oceans.

ex:

Pacific Ocean				
1	2	2	3	5
3	2	3	4	4
2	4	5	3	1
6	7	1	4	5
5	1	1	2	4

Atlantic Ocean

ip: heights = [[1,2,2,3,5], [3,2,3,4,4], [2,4,5,3,1], [6,7,1,4,5], [5,1,1,2,4]]

op: [[0,4], [1,3], [1,4], [2,2], [3,0], [3,1], [4,0]]

explanation: The following cells can flow to the Pacific Ocean & Atlantic ocean as shown below:

[0,4]: [0,4] → Pacific Ocean
[0,4] → Atlantic Ocean

[1,3]: [1,3] → [0,3] → PO
[1,3] → [1,4] → AO

[1,4]: [1,4] → [1,3] → [0,3] → PO
[1,4] → AO

[2,2]: → [1,2] → [0,2] → PO
→ [2,3] → [2,4] → AO

[3,0] → PO
→ [4,0] → AO

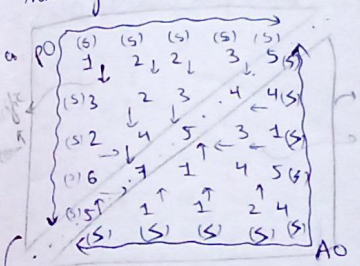
[3,1] → [3,0] → PO
→ [4,1] → AO

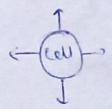
[4,0] → PO
→ AO

A) Apply multisource BFS → means we will input many sources in queue & rest everything will be followed as BFS does

Now we need those cells from which water can go to PO & AO. So those cell val > neighbours.

Thinking reverse; {we will see water from bottom to top}.





Water can go in 4 directions

$T_c = O(mn)$ (we are checking each cell)

Now intersection of these 2 final arrows will give us the cells where water will come & flow to both PO & AO.

Code (BFS)

```
vector<vector<int>> dir = {{1,0}, {-1,0}, {0,1}, {0,-1}};   // representing 4 directions (↓, ↑, →, ←)
int rows, cols;
vector<vector<int>> h;   // representing heights of each cell in input grid
vector<vector<bool>> bfs(queue<pair<int,int>>& qu){
    vector<vector<bool>> visited(rows, vector<bool>(cols, false));   // to keep track of visited cells
    while(!qu.empty()){
        auto cell = qu.front();
        qu.pop();
        int i = cell.first, j = cell.second;
        visited[i][j] = true;   // mark cell as visited
        for(int d=0; d<4; d++){   // checking all 4 directions from the current cell
            int newRow = i + dir[d][0], newCol = j + dir[d][1];
            if(newRow<0 or newCol<0 or newRow>=rows or newCol>=cols){
                continue;
            }
            if(visited[newRow][newCol]) continue;
            if(h[newRow][newCol] < h[i][j]) continue;   // when height of neighbouring cell >= curr cell
            qu.push({newRow, newCol});
        }
    }
    return visited;
}
```

```
vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights){
    rows = heights.size();
    cols = heights[0].size();
    h = heights;
    queue<pair<int,int>> pacificBFS;   // representing co-ord. of grid
    queue<pair<int,int>> atlanticBFS;
    for(int i=0; i<rows; i++){
        pacificBFS.push({i,0});        // stores starting points for BFS
        atlanticBFS.push({i,cols-1});
    }
    for(int j=1; j<cols; j++) pacificBFS.push({0,j});      // multisource BFS, covers all
    for(int j=0; j<cols-1; j++) atlanticBFS.push({rows-1,j});   points.
    vector<vector<bool>> pacific = bfs(pacificBFS);
    vector<vector<bool>> atlantic = bfs(atlanticBFS);
    vector<vector<int>> result;
    for(int i=0; i<rows; i++){
        for(int j=0; j<cols; j++){
            if(pacific[i][j] and atlantic[i][j])  result.push_back({i,j});
        }
    }
    return result;
}
```

Code DFS:

```
void dfs(int i, int j, vector<vector<bool>> &visited){
    visited[i][j] = true;
    for(int d=0; d<4; d++){
        int newRow = i + dir[d][0];
        int newCol = j + dir[d][1];
        if(newRow>=0 and newCol>=0 and newRow<rows and newCol<cols
           and !visited[newRow][newCol] and h[newRow][newCol]>=h[i][j]){
            dfs(newRow, newCol, visited);
        }
    }
}
```

(5)

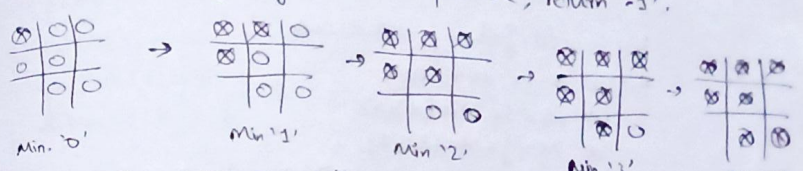
```
vector<vector<bool>> pacific(rows, vector<bool>(cols,false));
vector<vector<bool>> atlantic(rows, vector<bool>(cols,false));
for(int i=0; i<rows; i++){
    dfs(i,0,pacific);
    dfs(i,cols-1,atlantic);
}
for(int j=0; j<cols; j++){
    dfs(0,j,pacific);
    dfs(rows-1,j,atlantic);
}
```

994) ROTTING ORANGES

You are given an m×n grid where each cell can have 1 of 3 values:
- '0' ~ empty cell
- '1' ~ fresh orange
- '2' ~ rotten orange

Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.
Return the minimum no. of minutes that must elapse until no cell has a fresh orange. If this is impossible, return '-1'.

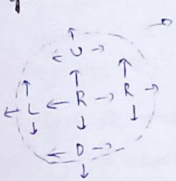
Eg:



grid = [[2,1,1],[1,1,0],[0,1,1]] , op: 4

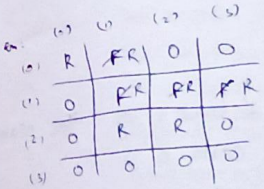
if grid: [[2,1,1],[0,1,1],[1,0,1]] → op: -1 (orange in bottom left = never rotten)

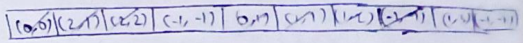
→ BFS



→ We can add all (rotten) into queue.
& fresh

Now after adding all, we can add {-1,-1} (so that we know the level/minutes)





Now mins++ → when (-1,-1), so answer: mins-1 (as we have an extra (-1,-1) to indicate level)

Code: (BFS)

```
int orangesRotting(vector<vector<int>>&grid){
    queue<pair<int,int>> qu;
    int freshOrange = 0, n = grid.size(), m = grid[0].size();
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(grid[i][j]==1) freshOrange++;
            else if(grid[i][j]==2) qu.push({i,j});   → multisource BFS step
        }
    }
    qu.push({-1,-1});   → we have add all src
                          → index can never be -1
    int mins = 0;
    vector<vector<int>> dir = {{-1,0},{1,0},{0,-1},{0,1}};
```

```
while(!qu.empty()){
    auto cell = qu.front();
    qu.pop();
    if(cell.first == -1 and cell.second == -1){
        mins++;
        if(!qu.empty()) qu.push({-1,-1});
        else break;
    }
    else{
        int i = cell.first, j = cell.second;
        for(int d = 0; d<4; d++){
            int nr = i + dir[d][0];
            int nc = j + dir[d][1];
            if(nr<0 or nc<0 or nr>=n or nc>=m) continue;
            if(grid[nr][nc]==2 or grid[nr][nc]==0) continue;
            freshOrange--;
            grid[nr][nc] = 2;
            qu.push({nr,nc});
        }
    }
}
return (freshOrange==0) ? mins-1 : -1;
}
```

Q) 01 MATRIX

Given an mxn binary matrix mat, return the distances of the nearest 0 for each cell. The distance btw 2 adjacent cells is 1.

Ex:

0	0	0
0	1	0
1	1	1

mat = [[0,0,0],[0,1,0],[1,1,1]] o/p: [[0,0,0],[0,1,0],[1,2,1]]

As Here we can apply multisource BFS;

Ex:

1	2 1	1 2	0	0
1	1 2	1 2	1 2	1 2
1 2	0	2 1	1 2	0
0	1 2	0	0	0
1	1	1 2	2 1	2 1

Brute force: → start bfs from every cell with a value 1

but we can optimize it, if we consider brute force, we have few problems, what if '1' is beside '1' & how to check distances?

How about we try multisource bfs from 0's.

Here we can initialise the distance → 2D vector by INT_MAX, as basically we are calculating shortest path, so it would be easier, as it can also act as unvisited! If visited → value will be overwritten

Code

```
vector<vector<int>> updateMatrix(vector<vector<int>>& mat){
    int m = mat.size();
    int n = mat[0].size();
    vector<vector<int>> distances(m, vector<int>(n, INT_MAX));
    queue<pair<int,int>> qu;

    for(int i=0; i<m; i++){                    → Adding all 0s as sources to the queue
        for(int j=0; j<n; j++){                  & set their distance to 0.
            if(mat[i][j]==0){
                qu.push({i,j});
                distances[i][j] = 0;
            }
        }
    }
    vector<vector<int>> dir = {{-1,0},{1,0},{0,-1},{0,1}};   → defining directions: up, down, left, right
    while(!qu.empty()){                        → perform BFS from each 0 source
        auto cell = qu.front();
        qu.pop();
        for(int d=0; d<4; d++){                  → explore all 4 directions
            int currRow = cell.first;
            int currCol = cell.second;
            int newRow = currRow + dir[d][0];
            int newCol = currCol + dir[d][1];
            if(newRow<0 or newCol<0 or newRow>=m or newCol>=n) continue;
            if(distances[newRow][newCol] <= distances[currRow][currCol]) continue;
            distances[newRow][newCol] = distances[currRow][currCol] + 1;
            qu.push({newRow,newCol});
        }
    }
    return distances;
}
```

GRAPHS - DISJOINT SET UNION

Linear DS - Stacks, Queues, arrays, ll, etc

Non-linear DS - Hashmaps, trees {BS, BST, heaps}

Clustering/Grouping : You will be having some elements & you need to add them/segregate them in diff groups/clusters & sometimes we might need to identify the group any element belongs to!

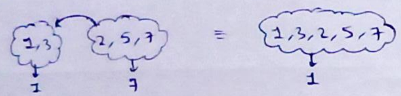
• **Terminologies:**

→ Leader/parent of the group: to uniquely identify a group, we will pick any element from the group & make it the representation/leader/parent of the graph.

Q) What func. dsu need to support?

(1) Union(a,b): adds the group where element 'b' belongs to the group where element A belongs or vice-versa

ex: union(1,5) →

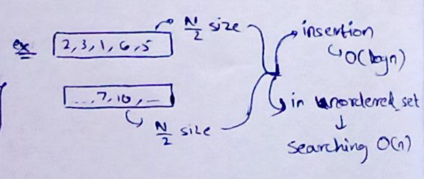


(2) Find(x) / get(x): this will be used to find which group 'x' belongs to. We will return the parent of the group that 'x' belongs to

APPROACH 1:

• represent every group as a set

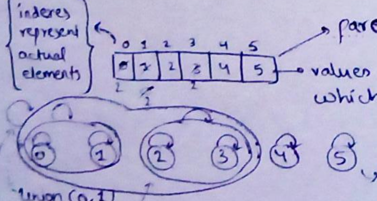
• further, show it as : array of sets



• This approach has high T_c!

APPROACH 2:

• Using arrays



Initially every ele belongs to different own groups

Here, you can see each time group '1' is getting union, we need to update it with every union. This is an issue which can be solved by other approaches.

• Pseudo code:

```
int find(x){
    return par[x];
}
```

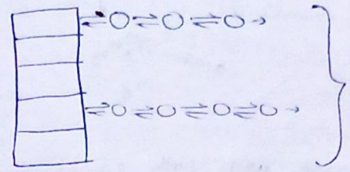
→ O(1)

```
void union(a,b){
  a = find(a);
  b = find(b);
  for(int i=0; i<n; i++){
    if(par[i]==b){
      par[i]=a;
    }
  }
}
```

→ O(n)

• Similarly, we can apply DLL too.

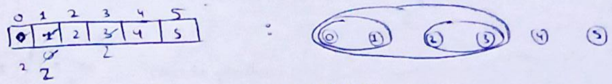
ex:



Here → find = O(n)
→ insert/union = O(1)

APPROACH 3: {Union by Size}

• Here we join smaller group with larger group



We will also maintain 'size' array to keep track of 'sizes'

union(0,1)
union(2,3)
union(1,2)

Note: This will update,

1
2
4
8
⋮
2^k

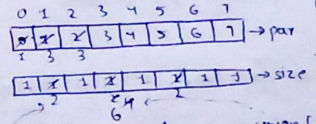
} k → logn

$\therefore T_c = \frac{n \times \log n}{n_{union}} \approx O(\log n)$ → Avg. T_c

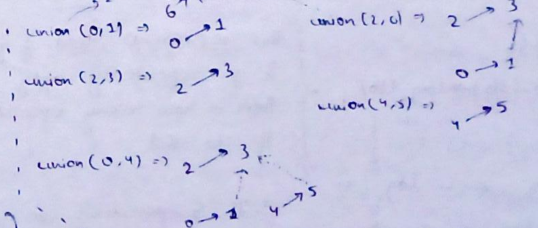
APPROACH 4: {Union by Rank}

• Using 'Tree'!

• Here we update the parent with the new parent



if we want parent, ex: parent 1 = 1 parent 3, 3 parent 3 → ans



• Pseudo code:

```
void union(a,b){
    a = find(a)
    b = find(b)
    if(sz[b]<sz[a]){
        sz[a] += sz[b]
        par[b] = a
    } else{
        sz[b] += sz[a]
        par[a] = b
}  }
```

(can also write) →

```
if(rank[a] < rank[b]){
    par[a] = b;
    rank[b]++;
} else {
    par[b] = a;
    rank[a]++;
}
```

T_c : const.

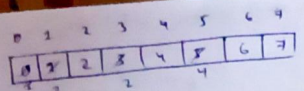
```
int find(x){
    if(par[x]==x) return x;
    return find(par[x])
}
```

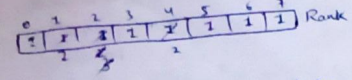
→ T_c : O(log n)

• Here 'rank' : in how many operations I can reach my parent

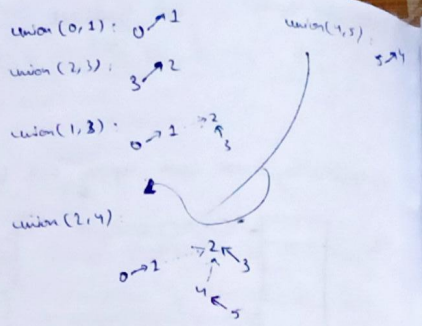
(see next page for ...)

• Here also smaller group merges with bigger group.



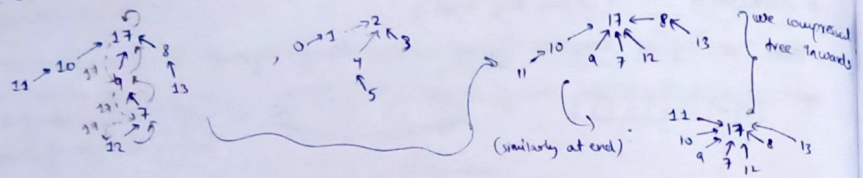


↳ like how many links there are, how many operations at max we need to do to find parent



APPROACH 5: {Union by size/Union by rank with path compression}

• ex



ex → union(9,11)

↳ whenever we apply 'find' operation, the element will get access to its root/parent

} Now what we do is, the ele.'s which are getting access of root → directly make them as child nodes

• pseudo code:
```
int find(x){
    if(par[x]==x) return x;
    return par[x] = find(par[x]);
}
```
} Now due to this, T_c almost becomes constant

• Using "Inverse Ackermann function" ↷ $T_c = O(\log^* n)$ $(n : \log_2 n)$

ex: n=65536

$\log_2(65536) \to \log_2(16)$ (1) (2)

$\log_2 1 \leftarrow \log_2 2 \leftarrow \log_2(4)$ (5) (4) (3)

} this represents that if you have a value of n, & you repeatedly apply $\log_2 n$ on this value, then in how many operations you can reduce it to ≤ 1

∴ In 5 operations, we got our answer!

• Code

```
int find(vector<int>&parent, int x){
    return parent[x] = (parent[x]==x) ? x : find(parent, parent[x]);
}
```

↑ This method returns which group/cluster 'x' belongs to

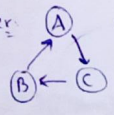
```
void Union(vector<int>&parent, vector<int>&rank, int a, int b){
    a = find(parent, a);
    b = find(parent, b);
    if(rank[a] >= rank[b]){
        rank[a]++;
        parent[b] = a;
    }else{
        rank[b]++;
        parent[a] = b;
    }
}
```

$T_c = O(\log * n)$

```
int main(){
    int n, m;   // n: elements, m: no. of queries
    cin>>n>>m;
    vector<int> parent(n+1);
    vector<int> rank(n+1,0);
    for(int i=0; i<=n; i++) parent[i] = i;
    while(m--){
        string str;
        cin>>str;
        if(str=="union"){
            int x,y;
            cin>>x>>y;
            Union(parent,rank,x,y);
        }else{
            int x;
            cin>>x;
            cout<<find(parent,x)<<endl;
        }
    }
    return 0;
}
```

TOPOLOGICAL SORTING & CYCLE DETECTION

We do this in directed graph, more specifically "DAG": directed acyclic graph

ex: 

• cyclic graph: can't be resolved for dependency

} It is like A needs C, C needs B, B needs A

QUE: {GOOGLE} 'n' inequalities : a<b, b>d, e>f, a<d, c>f, e>a, ...

Can these inequalities be resolved?

Hint: a<b means b → a (b dependent on a)

e<f : f → e

(inequalities resolved) ← Have to check if all of them are forming DAG(or) not. If yes ✓, no → can't be resolved

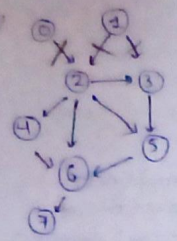
Now to do this dependency resolution, we have/use → Topological Sorting (can be done BFS / DFS)

• Kahn's Algorithm:

• Find a list of "start nodes" which have no incoming edges & insert them into a set S. Atleast 1 node must exist in a non-empty acyclic graph.

→ Indegree: indegree of a node is defined as how many incoming edges you have.

→ Here we use multisource BFS, (see next page)



Indegree:

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

Queue

0	1	2	3	4	5	6	7

(Insert those ele. only whose indegree = 0)

(After you process '0' → '2's dependency decreases by one.)

(After processing → Indegree of 2 & 3 decreases by 1)

Code:

```
vector<list<int>> graph;
int v;
void add_edge(int a, int b, bool bidir = true){
    graph[a].push_back(b);
    if(bidir) graph[b].push_back(a);
}

int main(){
    cin>>v;
    int e;
    cin>>e;
    graph.resize(v, list<int>());
    while(e--){
        int x,y;
        cin>>x>>y;
        add_edge(x,y,false);
    }
    topoBFS();
    return 0;
}
```

```
void topoBFS(){ //Kahn's Algo
    vector<int> indegree(v,0);
    for(int i=0; i<v; i++){
        for(auto neighbour: graph[i]){
            // i → neighbour, so we ↑ the indegree
            indegree[neighbour]++;
        }
    }
    queue<int> qu;
    unordered_set<int> vis;
    for(int i=0; i<v; i++){
        if(indegree[i]==0){
            qu.push(i);
            vis.insert(i);
        }
    }
    while(!qu.empty()){
        int node = qu.front();
        cout<<node<<" ";
        qu.pop();
        for(auto neighbour: graph[node]){
            if(!vis.count(neighbour)){ // means indegree != 0
                indegree[neighbour]--;
                if(indegree[neighbour]==0){
                    qu.push(neighbour);
                    vis.insert(neighbour);
                }
            }
        }
    }
}
```

---

1) There are total of 'numCourses' courses you have to take, labeled from 0 to numCourses-1.

You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

- ex: pair[0,1] = indicates that to take course 0, you have to first take course 1.

Return ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it's impossible to finish all courses, return an empty array?

A) Code:

```
vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites){
    vector<list<int>> graph;
    vector<int> indegree(numCourses,0);
    graph.resize(numCourses, list<int>());
    for(auto neighbour: prerequisites){
        int a = neighbour[0];
        int b = neighbour[1];
        graph[b].push_back(a);
        indegree[a]++;
    }

    queue<int> qu;
    ~~unordered_set<int> visited;~~
    vector<int> ans;
    for(int i=0; i<numCourses; i++){
        if(indegree[i]==0){
            qu.push(i);
            ~~visited.insert(i);~~
            ans.push_back(i);
        }
    }
    while(!qu.empty()){
        int node = qu.front();
        qu.pop();
        for(auto neighbour: graph[node]){
            indegree[neighbour]--;
            if(indegree[neighbour]==0){
                qu.push(neighbour);
                ans.push_back(neighbour);
            }
        }
    }
    if(ans.size()!=numCourses) return vector<int>();
    return ans;
}
```

Not needed, as once we get indegree as '0', we are storing it into 'node' & popping, so that element will not exist anymore. So, NO point of keeping 'visited.count' condition!

---

2) Given a grid of n×m size. Every cell of the grid, are marked as 'L', 'R', 'U', 'D'. Character on a cell denotes if you are standing at that cell, what direction you can move to. Check if we start from (0,0), can we reach (n-1, m-1)? {$S_c \equiv O(1)$}

ex:

R	R	D	R
D	L	D	L
U	D	L	L
U	R	R	R

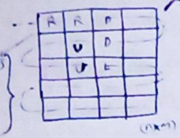
: U ← L D → R {end → non-modifiable}

A) Brute force: Apply recursion & go to each cell ⟶ prob. Space was being used!

Other method

Hint: 

} we have to go src → dest, but if we didn't reach, that means we are stuck in a cycle!

Approach 1: 

{ traversal DFS/BFS, uses space, so this method not allowed }

→ Worst case, the maximum no. of cells we will visit is 'n×m'. So if we go along each cell & we didn't reach within 'n×m' steps, we can say → it will never reach dest.

∴ Total = (n×m) steps

↳ this approach doesn't tell us location of cycle.

There won't be any cycle if it's possible to reach dest. We will get a cycle if we visit a cell twice.

Approach 2: Using Hare-Tortoise approach / Slow-Fast pointer approach

- We can initialise s, f from (0,0)
- Move 's' by 1 & 'f' by 2 → if cycle there, point of intersection ✓
  ↳ else → we can reach dest

↓

base case: (if we go out of grid)

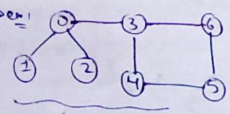
• Detect cycle in undirected graph:

→ BFS
→ DFS
→ DSU

→ Cycle Detection using DSU:

→ We will read each edge & all vertices of edge & apply Union operation

→ Input, in form of "edge list" → ex: [[0,1], [0,2], [0,3], ...] } ex:



Method:

- Create a dsu: (0) (1) (2) (3) (4) (5) (6)
- [0,1] → [0 1] (2) (3) ...
- [0,2] → [0 1 2] (3) (4) ...
  'O' parent 'O'
  '2' parent 2
  ∴ we can merge
- [0,3], [3,4] & let us read [5,6] } → [0 1 2 3 4] [5 6]
- [4,5] → [0 1 2 3 4 5 6]
- [3,6] ⇒ Same parent of '3' & '6', so cycle exists!

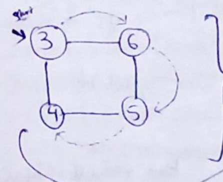
Code:

```
int find(vector<int>&parent, int x){
  return parent[x]=(parent[x]==x) ? x : find(parent,parent[x]);
bool Union (vector<int> & parent, vector<int>& rank, int a, int b){
    a=find(parent,a);
    b=find(parent,b);
    if(a==b) return true;
    if(rank[a]>=rank[b]){
        rank[a]++;
        parent[b]=a;
    }else{
        rank[b]++;
        parent[a]=b;
    }
    return false;
}
```

```
int main(){
    int n,m;        // elements = vertices, no. of queries = edges
    cin>>n>>m;
    vector<int> parent(n+1);
    vector<int> rank(n+1,0);
    for(int i=0; i<=n; i++) parent[i]=i;
    while(m--){
        int x,y;
        cin>>x>>y;
        bool b = Union(parent,rank,x,y);
        if(b==true){
            cout<<"cycle detected"<<endl;
        }
    }
    return 0;
}
```

∴ $T_c = O(E*(\log^* n)) \approx O(E)$

→ Cycle Detection using DFS:

ex: 

Now we don't want: (Parent ⇄ Child)

} → (6) → 2 edges ↗3 ↘5] → Since '3' is parent, we won't go there, so moving to '5'

(4) → 2 edges ↗3 ↘5] → Since '5' is parent, we won't go there/skip it, so moving to '3', but '3' is already visited!

(∴ Check if the visited is not your parent then there's a cycle)

Code: → $T_c = O(V+E)$ } same as that of DFS

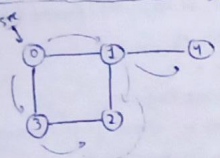
```
vector<list<int>> graph;
int v;
void add_edge(-){

}

void display(){
  for(int i=0;i<graph.size();i++){
    cout<<i<<"->";
    for(auto el : graph[i]){
        cout<<el<<", ";
    }
    cout<<endl;
}}
```

```
bool dfs(int src, int parent, unordered_set<int>&visited){
    visited.insert(src);
    for(auto neighbour : graph[src]){
        if(visited.count(neighbour) and neighbour != parent)
        { return true; }
        if(!visited.count(neighbour)){
            bool res = dfs(neighbour,src,visited);
            if(res==true) return true;
        }
    }
    return false;
}
```

```
bool has_cycle(){
    unordered_set<int> visited;
    for(int i=0; i<v; i++){
        if(!visited.count(i)){
            bool result = dfs(i, -1, visited);
            if(result == true) return true;
        }
    }
    return false;
}
```

```
int main(){
    cin>>v;
    :
    -edges input
    :
    display();
    bool b = has_cycle;
    cout<<"Cycle Detected "<<b<<endl;
    return 0;
}
```

→ Cycle Detection using BFS :



Visited → 0, 1, 3, 4, 2

Parent →

0	1	2	3	4
1	0	1	0	1

(parent)

Queue →

0	1	3	4	2

(eliminating from here → keeping it's parent value in array & adding it's next level in queue)

If a node is already visited & it's not your parent, then you have a cycle

Code: $T_c = O(V+E)$

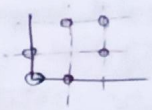
```
//Everything same as of above code except 2 changes → has_cycle()
                                                   → dfs → change to → bfs
bool bfs(int src){
    unordered_set<int> visited;
    queue<int> qu;
    vector<int> parent(v, -1);
    qu.push(src);
    visited.insert(src);
    while(!qu.empty()){
        int curr = qu.front();
        qu.pop();
        for(auto neighbour : graph[curr]){
            if(visited.count(neighbour) and parent[curr] != neighbour){
                return true;
            }
            if(!visited.count(neighbour)){
                visited.insert(neighbour);
                parent[neighbour] = curr;
                qu.push(neighbour);
            }
        }
    }
}
```

```
bool has_cycle(){
    :
    bool result = bfs(i);
}
```

Q) MOST STONES REMOVED WITH SAME ROW/COLUMN

On a 2D plane, we place 'n' stones at some integer co-ord. points. Each co-ord. point may have at most 1 stone.
A stone can be removed if it shares either same row/column as another stone that has not been removed. Given an array stones of length 'n' where stones[i] = [x_i, y_i] represents the location of the i^{th} stone, return the largest possible no. of stones that can be removed.

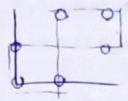
Ex: S = [[0,0], [1,0], [0,1], [1,2], [2,1], [2,2]] ; o/p = 5



A) Hint: Connected component, DFS

If 2 nodes share their same row/col then they're in same connected comp.

Ex:



This means for getting unique stone in 1 cc, we can destroy all it's other nodes. By the time we reach source, all other nodes will be destroyed.
∴ Answer: total nodes − no. of connected components
(will give us max no. of stones we can destroy)
(we can use DFS, BFS, DSU)

Code:

```
vector<list<int>> graph;
int v;
void dfs(int node, unordered_set<int> &visited){
    visited.insert(node);
    for(auto neighbour : graph[node]){
        if(!visited.count(neighbour)){
            dfs(neighbour, visited);
        }
    }
}
int connected_components(){
    int result=0;
    unordered_set<int> vis;
    for(int i=0; i<v; i++){
        if(vis.count(i)==0){
            result++;
            dfs(i, visited);
        }
    }
    return result;
}
```

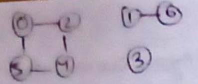
```
int removeStones(vector<vector<int>> &stones){
    v = stones.size();
    graph.resize(v, list<int>());
    for(int i=0; i<v; i++){
        for(int j=i+1; j<v; j++){
            if(stones[i][0]==stones[j][0] or
               stones[i][1]==stones[j][1]){
                graph[i].push_back(j);
                graph[j].push_back(i);
            }
        }
    }
    int ans = v - connected_components();
    return ans;
}
```

EXACT LOGIC OF CONNECTED COMPONENT

Q) COUNT UNREACHABLE PAIRS OF NODES IN AN UNDIRECTED GRAPH

You are given an integer 'n'. There is an undirected graph with 'n' nodes, numbered from 0 to n−1. You are given a 2D integer array edges where edges[i] = [a_i, b_i] denotes that there exists an undirected edge connecting nodes a_i & b_i. Return the no. of pairs of different nodes that are unreachable from each other

{1 <= n <= 10^5}

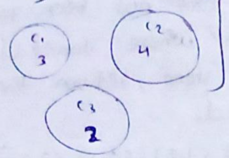
Ex: 

Ip: n = 7
edges = [[0,2],[0,5],[2,4],[1,6][5,4]]

O/P: 14

Unreachable pairs: [[0,1],[0,3],[0,6],[1,2],[1,3],[1,4],[1,5],[2,3],[2,6],[3,4],[3,5],
[3,6],[4,6],[5,6]]

Disconnected graphs: ex.



total unreachable pairs = 3×4 + 4×2 + 2×3

Now we can get size of connected component using DSU!

worst case: [3|4|2 ———] m } a: 3, 4, 2, 5, 2, 1, ...

= 3×4 + 3×2 + 3×5 + 3×2 + 3×1 + 4×2 + 4×5 ——
= 3×(4+2+5+2+1 ...) + 4×(2+5+2+1 ...)
pairs
+ 2×(5+2+1 ...) + 5×(2+1 ...) + 2×(1 ...)

Also,
3[size − 3] + (after this decrease size) 4(size − 4)

(technically we can make suffix sum)

doing this iteratively as n = 10^5, then recursion will consume lot of stack space! To avoid it, we do this way.

Code (using DSU)

```
int find(vector<int> &parent, int x){
    while(parent[x] != x){
        parent[x] = parent[parent[x]];
        x = parent[x];
    }
    return x;
}
void Union(vector<int> &parent, vector<int> &rank,
           int a, int b){
    a = find(parent, a);
    b = find(parent, b);
    if(rank[a] > rank[b]){
        rank[a]++;
        parent[b] = a;
    }
    else{
        rank[b]++;
        parent[a] = b;
    }
}
```

```
long long countPairs(int n,
            vector<vector<int>> &edges){
    vector<int> rank(n, 0);
    vector<int> parent(n);
    for(int i=0; i<n; i++) parent[i] = i;
    for(auto edge : edges){
        int a = find(parent, edge[0]);
        int b = find(parent, edge[1]);
        if(a != b) Union(parent, rank, a, b);
    }
    long long totalPairs = ((long long)n*(n-1))/2;
    vector<int> size(n, 0);
    for(int i=0; i<n; i++) size[find(parent, i)]++;
    for(int i=0; i<n; i++){
        if(parent[i] == i){
            totalPairs -= ((long long)size[i]
                          * (size[i]-1))/2;
        }
    }
    return totalPairs;
```

understood via example

If current node : i is representative of CC, then it subtracts pair of node within that component that can reach other from 'totalPairs'.

In set of size[i] nodes, each node can reach size[i]−1 other nodes, so there are: size[i] × (size[i]−1) edges.
But each pair of node contributes 2 directed edges (one in each dirn.), so we divide it by 2 to get no. of pairs.

Dry run: n = 7, edges = [[0,2],[0,5],[2,4],[1,6],[5,4]]

Initialization: parent = [0,1,2,3,4,5,6], rank = [0,0,0,0,0,0,0], totalPairs: 21

Iteration over edges
- Edge [0,2] → parents are diff., so we union them → parent[2]: 0.
- Edge [0,5] → " → parent[5]: 0
- Edge [2,4] → " → parent[4]: 0
- Edge [1,6] → " → parent[6]: 1
- Edge [5,4] → parent of 5 & 4 are same (i.e 0), so we don't do anything

∴ Parent = [0, 1, 0, 3, 0, 0, 1]

Counting size of each component: size = [4, 2, 0, 1, 0, 0, 0]

Calculating total unreachable pairs: totalPairs: $21 - \frac{4(4-1)}{2} - \frac{2(2-1)}{2} - \frac{1(1-1)}{2} = 14$

Q) DETECT CYCLES IN 2D GRID:

Given a 2D array of characters grid of size 'm×n', you need to find if there exists any cycle consisting of the same value in grid.
A cycle is a path of length 4 or more in the grid that starts & ends at the same cell. From a given cell, you can move to one of the cells adjacent to it - in one of the 4 directions (↑, ↓, ←, →), if it has the same value of the current cell.
Also, you can't move to the cell you visited in your last move.

Ex: Ip:

c	c	c	a
c	d	c	c
c	c	e	c
f	c	c	c

O/P: true

Ip:

a	b	b
b	z	b
b	b	a

grid = [['a','b','b'],['b','z','b'],['b','b','a']]

O/P: false.

A) Hint: Start DFS from each cell & keep marking cell as visited, if we get cycle from 1 cell, then we get ans.

Code:

```
vector<vector<int>> dir = {{0,1},{0,-1},{1,0},{-1,0}};   → 4 directions (→,←,↑,↓)
bool containsCycle(vector<vector<char>> &grid){
    int m = grid.size(), n = grid[0].size();
    vector<vector<int>> visited(m, vector<int>(n, 0));
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(!visited[i][j] and dfs(i, j, -1, -1, grid, visited)){
                return true;
            }
        }
    }
    return false;
}
```

```
bool dfs(int x, int y, int prev_x, int prev_y, vector<vector<char>>& grid,
         vector<vector<int>>& visited){
    if(visited[x][y]) return true;
    visited[x][y] = 1;
    for(auto d : dir){
        int nx = x + d[0];
        int ny = y + d[1];
        if(nx>=0 and nx<grid.size() and ny>=0 and ny<grid[0].size()){
            if(!(nx==prev_x and ny==prev_y) and grid[nx][ny]==grid[x][y])
            {
                if(dfs(nx, ny, x, y, grid, visited)) return true;
            }
        }
    }
    return false;
}
```

Suggestion: Try coding this again :)

KRUSKAL & PRIMS ALGORITHMS

Minimum Spanning Tree (MST): (graph having no cycles)

- Tree ——→ no cycles
- Includes all the nodes of graph
- Sum of edge wts is minimum

(Subgraph having all vertices but min. no. of edges available to keep it connected)

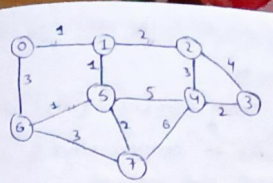
- Algo. to solve: Prims, Kruskals (both are 'greedy')
  - ↳ easier than Prims (Sanket Sir recommendation)

Kruskal's:

- One by one keep picking edges with min. weight
- If choosing an edge forms a cycle, avoid it, else use it
- Here ip in form of 'edge list' → we can sort by wt & use 'DSU'

* **Note** Read 'DSU' & 'Kruskal's'
  - ↳ (It will make soln. very easy)

Cycle detection: if 2 nodes have diff parent, no cycle! but if 2 nodes have same parent → cycle detected

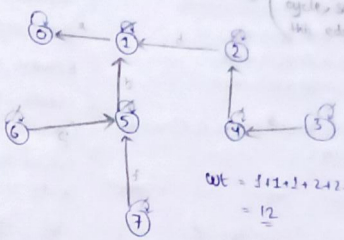


Sorted edge list

(0,1), (1,5), (5,6), (1,2), (5,4), (5,7), (0,6), (2,4), (6,7), (2,3), (5,4), (4,3)

(picking will create cycle, so we avoid this edge)

(order of drawing - alphabets)



wt = 1+1+1+2+2+2+3 = 12

Greedy choice → choosing smallest edge wt which doesn't form cycle

(0-based idx)

Code (Implementation) → $T_c = O(V + E \log E)$ (sorting) (while loop) → find, Union = $\log^* V \sim$ const.

```
#define ll long long int
int find(vector<int> &parent, int a){
    return parent[a] = ((parent[a]==a)? a : find(parent, parent[a]));
}
void Union(vector<int> &parent, vector<int> &rank, int a, int b){
    a = find(parent, a);
    b = find(parent, b);
    if(a==b) return;
    if(rank[a]>=rank[b]){
        rank[a]++;
        parent[b] = a;
    }else{
        rank[b]++;
        parent[a] = b;
    }
}
struct Edge{
    int src;
    int dest;
    int wt;
};
bool cmp(Edge e1, Edge e2){   // custom comparator for sorting
    return e1.wt < e2.wt;
}
ll kruskals(vector<Edge> &input, int n, int e){
    sort(input.begin(), input.end(), cmp);
    vector<int> parent(n+1), rank(n+1, 1);
    for(int i=0; i<=n; i++) parent[i] = i;   // initially every element is parent of themselves
    int edgeCount = 0, i = 0;
    ll ans = 0;
    while(edgeCount < n-1 and i < input.size()){   // we need exact 'n-1' nodes/edges to connect all vertices; will tell us at which edge we are; 0 based indexing
        Edge curr = input[i];   // → because ip is sorted, so we will get min. edge wt
        int srcPar = find(parent, curr.src);
        int destPar = find(parent, curr.dest);
        if(srcPar != destPar){
            // include edge as this will not make cycle
            Union(parent, rank, srcPar, destPar);
            ans += curr.wt;
            edgeCount++;
        }
        i++;   // → doesn't matter you picked last edge or not, we still need to go to next edge
    }   // adding this contn. because if we can't find edges btw all vertices, 'i' will go out of bounds & give error
    return ans;
}
```

```
int main(){
    int n,e;                    // vertices
    cin>>n>>e;                  // edges
    vector<Edge> v(e);
    for(int i=0; i<e; i++){
        cin>>v[i].src;
        cin>>v[i].dest;
        cin>>v[i].wt;
    }
    cout<<kruskals(v,n,e)<<endl;
    return 0;
}
```

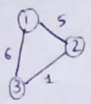
Q) CONNECTING CITIES WITH MIN COST

N cities → 1 to N.

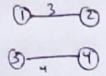
You are given `connections[i] = [city1, city2, cost]` representing cost to connect city1 & city2.

The connection is bidirectional.

Return min. cost so that for every pair of cities, there exists a path of connections (possibly of length 1) that connects those 2 cities together. The cost is sum of connection costs used. If task is impossible, return -1

eg: i/p: 

o/p: 6

i/p: 

o/p: -1

Hint: → we check no. of 'cc' → if >1 → return -1
→ else → we Kruskal algo.

Code

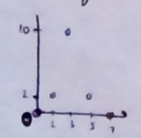
```
//Kruskal Algo.
//Finding connected_comp.
// algo. → 'n' → vertices → global
// remove ('add_edge') func.

int connectingCities(vector<Edge> v, int n, int e){
    // ans=0;
    if(connected_components()>=2) ans=-1;
    else{
        ans = kruskals(v,n,e);
    }
    return ans;
}
```

```
int main(){
    int e;                      // vertices, edges
    cin>>n>>e;
    vector<Edge> v(e);
    graph.resize(n, list<int>());
    for(int i=0; i<e; i++){
        cin>>v[i].src>>v[i].dest>>v[i].wt;
        graph[v[i].src-1].push_back(v[i].dest-1);   // here we have 1-based indexing
        graph[v[i].dest-1].push_back(v[i].src-1);
    }
    cout<<connectingCities(v,n,e)<<endl;
    return 0;
}
```

Q) MIN COST TO CONNECT ALL POINTS

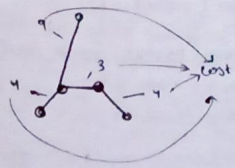
You're given an array 'points' representing integer co-ordinates of some points on a 2D-plane, where `points[i] = [xi, yi]`. The cost of connecting 2 points `[xi, yi]` & `[xj, yj]` is the manhattan distance btw them: $|x_i - x_j| + |y_i - y_j|$, where |val| denotes absolute value of val.

Return min. cost to make all points connected. All points are connected if there's exactly 1 simple path btw any 2 points.

eg: i/p: 

o/p: 20

[[0,0],[2,2],[3,10],[5,2],[7,0]]

eg: 

i/p: [[3,12],[-2,5],[-4,1]]

o/p: 18

A) Here points are given → now will make graph → connecting all vertices → we will get complete graph

{ Euclidian distance = $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$
Manhattan distance = $|x_2-x_1| + |y_2-y_1|$ }

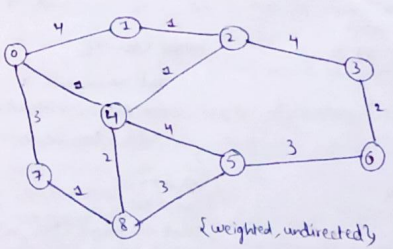
Now we will apply Kruskal's algo

Code

```
// Kruskal's Algo.
int minCostConnectPoints(vector<vector<int>>& points){
    int n = points.size();
    vector<Edge> edges;
    for(int i=0; i<n; i++){
        for(int j=i+1; j<n; j++){
            int dist = abs(points[i][0]-points[j][0]) + abs(points[i][1]-points[j][1]);
            edges.push_back({i,j,dist});
        }
    }
    return kruskals(edges, n, edges.size());   // vector, total vertices, total edges
}
```

Prim's:

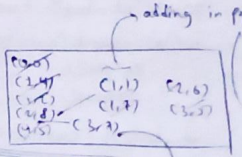
- Here 'priority queue' data str. is heavily used. (+BFS + Greedy)
- One specific node is fixed as the starting point of finding the subgraph.
- Includes all the nodes of graph } means from source node → we keep on finding
- Sum of edge wts is minimum. } min. wt edges & add to our tree

eg: 

{weighted, undirected}

- In prims we start with a src & try to discover other nodes

Priority queue → adding in PQ → (it will come to top again)



cout: (0,0) (1,4) (2,6) (1,1) (1,7) (3,5)

(3,7) came from '0', so we won't consider ⓪—① edge

Visited: 0, 4, 2, 8, 1, 7, 6, 3, 5

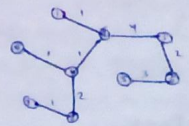
{ if dis.wt > curr wt → overwrite disc.wt }

(when comes out, check if it's there in 'visited'. If yes, then ignore that pair)

node	disc. wt (discovery weight)	parent
0	-1	X
1	~~∞~~ 1	2
2	~~∞~~ 1	4
3	~~∞~~ 4	2
4	~~∞~~ 1	0
5	~~∞~~ ~~4~~ 3	~~4~~ 6
6	~~∞~~ 2	3
7	~~∞~~ 1	~~8~~ 8
8	~~∞~~ 2	4

mapping

pair <wt, vertex>

Final tree: 

Now → greedy soln. → because initially ⓪—① we had 4, but later we got '3' → so we can update it ! ✓✓ not final ans but suitable answer

total cost = 15

Algo: (in English)
- DS: visited set, priority_queue<pair>, unordered_map
- Insert the pair of <-1, src> in the pq
- One by one remove the root element in the pq
- If the root element is already visited, then we will just continue
- We mark the root visited
- We store the weight of the pair in ans
- Update the mapping
- Go to every neighbour of the curr ele., & add only those which are non-visited & have a better weight proposition.

Code {1-based indexing} (Implementation) → $T_c = O(E \log E)$

```
#define ll long long int
vector<list<pair<int,int>>> gr;
void add_edge(int u, int v, int wt, bool bi_dir = true){
    gr[u].push_back({v,wt});
    if(bi_dir) gr[v].push_back({u,wt});
}
ll prims(int src, int n){
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    unordered_set<int> vis;
    vector<int> par(n+1);
    unordered_map<int,int> mp;
    for(int i=  ; i<=n; i++) mp[i]=INT_MAX;
    pq.push({0,src});
    mp[src]=0;
    int total_count=0, result=0;
    while(total_count<n and !pq.empty()){
        pair<int,int> curr = pq.top();
        if(vis.count(curr.second)){
            pq.pop();
            continue;
        }
        vis.insert(curr.second);
        total_count++;
        result+= curr.first;
        pq.pop();
        for(auto neighbour: gr[curr.second]){
            if(!vis.count(neighbour.first) and mp[neighbour.first] > neighbour.second){
                pq.push({neighbour.second, neighbour.first});
                par[neighbour.first] = neighbour.second;
                mp[neighbour.first] = neighbour.second;
            }
        }
    }
    return result;
}
```

by default → max heap, to convert into min heap

0 → n-1 edges atmost

sum of wts

curr ∈ priority queue

neighbour ∈ graph

```
int main(){
    int n,m;
    cin>>n>>m;
    gr.resize(n+1, list<pair<int,int>>());
    while(m--){
        int u,v,wt;
        cin>>u>>v>>wt;
        add_edge(u,v,wt);
    }
    int src;
    cin>>src;
    cout<<prims(src,n)<<endl;
    return 0;
}
```

1-based indexing

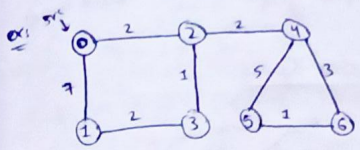
Prims (on)

Priority queue can still be optimized using 'fibonacci heap' → but in general we use binary heap only in C++/Java.

DIJKSTRA'S ALGORITHM

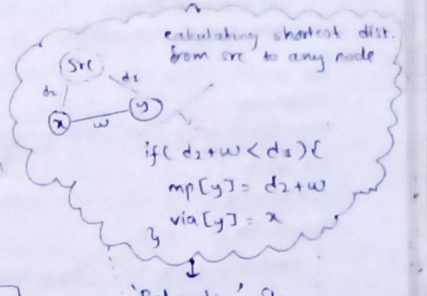
Given an undirected weighted graph, along with a source & dest. find the shortest path btw the src & dest. in terms of sum of edge cost!

Dijkstra's Algorithm!

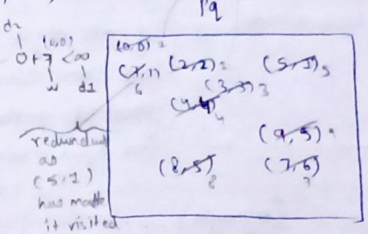


<dist, node>

order of removal from pq



Node	dist	Via (=parent)
0	0	X
1	~~5~~	~~3~~
2	~~2~~	0
3	~~3~~	2
4	~~4~~	2
5	~~8~~	~~6~~
6	7	4



Visited: 0 2 3 4 1 6 5

-> At every step, we pop ele. from 'pq' & the ele. which we popped, it means we got best distance for that & we do relaxation process!

Code: (Implementation) {0-based indexing} → $T_c = O(V \log V + E \log V)$

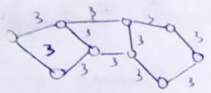
```
#define ll long long int
vector<list<pair<int,int>>>gr;
void add_edge(int u, int v, int wt, bool bi_dir=true){
    gr[u].pushback({v,wt});
    if(bi_dir) gr[v].push_back({u,wt});
}
unordered_map<int,int> djikstra(int src, int n){
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    unordered_set<int> vis;                                   // {wt, node}
    vector<int> via(n);
    unordered_map<int,int> mp;
    for(int i=0; i<n; i++) mp[i]=INT_MAX; // O(V)
    pq.push({0,src});
    mp[src]=0;
    while(!pq.empty()){ // O((V+E) log V)
        pair<int,int> curr = pq.top();
        if(vis.count(curr.second)){
            pq.pop();
            continue;
        }
        vis.insert(curr.second); pq.pop();
        for(auto neighbour: gr[curr.second]){
            if(!vis.count(neighbour.first) and mp[neighbour.first] > mp[curr.second] + neighbour.second){
                pq.push({mp[curr.second]+neighbour.second, neighbour.first});
                via[neighbour.first] = neighbour.second;
                mp[neighbour.first] = mp[curr.second] + neighbour.second;
            }
        }
    }
    return mp;
}
```

```
int main(){
    int n, m;
    cin>>n>>m;
    gr.resize(n, list<pair<int,int>>());
    while(m--){
        int u,v,wt;
        cin>>u>>v>>wt;
        add_edge(u,v,wt);
    }
    int src;
    cin>>src;
    unordered_map<int,int> sp= dijkstra(src,n);
    int dest;
    cin>>dest;
    for(auto p: sp){
        cout<<p.first<<" "<<p.second<<endl;
    }
    cout<<sp[dest]<<endl;
    return 0;
}
```

Kruskal — better to use when: no. of edges < no. of vertex — sparse graph

Prim's — better to use when: no. of edges > no. of vertex — dense graph

Special Cases



If all edges have same weight, you can simply apply BFS & find shortest path!

Just equal → so you can consider it as 0 also

0-1 BFS  {# google}

Here we have only '2' no.'s as edge wt's! Now to solve we have 2 methods

- Dijkstra
- BFS using deque
  - Here we will add path/node with wt '0' to front of deque & wt '1' to back of deque & solve

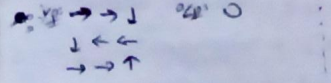
MIN COST TO MAKE ATLEAST ONE VALID PATH IN A GRID

Given 'm×n' grid. Each cell of the grid has a sign pointing to the next cell you should visit if you are currently in this cell. The sign of grid[i][j] can be:

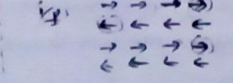
- 1 → go to cell to the right (i.e. grid[i][j] to grid[i][j+1])
- 2 → go to cell to the left (i.e. grid[i][j] to grid[i][j-1])
- 3 → go to the lower cell (i.e. grid[i][j] to grid[i+1][j])
- 4 → go to the upper cell (i.e. grid[i][j] to grid[i-1][j])

Notice that there could be some signs on the cells of the grid that point outside the grid.

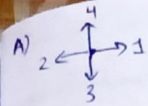
You will initially start at the upper left cell (0,0). A valid path in the grid is a path that starts from the upper left cell (0,0) & ends at the bottom-right cell (m-1, n-1) following the signs on the grid. The valid path doesn't have to be the shortest. You can modify the sign on a cell with cost = 1. You can modify the sign on a cell one time only. Return the minimum cost to make grid have atleast one valid path.



ex: 0



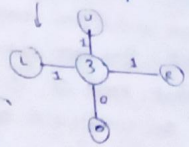
ex: 3 → change '(')

A) 

ex:

1	3	4
2	3	3
1	1	1

Now imagining it as a graph

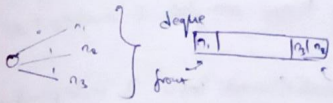


Dijkstra's Algo → (Now we need path with min. cost i.e. shortest path)

0-1 BFS

Now in this we use deque so insertion/deletion → $T_c = O(1)$

But in priority queue → insertion/deletion → $T_c = O(\log n)$ } So we can optimize our code



So now this prob T_c can be converted into normal BFS T_c i.e. $O(V+E)$

Code:

```
int minCost(vector<vector<int>>& grid){
    int n= grid.size();
    int m= grid[0].size();
    int dx[4]= {0,0,1,-1};
    int dy[4]= {1,-1,0,0};
    int dist[n][m]; → stores 'src' to each cell distance
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            dist[i][j]= INT_MAX;
        }
    }
    deque<pair<int,int>> dq;
    dq.push_back({0,0});
    dist[0][0]=0;
    while(!dq.empty()){
        auto curr = dq.front();
        dq.pop_back();
        int x= curr.first, y= curr.second;
        int dir = grid[x][y];
        for(int i=0; i<4; i++){
            int nx = x+ dx[i];
            int ny= y+ dy[i];
            int edgewt = 1; → 0-based indexing
            if(i+1==dir) edgewt = 0;
            if(nx<n and ny<m and nx>=0 and ny>=0){
                if(dist[nx][ny] > dist[x][y]+edgewt){
                    dist[nx][ny] = dist[x][y]+edgewt;
                    if(edgewt==1) dq.push_back({nx,ny});
                    else dq.push_front({nx,ny});
                }
            }
        }
    }
    return dist[n-1][m-1];
}
```

$\begin{pmatrix}0\\1\end{pmatrix}$ $\begin{pmatrix}0\\-1\end{pmatrix}$ $\begin{pmatrix}1\\0\end{pmatrix}$ $\begin{pmatrix}-1\\0\end{pmatrix}$ represent left, right, down, up → direction arrays

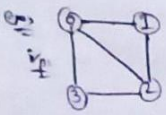
Q) IS GRAPH BIPARTITE ?

There is an undirected graph with 'n' nodes, where each node is numbered btw 0 & n-1. You are given a 2D array 'graph' where 'graph[u]' is an array of nodes that node 'u' is adjacent to. More formally, for each 'v' in graph[u], there's an undirected edge btw node 'u' & node 'v'. The graph has following properties:

- There are no self-edges (graph[u] doesn't contain u)
- There are no parallel edges (graph[u] doesn't contain duplicate values)
- If 'v' is in graph[u], then 'u' is in graph[v] (the graph is undirected)
- The graph may not be connected, meaning there maybe 2 nodes 'u' & 'v' such that there is no path btw them.

A graph is bipartite if the nodes can be partitioned into 2 independent sets 'A' & 'B' such that every edge in the graph connects a node in set A & a node in set B.

Return true if & only if it's bipartite.

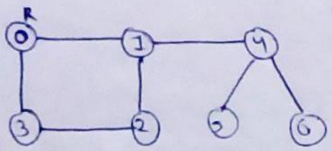
ex:  o/p: false

 o/p: true

ex: we can partition the nodes into 2 sets: {0,2} & {1,3}.

(* in last one, 0-2) connected i.e. same set → edge is present, so it was false)

A) (~ 2 color problem) → Set Red, Blue (0) (2)



* Input → in form of adjacency list

queue: 0 1 3 4 2 5 6

Visited: 0, 1, 3, 4, 2, 5, 6

vector:

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

0 → red → queue → pop_out → add children → & mark as visited → add color in array

1, 3 → blue → pop_out → add children → mark them as visited → add color in array

4, 2 → red → " → " → " → "

3 → 2) 0 → visited, so ignore

5, 6 → pop_out → mark as visited → add color in array!

Code → $T_c = O(V+E)$ ~same as that of BFS

```
bool isBipartite(vector<vector<int>>& grid){
    int n = graph.size();
    vector<int> color(n, -1);
    for(int i = 0; i<n; i++){
        if(color[i] == -1){
            if(bfs(i, color, n, graph) == false) return false;
        }
    }
    return true;
}
```

// If we have disconnected components, we need to call 'bfs' regularly, so we are doing it like this

```
bool bfs(int src, vector<int> color, int n, vector<vector<int>>& graph){
    queue<int> qu;
    qu.push(src);
    color[src] = 0;
    while (!qu.empty()){
        int curr = qu.front();
        qu.pop();
        for(auto neighbour : graph[curr]){
            if(color[neighbour] == -1){
                color[neighbour] = !color[curr];
                qu.push(neighbour);
            }
            else if(color[curr] == color[neighbour]){
                return false;
            }
        }
    }
    return true;
}
```

March, 2024

• **min_element()** : returns min ele.

Syntax: *min_element(first index, last index);

ex: vector<int> v {4,2,5,9,1};
cout << *min_element(v.begin(), v.end()); } → o/p: 1

• **max_element()** : returns max ele.

Syntax: *max_element(first index, last index);

ex: cout << *max_element(v.begin(), v.end()); } → o/p: 9

• **next_permutation()** : returns next lexographically greater permutation of the elements in the container passed to it as an argument

Syntax: next_permutation(begin, end);

(iterator pointing to 1st element of container) (iterator pointing to just after the last element of container)

ex: arr[] = {1,3,2}
next_permutation(arr, arr+3); (or) next_permutation(v.begin(), v.end())

o/p: arr[] = {2,1,3}

Explanation: all permutations of {1,2,3} are {{1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1}}. So next perm. after {1,3,2} is {2,1,3}.

• **__builtin_popcount()** : counts the no. of bit positions in binary representation of an integer which is set to 1.

ex: int n = 7
cout << __builtin_popcount(n); } → o/p: 3 {as 7 = 111 in binary}

If our dtype = long long → slight change in keyword

{ __builtin_popcountll(); }

• **string(a,b)** = returns character 'b' repeated 'a' times.

→

- **lower_bound()** : returns an iterator pointing to the first element that is not less than the given value in sorted array.

*syntax : lower_bound (arr.begin(), arr.end(), element) - arr.begin()

(if we subtract → we get lower_bound index)

ex: {1, 4, 4, 4, 4, 9, 9, 10, 11}

element = 4

↳ : lower_bound ⇒ 1

- **upper_bound()** : returns an iterator pointing to the first element that is greater than the given value in sorted array.

*syntax: upper_bound (arr.begin(), arr.end(), element) - arr.begin()

→ Here also need to check if index going out of bounds.

(if we subtract → we get upper_bound index)

ex: {1, 4, 4, 4, 4, 9, 9, 10, 11}

element = 4

↳ : upper_bound ⇒ 5

element = 12

↳ : upper_bound ⇒ 9 ~ {index → out of bounds}

↳ so decrement by '1'.