CRACK MAANG COMPANIES

Curated topic-wise DSA Questions with C++ codes

DAARIS AMEEN

TRIE

Implement Trie – 1

Problem Statement: Implementing insertion, search, and startWith operations in a trie or prefix-tree.

Implementation:

Type 1: To insert a string “word” in Trie.

Type 2: To check if the string “word” is present in Trie or not.

Type 3: To check if there is any string in the Trie that starts with the given prefix string “word”.

Example: Input: type = {1, 1, 2, 3, 2}; value = {"hello", "help", "help", "hel", "hel"}; Output: true true false Explanation: Trie object initialized “hello” inserted in the trie. “help” insertedin the trie “help” searched in the trie //returns true Checked if any previously inserted word has the prefix “hel” //return true “hel” searches in the trie //returns true

```
#include<bits/stdc++.h>
using namespace std;
#define IOS ios::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl "\n"
#define MAXN 100001
#define INF 1e18+1
struct Node {
	Node *links[26];
	bool flag = false;
	//checks if the reference trie is present or not
	bool containKey(char ch) {
		return (links[ch - 'a'] != NULL);
	}
	//creating reference trie
	void put(char ch, Node *node) {
		links[ch - 'a'] = node;
	}
	//to get the next node for traversal
	Node *get(char ch) {
		return links[ch - 'a'];
	}
	//setting flag to true at the end of the word
	void setEnd() {
		flag = true;
	}
	//checking if the word is completed or not
	bool isEnd() {
		return flag;
	}
};
class Trie {
```

```
private:
        Node* root;
public:
        Trie() {
                //creating new obejct
                root = new Node();
        }

        void insert(string word) {
                //initializing dummy node pointing to root initially
                Node *node = root;
                for (int i = 0; i < word.size(); i++) {
                        if (!node->containKey(word[i])) {
                                node->put(word[i], new Node());
                        }
                        //moves to reference trie
                        node = node->get(word[i]);
                }
                node->setEnd();
        }

        bool search(string word) {
                Node *node = root;
                for (int i = 0; i < word.size(); i++) {
                        if (!node->containKey(word[i])) {
                                return false;
                        }
                        node = node->get(word[i]);
                }
                return node->isEnd();
        }

        bool startsWith(string prefix) {
                Node* node = root;
                for (int i = 0; i < prefix.size(); i++) {
                        if (!node->containKey(prefix[i])) {
                                return false;
                        }
                        node = node->get(prefix[i]);
                }
                return true;
        }
};

int main()
{
        int n = 5;
        vector<int>type = {1, 1, 2, 3, 2};
        vector<string>value = {"hello", "help", "help", "hel", "hel"};
        Trie trie;
```

```
	for (int i = 0; i < n; i++) {
		if (type[i] == 1) {
			trie.insert(value[i]);
		}
		else if (type[i] == 2) {
			if (trie.search(value[i])) {
				cout << "true" << "\n";
			}
			else {
				cout << "false" << "\n";
			}
		}
		else {
			if (trie.startsWith(value[i])) {
				cout << "true" << "\n";
			}
			else {
				cout << "false" << "\n";
			}
		}
	}
}
```

Implement Trie – II

Problem Statement: Implement a data structure "TRIE" from scratch. Complete some functions.

1) Trie(): Initialize the object of this "TRIE" data structure.

2) insert("WORD"): Insert the string "WORD" into this "TRIE" data structure.

3) countWordsEqualTo("WORD"): Return how many times this "WORD" is present in this "TRIE".

4) countWordsStartingWith("PREFIX"): Return how many words are there in this "TRIE" that have the string "PREFIX" as a prefix.

5) erase("WORD"): Delete this string "WORD" from the "TRIE".

Note:

1. *If erase("WORD") function is called then it is guaranteed that the "WORD" is presentin the "TRIE".*
2. *If you are going to use variables with dynamic memory allocation then you need to release the memory associated with them at the end of your solution.*

Example: Input: 1 6 insert samsung insert samsung insert vivo erase vivo countWordsEqualTo samsung countWordsStartingWith vi Output: 2 0

Explanation:

Insert "samsung": we are going to insert the word "samsung" into the "TRIE". Insert "samsung": we are going to insert the word "samsung" again into the "TRIE". Insert "vivo": we are going to insert the word "vivo" into the "TRIE". Erase "vivo": we are going to delete the word "vivo" from the "TRIE". countWordsEqualTo "samsung": There are two instancesof "samsung" is present in "TRIE". countWordsStartWith "vi":There is not a single word inthe "TRIE" that starts from the prefix "vi".

```
#include <bits/stdc++.h>

using namespace std;
struct Node {
  Node * links[26];
  int cntEndWith = 0;
  int cntPrefix = 0;

  bool containsKey(char ch) {
    return (links[ch - 'a'] != NULL);
  }
  Node * get(char ch) {
    return links[ch - 'a'];
  }
  void put(char ch, Node * node) {
    links[ch - 'a'] = node;
  }
  void increaseEnd() {
    cntEndWith++;
  }
  void increasePrefix() {
    cntPrefix++;
  }
  void deleteEnd() {
    cntEndWith--;
  }
  void reducePrefix() {
    cntPrefix--;
  }
  int getEnd() {
    return cntEndWith;
  }
  int getPrefix() {
    return cntPrefix;
  }
};
class Trie {
  private:
    Node * root;

  public:
    /** Initialize your data structure here. */
    Trie() {
      root = new Node();
    }

  /** Inserts a word into the trie. */
  void insert(string word) {
    Node * node = root;
    for (int i = 0; i < word.length(); i++) {
      if (!node -> containsKey(word[i])) {
        node -> put(word[i], new Node());
      }
      node = node -> get(word[i]);
      node -> increasePrefix();
    }
    node -> increaseEnd();
  }

 int countWordsEqualTo(string &word)
    {
        Node *node = root;
        for (int i = 0; i < word.length(); i++)
        {
            if (node->containsKey(word[i]))
            {
                node = node->get(word[i]);
            }
            else
            {
                return 0;
            }
        }
        return node->getEnd();
    }


  int countWordsStartingWith(string & word) {
    Node * node = root;
    for (int i = 0; i < word.length(); i++) {
      if (node -> containsKey(word[i])) {
        node = node -> get(word[i]);
      } else {
        return 0;
      }
    }
    return node -> getPrefix();
  }

  void erase(string & word) {
    Node * node = root;
    for (int i = 0; i < word.length(); i++) {
      if (node -> containsKey(word[i])) {
        node = node -> get(word[i]);
        node -> reducePrefix();
      } else {
        return;
      }
    }
    node -> deleteEnd();
  }
};

int main() {
  Trie T;
  T.insert("apple");
  T.insert("apple");
  T.insert("apps");
  T.insert("apps");
  string word1 = "apps";
  cout<<"Count Words Equal to "<< word1<<"
"<<T.countWordsEqualTo(word1)<<endl;
  string word2 = "abc";
  cout<<"Count Words Equal to "<< word2<<"
"<<T.countWordsEqualTo(word2)<<endl;
  string word3 = "ap";
  cout<<"Count Words Starting With "<<word3<<"
"<<T.countWordsStartingWith(word3)
  <<endl;
  string word4 = "appl";
  cout<<"Count Words Starting With "<< word4<<"
"<<T.countWordsStartingWith(word4)
  <<endl;
  T.erase(word1);
  cout<<"Count Words equal to "<< word1<<"
"<<T.countWordsEqualTo(word1)<<endl;
  return 0;
}
```

Number of Distinct Substrings in a String Using Trie

Problem Statement: Given a string of alphabetic characters. Return the count of distinctsubstrings of the string(including the empty string) using the Trie data structure.

Example 1: Input: S1= “ababa”

Output: Total number of distinct substrings : 10 Explanation: All the substrings of the string are a, ab, aba, abab, ababa, b, ba, bab, baba, a, ab, aba, b, ba, a. Many of the substrings are duplicated. The distinct substrings are a, ab, aba,abab, ababa, b, ba, bab, baba. Total Count of the distinct substrings is 9 + 1(for empty string) = 10.

Example 2: Input: S1= “ccfdf”

Output: Total number of distinct substrings : 14

Explanation: All the substrings of the stringare c, cc, ccf, ccfd, ccfdf, c, cf, cfd, cfdf, f, fd, fdf, d, df, f. Many of the substrings are duplicated. The distinct substrings are c, cc, ccf, ccfd, ccfdf, cf, cfd, cfdf, f, fd, fdf, d, df. Total Count of the distinct substrings is 13 + 1(for empty string) = 14.

```
#include<iostream>

using namespace std;

struct Node {
  Node * links[26];

  bool containsKey(char ch) {
    return (links[ch - 'a'] != NULL);
  }

  Node * get(char ch) {
    return links[ch - 'a'];
  }

  void put(char ch, Node * node) {
    links[ch - 'a'] = node;
  }
};

int countDistinctSubstrings(string & s) {
  Node * root = new Node();
  int count = 0;
  int n = s.size();
  for (int i = 0; i < n; i++) {
    Node * node = root;

    for (int j = i; j < n; j++) {
      if (!node -> containsKey(s[j])) {
        node -> put(s[j], new Node());
        count++;
      }
      node = node -> get(s[j]);
    }

  }
  return count + 1;
}

int main() {
  string s1 = "ababa";
  cout << "Total number of distinct substrings : " <<
countDistinctSubstrings(s1);
  cout << "\n";
  string s2 = "ccfdf";
  cout << "Total number of distinct substrings : " <<
countDistinctSubstrings(s2);

  return 0;
}
```

Maximum Xor Queries | Trie

Problem Statement:

You are given an array/list 'ARR' consisting of 'N' non-negative integers. You are also givena list 'QUERIES' consisting of 'M' queries, where the 'i-th' query is a list/array of two non-negative integers 'Xi', 'Ai', i.e 'QUERIES[i]' = ['Xi', 'Ai'].

The answer to the ith query, i.e 'QUERIES[i]' is the maximum bitwise xor value of 'Xi' withany integer less than or equal to 'Ai' in 'ARR'.

You should return an array/list consisting of 'N' integers where the 'i-th' integer is the answer of 'QUERIES[i]'.

Example 1: Input:

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

Output: 3 7 -1

Explanation: In the first test case, the answer of query [1, 3] is 3 because 1^2 = 3 and 2 <=3, and the answer of query [5, 6] is 7 because 5 ^ 2 = 7 and 2 <= 6.

In the second test case, no element is less than or equal to 0 in the given array 'ARR'.

Example 2: Input: 2 6 3 6 6 3 5 2 4 6 3 8 1 12 4 5 2 0 0 0 0 0 1 0 1 1

Output: 5 -1 15 1 1

```
#include<bits/stdc++.h>

using namespace std;

struct Node {
  Node * links[2];

  bool containsKey(int ind) {
    return (links[ind] != NULL);
  }
  Node * get(int ind) {
    return links[ind];
  }
  void put(int ind, Node * node) {
    links[ind] = node;
  }
};
class Trie {
  private: Node * root;
  public:
    Trie() {
      root = new Node();
    }

  public:
    void insert(int num) {
      Node * node = root;
      for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        if (!node -> containsKey(bit)) {
          node -> put(bit, new Node());
        }
        node = node -> get(bit);
      }
    }
  public:
    int findMax(int num) {
      Node * node = root;
      int maxNum = 0;
      for (int i = 31; i >= 0; i--) {
        int bit = (num >> i) & 1;
        if (node -> containsKey(!bit)) {
          maxNum = maxNum | (1 << i);
          node = node -> get(!bit);
        } else {
          node = node -> get(bit);
        }
      }
      return maxNum;
    }
};
vector < int > maxXorQueries(vector < int > & arr, vector < vector <
int
>> & queries) {
  vector < int > ans(queries.size(), 0);
  vector < pair < int, pair < int, int >>> offlineQueries;
  sort(arr.begin(), arr.end());
  int index = 0;
  for (auto & it: queries) {
    offlineQueries.push_back({
      it[1],
      {
        it[0],
        index++
      }
    });
  }
  sort(offlineQueries.begin(), offlineQueries.end());
  int i = 0;
  int n = arr.size();
  Trie trie;

  for (auto & it: offlineQueries) {
    while (i < n && arr[i] <= it.first) {
      trie.insert(arr[i]);
      i++;
    }
    if (i != 0) ans[it.second.second] = trie.findMax(it.second.first);
    else ans[it.second.second] = -1;
  }
  return ans;
}
int main() {
  int t;
  cin >> t;
  while (t--) {
    int n, m;
    cin >> n >> m;
    vector < int > arr(n);
    for (int i = 0; i < n; i++) {
      cin >> arr[i];
    }
    vector < vector < int >> queries;

    for (int i = 0; i < m; i++) {
      vector < int > temp;
      int xi, ai;
      cin >> xi >> ai;
      temp.push_back(xi);
      temp.push_back(ai);
      queries.push_back(temp);
    }

    vector < int > ans = maxXorQueries(arr, queries);
    for (int j = 0; j < ans.size(); j++) {
      cout << ans[j] << " ";
    }
    cout << endl;
  }

}
```

TREE

Prorder - Iterative

```
class Solution {

public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> preorder;
        if(root == NULL) return preorder;

        stack<TreeNode*> st;
        st.push(root);
        while(!st.empty()){
            root = st.top();
            st.pop();
            preorder.push_back(root->val);
            if(root->right != NULL){
                st.push(root->right);
            }
            if(root->left!= NULL){
                st.push(root->left);
            }
        }
        return preorder;
    }
};
```

Inorder - Recursive

```
class Solution {
private:
    void dfs(TreeNode *node, vector<int> &inorder) {
        if(node == NULL) return;

        dfs(node->left, inorder);
        inorder.push_back(node->val);
        dfs(node->right, inorder);
    }
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> inorder;
        dfs(root, inorder);
        return inorder;
    }
};
```

postorder 2 stacks

```
class Solution {

public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> postorder;
        if(root == NULL) return postorder;
        stack<TreeNode*> st1, st2;
        st1.push(root);
        while(!st1.empty()) {
            root = st1.top();
            st1.pop();
            st2.push(root);
            if(root->left != NULL) {
                st1.push(root->left);
            }
            if(root->right != NULL) {
                st1.push(root->right);
            }
        }
        while(!st2.empty()) {
            postorder.push_back(st2.top()->val);
            st2.pop();
        }
        return postorder;
    }
};
```

postorder 1 stack

```
class Solution {

public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> postorder;
        if(root == NULL) return postorder;
        stack<TreeNode*> st1;
        TreeNode* current = root;
        while(current != NULL || !st1.empty()) {
            if(current != NULL){
                st1.push(current);
                current = current->left;
            }else{
                TreeNode* temp = st1.top()->right;
                if (temp == NULL) {
                    temp = st1.top();
                    st1.pop();
                    postorder.push_back(temp->val);
                    while (!st1.empty() && temp == st1.top()->right) {
                        temp = st1.top();
                        st1.pop();
                        postorder.push_back(temp->val);
                    }
                } else {
                    current = temp;
                }
            }
        }
        return postorder;
    }
};
```

level order - Iterative

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(root == NULL) return ans;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()) {
            int size = q.size();
            vector<int> level;
            for(int i = 0;i<size;i++) {
                TreeNode *node = q.front();
                q.pop();
                if(node->left != NULL) q.push(node->left);
                if(node->right != NULL) q.push(node->right);
                level.push_back(node->val);
            }
            ans.push_back(level);
        }
        return ans;
    }
};
```

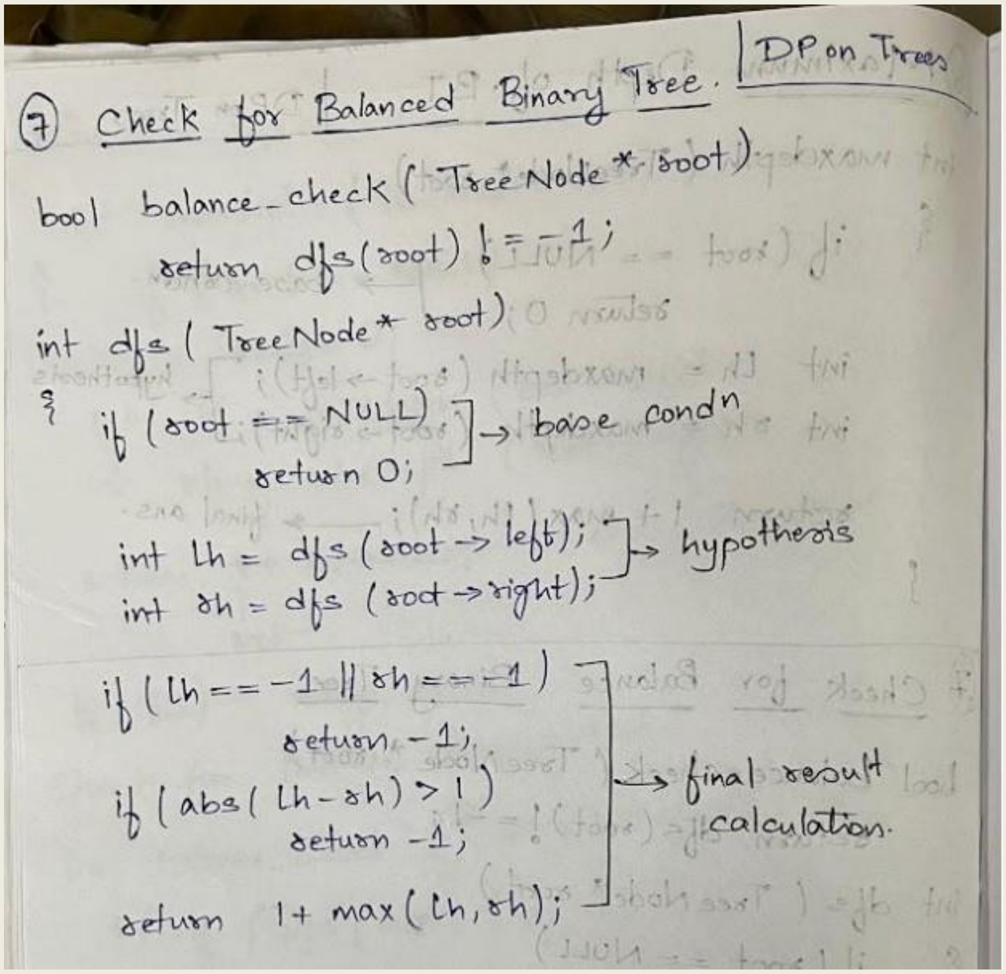
All 3 traversals in ONE code

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {

public:
    vector<int> postorderTraversal(TreeNode* root) {
        stack<pair<TreeNode*,int>> st;
        st.push({root, 1});
        vector<int> pre, in, post;
        if(root == NULL) return post;

        while(!st.empty()) {
            auto it = st.top();
            st.pop();

            // this is part of pre
            // increment 1 to 2
            // push the left side of the tree
            if(it.second == 1) {
                pre.push_back(it.first->val);
                it.second++;
                st.push(it);

                if(it.first->left != NULL) {
                    st.push({it.first->left, 1});
                }
            }

            // this is a part of in
            // increment 2 to 3
            // push right
            else if(it.second == 2) {
                in.push_back(it.first->val);
                it.second++;
                st.push(it);

                if(it.first->right != NULL) {
                    st.push({it.first->right, 1});
                }
            }
            // don't push it back again
            else {
                post.push_back(it.first->val);
            }
        }

        return post;
    }
};
```

Max depth of BT

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;

        int lh = maxDepth(root->left);
        int rh = maxDepth(root->right);

        return 1 + max(lh, rh);
    }
};
```

Check for balanced BT

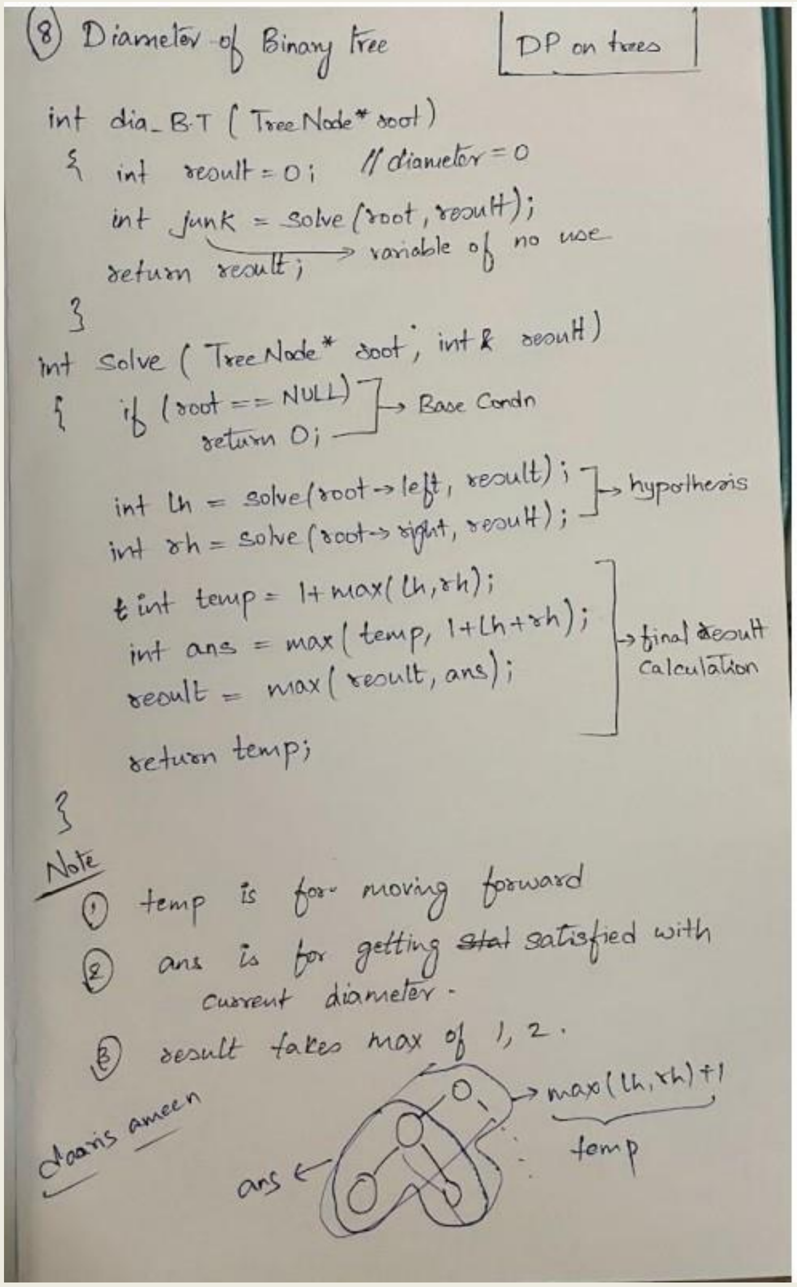


DP on Trees

⑦ Check for Balanced Binary Tree.

```
bool balance_check( TreeNode* root)
    return dfs(root) != -1;

int dfs( TreeNode* root)
{
    if (root == NULL)      ]→ base condn
        return 0;

    int lh = dfs(root -> left);    ]→ hypothesis
    int rh = dfs (root -> right);

    if (lh == -1 || rh == -1)      ]
        return -1;                 ]
    if (abs( lh - rh) > 1)         ]→ final result
        return -1;                 ]  calculation.
    return 1 + max (lh, rh);       ]
```

Diameter of BT



(8) Diameter of Binary tree — DP on trees

```
int dia_B.T ( TreeNode* root)
{  int  result = 0;     // diameter = 0
   int junk = solve(root, result);
                → variable of no use
   return result;
}
int solve ( TreeNode* root, int& result)
{    if (root == NULL)     ⎤→ Base Condn
         return 0;         ⎦

     int lh = solve(root->left, result);   ⎤→ hypothesis
     int rh = solve(root->right, result);  ⎦
     int temp = 1+max(lh,rh);              ⎤
     int ans = max(temp, 1+lh+rh);         ⎥→ final result
     result = max(result, ans);            ⎥  calculation
                                           ⎦
     return temp;
}
```

Note

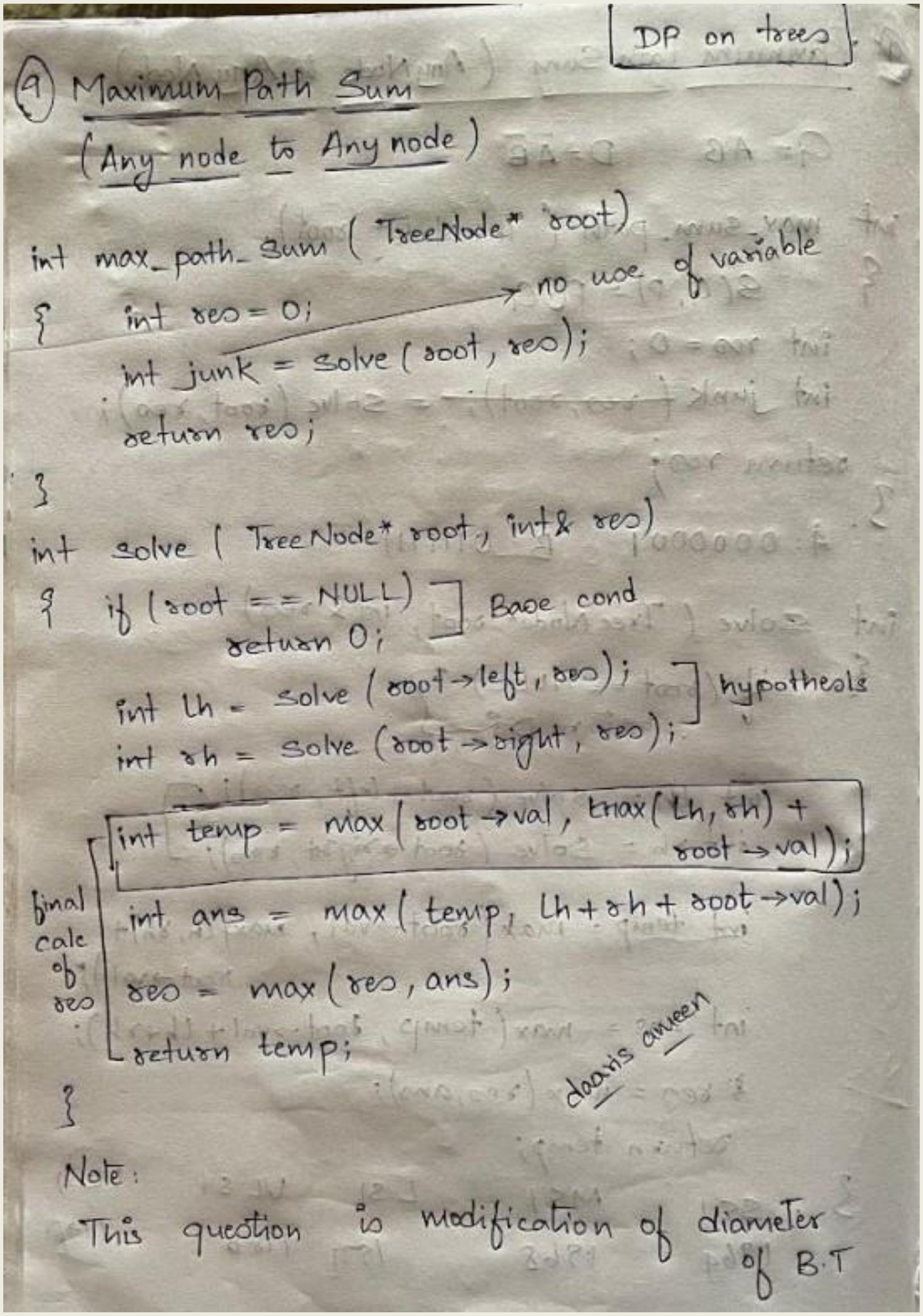
1. temp is for moving forward
2. ans is for getting ~~stat~~ satisfied with current diameter.
3. result takes max of 1, 2.

ans

$\underbrace{\max(lh,rh)+1}_{temp}$

daaris ameen

Maximum path sum (Any node to Any node)



DP on trees

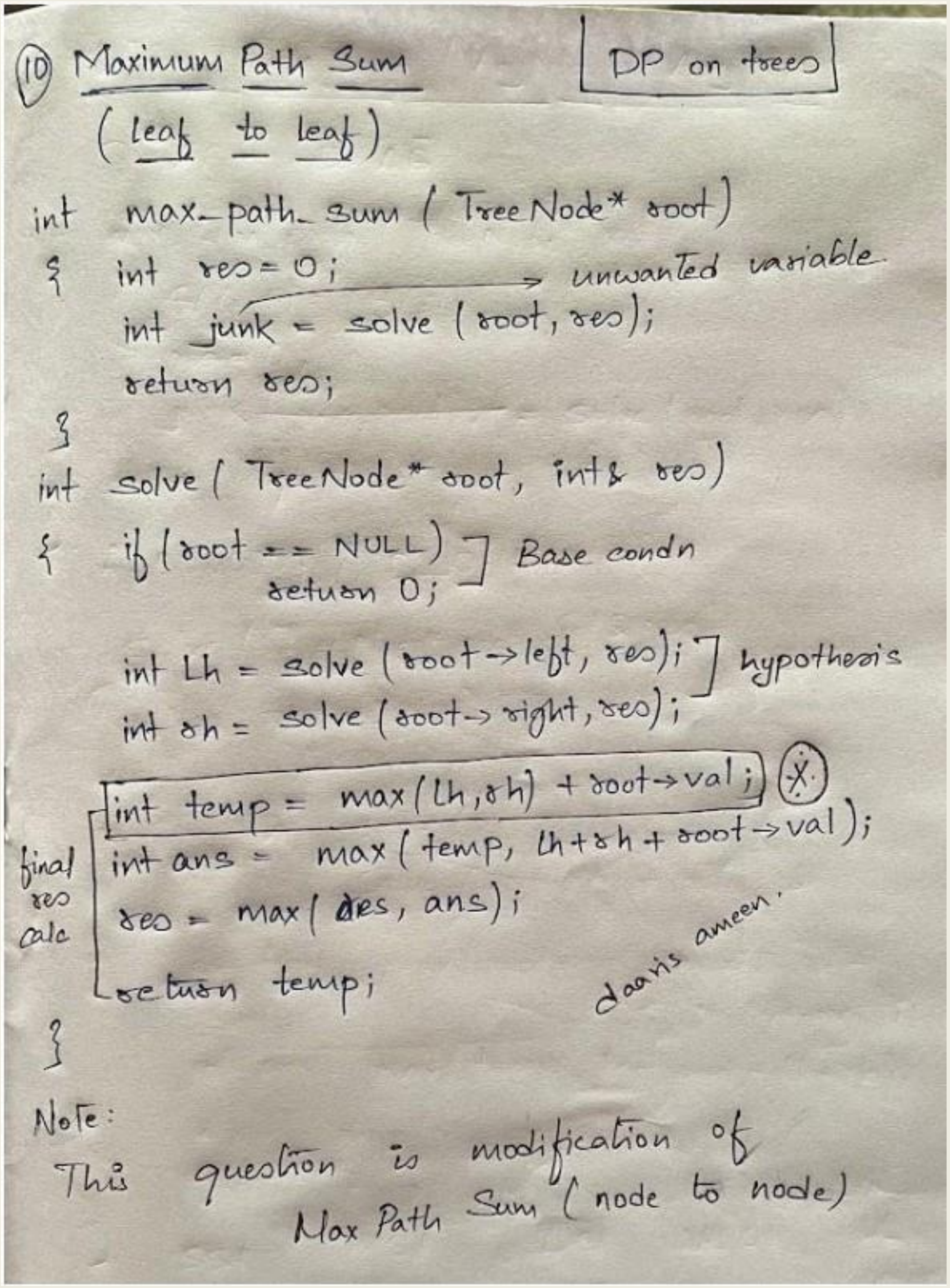
(9) Maximum Path Sum
(Any node to Any node)

```
int max_path_sum( TreeNode* root)
{     int res = 0;          → no use of variable
      int junk = solve ( root, res);
      return res;
}
int  solve ( TreeNode* root, int& res)
{    if (root == NULL)   ] Base cond
          return 0;
     int lh = solve (root->left, res);   ] hypothesis
     int rh = solve (root->right, res);
final [ int temp = max( root->val, max(lh, rh) +
calc                                root->val);
of    int ans = max( temp, lh + rh + root->val);
res   res = max(res, ans);
      return temp;
}
```

daaris ameen

Note:
This question is modification of diameter of B.T

Maximum path sum - Leaf node to Leaf node



(10) Maximum Path Sum

DP on trees

(leaf to leaf)

```
int max_path_sum (TreeNode* root)
{   int res = 0;                          → unwanted variable
    int junk = solve (root, res);
    return res;
}
int solve (TreeNode* root, int& res)
{   if (root == NULL)       ] Base condn
          return 0;

    int lh = solve (root->left, res);   ] hypothesis
    int rh = solve (root->right, res);
final  int temp = max(lh, rh) + root->val;   (X)
res    int ans = max (temp, lh + rh + root->val);
calc   res = max( res, ans);
       return temp;
}
```

daaris ameen.

Note:
This question is modification of
Max Path Sum (node to node)

Check if 2 trees are Identical

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(!p) return !q;
        if(!q) return !p;
        return (p->val == q->val) && isSameTree(p->left, q->left) &&
isSameTree(p->right, q->right);
    }
};
```

Zig-Zag traversal

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (root == NULL) {
            return result;
        }

        queue<TreeNode*> nodesQueue;
        nodesQueue.push(root);
        bool leftToRight = true;

        while ( !nodesQueue.empty()) {
            int size = nodesQueue.size();
            vector<int> row(size);
            for (int i = 0; i < size; i++) {
                TreeNode* node = nodesQueue.front();
                nodesQueue.pop();

                // find position to fill node's value
                int index = (leftToRight) ? i : (size - 1 - i);

                row[index] = node->val;
                if (node->left) {
                    nodesQueue.push(node->left);
                }
                if (node->right) {
                    nodesQueue.push(node->right);
                }
            }
            // after this level
            leftToRight = !leftToRight;
            result.push_back(row);
        }
        return result;
    }
};
```

Boundary Traversal

```
class Solution {
    bool isLeaf(Node* root) {
        return !root->left && !root->right;
    }

    void addLeftBoundary(Node* root, vector<int> &res) {
        Node* cur = root->left;
        while (cur) {
            if (!isLeaf(cur)) res.push_back(cur->data);
            if (cur->left) cur = cur->left;
            else cur = cur->right;
        }
    }
    void addRightBoundary(Node* root, vector<int> &res) {
        Node* cur = root->right;
        vector<int> tmp;
        while (cur) {
            if (!isLeaf(cur)) tmp.push_back(cur->data);
            if (cur->right) cur = cur->right;
            else cur = cur->left;
        }
        for (int i = tmp.size()-1; i >= 0; --i) {
            res.push_back(tmp[i]);
        }
    }

    void addLeaves(Node* root, vector<int>& res) {
        if (isLeaf(root)) {
            res.push_back(root->data);
            return;
        }
        if (root->left) addLeaves(root->left, res);
        if (root->right) addLeaves(root->right, res);
    }
public:
    vector <int> printBoundary(Node *root)
    {
        vector<int> res;
        if (!root) return res;

        if (!isLeaf(root)) res.push_back(root->data);

        addLeftBoundary(root, res);

        // add leaf nodes
        addLeaves(root, res);

        addRightBoundary(root, res);
        return res;
    }
};
```

Vertical Order traversal

```
class Solution {
public:
    vector<vector<int>> verticalTraversal(TreeNode* root) {
        map<int, map<int, multiset<int>>> nodes;
        queue<pair<TreeNode*, pair<int, int>>> todo;
        todo.push({root, {0, 0}});
        while (!todo.empty()) {
            auto p = todo.front();
            todo.pop();
            TreeNode* node = p.first;
            int x = p.second.first, y = p.second.second;
            nodes[x][y].insert(node -> val);
            if (node -> left) {
                todo.push({node -> left, {x - 1, y + 1}});
            }
            if (node -> right) {
                todo.push({node -> right, {x + 1, y + 1}});
            }
        }
        vector<vector<int>> ans;
        for (auto p : nodes) {
            vector<int> col;
            for (auto q : p.second) {
                col.insert(col.end(), q.second.begin(),
q.second.end());
            }
            ans.push_back(col);
        }
        return ans;
    }
};
```

Top view of BT

```
class Solution
{
    public:
    //Function to return a list of nodes visible from the top view
    //from left to right in Binary Tree.
    vector<int> topView(Node *root)
    {
        vector<int> ans;
        if(root == NULL) return ans;
        map<int,int> mpp;
        queue<pair<Node*, int>> q;
        q.push({root, 0});
        while(!q.empty())  {
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;
            if(mpp.find(line) == mpp.end()) mpp[line] = node->data;

            if(node->left != NULL) {
                q.push({node->left, line-1});
            }
            if(node->right != NULL) {
                q.push({node->right, line + 1});
            }

        }

        for(auto it : mpp) {
            ans.push_back(it.second);
        }
        return ans;
    }

};
```

Bottom View of BT

```
class Solution {
  public:
    vector <int> bottomView(Node *root) {
        vector<int> ans;
        if(root == NULL) return ans;
        map<int,int> mpp;
        queue<pair<Node*, int>> q;
        q.push({root, 0});
        while(!q.empty())  {
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;
            mpp[line] = node->data;

            if(node->left != NULL) {
                q.push({node->left, line-1});
            }
            if(node->right != NULL) {
                q.push({node->right, line + 1});
            }

        }

        for(auto it : mpp) {
            ans.push_back(it.second);
        }
        return ans;
    }
};
```

Right/Left View of BT

```
class Solution {
public:
    void recursion(TreeNode *root, int level, vector<int> &res)
    {
        if(root==NULL) return ;
        if(res.size()==level) res.push_back(root->val);
        recursion(root->right, level+1, res);
        recursion(root->left, level+1, res);
    }

    vector<int> rightSideView(TreeNode *root) {
        vector<int> res;
        recursion(root, 0, res);
        return res;
    }
};
```

Symmetrical BT

```
class Solution {
public:
    bool f(TreeNode *root1, TreeNode* root2) {
        if(!root1) return !root2;
        if(!root2) return !root1;
        return (root1->val == root2->val) && f(root1->left, root2-
>right) && f(root1->right, root2->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return f(root->left, root->right);
    }
};
```

Root to Node Path in BT

```
bool getPath(TreeNode *root, vector<int> &arr, int x) {
    // if root is NULL
    // there is no path
    if (!root)
        return false;

    // push the node's value in 'arr'
    arr.push_back(root->val);

    // if it is the required node
    // return true
    if (root->val == x)
        return true;

    // else check whether the required node lies
    // in the left subtree or right subtree of
    // the current node
    if (getPath(root->left, arr, x) || getPath(root->right, arr, x))
        return true;

    // required node does not lie either in the
    // left or right subtree of the current node
    // Thus, remove current node's value from
    // 'arr'and then return false
    arr.pop_back();
    return false;
}
vector<int> Solution::solve(TreeNode* A, int B) {
    vector<int> arr;
    if(A == NULL) {
        return arr;
    }
    getPath(A, arr, B);
    return arr;
}
```

Lowest Common Ancestor in BT

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
        //base case
        if (root == NULL || root == p || root == q) {
            return root;
        }
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        //result
        if(left == NULL) {
            return right;
        }
        else if(right == NULL) {
            return left;
        }
        else { //both left and right are not null, we found our result
            return root;
        }
    }
};
```

Maximum width of BT

```
class Solution {
public:
    int widthOfBinaryTree(TreeNode* root) {
        if(!root)
            return 0;
        int ans = 0;
        queue<pair<TreeNode*, int>> q;
        q.push({root,0});
        while(!q.empty()){
            int size = q.size();
            int mmin = q.front().second;     //to make the id starting
from zero
            int first,last;
            for(int i=0; i<size; i++){
                int cur_id = q.front().second-mmin;
                TreeNode* node = q.front().first;
                q.pop();
                if(i==0) first = cur_id;
                if(i==size-1) last = cur_id;
                if(node->left)
                    q.push({node->left, cur_id*2+1});
                if(node->right)
                    q.push({node->right, cur_id*2+2});
            }
            ans = max(ans, last-first+1);
        }
        return ans;
    }
};
```

Children Sum Property in BT - O(N)

```
void reorder(TreeNode* root) {
    if(root == NULL) return;
    int child = 0;
    if(root->left) {
        child += root->left->data;
    }
    if(root->right) {
        child += root->right->data;
    }

    if(child >= root->data) root->data = child;
    else {
        if(root->left) root->left->data = root->data;
        else if(root->right) root->right->data = root->data;
    }

    reorder(root->left);
    reorder(root->right);

    int tot = 0;
    if(root->left) tot += root->left->data;
    if(root->right) tot+= root->right->data;
    if(root->left or root->right) root->data = tot;
}
void changeTree(TreeNode* root) {
    reorder(root);
}
```

Nodes at distance K in BT

```
class Solution {
    void markParents(TreeNode* root, unordered_map<TreeNode*,
TreeNode*> &parent_track, TreeNode* target) {
        queue<TreeNode*> queue;
        queue.push(root);
        while(!queue.empty()) {
            TreeNode* current = queue.front();
            queue.pop();
            if(current->left) {
                parent_track[current->left] = current;
                queue.push(current->left);
            }
            if(current->right) {
                parent_track[current->right] = current;
                queue.push(current->right);
            }
        }
    }
public:
    vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
        unordered_map<TreeNode*, TreeNode*> parent_track; // node ->
parent
        markParents(root, parent_track, target);

        unordered_map<TreeNode*, bool> visited;
        queue<TreeNode*> queue;
        queue.push(target);
        visited[target] = true;
        int curr_level = 0;
        while(!queue.empty()) { /*Second BFS to go upto K level from
target node and using our hashtable info*/
            int size = queue.size();
            if(curr_level++ == k) break;
            for(int i=0; i<size; i++) {
                TreeNode* current = queue.front(); queue.pop();
                if(current->left && !visited[current->left]) {
                    queue.push(current->left);
                    visited[current->left] = true;
                }
                if(current->right && !visited[current->right]) {
                    queue.push(current->right);
                    visited[current->right] = true;
                }
                if(parent_track[current] && !
visited[parent_track[current]]) {
                    queue.push(parent_track[current]);
                    visited[parent_track[current]] = true;
                }
            }
        }
        vector<int> result;
        while(!queue.empty()) {
            TreeNode* current = queue.front(); queue.pop();
            result.push_back(current->val);
        }
        return result;
    }
};
```

Time taken to burn entire tree in BT

```
#include<bits/stdc++.h>

int findMaxDistance(map<TreeNode*> &mpp, TreeNode* target) {
    queue<TreeNode*> q;
    q.push(target);
    map<TreeNode*,int> vis;
    vis[target] = 1;
    int maxi = 0;
    while(!q.empty()) {
        int sz = q.size();
        int fl = 0;
        for(int i = 0;i<sz;i++) {
            auto node = q.front();
            q.pop();
            if(node->left && !vis[node->left]) {
                fl = 1;
                vis[node->left] = 1;
                q.push(node->left);
            }
            if(node->right && !vis[node->right]) {
                fl = 1;
                vis[node->right] = 1;
                q.push(node->right);
            }

            if(mpp[node] && !vis[mpp[node]]) {
                fl = 1;
                vis[mpp[node]] = 1;
                q.push(mpp[node]);
            }
        }
        if(fl) maxi++;
    }
    return maxi;
}
TreeNode* bfsToMapParents(TreeNode* root, map<TreeNode*, TreeNode*>
&mpp, int start) {
    queue<TreeNode*> q;
    q.push(root);
    TreeNode* res;
    while(!q.empty()) {
        TreeNode* node = q.front();
        if(node->data == start) res = node;
        q.pop();
        if(node->left) {
            mpp[node->left] = node;
            q.push(node->left);
        }
        if(node->right) {
            mpp[node->right] = node;
            q.push(node->right);
        }
    }
    return res;
}
int timeToBurnTree(TreeNode* root, int start)
{
    map<TreeNode*, TreeNode*> mpp;
    TreeNode* target = bfsToMapParents(root, mpp, start);
    int maxi = findMaxDistance(mpp, target);
    return maxi;
}
```

Count nodes in Complete BT

```
class Solution {
public:
    int countNodes(TreeNode* root) {
        if(root == NULL) return 0;

        int lh = findHeightLeft(root);
        int rh = findHeightRight(root);

        if(lh == rh) return (1<<lh) - 1;

        return 1 + countNodes(root->left) + countNodes(root->right);
    }
    int findHeightLeft(TreeNode* node) {
        int hght = 0;
        while(node) {
            hght++;
            node = node->left;
        }
        return hght;
    }
    int findHeightRight(TreeNode* node) {
        int hght = 0;
        while(node) {
            hght++;
            node = node->right;
        }
        return hght;
    }
};
```

Construct BT from inorder & preorder

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        map<int, int> inMap;

        for(int i = 0; i < inorder.size(); i++) {
            inMap[inorder[i]] = i;
        }

        TreeNode* root = buildTree(preorder, 0, preorder.size() - 1,
inorder, 0, inorder.size() - 1, inMap);
        return root;
    }
   TreeNode* buildTree(vector<int>& preorder, int preStart, int
preEnd, vector<int>& inorder, int inStart, int inEnd, map<int, int>
&inMap) {
        if(preStart > preEnd || inStart > inEnd) return NULL;

        TreeNode* root = new TreeNode(preorder[preStart]);
        int inRoot = inMap[root->val];
        int numsLeft = inRoot - inStart;

        root->left = buildTree(preorder, preStart + 1, preStart +
numsLeft, inorder, inStart, inRoot - 1, inMap);
        root->right = buildTree(preorder, preStart + numsLeft + 1,
preEnd, inorder, inRoot + 1, inEnd, inMap);

        return root;
    }
};
```

Construct BT from inorder & postorder

```
TreeNode* buildtree(vector<int>& inorder, vector<int>& postorder)
{
    if(inorder.size()!=postorder.size())
        return NULL;
    map<int,int>mp;
    for(int i=0;i<inorder.size();i++)
    {
        mp[inorder[i]]=i;
    }
    TreeNode* root = buildingindetail(inorder,0,inorder.size()-
1,postorder,0,postorder.size()-1);
    return root;
}

TreeNode* buildingindetail(vector<int>& inorder, int instart, int
inend, vector<int>& postorder, int poststart, int postend,
map<int,int>mp)
{
    if(poststart>postend || instart>inend)
        return NULL;

    TreeNode* root = new TreeNode(postorder[postend]);
    int inroot = mp[postorder[postend]];
    int numsleft = inroot-instart;

    root->left = buildingindetail(inorder, instart, inroot-1,
postorder, poststart, poststart+numsleft-1, mp);
    root->right = buildingindetail(inorder, inroot+1, inend,
postorder, poststart+numsleft, postend-1, mp);

    return root;
}
```

Serialize and Deserialize BT

```
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if(!root) return "";

        string s ="";
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()) {
           TreeNode* curNode = q.front();
           q.pop();
           if(curNode==NULL) s.append("#,");
```

```
            else s.append(to_string(curNode->val)+',');
            if(curNode != NULL){
                q.push(curNode->left);
                q.push(curNode->right);
            }
        }
        return s;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if(data.size() == 0) return NULL;
        stringstream s(data);
        string str;
        getline(s, str, ',');
        TreeNode *root = new TreeNode(stoi(str));
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()) {

            TreeNode *node = q.front();
            q.pop();

            getline(s, str, ',');
            if(str == "#") {
                node->left = NULL;
            }
            else {
                TreeNode* leftNode = new TreeNode(stoi(str));
                node->left = leftNode;
                q.push(leftNode);
            }

            getline(s, str, ',');
            if(str == "#") {
                node->right = NULL;
            }
            else {
                TreeNode* rightNode = new TreeNode(stoi(str));
                node->right = rightNode;
                q.push(rightNode);
            }
        }
        return root;
    }
};
```

Morris’s traversal Inorder

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> inorder;

        TreeNode* cur = root;
        while(cur != NULL) {
            if(cur->left == NULL) {
                inorder.push_back(cur->val);
                cur = cur->right;
            }
            else {
                TreeNode* prev = cur->left;
                while(prev->right != NULL && prev->right != cur) {
                    prev = prev->right;
                }

                if(prev->right == NULL) {
                    prev->right = cur;
                    cur = cur->left;
                }
                else {
                    prev->right = NULL;
                    inorder.push_back(cur->val);
                    cur = cur->right;
                }
            }
        }
        return inorder;
    }
};
```

Morris's traversal Preorder

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> preorder;

        TreeNode* cur = root;
        while(cur != NULL) {
            if(cur->left == NULL) {
                preorder.push_back(cur->val);
                cur = cur->right;
            }
            else {
                TreeNode* prev = cur->left;
                while(prev->right != NULL && prev->right != cur) {
                    prev = prev->right;
                }

                if(prev->right == NULL) {
                    prev->right = cur;
                    preorder.push_back(cur->val);
                    cur = cur->left;
                }
                else {
                    prev->right = NULL;
                    cur = cur->right;
                }
            }
        }
        return preorder;
    }
};
```

Flatten a BT to Linked List

```
// TC - O(N)
// SC - O(N)
class Solution {
    TreeNode* prev = NULL;
public:
    void flatten(TreeNode* root) {
        if(root == NULL) return;

        flatten(root->right);
        flatten(root->left);

        root->right = prev;
        root->left = NULL;
        prev = root;
    }
};

// TC - O(N)
// SC - O(N)
// Iterative
class Solution {
public:
    void flatten(TreeNode* root) {
        if(root == NULL) return;
        stack<TreeNode*> st;
        st.push(root);
        while(!st.empty()) {
            TreeNode* cur = st.top();
            st.pop();

            if(cur->right != NULL) {
                st.push(cur->right);
            }
            if(cur->left != NULL) {
                st.push(cur->left);
            }
            if(!st.empty()) {
                cur->right = st.top();
            }
            cur->left = NULL;
        }

    }
};

// TC - O(N)
// SC - O(1)
class Solution {
public:
    void flatten(TreeNode* root) {
        TreeNode* cur = root;
           while (cur)
           {
                if(cur->left)
                {
                     TreeNode* pre = cur->left;
                     while(pre->right)
                     {
                          pre = pre->right;
                     }
                     pre->right = cur->right;
                     cur->right = cur->left;
                     cur->left = NULL;
                }
                cur = cur->right;
           }
    }
};
```

BST

Search in BST

```
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
         while(root != NULL && root->val != val){
            root = val<root->val? root->left:root->right;
        }
        return root;
    }
};
```

Ceil in BST

```
 int findCeil(TreeNode *root, int key){

      int ceil = -1;
    while (root) {

        if (root->data == key) {
            ceil = root->data;
            return ceil;
        }

        if (key > root->data) {
            root = root->right;
        }
        else {
            ceil = root->data;
            root = root->left;
        }
    }
    return ceil;
}
```

Floor in BST

```
int floorInBST(TreeNode * root, int key)
{
    int floor = -1;
    while (root) {

        if (root->val == key) {
            floor = root->val;
            return floor;
        }

        if (key > root->val) {
            floor = root->val;
            root = root->right;
        }
        else {
            root = root->left;
        }
    }
    return floor;
}
```

Insert a node in BST

```
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(root == NULL) return new TreeNode(val);
        TreeNode *cur = root;
        while(true) {
            if(cur->val <= val) {
                if(cur->right != NULL) cur = cur->right;
                else {
                    cur->right = new TreeNode(val);
                    break;
                }
            } else {
                if(cur->left != NULL) cur = cur->left;
                else {
                    cur->left = new TreeNode(val);
                    break;
                }
            }
        }
        return root;
    }
};
```

Delete a node in BST

```
TreeNode* find_predecessor(TreeNode *root)
{
    if (root->right == NULL)
        return root;
    return find_predecessor(root->right);
}

TreeNode* helper(TreeNode *root)
{
    if (root->left == NULL)
        return root->right;
    else if (root->right == NULL)
        return root->left;
    TreeNode *inordersuccessor = root->right;
    TreeNode *inorderpredecessor = find_predecessor(root->left);
    inorderpredecessor->right = inordersuccessor;
    return root->left;
}

TreeNode* delete_node_bst(TreeNode *root, int key)
{
    if (root == NULL) return NULL;

    if (root->val == key) return helper(root);

    TreeNode *dummy = root;
    while (root != NULL)
    {
        if (key > root->val)
```

```
            {
                if (root->right != NULL and root->right->val == key)
                {
                    root->right = helper(root->right);
                    break;
                }
                else
                    root = root->right;
            }
            else
            {
                if (root->left != NULL and root->left->val == key)
                {
                    root->left = helper(root->left);
                    break;
                }
                else
                    root = root->left;
            }
        }
        return dummy;
}
```

Kth smallest & Kth largest in BST

```
TreeNode* kthlargest(TreeNode* root,int& k)
{
        if(root==NULL)
        return NULL;

        TreeNode* right=kthlargest(root->right,k);
        if(right!=NULL)
        return right;
        k--;

        if(k==0)
        return root;

        return kthlargest(root->left,k);
}

TreeNode* kthsmallest(TreeNode* root,int &k)
{
        if(root==NULL)
        return NULL;

        TreeNode* left=kthsmallest(root->left,k);
        if(left!=NULL)
        return left;
        k--;
        if(k==0)
```

```
      return root;

      return kthsmallest(root->right,k);
}
```

Check if BT is BST

```
    bool isValidBST(TreeNode* root) {
        return isValidBST(root, INT_MIN, INT_MAX);
    }

    bool isValidBST(TreeNode* root, long minVal, long maxVal) {
        if (root == null) return true;
        if (root->val >= maxVal || root->val <= minVal) return false;
        return isValidBST(root->left, minVal, root->val) &&
isValidBST(root->right, root->val, maxVal);
    }
```

LCA in BST

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
TreeNode* q) {
        if(!root) return NULL;
        int curr = root->val;
        if(curr < p->val && curr < q->val) {
            return lowestCommonAncestor(root->right, p, q);
        }
        if(curr > p->val && curr > q->val) {
            return lowestCommonAncestor(root->left, p, q);
        }
        return root;
    }
};
```

BST from preorder

```
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& A) {
        int i = 0;
        return build(A, i, INT_MAX);
    }

    TreeNode* build(vector<int>& A, int& i, int bound) {
        if (i == A.size() || A[i] > bound) return NULL;
        TreeNode* root = new TreeNode(A[i++]);
        root->left = build(A, i, root->val);
        root->right = build(A, i, bound);
        return root;
    }
};
```

BST from postorder

```
Node* build(vector<int>& post, int& i, int bound) {
        if (i < 0 || post[i] < bound) return NULL;
        Node* root = new Node(post[i--]);
        root->right = build(post, i, root->data);
        root->left = build(post, i, bound);
        return root;
}
Node *constructTree (vector<int>& post, int size)
{
    int i = size - 1;
    return build(post, i, INT_MIN);
}
```

Inorder successor/predecessor in BST

```
// successor = ceil
// predecessor = floor

class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
        TreeNode* successor = NULL;

        while (root != NULL) {

            if (p->val >= root->val) {
                root = root->right;
            } else {
                successor = root;
                root = root->left;
            }
        }

        return successor;
    }
};
```

BST Iterator - O(H)

```
class BSTIterator {
    stack<TreeNode *> myStack;
public:
    BSTIterator(TreeNode *root) {
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !myStack.empty();
    }
    /** @return the next smallest number */
    int next() {
        TreeNode *tmpNode = myStack.top();
        myStack.pop();
        pushAll(tmpNode->right);
        return tmpNode->val;
    }

private:
    void pushAll(TreeNode *node) {
        for (; node != NULL; myStack.push(node), node = node->left);
    }
};
```

Check if there exists a pair of sum k in BST

```
class BSTIterator {
    stack<TreeNode *> myStack;
    bool reverse = true;
public:
    BSTIterator(TreeNode *root, bool isReverse) {
        reverse = isReverse;
        pushAll(root);
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !myStack.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *tmpNode = myStack.top();
        myStack.pop();
        if(!reverse) pushAll(tmpNode->right);
        else pushAll(tmpNode->left);
        return tmpNode->val;
    }

private:
    void pushAll(TreeNode *node) {
        for(;node != NULL; ) {
             myStack.push(node);
             if(reverse == true) {
                 node = node->right;
             } else {
                 node = node->left;
             }
        }
    }
};
class Solution {
public:
    bool findTarget(TreeNode* root, int k) {
        if(!root) return false;
        BSTIterator l(root, false);
        BSTIterator r(root, true);

        int i = l.next();
        int j = r.next();
        while(i<j) {
            if(i + j == k) return true;
            else if(i + j < k) i = l.next();
            else j = r.next();
        }
        return false;
    }
};
```

Recover BST - where exactly 2 nodes swaped

```
class Solution {
private:
    TreeNode* first;
    TreeNode* prev;
    TreeNode* middle;
    TreeNode* last;
private:
    void inorder(TreeNode* root) {
        if(root == NULL) return;

        inorder(root->left);

        if (prev != NULL && (root->val < prev->val))
        {

            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( first == NULL )
            {
                first = prev;
                middle = root;
            }

            // If this is second violation, mark this node as last
            else
                last = root;
        }

        // Mark this node as previous
        prev = root;
        inorder(root->right);
    }
public:
    void recoverTree(TreeNode* root) {
        first = middle = last = NULL;
        prev = new TreeNode(INT_MIN);
        inorder(root);
        if(first && last) swap(first->val, last->val);
        else if(first && middle) swap(first->val, middle->val);
    }
};
```

Largest BST from BT

```
class NodeValue {
public:
    int maxNode, minNode, maxSize;

    NodeValue(int minNode, int maxNode, int maxSize) {
        this->maxNode = maxNode;
        this->minNode = minNode;
        this->maxSize = maxSize;
    }
};

class Solution {
private:
    NodeValue largestBSTSubtreeHelper(TreeNode* root) {
        // An empty tree is a BST of size 0.
        if (!root) {
            return NodeValue(INT_MAX, INT_MIN, 0);
        }

        // Get values from left and right subtree of current tree.
        auto left = largestBSTSubtreeHelper(root->left);
        auto right = largestBSTSubtreeHelper(root->right);

        // Current node is greater than max in left AND smaller than
min in right, it is a BST.
        if (left.maxNode < root->val && root->val < right.minNode) {
            // It is a BST.
            return NodeValue(min(root->val, left.minNode), max(root-
>val, right.maxNode), left.maxSize + right.maxSize + 1);
        }

        // Otherwise, return [-inf, inf] so that parent can't be valid
BST
        return NodeValue(INT_MIN, INT_MAX, max(left.maxSize,
right.maxSize));
    }
    public:
```

```
    int largestBSTSubtree(TreeNode* root) {
        return largestBSTSubtreeHelper(root).maxSize;
    }
};
```

LINKED LIST

Reverse a Linked List

Given the head of a singly linked list, write a program to reverse the linked list, and return the head pointer to the reversed list.

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */

class Solution {
public:

    ListNode* reverseList(ListNode* head) {

       //step 1
        ListNode* prev_p = NULL;
        ListNode* current_p = head;
        ListNode* next_p;

       //step 2
        while(current_p) {

            next_p = current_p->next;
            current_p->next = prev_p;

            prev_p = current_p;
            current_p = next_p;
        }

        head = prev_p; //step 3
        return head;
    }
};
```

```
ListNode* reverseList(ListNode* &head) {

        if (head == NULL||head->next==NULL)
            return head;

        ListNode* nnode = reverseList(head->next);
```

```
            head->next->next = head;
            head->next = NULL;
            return nnode;
        }
```

Find middle element in a Linked List

Given the head of a singly linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.

```
class Solution {
public:
        ListNode* middleNode(ListNode* head) {
        int n = 0;
        ListNode* temp = head;
        while(temp) {
                n++;
                        temp = temp->next;
        }

        temp = head;

        for(int i = 0; i < n / 2; i++) {
                        temp = temp->next;
        }

        return temp;
        }
};
```

Merge two sorted Linked Lists

Given two singly linked lists that are sorted in increasing order of node values, merge two sorted linked lists and return them as a sorted list. The list should be made by splicing together the nodes of the first two lists.

```
class Solution {

public:
      ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {

            // when list1 is empty then
            // our output will be same as list2
            if(l1 == NULL) return l2;

            // when list2 is empty then our output
            // will be same as list1
            if(l2 == NULL) return l1;

            // pointing l1 and l2 to smallest and greatest one
```

```
        if(l1->val > l2->val) std::swap(l1,l2);

        // act as head of resultant merged list
        ListNode* res = l1;

        while(l1 != NULL && l2 != NULL) {

            ListNode* temp = NULL;

            while(l1 != NULL && l1->val <= l2->val) {

                temp = l1;//storing last sorted node
                l1 = l1->next;
            }

            // link previous sorted node with
            // next larger node in list2
            temp->next = l2;
            std::swap(l1,l2);
        }

        return res;
    }
};
```

Remove N-th node from the end of a Linked List

Given a linked list, and a number N. Find the Nth node from the end of this linked list and delete it. Return the head of the new modified linked list.

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode * start = new ListNode();
        start -> next = head;
        ListNode* fast = start;
        ListNode* slow = start;

        for(int i = 1; i <= n; ++i)
            fast = fast->next;

        while(fast->next != NULL)
        {
            fast = fast->next;
            slow = slow->next;
        }

        slow->next = slow->next->next;

        return start->next;
    }
};
TC- O(N), SC- O(1)
```

Add two numbers represented as Linked Lists

Given the heads of two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode();
        ListNode *temp = dummy;
        int carry = 0;
        while( (l1 != NULL || l2 != NULL) || carry) {
            int sum = 0;
            if(l1 != NULL) {
                sum += l1->val;
                l1 = l1 -> next;
            }

            if(l2 != NULL) {
                sum += l2 -> val;
                l2 = l2 -> next;
            }

            sum += carry;
            carry = sum / 10;
            ListNode *node = new ListNode(sum % 10);
            temp -> next = node;
            temp = temp -> next;
        }
        return dummy -> next;
    }
};

TC and SC is O(max(M,N))
```

Delete given node in a Linked List : O(1) approach

Write a function to delete a node in a singly-linked list. You will not be given access to the head of the list instead, you will be given access to the node to be deleted directly. It is guaranteed that the node to be deleted is not a tail node in the list.

```
#include<iostream>
using namespace std;

class node {
    public:
        int num;
        node* next;
        node(int a) {
            num = a;
            next = NULL;
        }
};
//function to insert node at the end of the list
void insertNode(node* &head,int val) {
    node* newNode = new node(val);
    if(head == NULL) {
        head = newNode;
        return;
    }
    node* temp = head;
    while(temp->next != NULL) temp = temp->next;
    temp->next = newNode;
}
//function to get reference of the node to delete
node* getNode(node* head,int val) {
    while(head->num != val) head = head->next;

    return head;
}
//delete function as per the question
void deleteNode(node* t) {
    t->num = t->next->num;
    t->next = t->next->next;
    return;
}
//printing the list function
void printList(node* head) {
    while(head->next != NULL) {
        cout<<head->num<<"->";
        head = head->next;
    }
    cout<<head->num<<"\n";
}

int main() {
    node* head = NULL;
    //inserting node
    insertNode(head,1);
    insertNode(head,4);
    insertNode(head,2);
    insertNode(head,3);
    //printing given list
    cout<<"Given Linked List:\n";
    printList(head);
    node* t = getNode(head,2);
    //delete node
    deleteNode(t);
    //list after deletion operation
    cout<<"Linked List after deletion:\n";
    printList(head);
    return 0;
}
```

Find intersection of Two Linked Lists

Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

```
node* intersectionPresent(node* head1,node* head2) {
     unordered_set<node*> st;
    while(head1 != NULL) {
       st.insert(head1);
       head1 = head1->next;
    }
    while(head2 != NULL) {
        if(st.find(head2) != st.end()) return head2;
        head2 = head2->next;
    }
    return NULL;

}
```

Detect a Cycle in a Linked List

Given head, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.

Return true if there is a cycle in the linked list. Otherwise, return false.

```
//utility function to create cycle
void createCycle(node* &head,int a,int b) {
    int cnta = 0,cntb = 0;
    node* p1 = head;
    node* p2 = head;
    while(cnta != a || cntb != b) {
        if(cnta != a) p1 = p1->next, ++cnta;
        if(cntb != b) p2 = p2->next, ++cntb;
    }
    p2->next = p1;
}

//utility function to detect cycle
bool cycleDetect(node* head) {
    unordered_set<node*> hashTable;
    while(head != NULL) {
        if(hashTable.find(head) != hashTable.end()) return true;
        hashTable.insert(head);
        head = head->next;
    }
    return false;
}
```

Reverse Linked List in groups of Size K

Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list. k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

```
int lengthOfLinkedList(node* head) {
    int length = 0;
    while(head != NULL) {
        ++length;
        head = head->next;
    }
    return length;
}
//utility function to reverse k nodes in the list
node* reverseKNodes(node* head,int k) {
    if(head == NULL||head->next == NULL) return head;

    int length = lengthOfLinkedList(head);

    node* dummyHead = new node(0);
    dummyHead->next = head;

    node* pre = dummyHead;
    node* cur;
    node* nex;

    while(length >= k) {
        cur = pre->next;
        nex = cur->next;
        for(int i=1;i<k;i++) {
            cur->next = nex->next;
            nex->next = pre->next;
            pre->next = nex;
            nex = cur->next;
        }
        pre = cur;
        length -= k;
    }
    return dummyHead->next;
}
```

Check if given Linked List is Plaindrome

Given the head of a singly linked list, return true if it is a palindrome.

```
//use extra data structure
//or use reverse link list & middle ele in linked list

_ _ _ _
node* reverse(node* ptr) {
    node* pre=NULL;
    node* nex=NULL;
    while(ptr!=NULL) {
        nex = ptr->next;
        ptr->next = pre;
        pre=ptr;
        ptr=nex;
    }
    return pre;
}

bool isPalindrome(node* head) {
    if(head==NULL||head->next==NULL) return true;

    node* slow = head;
    node* fast = head;

    while(fast->next!=NULL&&fast->next->next!=NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    slow->next = reverse(slow->next);
    slow = slow->next;
    node* dummy = head;

    while(slow!=NULL) {
        if(dummy->num != slow->num) return false;
        dummy = dummy->next;
        slow = slow->next;
    }
    return true;
}
```

Starting point of loop in a Linked List

Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that the tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

```
node* detectCycle(node* head) {
    unordered_set<node*> st;
    while(head != NULL) {
        if(st.find(head) != st.end()) return head;
        st.insert(head);
        head = head->next;
    }
    return NULL;
}
```

```
node* detectCycle(node* head) {
    if(head == NULL||head->next == NULL) return NULL;

    node* fast = head;
    node* slow = head;
    node* entry = head;

    while(fast->next != NULL&&fast->next->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;

        if(slow == fast) {
            while(slow != entry) {
                slow = slow->next;
                entry = entry->next;
            }
            return slow;
        }
    }
    return NULL;
}
```

Flattening a Linked List

Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:

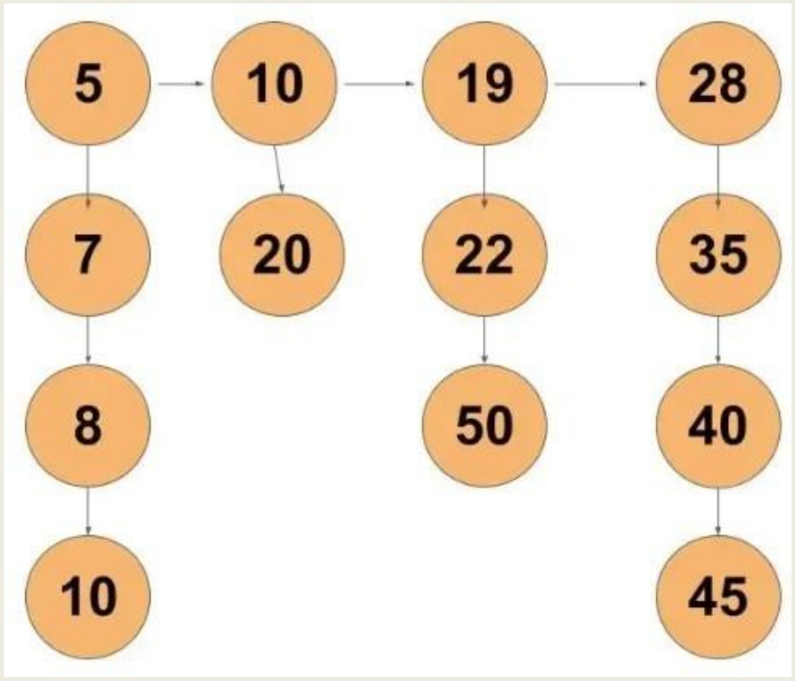
(i) a next pointer to the next node,

(ii) a bottom pointer to a linked list where this node is head.

Each of the sub-linked-list is in sorted order.

Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order.

Note: The flattened list will be printed using the bottom pointer instead of the next pointer.



```
Node* mergeTwoLists(Node* a, Node* b) {

    Node *temp = new Node(0);
    Node *res = temp;

    while(a != NULL && b != NULL) {
        if(a->data < b->data) {
            temp->bottom = a;
            temp = temp->bottom;
            a = a->bottom;
        }
        else {
            temp->bottom = b;
            temp = temp->bottom;
            b = b->bottom;
        }
    }

    if(a) temp->bottom = a;
    else temp->bottom = b;

    return res -> bottom;

}
Node *flatten(Node *root)
{

        if (root == NULL || root->next == NULL)
```

```
            return root;

        // recur for list on right
        root->next = flatten(root->next);

        // now merge
        root = mergeTwoLists(root, root->next);

        // return the root
        // it will be in turn merged with its left
        return root;
}
```

Rotate a Linked List

Given the head of a linked list, rotate the list to the right by k places.

Input: head = [1,2,3,4,5] k = 2 Output: head = [4,5,1,2,3]

```
node* rotateRight(node* head,int k) {
    if(head == NULL||head->next == NULL||k == 0) return head;
    //calculating length
    node* temp = head;
    int length = 1;
    while(temp->next != NULL) {
        ++length;
        temp = temp->next;
    }
    //link last node to first node
    temp->next = head;
    k = k%length; //when k is more than length of list
    int end = length-k; //to get end of the list
    while(end--) temp = temp->next;
    //breaking last node link and pointing to NULL
    head = temp->next;
    temp->next = NULL;

    return head;
}
```

BINARY SEARCH

BS on Reverse sorted array

```
#include<bits/stdc++.h>
using namespace std;

int solve()
{
    int n,target; cin>>n>>target;
    vector<int>nums(n);

    for(int i=0;i<n;i++)
        cin>>nums[i];

    int start=0, end=n-1;
    while(start<=end)
    {
        int mid = (end-start)/2 + start;
        if(nums[i]==mid)
            return mid;
        if(nums[i]>target)
            start=mid+1;
        else
            end=mid-1;
    }
    return -1;
}
```

Order not known search

```
#include<bits/stdc++.h>
using namespace std;

int solve()
{
    int n,target; cin>>n>>target;
    vector<int>nums(n);

    for(int i=0;i<n;i++)
        cin>>nums[i];

    int  start=0,  end=n-1;
    int order=1;
    if(nums[start]<nums[end])
        order=1; // increasing order
    else
        order=0; // decreasing order
    if(order==0)
    {
        while(start<=end)
        {
            int mid = (end-start)/2 + start;
            if(nums[i]==mid)
                return mid;
            if(nums[i]>target)
                start=mid+1;
            else
                end=mid-1;
        }
    }
    else
    {
        while(start<=end)
        {
            int mid = (end-start)/2 + start;
            if(nums[i]==mid)
                return mid;
            if(nums[i]>target)
                end=mid-1;
            else
                start=mid+1;
        }
    }
    return -1;
}
```

1st & last occurence of an element

```
vector<int> searchRange(vector<int>& nums, int target) {
        int first = -1, last = -1;
        int start=0,end=nums.size()-1;
        while(start<=end)
        {
            int mid=(end-start)/2+start;
            if(nums[mid]>=target)
            {
                if(nums[mid]==target)
                    first=mid;
                end=mid-1;
            }
            else
                start=mid+1;
        }
        start=0,end=nums.size()-1;
        while(start<=end)
        {
            int mid=(end-start)/2+start;
            if(nums[mid]<=target)
            {
                if(nums[mid]==target)
                    last=mid;
                start=mid+1;
            }
            else
                end=mid-1;
        }

        vector<int>v(2); v[0]=first, v[1]=last;
        return v;
    }
```

Count of an element in sorted array

```
vector<int> searchRange(vector<int>& nums, int target) {
        int first = -1, last = -1;
        int start=0,end=nums.size()-1;
        while(start<=end)
        {
            int mid=(end-start)/2+start;
            if(nums[mid]>=target)
            {
                if(nums[mid]==target)
                    first=mid;
                end=mid-1;
            }
            else
                start=mid+1;
        }
        start=0,end=nums.size()-1;
        while(start<=end)
        {
            int mid=(end-start)/2+start;
            if(nums[mid]<=target)
            {
                if(nums[mid]==target)
                    last=mid;
                start=mid+1;
            }
            else
                end=mid-1;
        }

        if(first==-1 || last==-1)
        return 0;
        else
        return last-first+1;
    }
```

Find an Element in Rotated Sorted Array

```
int search(vector<int> &nums, int target)
        {
                // find minimum element
            int pos_min = 0;
            int n = nums.size(), start = 0, end = n - 1;
            while (start <= end)
            {
                int mid = (end - start) / 2 + start;
                int next = (mid + 1) % n;
                int prev = (n + mid - 1) % n;

                if (nums[mid] < nums[prev] and nums[mid] < nums[next])
                {
                    pos_min = mid;
                    break;
                }
                if (nums[mid] > nums[end])
                    start = mid + 1;
                else
                    end = mid - 1;
            }

            if (target >= nums[pos_min] and target <= nums[n - 1])
            {
                start = pos_min, end = n - 1;
                while (start <= end)
                {
                    int mid = (end - start) / 2 + start;
                    if (nums[mid] == target)
                        return mid;
                    if (nums[mid] > target)
                        end = mid - 1;
                    else
                        start = mid + 1;
                }
                return -1;
            }
            else
            {
                start = 0, end = pos_min - 1;
                while (start <= end)
                {
                    int mid = (end - start) / 2 + start;
                    if (nums[mid] == target)
                        return mid;
                    if (nums[mid] > target)
                        end = mid - 1;
                    else
                        start = mid + 1;
                }
                return -1;
            }
            return 0;
        }
```

Number of times a Sorted array is Rotated

```
int findKRotation(int arr[], int n) {
        // code here
        int start=0,end=n-1;
        while(start<=end)
        {
            int mid=(end-start)/2+start;
            int prev=(n+mid-1)%n, next=(mid+1)%n;

            if(arr[mid]<arr[prev] and arr[mid]<arr[next])
                return mid;
            if(arr[mid]>arr[end])
                start=mid+1;
            else
                end=mid-1;
        }
        return 0;
    }
```

Searching in a Nearly sorted array

```
int solve()
{
    int n, target;
    cin >> n >> target;
    vector<int> nums(n);
    for (int i = 0; i < n; i++)
        cin >> v[i];

    int start = 0, end = n - 1;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (nums[mid] == target)
            return mid;
        if (mid - 1 >= start and nums[mid - 1] == target)
            return mid - 1;
        if (mid + 1 <= end and nums[mid + 1] == target)
            return mid + 1;
        if (nums[mid] > target)
            end = mid - 1;
        else
            start = mid + 1;
    }
    return -1;
}
```

Floor of an element in sorted array

```
int findFloor(long long int arr[], int N, long long int target)
{

    int start = 0, end = N - 1;
    long long int floor_ele = -1;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (arr[mid] <= target)
        {
            floor_ele = mid;
            start = mid + 1;
        }
        else
            end = mid - 1;
    }
    return floor_ele;
}
```

Ceil of an element in sorted array

```
int findCeil(long long int arr[], int N, long long int target)
{

    int start = 0, end = N - 1;
    long long int ceil_ele = -1;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (arr[mid] >= target)
        {
            ceil_ele = mid;
            end = mid - 1;
        }
        else
            start = mid + 1;
    }
    return ceil_ele;
}
```

Next Alphabet element

```
char nextGreatestLetter(vector<char> &letters, char target)
        {
            int start = 0, end = letters.size() - 1;
            char z = letters[0];
            while (start <= end)
            {
                int mid = (end - start) / 2 + start;
                if (letters[mid] > target)
                {
                    z = letters[mid];
                    end = mid - 1;
                }
                else
                    start = mid + 1;
            }
            return z;
        }
```

Find position of an element in an Infinite sorted array

```
// assume nums length is infinite

int find_elementpos_infinite_array(vector<int> nums, int target)
{
    int start = 0, end = 0;
    while (nums[start] <= target)
    {
        start *= 2;
    }
    end = start;
    start /= 2;

    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (nums[mid] == target)
            return mid;
        if (nums[mid] > target)
            end = mid - 1;
        else
            start = mid + 1;
    }
    return -1;
}
```

Index of first 1 in binary sorted infinite array

```
// assume nums length is infinite containing 0's and 1's

int find_elementpos_infinite_array(vector<int> nums, int target)
{
    int start = 0, end = 0;
    while (nums[start] == 1)
    {
        start *= 2;
    }
    end = start;
    start /= 2;

    int first_one = start;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (nums[mid] == 1)
        {
            first_one = mid;
            end = mid - 1;
        }
        else
            start = mid + 1;
    }
    return first_one;
}
```

Minimum difference element in a Sorted array

```
// minimum difference with a given target in sorted array

int min_diff_with_target(vector<int> nums, int target)
{
    int n = nums.size(), start = 0, end = n - 1;
    int floor_ele = -1, ceil_ele = -1;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (nums[mid] >= target)
        {
            floor_ele = nums[mid];
            end = mid - 1;
        }
        else
            start = mid + 1;
    }

    start = 0, end = n - 1;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (nums[mid] <= target)
        {
            ceil_ele = nums[mid];
            start = mid + 1;
        }
        else
            end = mid - 1;
    }
    return min(ceil_ele - target, target - floor_ele);
}
```

BS on Answer concept

```
Binary Search on answer concept - BSA
    1. Applicable on unsorted array.
    2. criteria is needed to check whether mid is the answer.
    3. criteria is needed to move the array left of right.
```

Peak Element

```
// Binary Search on answer concept - BSA
//     1. Applicable on unsorted array.
//     2. criteria is needed to check whether mid is the answer.
//     3. criteria is needed to move the array left of right.

class Solution
{
    public:
        int findPeakElement(vector<int> &nums)
        {
            int n = nums.size(), start = 0, end = n - 1;
            if (n == 1)
                return 0;
            while (start <= end)
            {
                int mid = (end - start) / 2 + start;
                if (mid > 0 and mid < n - 1)
                {
                    if (nums[mid] > nums[mid - 1] and nums[mid] >
nums[mid + 1])
                        return mid;
                    else if (nums[mid + 1] > nums[mid - 1])
                        start = mid + 1;
                    else
                        end = mid - 1;
                }
                if (mid == 0)
                {
                    if (nums[mid] > nums[mid + 1])
                        return mid;
                    else
                        return mid + 1;
                }
                if (mid == n - 1)
                {
                    if (nums[mid] > nums[mid - 1])
                        return mid;
                    else
                        return mid - 1;
                }
            }
            return -1;
        }
};
```

Find maximum element in Bitonic array

```
// Binary Search on answer concept - BSA
//     1. Applicable on unsorted array.
//     2. criteria is needed to check whether mid is the answer.
//     3. criteria is needed to move the array left of right.


// bitonic is array where elements increase monotonically and decrease
monotonically

int findMaximum_bitonic_array(int arr[], int n)
{
    int start = 0, end = n - 1;
    if (n == 1)
        return arr[0];
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (mid > 0 and mid < n - 1)
        {
            if (arr[mid] > arr[mid - 1] and arr[mid] > arr[mid + 1])
                return arr[mid];
            else if (arr[mid + 1] > arr[mid - 1])
                start = mid + 1;
            else
                end = mid - 1;
        }
        if (mid == 0)
        {
            if (arr[mid] > arr[mid + 1])
                return arr[mid];
            else
                return arr[mid + 1];
        }
        if (mid == n - 1)
        {
            if (arr[mid] > arr[mid - 1])
                return arr[mid];
            else
                return arr[mid - 1];
        }
    }
    return -1;
}
```

Search an element in Bitonic Array

```
// Binary Search on answer concept - BSA
//     1. Applicable on unsorted array.
//     2. criteria is needed to check whether mid is the answer.
//     3. criteria is needed to move the array left of right.

// bitonic is array where elements increase monotonically and decrease
monotonically

int findMaximum_bitonic_array(vector<int> arr, int n)
{
    int start = 0, end = n - 1;
    if (n == 1)
        return 0;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (mid > 0 and mid < n - 1)
        {
            if (arr[mid] > arr[mid - 1] and arr[mid] > arr[mid + 1])
                return mid;
            else if (arr[mid + 1] > arr[mid - 1])
                start = mid + 1;
            else
                end = mid - 1;
        }
        if (mid == 0)
        {
            if (arr[mid] > arr[mid + 1])
                return mid;
            else
                return mid + 1;
        }
        if (mid == n - 1)
        {
            if (arr[mid] > arr[mid - 1])
                return mid;
            else
                return mid - 1;
        }
    }
    return -1;
}
int search_element_bitonic_array(vector<int> &A, int B)
{
    int maximum_bitonic_array = findMaximum_bitonic_array(A,
A.size());
      // cout<<maximum_bitonic_array;
    int start = 0, end = maximum_bitonic_array - 1;
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (A[mid] == B)
            return mid;
        if (A[mid] > B)
            end = mid - 1;
        else
            start = mid + 1;
    }

    start = maximum_bitonic_array, end = A.size();
    while (start <= end)
    {
        int mid = (end - start) / 2 + start;
        if (A[mid] == B)
            return mid;
        if (A[mid] > B)
            start = mid + 1;
        else
            end = mid - 1;
    }
    return -1;
}
```

Search in Row-wise & Column-wise sorted array

```
bool searchMatrix(vector<vector < int>> &matrix, int target)
        {
            int rownum = 0, colnum = 0;
            int start = 0, end = matrix.size() - 1;
            while (start <= end)
            {
                int mid = (end - start) / 2 + start;
                if (matrix[mid][0] <= target)
                {
                    rownum = mid;
                    start = mid + 1;
                }
                else
                    end = mid - 1;
            }
            start = 0, end = matrix[0].size() - 1;
            while (start <= end)
            {
                int mid = (end - start) / 2 + start;
                if (matrix[rownum][mid] == target)
                    return true;
                if (matrix[rownum][mid] < target)
                    start = mid + 1;
                else
                    end = mid - 1;
            }
            return false;
        }
```

Allocate minimum number of pages

```
#include <bits/stdc++.h>
using namespace std;

bool isPossible(int arr[], int n, int m, int curr_min)
{
      int studentsRequired = 1;
      int curr_sum = 0;

      // iterate over all books
      for (int i = 0; i < n; i++) {
            // check if current number of pages are greater
            // than curr_min that means we will get the result
            // after mid no. of pages
            if (arr[i] > curr_min)
                  return false;

            // count how many students are required
            // to distribute curr_min pages
            if (curr_sum + arr[i] > curr_min) {
                  // increment student count
                  studentsRequired++;

                  // update curr_sum
                  curr_sum = arr[i];

                  // if students required becomes greater
                  // than given no. of students,return false
                  if (studentsRequired > m)
                        return false;
            }

            // else update curr_sum
            else
                  curr_sum += arr[i];
      }
      return true;
}

// function to find minimum pages
int findPages(int arr[], int n, int m)
{
      long long sum = 0;

      // return -1 if no. of books is less than
      // no. of students
```

```
	if (n < m)
		return -1;

	// Count total number of pages
	for (int i = 0; i < n; i++)
		sum += arr[i];

	// initialize start as 0 pages and end as
	// total pages
	int start = 0, end = sum;
	int result = INT_MAX;

	// traverse until start <= end
	while (start <= end) {
		// check if it is possible to distribute
		// books by using mid as current minimum
		int mid = (start + end) / 2;
		if (isPossible(arr, n, m, mid)) {
			// update result to current distribution
			// as it's the best we have found till now.
			result = mid;

			// as we are finding minimum and books
			// are sorted so reduce end = mid -1
			// that means
			end = mid - 1;
		}

		else
			// if not possible means pages should be
			// increased so update start = mid + 1
			start = mid + 1;
	}

	// at-last return minimum no. of pages
	return result;
}

// Drivers code
int main()
{
	// Number of pages in books
	int arr[] = { 12, 34, 67, 90 };
	int n = sizeof arr / sizeof arr[0];
	int m = 2; // No. of students

	cout << "Minimum number of pages = "
		<< findPages(arr, n, m) << endl;
	return 0;
}
```

Nth Root of a Number

Given two numbers N and M, find the Nth root of M.

```
#include <bits/stdc++.h>
using namespace std;
double multiply(double number, int n) {
    double ans = 1.0;
    for(int i = 1;i<=n;i++) {
        ans = ans * number;
    }
    return ans;
}

void getNthRoot(int n, int m) {
    double low = 1;
    double high = m;
    double eps = 1e-6;

    while((high - low) > eps) {
        double mid = (low + high) / 2.0;
        if(multiply(mid, n) < m) {
            low = mid;
        }
        else {
            high = mid;
        }
    }

    cout <<n<<"th root of "<<m<<" is "<<low<<endl;

}
int main() {
     int n=3, m=27;
     getNthRoot(n, m);
     return 0;
}

Time Complexity: N x log(M x 10^d)

Space Complexity: O(1)
```

Find median in row wise sorted matrix

We are given a row-wise sorted matrix of size rc*, we need to find the median of the matrix given. It is assumed that* rc is always odd.

```
// C++ program to find median of a matrix
// sorted row wise
```

```cpp
#include<bits/stdc++.h>
using namespace std;

const int MAX = 100;

// function to find median in the matrix
int binaryMedian(int m[][MAX], int r ,int c)
{
	int min = INT_MAX, max = INT_MIN;
	for (int i=0; i<r; i++)
	{
		// Finding the minimum element
		if (m[i][0] < min)
			min = m[i][0];

		// Finding the maximum element
		if (m[i][c-1] > max)
			max = m[i][c-1];
	}

	int desired = (r * c + 1) / 2;
	while (min < max)
	{
		int mid = min + (max - min) / 2;
		int place = 0;

		// Find count of elements smaller than or equal to mid
		for (int i = 0; i < r; ++i)
			place += upper_bound(m[i], m[i]+c, mid) - m[i];
		if (place < desired)
			min = mid + 1;
		else
			max = mid;
	}
	return min;
}

// driver program to check above functions
int main()
{
	int r = 3, c = 3;
	int m[][MAX]= { {1,3,5}, {2,6,9}, {3,6,9} };
	cout << "Median is " << binaryMedian(m, r, c) << endl;
	return 0;
}

Time Complexity: O(32 * r * log(c))
Auxiliary Space: O(1)
```

Search Single Element in a sorted array

Given a sorted array of N integers, where every element except one appears exactly twice and one element appears only once. Search Single Element in a sorted array.

```
#include<bits/stdc++.h>

using namespace std;
class Solution {
    public:
        int findSingleElement(vector < int > & nums)
        {
            int low = 0;
            int high = n - 2;

            while (low <= high) {
                int mid = (low + high) / 2;

                if (mid % 2 == 0) {
                    if (nums[mid] != nums[mid + 1])
                    //Checking whether we are in right half

                        high = mid - 1; //Shrinking the right half
                    else
                        low = mid + 1; //Shrinking the left half
                } else {

                    //Checking whether we are in right half
                    if (nums[mid] == nums[mid + 1])
                        high = mid - 1; //Shrinking the right half
                    else
                        low = mid + 1; //Shrinking the left half
                }
            }

            return nums[low];
        }
};

int main() {
    Solution obj;
    vector < int > v {1,1,2,3,3,4,4,8,8
    };

    int elem = obj.findSingleElement(v);
    cout << "The single occurring element is " +
    " << elem << endl;

}
```

```
Time Complexity: O(log(N))

Space Complexity: O(1)
```

Median of Two Sorted Arrays of different sizes

Given two sorted arrays arr1 and arr2 of size m and n respectively, return the median of the two sorted arrays.

```
#include<bits/stdc++.h>
using namespace std;

float median(int num 1[],int num2[],int m,int n) {
    if(m>n)
        return median(nums2,nums1,n,m);//ensuring that binary search
happens on minimum size array

    int low=0,high=m,medianPos=((m+n)+1)/2;
    while(low<=high) {
        int cut1 = (low+high)>>1;
        int cut2 = medianPos - cut1;

        int l1 = (cut1 == 0)? INT_MIN:nums1[cut1-1];
        int l2 = (cut2 == 0)? INT_MIN:nums2[cut2-1];
        int r1 = (cut1 == m)? INT_MAX:nums1[cut1];
        int r2 = (cut2 == n)? INT_MAX:nums2[cut2];

        if(l1<=r2 && l2<=r1) {
            if((m+n)%2 != 0)
                return max(l1,l2);
            else
                return (max(l1,l2)+min(r1,r2))/2.0;
        }
        else if(l1>r2) high = cut1-1;
        else low = cut1+1;
    }
    return 0.0;
}

int main() {
    int nums1[] = {1,4,7,10,12};
    int nums2[] = {2,3,6,15};
    int m = sizeof(nums1)/sizeof(nums1[0]);
    int n = sizeof(nums2)/sizeof(nums2[0]);
    cout<<"The Median of two sorted arrays is"<<fixed<<setprecision(5)
    <<median(nums1,nums2,m,n);
    return 0;
}
```

```
Time Complexity : O(log(m,n))

Space Complexity: O(1)
```

K-th Element of two sorted arrays

Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the kth position of the final sorted array.

```
#include<bits/stdc++.h>
using namespace std;
int kthelement(int arr1[], int arr2[], int m, int n, int k) {
    if(m > n) {
        return kthelement(arr2, arr1, n, m, k);
    }

    int low = max(0,k-m), high = min(k,n);

    while(low <= high) {
        int cut1 = (low + high) >> 1;
        int cut2 = k - cut1;
        int l1 = cut1 == 0 ? INT_MIN : arr1[cut1 - 1];
        int l2 = cut2 == 0 ? INT_MIN : arr2[cut2 - 1];
        int r1 = cut1 == n ? INT_MAX : arr1[cut1];
        int r2 = cut2 == m ? INT_MAX : arr2[cut2];

        if(l1 <= r2 && l2 <= r1) {
            return max(l1, l2);
        }
        else if (l1 > r2) {
            high = cut1 - 1;
        }
        else {
            low = cut1 + 1;
        }
    }
    return 1;
}
int main() {
    int array1[] = {2,3,6,7,9};
    int array2[] = {1,4,8,10};
    int m = sizeof(array1)/sizeof(array1[0]);
    int n = sizeof(array2)/sizeof(array2[0]);
    int k = 5;
    cout<<"The element at the kth position in the final sorted array 
is "
    <<kthelement(array1,array2,m,n,k);
    return 0;
}
```

```
Time Complexity : log(min(m,n))
Space Complexity: O(1)
```

Aggressive Cows

There is a new barn with N stalls and C cows. The stalls are located on a straight line at positions x1,....,xN (0 <= xi <= 1,000,000,000). We want to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?

Input: No of stalls = 5 Array: {1,2,8,4,9} And number of cows: 3

Output: One integer, the largest minimum distance 3

```
#include <bits/stdc++.h>

  using namespace std;
  bool isPossible(int a[], int n, int cows, int minDist) {
    int cntCows = 1;
    int lastPlacedCow = a[0];
    for (int i = 1; i < n; i++) {
      if (a[i] - lastPlacedCow >= minDist) {
        cntCows++;
        lastPlacedCow = a[i];
      }
    }
    if (cntCows >= cows) return true;
    return false;
  }
  int main() {
    int n = 5, cows = 3;
    int a[]={1,2,8,4,9};
    sort(a, a + n);

    int low = 1, high = a[n - 1] - a[0];

    while (low <= high) {
      int mid = (low + high) >> 1;

      if (isPossible(a, n, cows, mid)) {
        low = mid + 1;
      } else {
        high = mid - 1;
      }
    }
    cout << "The largest minimum distance is " << high << endl;

    return 0;
  }

  TC : O(N*log(M))
  SC: O(1)
```

RECURSION & BACKTRACKING

Subset Sum : Sum of all Subsets

Given an array print all the sum of the subset generated from it, in the increasing order.

```
#include<bits/stdc++.h>

using namespace std;
class Solution {
  public:
    void solve(int ind, vector < int > & arr, int n, vector < int > &
ans, int sum) {
      if (ind == n) {
        ans.push_back(sum);
        return;
      }
      //element is picked
      solve(ind + 1, arr, n, ans, sum + arr[ind]);
      //element is not picked
      solve(ind + 1, arr, n, ans, sum);
    }
  vector < int > subsetSums(vector < int > arr, int n) {
    vector < int > ans;
    solve(0, arr, n, ans, 0);
    sort(ans.begin(), ans.end());
    return ans;
  }
};


int main() {
  vector < int > arr{3,1,2};
  Solution ob;
  vector < int > ans = ob.subsetSums(arr, arr.size());
  sort(ans.begin(), ans.end());
  cout<<"The sum of each subset is "<<endl;
  for (auto sum: ans) {
    cout << sum << " ";
  }
  cout << endl;

  return 0;
}
```

Output:

```
The sum of each subset is
0 1 2 3 3 4 5 6
```

Time Complexity: O(2^n)+O(2^n log(2^n)). Each index has two ways. You can either pick it up or not pick it. So for n index time complexity

```
for O(2^n) and for sorting it will take (2^n log(2^n)).

Space Complexity: O(2^n) for storing subset sums, since 2^n subsets
can be generated for an array of size n.
```

Subset – II | Print all the Unique Subsets

Given an array of integers that may contain duplicates the task is to return all possible subsets. Return only unique subsets and they can be in any order.

```
#include <bits/stdc++.h>

using namespace std;
void printAns(vector < vector < int >> & ans) {
   cout<<"The unique subsets are "<<endl;
   cout << "[ ";
   for (int i = 0; i < ans.size(); i++) {
      cout << "[ ";
      for (int j = 0; j < ans[i].size(); j++)
         cout << ans[i][j] << " ";
      cout << "]";
   }
   cout << " ]";
}
class Solution {
   private:
      void findSubsets(int ind, vector < int > & nums, vector < int >
& ds, vector < vector < int >> & ans) {
         ans.push_back(ds);
         for (int i = ind; i < nums.size(); i++) {
            if (i != ind && nums[i] == nums[i - 1]) continue;
            ds.push_back(nums[i]);
            findSubsets(i + 1, nums, ds, ans);
            ds.pop_back();
         }
      }
   public:
      vector < vector < int >> subsetsWithDup(vector < int > & nums) {
         vector < vector < int >> ans;
         vector < int > ds;
         sort(nums.begin(), nums.end());
         findSubsets(0, nums, ds, ans);
         return ans;
      }
};
int main() {
   Solution obj;
   vector < int > nums = {1,2,2 };
   vector < vector < int >> ans = obj.subsetsWithDup(nums);
   printAns(ans);
```

```
    return 0;
}
Output:

The unique subsets are
[ [ ][ 1 ][ 1 2 ][ 1 2 2 ][ 2 ][ 2 2 ] ]

Time Complexity: O(2^n) for generating every subset and O(k)  to
insert every subset in another data structure if the average length of
every subset is k. Overall O(k * 2^n).

Space Complexity: O(2^n * k) to store every subset of average length
k. Auxiliary space is O(n)  if n is the depth of the recursion tree.
```

Combination Sum – 1

Given an array of distinct integers and a target, you have to return the list of all unique combinations where the chosen numbers sum to target. You may return the combinations in any order.

The same number may be chosen from the given array an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is guaranteed that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

```
#include<bits/stdc++.h>

using namespace std;
class Solution {
  public:
    void findCombination(int ind, int target, vector < int > & arr,
vector < vector < int >> & ans, vector < int > & ds) {
      if (ind == arr.size()) {
        if (target == 0) {
          ans.push_back(ds);
        }
        return;
      }
      // pick up the element 
      if (arr[ind] <= target) {
        ds.push_back(arr[ind]);
        findCombination(ind, target - arr[ind], arr, ans, ds);
        ds.pop_back();
      }

      findCombination(ind + 1, target, arr, ans, ds);

    }
  public:
    vector < vector < int >> combinationSum(vector < int > &
candidates, int target) {
      vector < vector < int >> ans;
      vector < int > ds;
      findCombination(0, target, candidates, ans, ds);
      return ans;
    }
};
int main() {
  Solution obj;
  vector < int > v {2,3,6,7};
  int target = 7;

  vector < vector < int >> ans = obj.combinationSum(v, target);
  cout << "Combinations are: " << endl;
  for (int i = 0; i < ans.size(); i++) {
    for (int j = 0; j < ans[i].size(); j++)
      cout << ans[i][j] << " ";
    cout << endl;
  }
}
Output:

Combinations are:
2 2 3
7

Time Complexity: O(2^t * k) where t is the target, k is the average
length

Reason: Assume if you were not allowed to pick a single element
multiple times, every element will have a couple of options: pick or
not pick which is 2^n different recursion calls, also assuming that
the average length of every combination generated is k. (to put length
k data structure into another data structure)

Why not (2^n) but (2^t) (where n is the size of an array)?

Assume that there is 1 and the target you want to reach is 10 so 10
times you can "pick or not pick" an element.

Space Complexity: O(k*x), k is the average length and x is the no. of
combinations
```

Combination Sum II – Find all unique combinations

Problem Statement: Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination.

```
#include<bits/stdc++.h>

using namespace std;
void findCombination(int ind, int target, vector < int > & arr, vector
< vector < int >> & ans, vector < int > & ds) {
  if (target == 0) {
    ans.push_back(ds);
    return;
  }
  for (int i = ind; i < arr.size(); i++) {
    if (i > ind && arr[i] == arr[i - 1]) continue;
    if (arr[i] > target) break;
    ds.push_back(arr[i]);
    findCombination(i + 1, target - arr[i], arr, ans, ds);
    ds.pop_back();
  }
}
vector < vector < int >> combinationSum2(vector < int > & candidates,
int target) {
  sort(candidates.begin(), candidates.end());
  vector < vector < int >> ans;
  vector < int > ds;
  findCombination(0, target, candidates, ans, ds);
  return ans;
}
int main() {
  vector<int> v{10,1,2,7,6,1,5};
  vector < vector < int >> comb = combinationSum2(v, 8);
  cout << "[ ";
  for (int i = 0; i < comb.size(); i++) {
    cout << "[ ";
    for (int j = 0; j < comb[i].size(); j++) {
      cout << comb[i][j] << " ";
    }
    cout << "]";
  }
  cout << " ]";
}
```

Output:

```
[ [ 1 1 6 ][ 1 2 5 ][ 1 7 ][ 2 6 ] ]
```

Time Complexity:O(2^n*k)

Reason: Assume if all the elements in the array are unique then the no. of subsequence you will get will be O(2^n). we also add the ds to our ans when we reach the base case that will take "k"//average space for the ds.

Space Complexity:O(k*x)

```
Reason: if we have x combinations then space will be x*k where k is
the average length of the combination.
```

Palindrome Partitioning

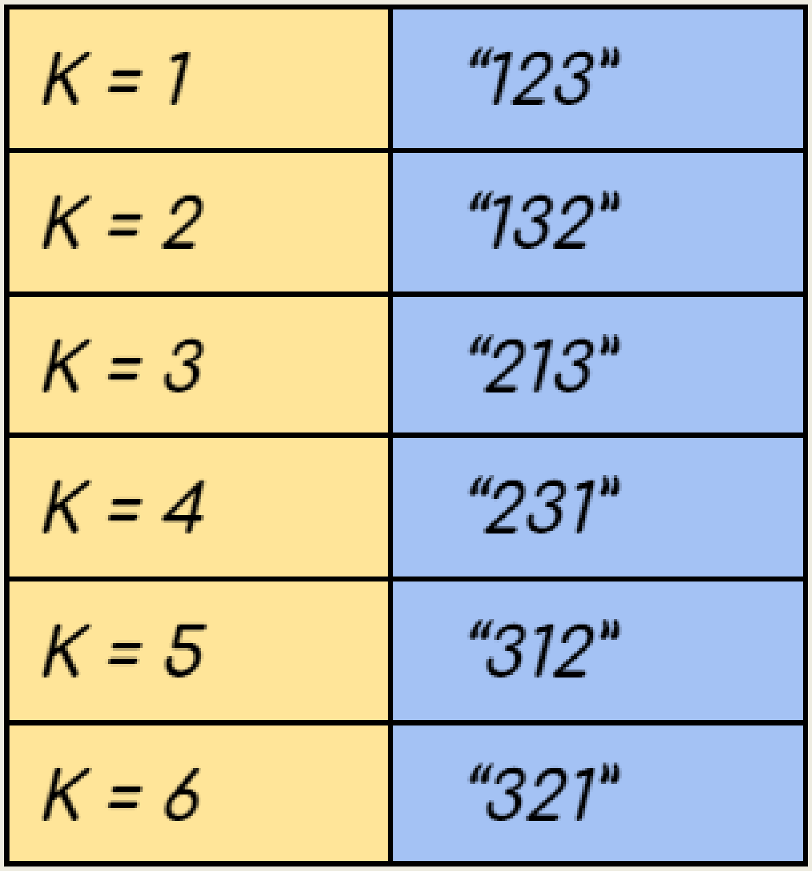
You are given a string s, partition it in such a way that every substring is a palindrome. Return all such palindromic partitions of s.

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
  public:
    vector < vector < string >> partition(string s) {
      vector < vector < string > > res;
      vector < string > path;
      func(0, s, path, res);
      return res;
    }

  void func(int index, string s, vector < string > & path,
    vector < vector < string > > & res) {
    if (index == s.size()) {
      res.push_back(path);
      return;
    }
    for (int i = index; i < s.size(); ++i) {
      if (isPalindrome(s, index, i)) {
        path.push_back(s.substr(index, i - index + 1));
        func(i + 1, s, path, res);
        path.pop_back();
      }
    }
  }

  bool isPalindrome(string s, int start, int end) {
    while (start <= end) {
      if (s[start++] != s[end--])
        return false;
    }
    return true;
  }
};
int main() {
  string s = "aabb";
  Solution obj;
  vector < vector < string >> ans = obj.partition(s);
  int n = ans.size();
  cout << "The Palindromic partitions are :-" << endl;
  cout << " [ ";
  for (auto i: ans) {
    cout << "[ ";
    for (auto j: i) {
      cout << j << " ";
    }
    cout << "] ";
  }
  cout << "]";

  return 0;
}
Output:

The Palindromic partitions are :-
[ [ a a b b ] [ a a bb ] [ aa b b ] [ aa bb ] ]

Time Complexity: O( (2^n) *k*(n/2) )

Reason: O(2^n) to generate every substring and O(n/2) to check if the
substring generated is a palindrome. O(k) is for inserting the
palindromes in another data structure, where k  is the average length
of the palindrome list.

Space Complexity: O(k * x)

Reason: The space complexity can vary depending upon the length of the
answer. k is the average length of the list of palindromes and if we
have x such list of palindromes in our final answer. The depth of the
recursion tree is n, so the auxiliary space required is equal to the
O(n).
```

Find K-th Permutation Sequence

Given N and K, where N is the sequence of numbers from 1 to N([1,2,3..... N]) find the Kth permutation sequence.

For N = 3 the 3! Permutation sequences in order would look like this:-



K = 1	“123”
K = 2	“132”
K = 3	“213”
K = 4	“231”
K = 5	“312”
K = 6	“321”

Note: 1<=K<=N!

Hence for a given input its Kth permutation always exists

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
  public:
    string getPermutation(int n, int k) {
      int fact = 1;
      vector < int > numbers;
      for (int i = 1; i < n; i++) {
        fact = fact * i;
        numbers.push_back(i);
      }
      numbers.push_back(n);
      string ans = "";
      k = k - 1;
      while (true) {
        ans = ans + to_string(numbers[k / fact]);
        numbers.erase(numbers.begin() + k / fact);
        if (numbers.size() == 0) {
          break;
        }

        k = k % fact;
        fact = fact / numbers.size();
      }
      return ans;
    }
};

int main() {
  int n = 3, k = 3;
  Solution obj;
  string ans = obj.getPermutation(n, k);
  cout << "The Kth permutation sequence is " << ans << endl;

  return 0;
}
```

```
Output:

The Kth permutation sequence is 213
```

Time Complexity: O(N) * O(N) = O(N^2)

Reason: We are placing N numbers in N positions. This will take O(N) time. For every number, we are reducing the search space by removing the element already placed in the previous step. This takes another O(N) time.

Space Complexity: O(N)

Reason: We are storing the numbers in a data structure(here vector)

Print All Permutations of a String/Array

Given an array arr of distinct integers, print all permutations of String/Array.

```
#include<bits/stdc++.h>

using namespace std;
class Solution {
  private:
    void recurPermute(vector < int > & ds, vector < int > & nums,
vector < vector < int >> & ans, int freq[]) {
      if (ds.size() == nums.size()) {
        ans.push_back(ds);
        return;
      }
      for (int i = 0; i < nums.size(); i++) {
        if (!freq[i]) {
          ds.push_back(nums[i]);
          freq[i] = 1;
          recurPermute(ds, nums, ans, freq);
          freq[i] = 0;
          ds.pop_back();
        }
      }
    }
  public:
    vector < vector < int >> permute(vector < int > & nums) {
      vector < vector < int >> ans;
      vector < int > ds;
      int freq[nums.size()];
      for (int i = 0; i < nums.size(); i++) freq[i] = 0;
      recurPermute(ds, nums, ans, freq);
      return ans;
    }
};

int main() {
  Solution obj;
  vector<int> v{1,2,3};
  vector < vector < int >> sum = obj.permute(v);
  cout << "All Permutations are " << endl;
  for (int i = 0; i < sum.size(); i++) {
    for (int j = 0; j < sum[i].size(); j++)
      cout << sum[i][j] << " ";
    cout << endl;
  }
}
```

Time Complexity: N! x N

Space Complexity: O(N)

```
#include<bits/stdc++.h>

using namespace std;
class Solution {
  private:
    void recurPermute(int index, vector < int > & nums, vector <
vector < int >> & ans) {
      if (index == nums.size()) {
        ans.push_back(nums);
        return;
      }
      for (int i = index; i < nums.size(); i++) {
        swap(nums[index], nums[i]);
        recurPermute(index + 1, nums, ans);
        swap(nums[index], nums[i]);
      }
    }
  public:
    vector < vector < int >> permute(vector < int > & nums) {
      vector < vector < int >> ans;
      recurPermute(0, nums, ans);
      return ans;
    }
};

int main() {
  Solution obj;
  vector < int > v {1,2,3};
  vector < vector < int >> sum = obj.permute(v);
  cout << "All Permutations are" << endl;
  for (int i = 0; i < sum.size(); i++) {
    for (int j = 0; j < sum[i].size(); j++)
      cout << sum[i][j] << " ";
    cout << endl;
  }
}
```

Time Complexity: O(N! X N)

Space Complexity: O(1)

N Queen Problem | Return all Distinct Solutions to the N-Queens Puzzle

The n-queens is the problem of placing n queens on n × n chessboard such that no two queens can attack each other. Given an integer n, return all distinct solutions to the n - queens puzzle. Each solution contains a distinct boards configuration of the queen's placement, where 'Q' and '.' indicate queen and empty space respectively.

```
#include <bits/stdc++.h>

using namespace std;
class Solution {
  public:
    bool isSafe1(int row, int col, vector < string > board, int n) {
      // check upper element
      int duprow = row;
      int dupcol = col;

      while (row >= 0 && col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        row--;
```

```
          col--;
        }

        col = dupcol;
        row = duprow;
        while (col >= 0) {
          if (board[row][col] == 'Q')
            return false;
          col--;
        }

        row = duprow;
        col = dupcol;
        while (row < n && col >= 0) {
          if (board[row][col] == 'Q')
            return false;
          row++;
          col--;
        }
        return true;
      }

    public:
      void solve(int col, vector < string > & board, vector < vector <
  string >> & ans, int n) {
        if (col == n) {
          ans.push_back(board);
          return;
        }
        for (int row = 0; row < n; row++) {
          if (isSafe1(row, col, board, n)) {
            board[row][col]  =  'Q';
            solve(col + 1, board, ans, n);
            board[row][col] = '.';
          }
        }
      }

    public:
      vector < vector < string >> solveNQueens(int n) {
        vector < vector < string >> ans;
        vector < string > board(n);
        string s(n, '.');
        for (int i = 0; i < n; i++) {
          board[i] = s;
        }
        solve(0, board, ans, n);
        return ans;
      }
  };
```

```
int main() {
  int n = 4; // we are taking 4*4 grid and 4 queens
  Solution obj;
  vector < vector < string >> ans = obj.solveNQueens(n);
  for (int i = 0; i < ans.size(); i++) {
    cout << "Arrangement " << i + 1 << "\n";
    for (int j = 0; j < ans[0].size(); j++) {
      cout << ans[i][j];
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

Time Complexity: Exponential in nature, since we are trying out all ways. To be precise it goes as O

(N! * N) nearly.

Space Complexity: O(N^2)

```
class Solution {
  public:
    void solve(int col, vector < string > & board, vector < vector <
string >> & ans, vector < int > & leftrow, vector < int > &
upperDiagonal, vector < int > & lowerDiagonal, int n) {
      if (col == n) {
        ans.push_back(board);
        return;
      }
      for (int row = 0; row < n; row++) {
        if (leftrow[row] == 0 && lowerDiagonal[row + col] == 0 &&
upperDiagonal[n - 1 + col - row] == 0) {
          board[row][col] = 'Q';
          leftrow[row] = 1;
          lowerDiagonal[row + col] = 1;
          upperDiagonal[n - 1 + col - row] = 1;
          solve(col + 1, board, ans, leftrow, upperDiagonal,
lowerDiagonal, n);
          board[row][col] = '.';
          leftrow[row] = 0;
          lowerDiagonal[row + col] = 0;
          upperDiagonal[n - 1 + col - row] = 0;
        }
      }
    }

  public:
    vector < vector < string >> solveNQueens(int n) {
      vector < vector < string >> ans;
      vector < string > board(n);
      string s(n, '.');
      for (int i = 0; i < n; i++) {
        board[i] = s;
      }
      vector < int > leftrow(n, 0), upperDiagonal(2 * n - 1, 0),
lowerDiagonal(2 * n - 1, 0);
      solve(0, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);
      return ans;
    }
};
int main() {
  int n = 4; // we are taking 4*4 grid and 4 queens
  Solution obj;
  vector < vector < string >> ans = obj.solveNQueens(n);
  for (int i = 0; i < ans.size(); i++) {
    cout << "Arrangement " << i + 1 << "\n";
    for (int j = 0; j < ans[0].size(); j++) {
      cout << ans[i][j];
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

```
Time Complexity: Exponential in nature since we are trying out all
ways, to be precise it is O(N! * N).

Space Complexity: O(N)
```

Sudoku Solver

Given a 9×9 incomplete sudoku, solve it such that it becomes valid sudoku. Valid sudoku has the following properties.

```
     1. All the rows should be filled with numbers(1 - 9) exactly
once.

     2. All the columns should be filled with numbers(1 - 9) exactly
once.
```

3. Each 3×3 submatrix should be filled with numbers(1 – 9) exactly once.

Note: Character '.' indicates empty cell.

```
#include <iostream>

#include <vector>

using namespace std;

bool isValid(vector < vector < char >> & board, int row, int col, char
c) {
  for (int i = 0; i < 9; i++) {
    if (board[i][col] == c)
      return false;

    if (board[row][i] == c)
      return false;

    if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
      return false;
  }
  return true;
}

bool solve(vector < vector < char >> & board) {
  for (int i = 0; i < board.size(); i++) {
    for (int j = 0; j < board[0].size(); j++) {
      if (board[i][j] == '.') {
        for (char c = '1'; c <= '9'; c++) {
          if (isValid(board, i, j, c)) {
            board[i][j] = c;

            if (solve(board))
              return true;
            else
              board[i][j] = '.';
          }
        }

        return false;
      }
    }
  }
  return true;
}
int main() {
    vector<vector<char>>board{
        {'9', '5', '7', '.', '1', '3', '.', '8', '4'},
        {'4', '8', '3', '.', '5', '7', '1', '.', '6'},
        {'.', '1', '2', '.', '4', '9', '5', '3', '7'},
        {'1', '7', '.', '3', '.', '4', '9', '.', '2'},
        {'5', '.', '4', '9', '7', '.', '3', '6', '.'},
        {'3', '.', '9', '5', '.', '8', '7', '.', '1'},
        {'8', '4', '5', '7', '9', '.', '6', '1', '3'},
        {'.', '9', '1', '.', '3', '6', '.', '7', '5'},
        {'7', '.', '6', '1', '8', '5', '4', '.', '9'}
    };

    solve(board);

    for(int i= 0; i< 9; i++){
        for(int j= 0; j< 9; j++)
            cout<<board[i][j]<<" ";
            cout<<"\n";
    }
    return 0;
}
```

Time Complexity: O(9(n ^ 2)), in the worst case, for each cell in the n2 board, we have 9 possible numbers.

Space Complexity: O(1), since we are refilling the given board itself, there is no extra space required, so constant space complexity.

M – Coloring Problem

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color.

Example 1: Input:

1. N = 4
2. M = 3
3. E = 5

Edges[] = { (0, 1), (1, 2), (2, 3), (3, 0), (0, 2) }

Output: 1

Explanation: It is possible to colour the given graph using 3 colours.

```
#include<bits/stdc++.h>

using namespace std;
bool isSafe(int node, int color[], bool graph[101][101], int n, int
col) {
  for (int k = 0; k < n; k++) {
    if (k != node && graph[k][node] == 1 && color[k] == col) {
      return false;
    }
  }
  return true;
}
bool solve(int node, int color[], int m, int N, bool graph[101][101])
{
  if (node == N) {
    return true;
  }

  for (int i = 1; i <= m; i++) {
    if (isSafe(node, color, graph, N, i)) {
      color[node] = i;
      if (solve(node + 1, color, m, N, graph)) return true;
      color[node] = 0;
    }

  }
  return false;
}

//Function to determine if graph can be coloured with at most M
colours such
//that no two adjacent vertices of graph are coloured with same
colour.
bool graphColoring(bool graph[101][101], int m, int N) {
  int color[N] = {
    0
  };
  if (solve(0, color, m, N, graph)) return true;
  return false;
}

int main() {
  int N = 4;
  int m = 3;

  bool graph[101][101] = {
    (0, 1),
    (1, 2),
    (2, 3),
    (3, 0),
    (0, 2)
  };
  cout << graphColoring(graph, m, N);

}
```

```
Time Complexity: O( N^M) (n raised to m)

Space Complexity: O(N)
```

Rat in a Maze

Consider a rat placed at (0, 0) in a square matrix of order N * N. It has to reach the destination at (N – 1, N – 1). Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are 'U'(up), 'D'(down), 'L' (left), 'R' (right). Value 0 at a cell in the matrix represents that it is blocked and the rat cannot move to it while value 1 at a cell in the matrix represents that rat can travel through it.

Note: In a path, no cell can be visited more than one time.

Print the answer in lexicographical(sorted) order

Input: N = 4

m[][] = {{1, 0, 0, 0}, {1, 1, 0, 1}, {1, 1, 0, 0}, {0, 1, 1, 1}}

Output: DDRDRR DRDDRR

```
#include <bits/stdc++.h>

using namespace std;

class Solution {
  void solve(int i, int j, vector < vector < int >> & a, int n, vector
< string > & ans, string move,
    vector < vector < int >> & vis) {
    if (i == n - 1 && j == n - 1) {
      ans.push_back(move);
      return;
    }

    // downward
    if (i + 1 < n && !vis[i + 1][j] && a[i + 1][j] == 1) {
      vis[i][j] = 1;
      solve(i + 1, j, a, n, ans, move + 'D', vis);
      vis[i][j] = 0;
    }

    // left
    if (j - 1 >= 0 && !vis[i][j - 1] && a[i][j - 1] == 1) {
      vis[i][j] = 1;
      solve(i, j - 1, a, n, ans, move + 'L', vis);
      vis[i][j] = 0;
    }
    // right
    if (j + 1 < n && !vis[i][j + 1] && a[i][j + 1] == 1) {
      vis[i][j] = 1;
      solve(i, j + 1, a, n, ans, move + 'R', vis);
      vis[i][j] = 0;
    }

    // upward
    if (i - 1 >= 0 && !vis[i - 1][j] && a[i - 1][j] == 1) {
      vis[i][j] = 1;
      solve(i - 1, j, a, n, ans, move + 'U', vis);
      vis[i][j] = 0;
    }

  }
  public:
    vector < string > findPath(vector < vector < int >> & m, int n) {
      vector < string > ans;
      vector < vector < int >> vis(n, vector < int > (n, 0));

      if (m[0][0] == 1) solve(0, 0, m, n, ans, "", vis);
      return ans;
    }
};

int main() {
  int n = 4;

   vector < vector < int >> m = {{1,0,0,0},{1,1,0,1},{1,1,0,0},
{0,1,1,1}};

  Solution obj;
  vector < string > result = obj.findPath(m, n);
  if (result.size() == 0)
    cout << -1;
  else
    for (int i = 0; i < result.size(); i++) cout << result[i] << " ";
  cout << endl;

  return 0;
}

Time Complexity: O(4^(m*n)), because on every cell we need to try 4
different directions.

Space Complexity:  O(m*n) ,Maximum Depth of the recursion
tree(auxiliary space).
```

Word Break I

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.

Consider the following dictionary

{ i, like, sam, sung, samsung, mobile, ice, cream, icecream, man, go, mango}

Input: ilike

Output: Yes

The string can be segmented as "i like".

```
RECURSION---

#include <iostream>
using namespace std;

int dictionaryContains(string word)
{
      string dictionary[] = {"mobile","samsung","sam","sung",
                                          "man","mango","icecream","and",
                                          "go","i","like","ice","cream"};
      int size = sizeof(dictionary)/sizeof(dictionary[0]);
      for (int i = 0; i < size; i++)
            if (dictionary[i].compare(word) == 0)
            return true;
      return false;
}

bool wordBreak(string str)
{
      int size = str.size();

      // Base case
      if (size == 0) return true;

      // Try all prefixes of lengths from 1 to size
      for (int i=1; i<=size; i++)
      {
            if (dictionaryContains( str.substr(0, i) ) && wordBreak(
str.substr(i, size-i) ))
                  return true;
      }


      return false;
}
```

```
int main()
{
      wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
      return 0;
}
```

Word Break II - Print all Ways

INPUT:

6

god is now no where here

godisnowherenowhere

OUTPUT:

god is no where no where

god is no where now here

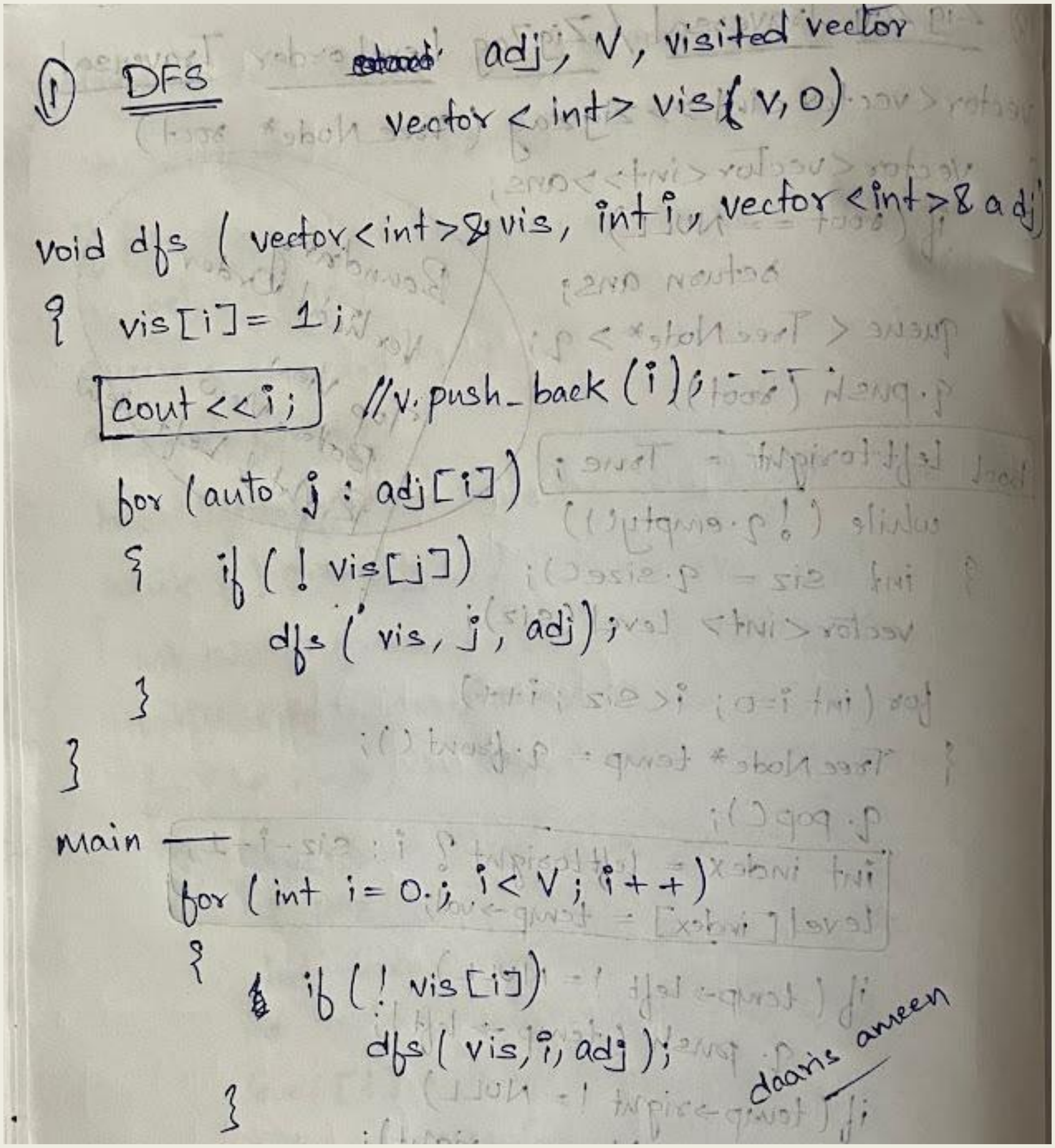
god is now here no where

god is now here now here

```
    Time Complexity: O(N * (2 ^ (N - 1)))
    Space Complexity: O(N* (2 ^ (N - 1))

    Where N is the length of the string S.

#include <unordered_set>
using namespace std;

vector<string> wordBreakHelper(string &s, int idx,
unordered_set<string> &dictSet, int size)
{
    // Base Condition
    if (idx == size)
    {
        return {""};
    }

    vector<string> subPart, completePart;
    string word = "";

    /*
        Start exploring the sentence from the index until we wouldn't
find 'j' such that substring [index,j] exists in the dictionary as a
word.
```

```
    */
    for (int j = idx; j < size; j++)
    {

        word.push_back(s[j]);
        if (dictSet.count(word) == 0)
        {
            continue;
        }

        //  Get the answer for rest of sentence from 'j' to s.size().
        subPart = wordBreakHelper(s, j + 1, dictSet, size);

        //  Append "word" with all the answer that we got.
        for (int i = 0; i < subPart.size(); i++)
        {
            if (subPart[i].size() != 0)
            {
                string temp = word;
                temp.append(" ");
                temp.append(subPart[i]);
                subPart[i] = temp;
            }
            else
            {
                subPart[i] = word;
            }
        }

        for (int i = 0; i < subPart.size(); i++)
        {
            completePart.push_back(subPart[i]);
        }
    }
    return completePart;
}

vector<string> wordBreak(string &s, vector<string> &dictionary)
{
    //  Set to check the whether any word exists in the dictionary or
not.
    unordered_set<string> dictSet;

    for (int i = 0; i < dictionary.size(); i++)
    {
        dictSet.insert(dictionary[i]);
    }
    return wordBreakHelper(s, 0, dictSet, s.size());
}
```

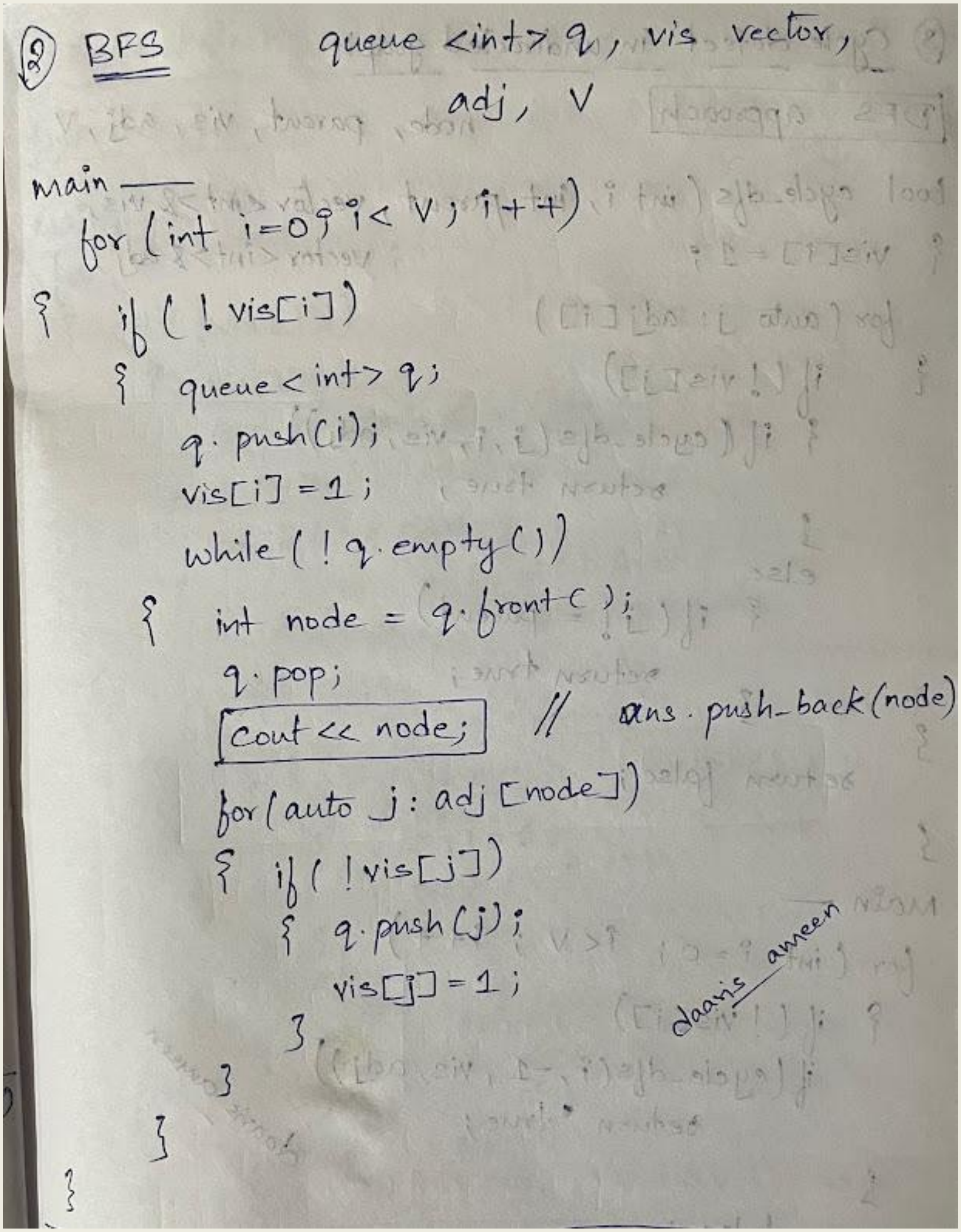
GRAPHS



① DFS   adj, V, visited vector

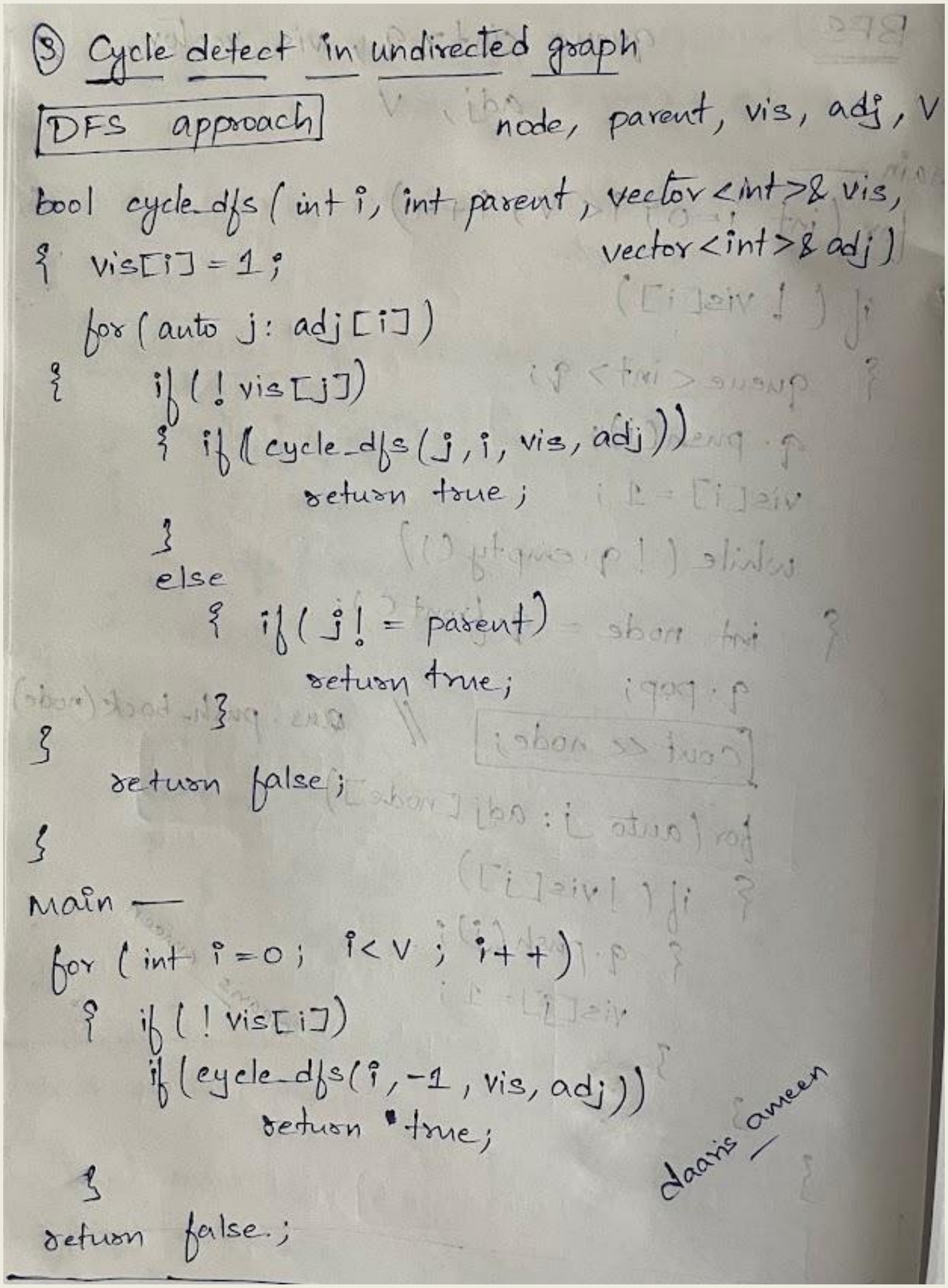
```
vector<int> vis(V, 0)

void dfs (vector<int>& vis, int i, vector<int>& adj)
{   vis[i] = 1;
    cout<<i;   //v.push_back(i);
    for (auto j : adj[i])
    {   if (!vis[j])
            dfs(vis, j, adj);
    }
}

main —
    for (int i = 0; i < V; i++)
    {   if (!vis[i])
            dfs(vis, i, adj);
    }
```

daaris ameen



② **BFS** queue <int> q, vis vector,
adj, V

```
main ____
  for (int i=0; i< V; i++)
  {   if ( ! vis[i])
      {  queue< int> q;
         q. push(i);
         vis[i] = 1;
         while ( ! q. empty())
        {  int node = q. front();
           q. pop;
           cout << node;      // ans. push_back(node)
           for(auto j : adj [node])
           {  if( !vis[j])
              {  q. push(j);
                 vis[j] = 1;
              }
           }
        }
  }
```

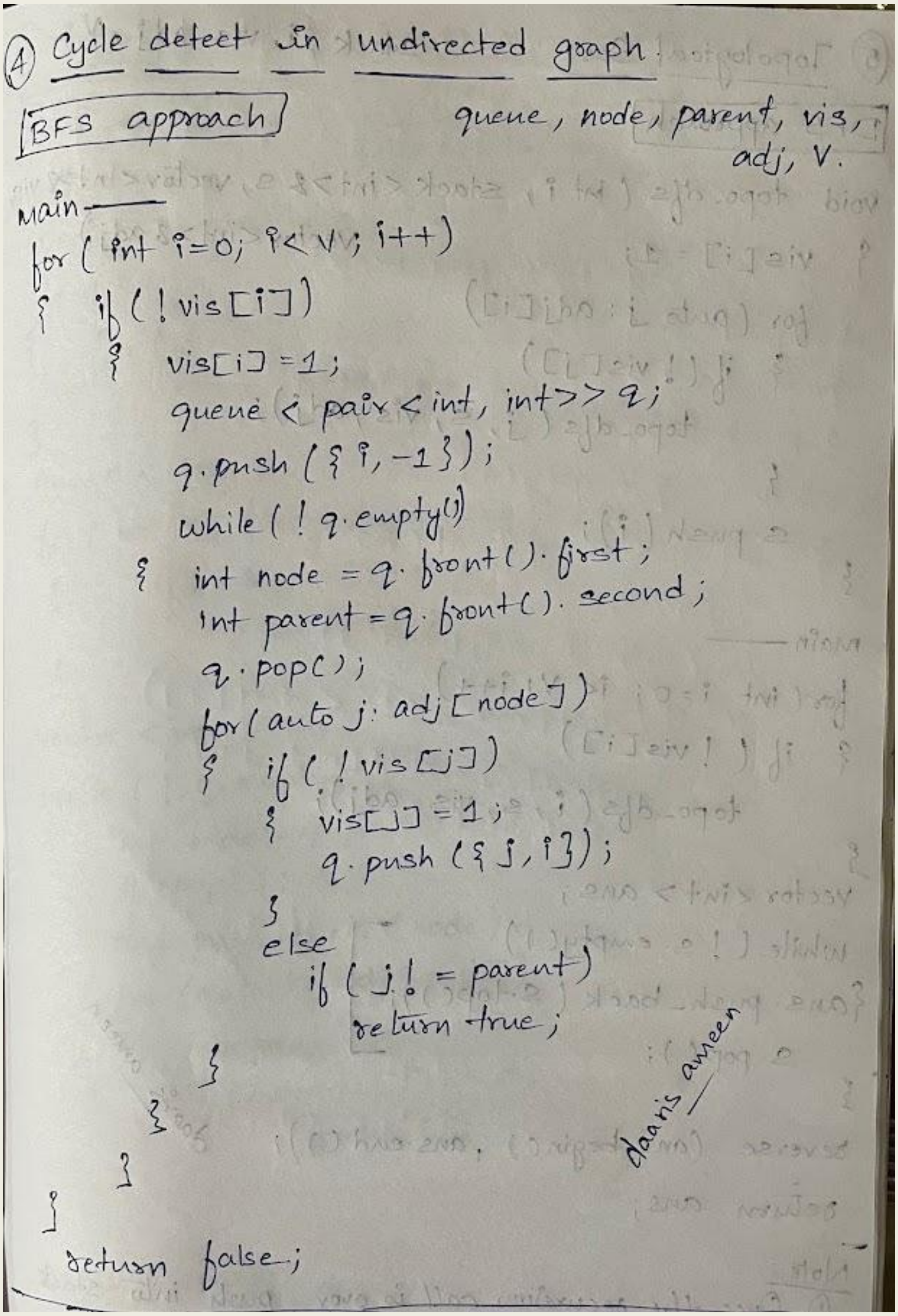


③ Cycle detect in undirected graph

DFS approach   node, parent, vis, adj, V

```
bool cycle_dfs( int i, int parent, vector<int>& vis,
                                   vector<int>& adj)
{   vis[i] = 1;
   for( auto j: adj[i])
  {       if( ! vis[j])
          {  if((cycle_dfs( j, i, vis, adj))
                     return true;
          }
          else
              {  if( j! = parent)
                     return true;
          }}
}
     return false;
}
Main —
 for ( int  i = 0;   i < V ;  i++)
   {  if( ! vis[i])
         if( cycle_dfs( i, -1 , vis, adj))
                 return true;
   }
 return  false;
```

daaris ameen



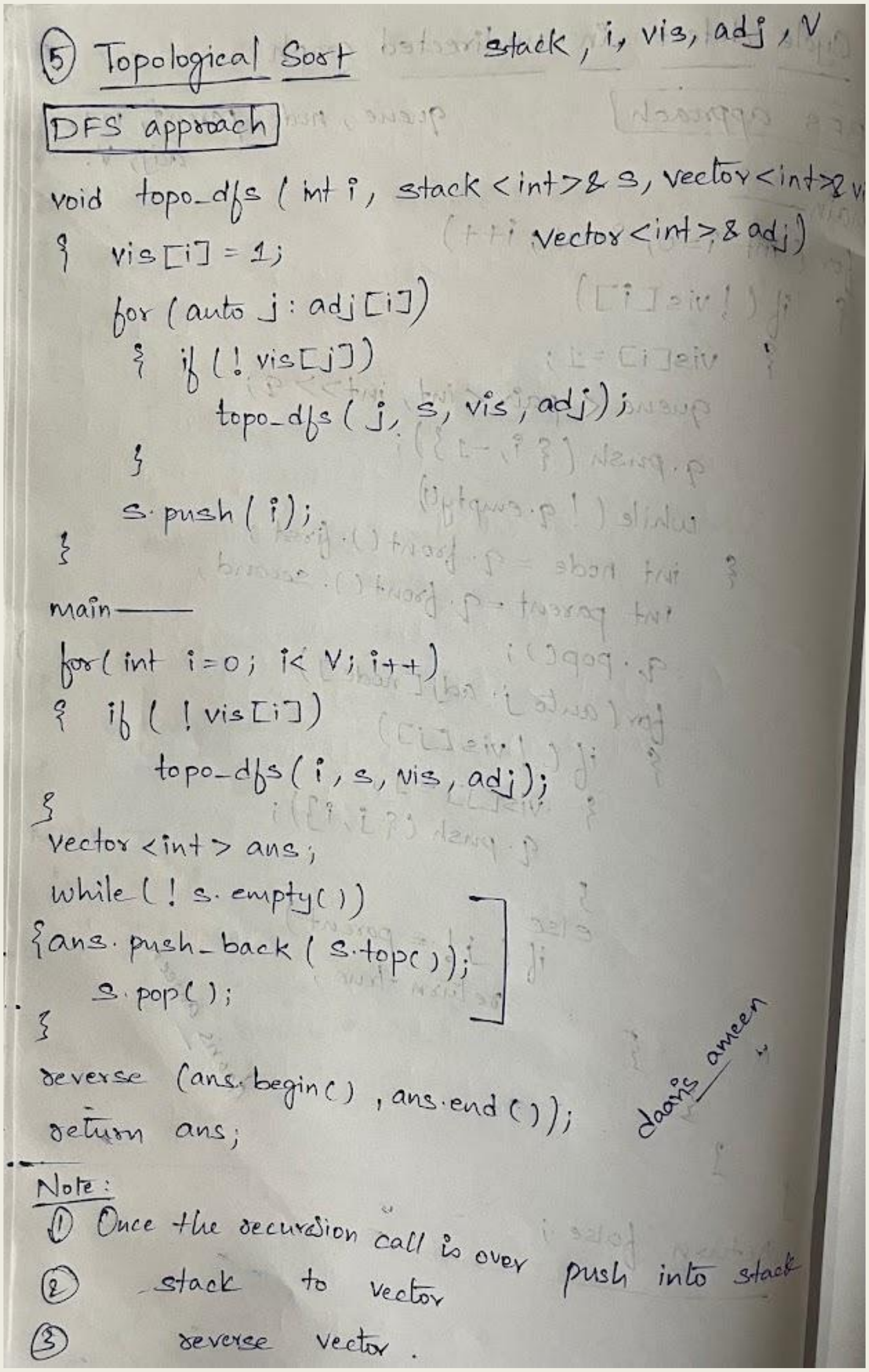
④ Cycle detect in undirected graph

BFS approach

queue, node, parent, vis, adj, V.

```
main
for ( int i=0; i< V; i++)
{   if ( ! vis[i])
     {   vis[i] = 1;
         queue < pair<int, int>> q;
         q.push ({i, -1});
         while ( ! q.empty())
       {   int node = q.front().first;
           int parent = q.front().second;
           q.pop();
           for( auto j: adj[node])
           {   if ( ! vis[j])
               {   vis[j] = 1;
                   q.push ({j, i});
               }
               else
                  if ( j != parent)
                     return true;
           }
       }
     }
}
  return false;
```

daaris ameen



⑤ Topological Sort   stack, i, vis, adj, V

DFS approach

```
void topo_dfs ( int i, stack<int>& s, vector<int>& v
                                    vector<int>& adj)
{  vis[i] = 1;
   for (auto j : adj[i])
    {  if (!vis[j])
          topo_dfs ( j, s, vis, adj);
    }
   s.push ( i);
}
main ———
for( int  i=0; i< V; i++)
{  if ( ! vis[i])
       topo_dfs( i, s, vis, adj);
}
vector <int> ans;
while ( ! s.empty( ))
{ans.push_back ( s.top( ));
   s.pop( );
}
reverse (ans.begin( ) , ans.end ( ));
return ans;
```

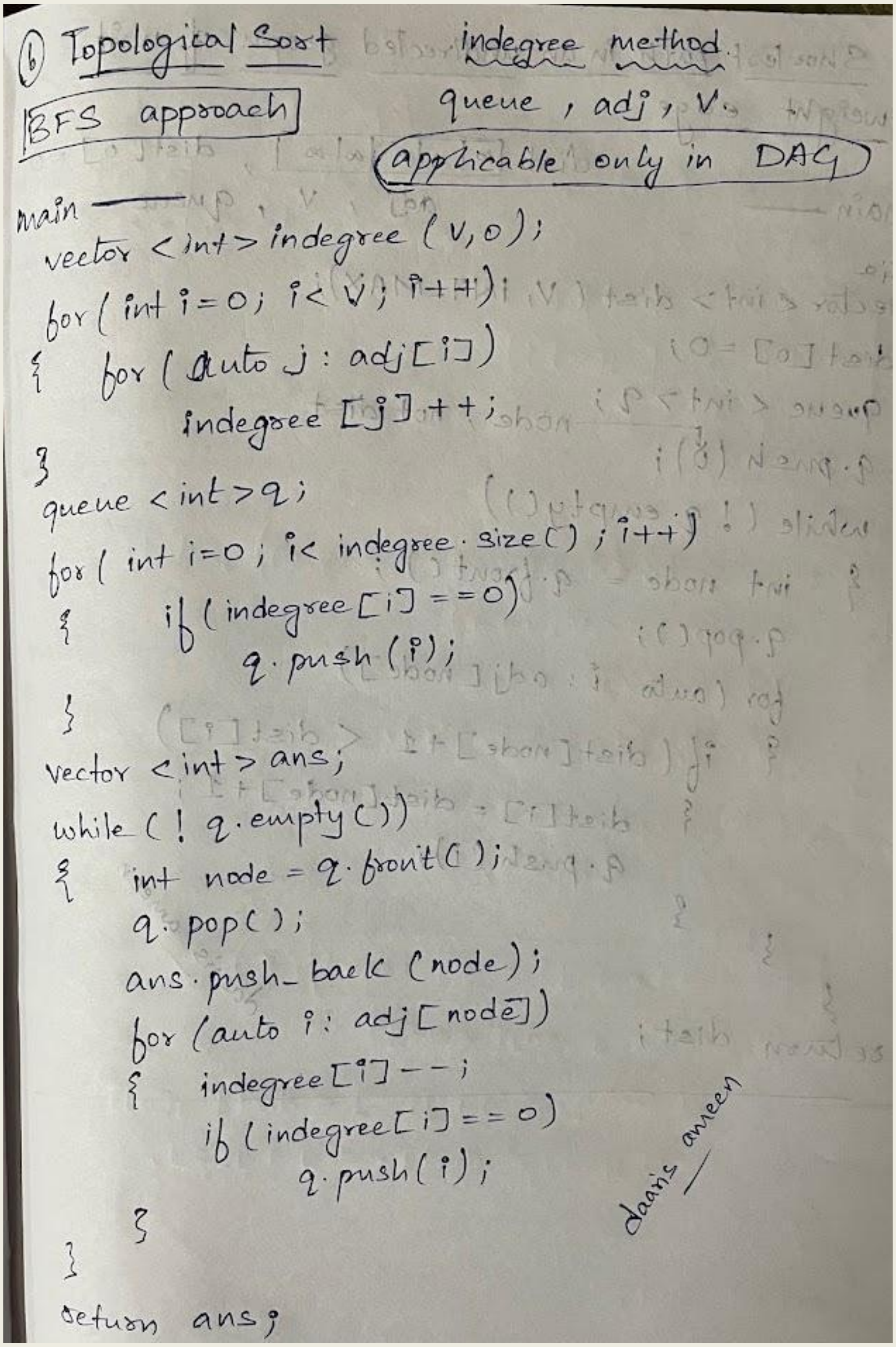
daaris ameen

Note:

① Once the recursion call is over push into stack

② stack to vector

③ reverse vector.



(b) Topological Sort indegree method

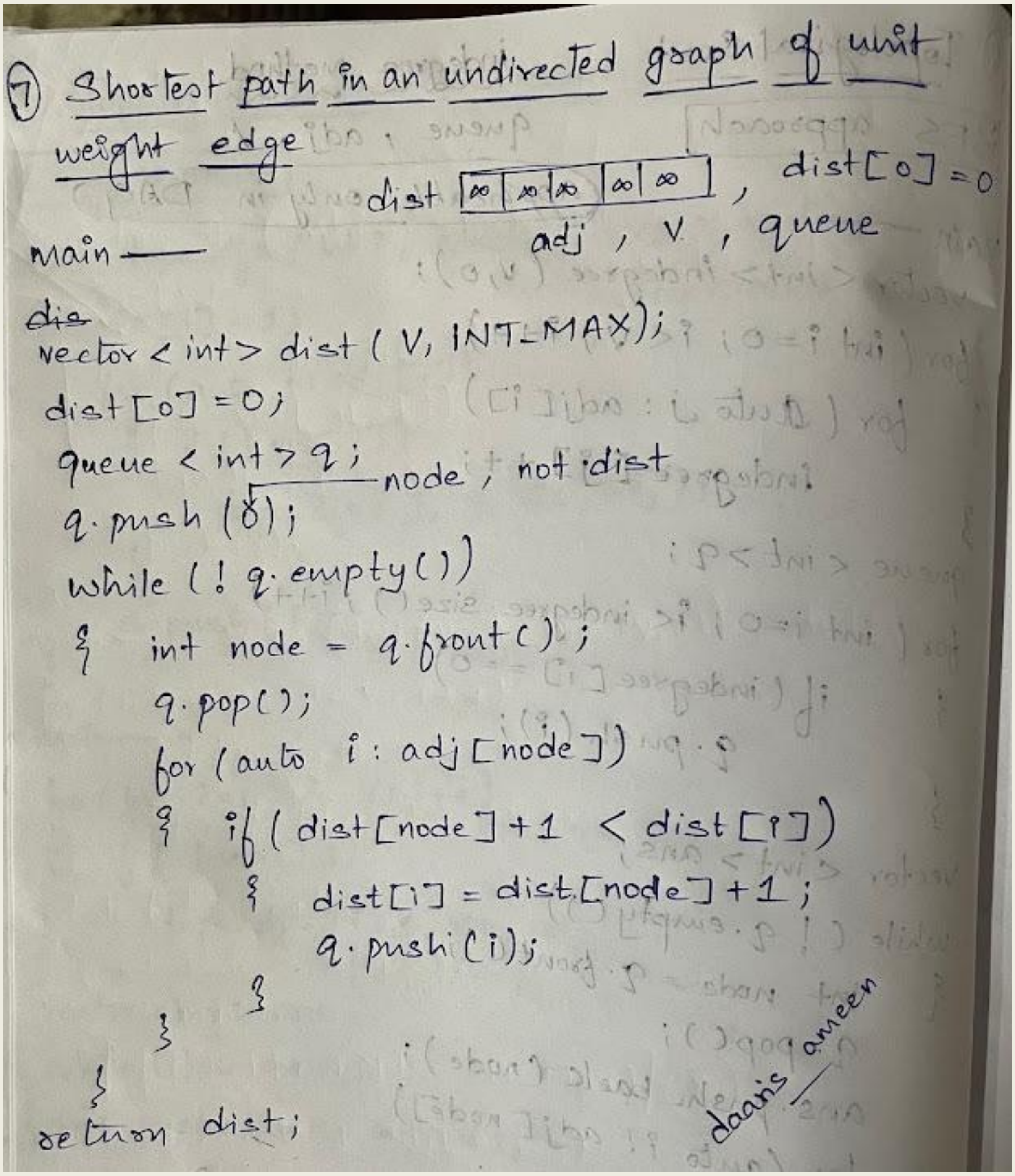
BFS approach   queue, adj, V.

(applicable only in DAG)

main ——

```
vector <int> indegree (V,0);
for ( int i=0; i< V; i++)
{    for ( auto j : adj[i])
         indegree [j] ++;
}
queue <int> q;
for ( int i=0; i< indegree.size() ; i++)
{       if ( indegree[i] ==0)
             q.push (i);
}
vector <int> ans;
while ( ! q.empty())
{    int node = q.front();
     q.pop();
     ans.push_back (node);
     for (auto i: adj[node])
     {    indegree[i] --;
          if (indegree[i] == 0)
               q.push(i);
     }
}
return ans;
```

daaris ameen



⑦ Shortest path in an undirected graph of unit weight edge

dist | ∞ | ∞ | ∞ | ∞ | ∞ | , dist[0] = 0

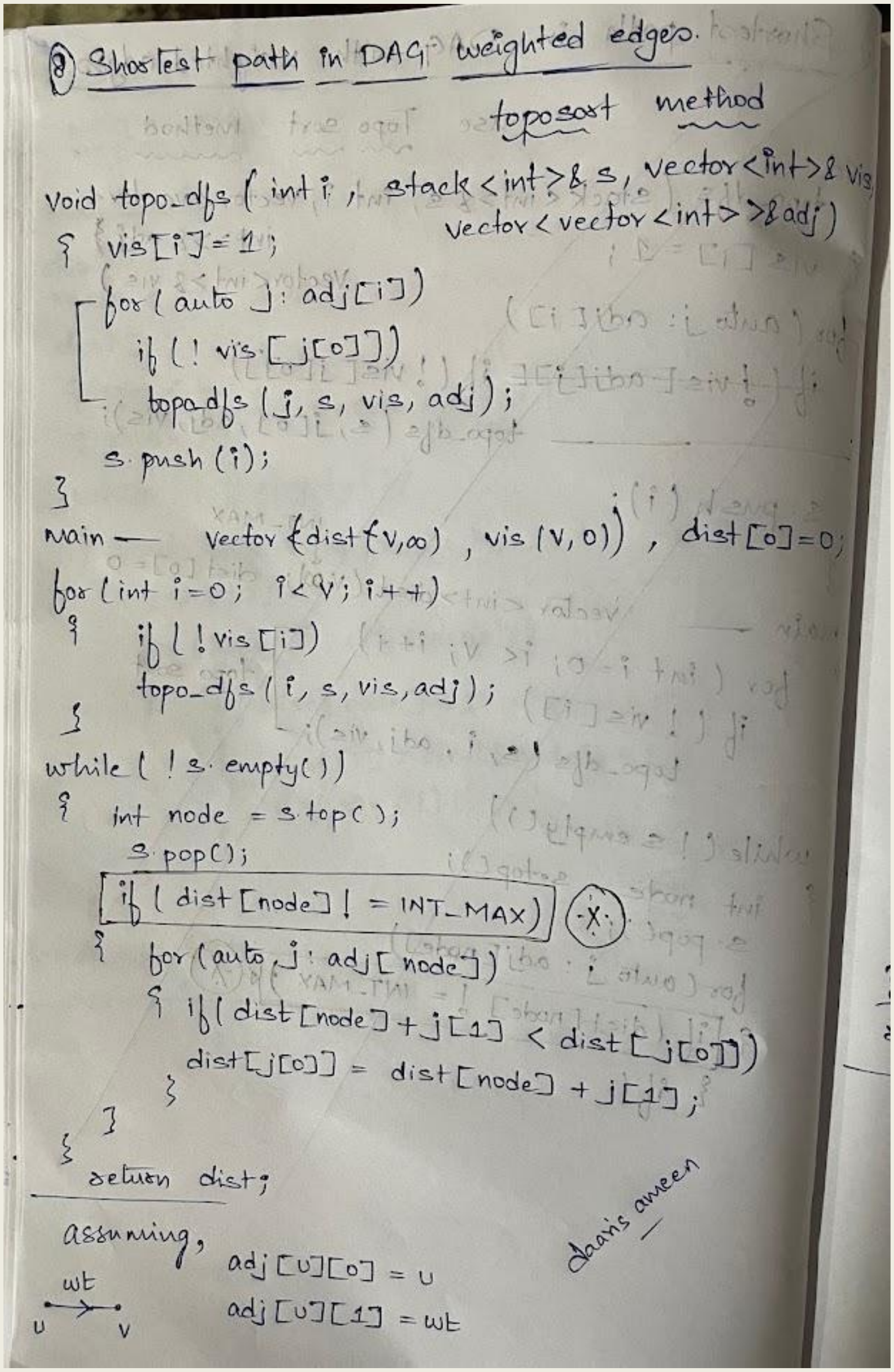
adj , V , queue

main ——

~~dis~~

```
vector<int> dist( V, INT_MAX);
dist[0] = 0;
queue<int> q;
q.push(0);          node, not dist
while(!q.empty())
{   int node = q.front();
    q.pop();
    for(auto i : adj[node])
    {   if(dist[node]+1 < dist[i])
        {   dist[i] = dist[node]+1;
            q.push(i);
        }
    }
}
return dist;
```

daaris ameen



(8) Shortest path in DAG weighted edges.

toposort method

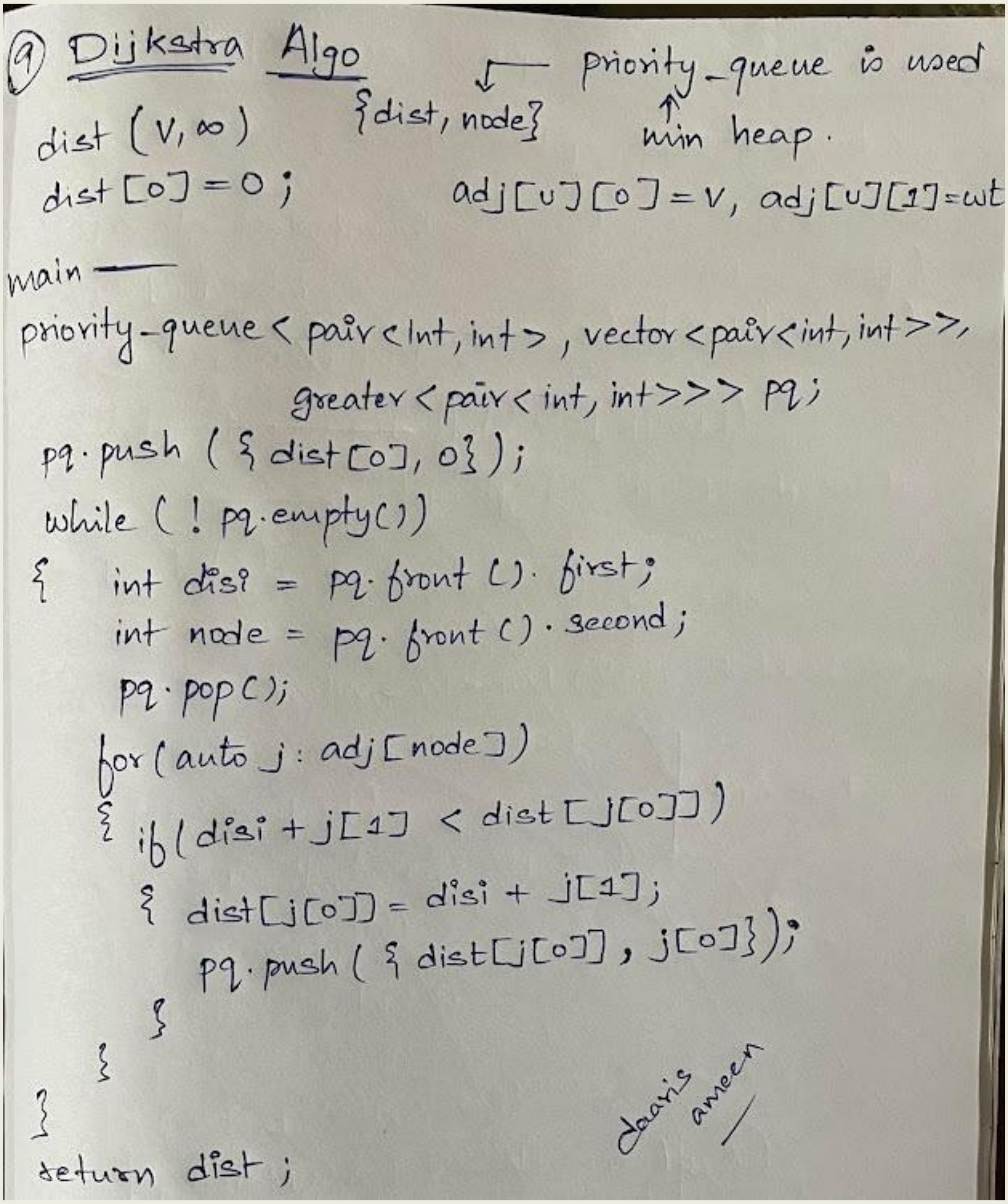
```
void topo_dfs(int i, stack<int>& s, vector<int>& vis,
                              vector<vector<int>>& adj)
{  vis[i] = 1;
   for(auto j : adj[i])
      if(!vis[j[0]])
       topo_dfs(j, s, vis, adj);
   s.push(i);
}
main —   vector {dist(V,∞) , vis(V,0)} , dist[0]=0;
for(int i=0;  i<V; i++)
{     if(!vis[i])
      topo_dfs(i, s, vis, adj);
 }
while(!s.empty())
{   int node = s.top();
    s.pop();
   if(dist[node] != INT_MAX)  (X)
   {   for(auto j : adj[node])
       {  if(dist[node] + j[1]  < dist[j[0]])
          dist[j[0]] = dist[node] + j[1];
       }
   }
}
  return dist;
```

assuming, adj[u][0] = u

adj[u][1] = wt

$$\begin{tikzcd} u \arrow[r, "wt"] & v \end{tikzcd}$$

Daaris ameen



(9) Dijkstra Algo

{dist, node} ← priority_queue is used
↑ min heap.

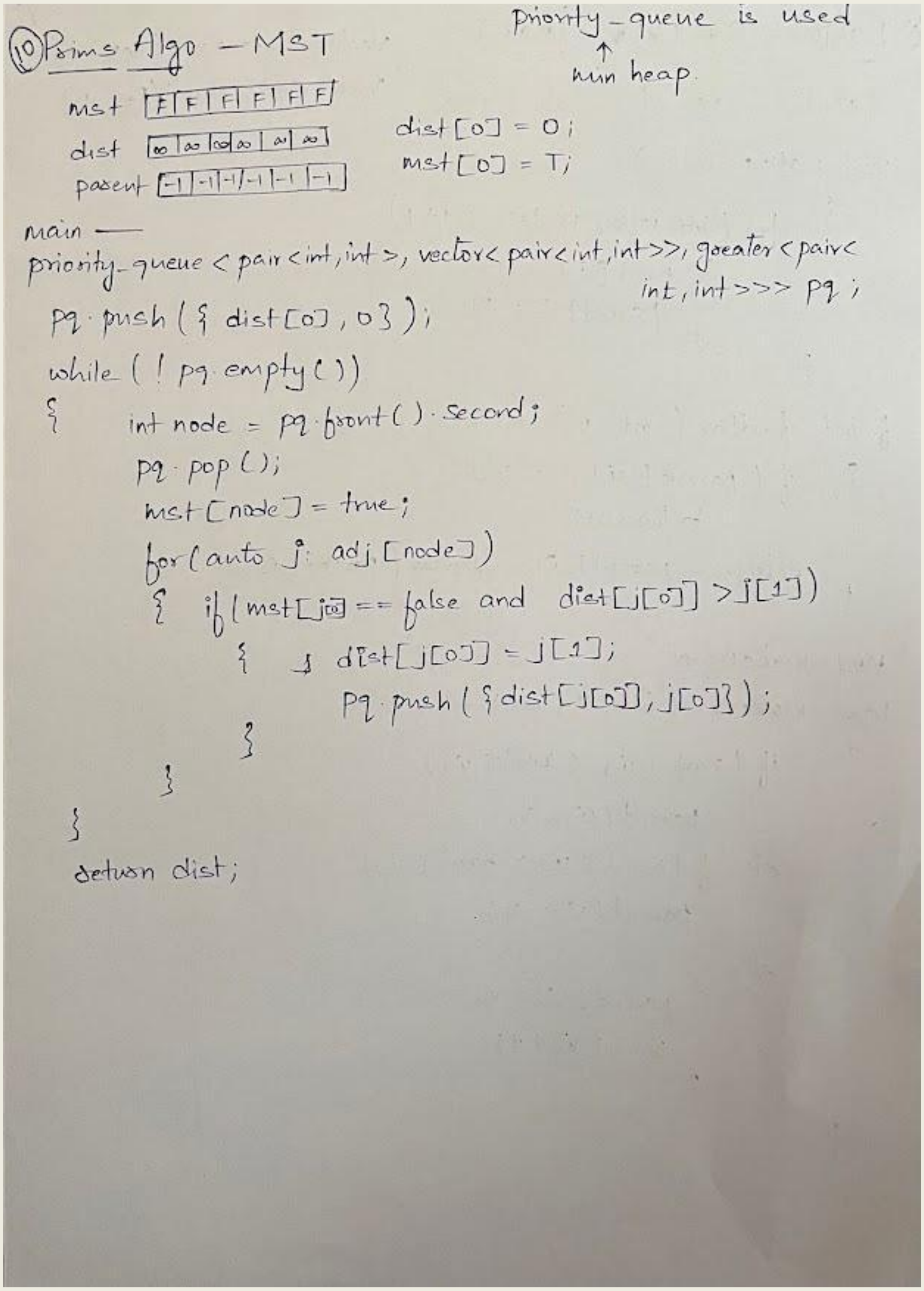
```
dist (V, ∞)
dist [0] = 0 ;
```

adj[u][0] = v, adj[u][1] = wt

main —

```
priority_queue < pair<int, int> , vector <pair<int, int>>,
                     greater < pair < int, int>>> pq;
pq.push ( { dist[0], 0} );
while ( ! pq.empty())
{    int disi = pq.front (). first;
     int node = pq.front () . second;
     pq.pop();
    for (auto j : adj[node])
    { if(disi + j[1]  < dist [j[0]])
      { dist[j[0]] = disi + j[1];
         pq.push ( { dist[j[0]] , j[0]});
      }
    }
}
return dist ;
```

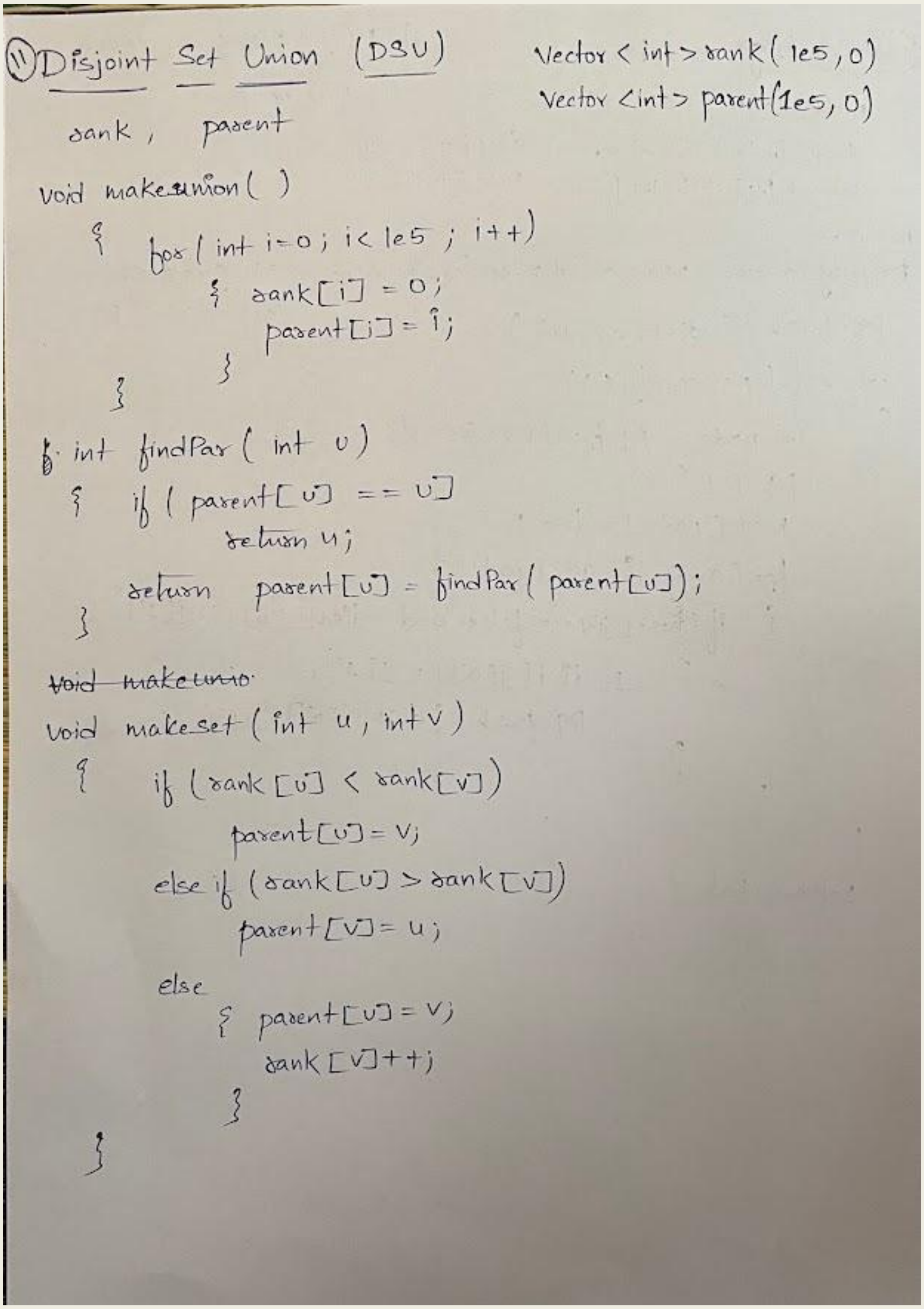
Daaris ameen



(10) Prims Algo — MST

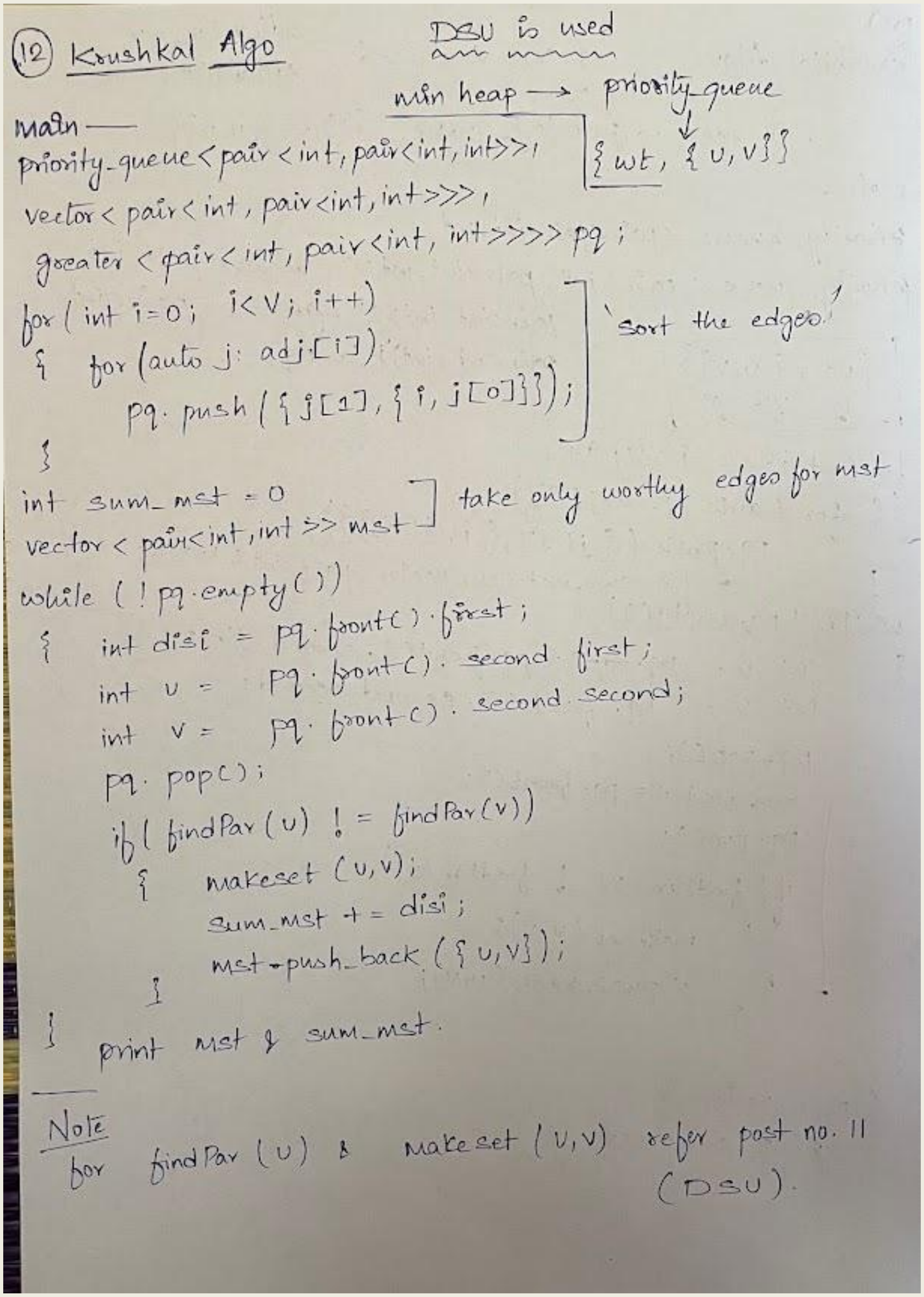
priority_queue is used
↑
min heap.

```
mst    | F | F | F | F | F | F |
dist   | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
parent | -1 | -1 | -1 | -1 | -1 | -1 |

dist[0] = 0;
mst[0] = T;

main —
priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<
                                              int,int>>> pq;
 pq.push({dist[0], 0});
 while(!pq.empty())
 {    int node = pq.front().second;
      pq.pop();
      mst[node] = true;
      for(auto j: adj[node])
      {   if(mst[j[0]] == false and  dist[j[0]] > j[1])
             {    dist[j[0]] = j[1];
                  pq.push({dist[j[0]], j[0]});
             }
      }
 }
 return dist;
```



11) Disjoint Set Union (DSU)

```
Vector < int > rank( 1e5, 0)
Vector <int> parent(1e5, 0)

rank ,   parent

void ~~make~~union( )
  {   for ( int i=0 ; i< 1e5 ; i++)
         {  rank[i] = 0;
            parent[i] = i;
         }
  }

~~f~~ int findPar ( int u )
  {   if ( parent[u] == u]
            return u;
      return  parent[u] = findPar( parent[u]);
  }

~~void makeunio~~
void makeset ( int u , int v )
  {     if (rank [u] < rank[v])
             parent[u] = v;
        else if (rank[u] > rank[v])
             parent[v] = u;
        else
           {  parent[u] = v;
              rank [v]++;
           }
  }
```

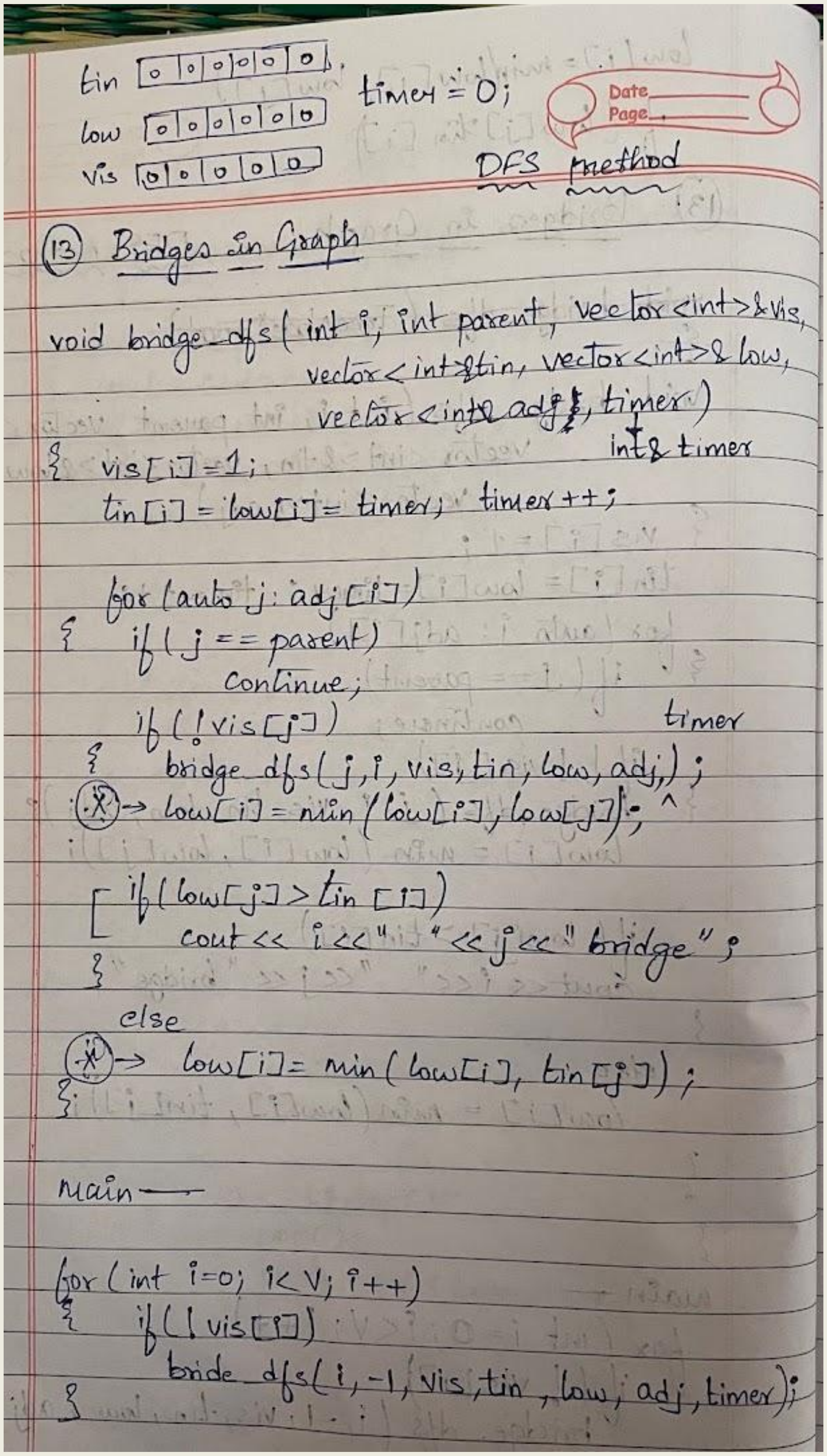


(12) Krushkal Algo   DSU is used

min heap → priority_queue → {wt, {u,v}}

```
main ——
priority_queue<pair<int, pair<int,int>>,
vector<pair<int, pair<int,int>>>,
greater<pair<int, pair<int, int>>>> pq;

for ( int i=0;  i<V; i++)                         'sort the edges'
{    for (auto j: adj[i])
        pq.push({j[1], {i, j[0]}});
}

int sum_mst = 0                                    take only worthy edges for mst
vector<pair<int,int>> mst

while (!pq.empty())
{    int disi = pq.front().first;
     int u = pq.front().second.first;
     int v = pq.front().second.second;
     pq.pop();
     if( findPar(u) != findPar(v))
       {   makeset (u,v);
           sum_mst += disi;
           mst.push_back({u,v});
       }
}    print mst & sum_mst.
```

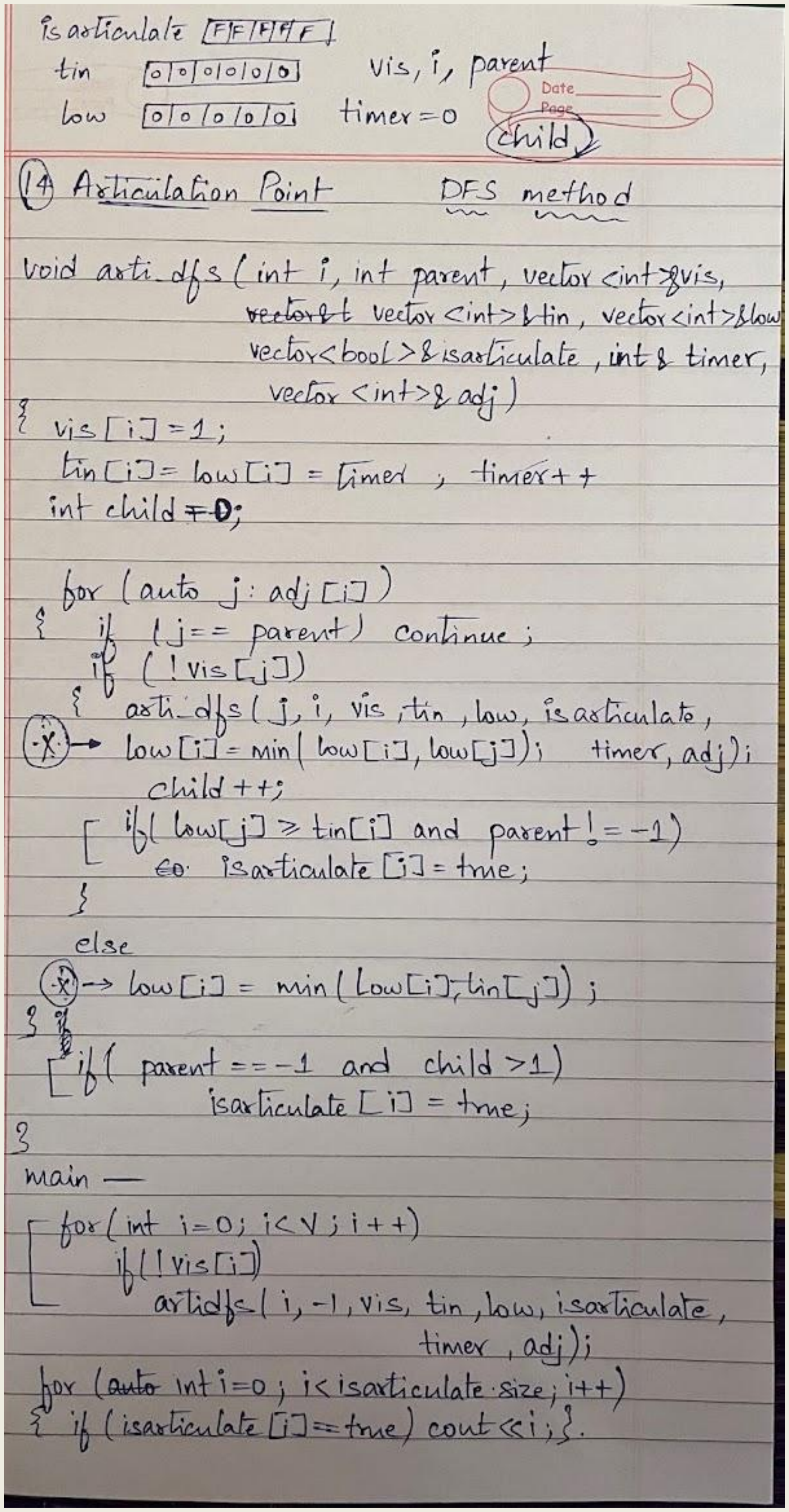
Note for findPar(u) & makeset(u,v) refer post no. 11 (DSU).



```
tin  [0|0|0|0|0|0|0]
low  [0|0|0|0|0|0|0]     timer = 0;
vis  [0|0|0|0|0|0|0]
                              DFS method
```

(13) Bridges in Graph

```
void bridge_dfs( int i, int parent, vector<int>&vis,
                 vector<int>&tin, vector<int>& low,
                 vector<int> adj[], timer)
                                        int& timer
{ vis[i] = 1;
  tin[i] = low[i] = timer;   timer++;

  for (auto j: adj[i])
  {  if (j == parent)
           continue;
     if (!vis[j])                              timer
     {    bridge_dfs( j, i, vis, tin, low, adj, ) ;
  (X)->   low[i] = min (low[i], low[j]);

     [ if (low[j] > tin [i])
          cout << i << " " << j << " bridge" ;
     }
       else
  (X)->  low[i] = min ( low[i], tin[j] ) ;
  }


main ——

for (int i=0; i< V; i++)
{     if (! vis[i])
         bride_dfs( i, -1, vis, tin, low, adj, timer);
}
```



isarticulate [F|F|F|F|F]

tin [0|0|0|0|0|0|0]   vis, i, parent

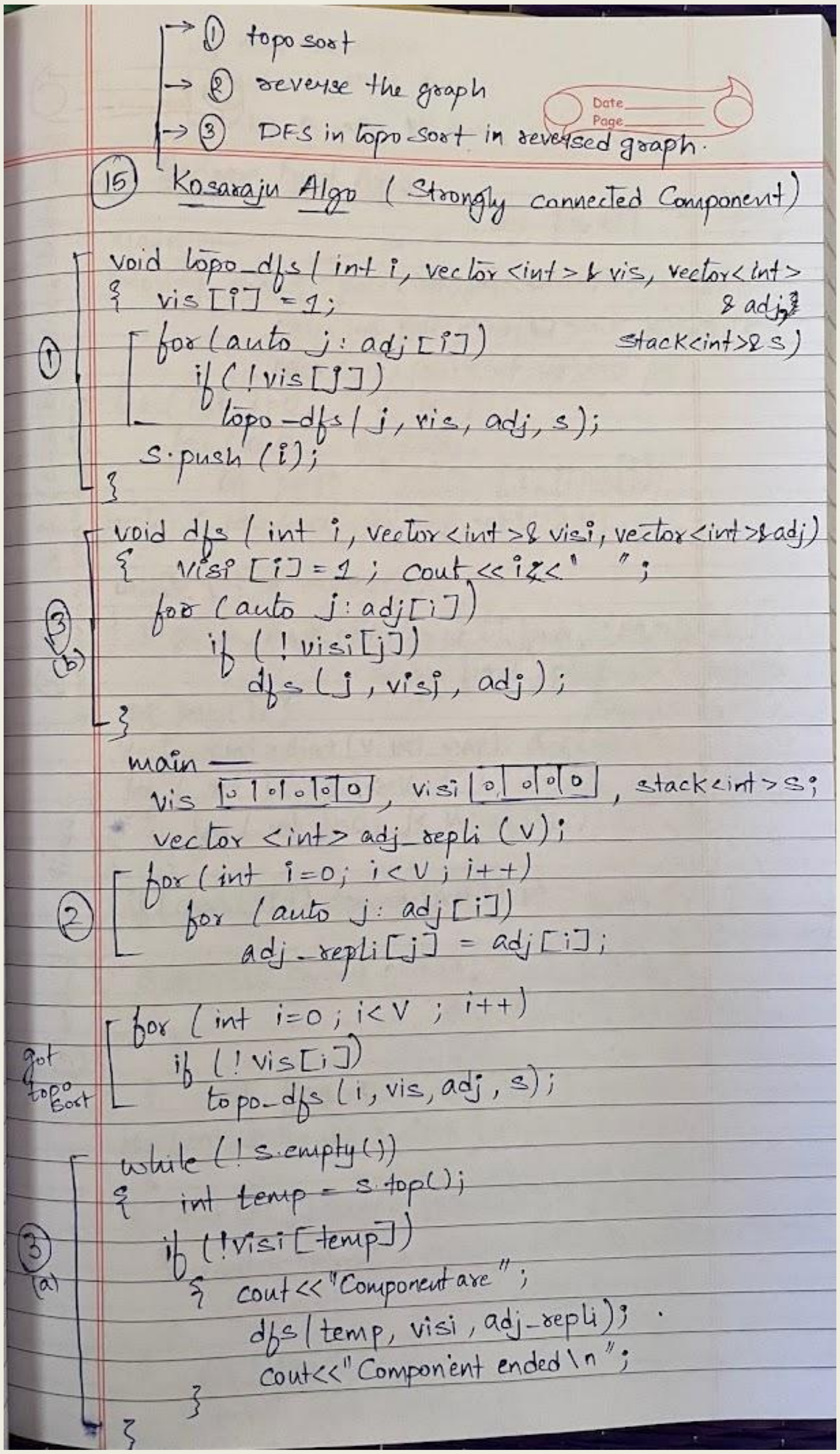
low [0|0|0|0|0|0]   timer = 0

Date_____ Page_____

child

(14) Articulation Point   DFS method

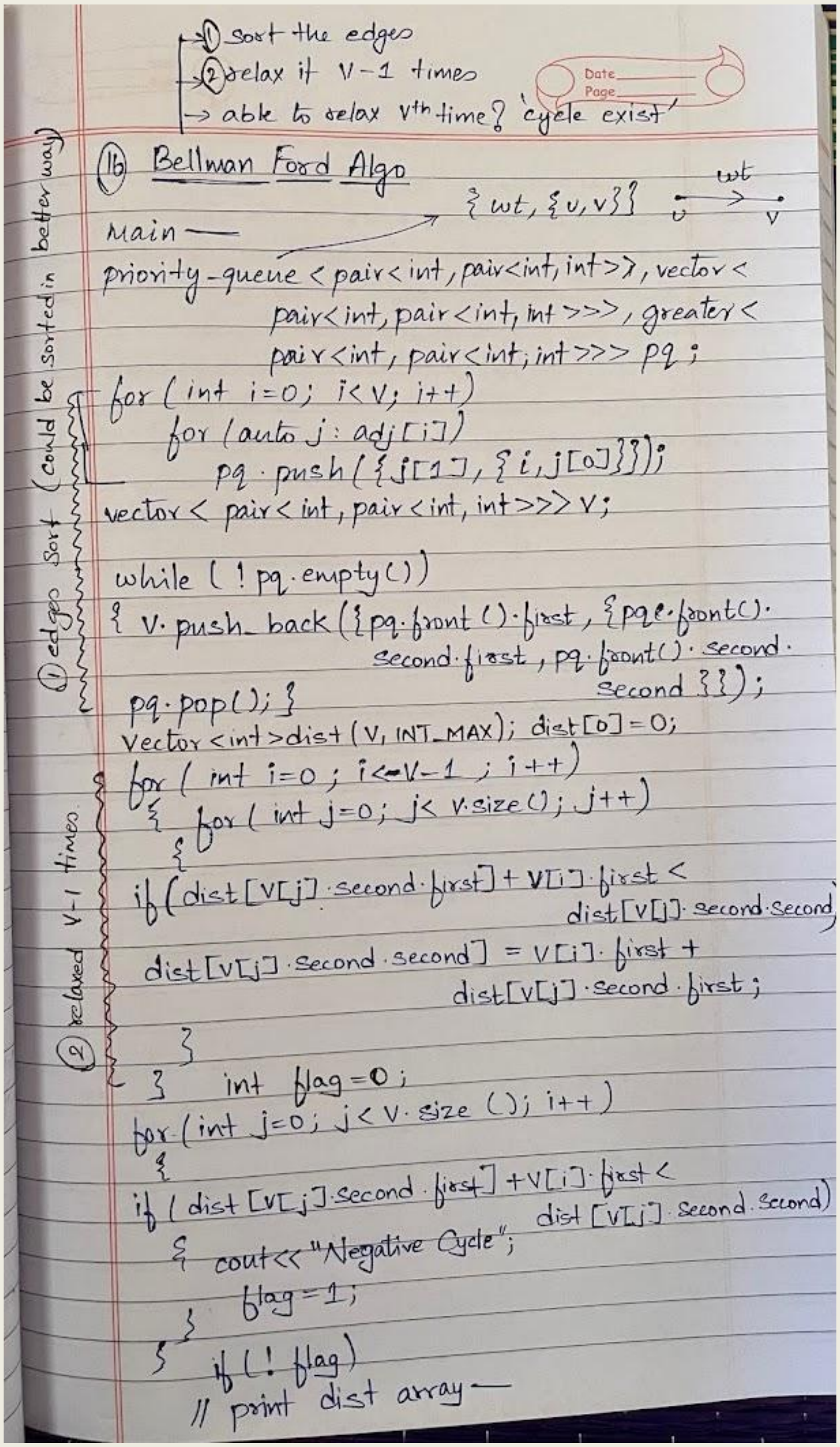
```
void arti_dfs(int i, int parent, vector<int>&vis,
              ~~vector&t~~ vector<int>&tin, vector<int>&low
              vector<bool>&isarticulate, int& timer,
                vector<int>& adj)
{ vis[i] = 1;
  tin[i] = low[i] = timer ; timer++
  int child = 0;

  for (auto j : adj[i])
  {   if (j == parent) continue;
      if (!vis[j])
      { arti_dfs(j, i, vis, tin, low, isarticulate,
(X)→ low[i] = min(low[i], low[j]);     timer, adj);
        child++;
     [ if( low[j] >= tin[i] and parent != -1)
           isarticulate [i] = true;
      }
      else
  (X)→ low[i] = min(Low[i], tin[j]);
  }
 [ if( parent == -1 and child > 1)
            isarticulate [i] = true;
}
main —
[ for(int i=0; i<V; i++)
      if(!vis[i])
         artidfs( i, -1, vis, tin, low, isarticulate,
                              timer, adj);
 for (~~auto~~ int i=0; i<isarticulate.size; i++)
 {  if (isarticulate[i]==true) cout<<i; }.
```



→ ① topo sort
→ ② reverse the graph
→ ③ DFS in topo-sort in reversed graph.

(15) Kosaraju Algo (Strongly connected Component)

```
①   void topo_dfs( int i, vector<int>& vis, vector<int>
                                                  & adj[],
    {  vis[i] = 1;                           stack<int>& s)
       for(auto j : adj[i])
          if(!vis[j])
             topo_dfs(j, vis, adj, s);
       s.push(i);
    }

③   void dfs ( int i, vector<int>& visi, vector<int>& adj)
(b) {   visi[i] = 1;  cout << i << "  ";
        for (auto j : adj[i])
            if (!visi[j])
               dfs(j, visi, adj);
    }

    main —
      vis [0|1|0|0|0|0], visi [0| 0|0|0], stack<int> s;
      vector <int> adj_repli (V);
②     for (int i=0; i<V; i++)
          for (auto j : adj[i])
              adj_repli[j] = adj[i];

got   for ( int i=0; i<V ; i++)
topo     if (!vis[i])
sort       topo_dfs(i, vis, adj, s);

③     while (!s.empty())
(a)   {  int temp = s.top();
         if (!visi[temp])
          {  cout << "Component are";
             dfs(temp, visi, adj_repli);
             cout << "Component ended\n";
          }
      }
```



→ ① sort the edges
→ ② relax it V − 1 times
→ able to relax V^th time? 'cycle exist'

(16) Bellman Ford Algo

{wt, {u, v}} $u \xrightarrow{wt} v$

① edges sort (could be sorted in better way)

② relaxed V-1 times

```
Main —
priority_queue<pair<int,pair<int,int>>, vector<
            pair<int,pair<int,int>>>, greater<
            pair<int,pair<int;int>>> pq;
for (int i=0; i<V; i++)
    for (auto j : adj[i])
        pq.push({j[1], {i, j[0]}});
vector< pair<int,pair<int,int>>> V;

while (!pq.empty())
{ V.push_back({pq.front().first, {pq.front().
            second.first, pq.front().second.
                                 second}});
 pq.pop(); }
vector<int>dist(V, INT_MAX); dist[0]=0;
for ( int i=0 ; i<V-1 ; i++)
 {  for( int j=0; j< V.size(); j++)
   {
 if(dist[V[j].second.first]+V[i].first <
                       dist[V[j].second.second])
  dist[V[j].second.second] = V[i].first +
                    dist[V[j].second.first];
     }
 }    int flag=0;
for(int j=0; j< V.size (); i++)
 {
 if ( dist [V[j].second.first] +V[i].first <
                   dist [V[j].second.second])
   { cout<<"Negative Cycle";
       flag=1;
   }
 }   if(!flag)
   // print dist array —
```

STACKS & QUEUES

Nearest Greater to Left - NGL

```
vector < long long > nextLargerElement(vector < long long > arr, int
n)
{
      // Your code here
    stack < long long > s;
    vector < long long > ans;
    for (int i = 0; i < n; i++)
    {
        if (s.size() == 0)
            ans.push_back(-1);

        if (s.size() > 0 and s.top() >= arr[i])
            ans.push_back(s.top());

        else if (s.size() > 0 and s.top() < arr[i])
        {
            while (s.size() > 0 and s.top() < arr[i])
            {
                s.pop();
            }
            if (s.size() == 0)
                ans.push_back(-1);
            else
                ans.push_back(s.top());
        }
        s.push(arr[i]);
    }

    return ans;
}
```

Nearest Greater to Right - NGR

```
vector < long long > nextLargerElement(vector < long long > arr, int
n)
{
      // Your code here
    stack < long long > s;
    vector < long long > ans;
    for (int i = n - 1; i >= 0; i--)
    {
        if (s.size() == 0)
            ans.push_back(-1);

        if (s.size() > 0 and s.top() >= arr[i])
            ans.push_back(s.top());

        else if (s.size() > 0 and s.top() < arr[i])
        {
            while (s.size() > 0 and s.top() < arr[i])
            {
                s.pop();
            }
            if (s.size() == 0)
                ans.push_back(-1);
            else
                ans.push_back(s.top());
        }
        s.push(arr[i]);
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

Nearest Smaller to Left - NSL

```
vector<int> nearest_smallestonleft(int arr[], int n)
{
     // Complete the function
    stack<int> s;
    vector<int> ans;
    for (int i = 0; i < n; i++)
    {
        if (s.size() == 0)
            ans.push_back(-1);

        if (s.size() > 0 and s.top() <= arr[i])
            ans.push_back(s.top());

        else if (s.size() > 0 and s.top() > arr[i])
        {
            while (s.size() > 0 and s.top() > arr[i])
            {
                s.pop();
            }
            if (s.size() == 0)
                ans.push_back(-1);
            else
                ans.push_back(s.top());
        }
        s.push(arr[i]);
    }
    return ans;
}
```

Nearest Smaller to Right - NSR

```
vector<int> nearest_smallestonright(int arr[], int n)
{
     // Complete the function
```

```
    stack<int> s;
    vector<int> ans;
    for (int i = n - 1; i >= 0; i--)
    {
        if (s.size() == 0)
            ans.push_back(-1);

        if (s.size() > 0 and s.top() <= arr[i])
            ans.push_back(s.top());

        else if (s.size() > 0 and s.top() > arr[i])
        {
            while (s.size() > 0 and s.top() > arr[i])
            {
                s.pop();
            }
            if (s.size() == 0)
                ans.push_back(-1);
            else
                ans.push_back(s.top());
        }
        s.push(arr[i]);
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

Stock Span Problem

```
vector<int> calculateSpan(int price[], int n)
{
      // Your code here
    stack<pair<int, int>> s;

    vector<int> ans;
    for (int i = 0; i < n; i++)
    {
        if (s.size() == 0)
            ans.push_back(i + 1);
        else if (s.size() > 0 and s.top().first > price[i])
            ans.push_back(i - s.top().second);
        else if (s.size() > 0 and s.top().first <= price[i])
        {
            while (s.size() > 0 and s.top().first <= price[i])
            {
                s.pop();
            }
            if (s.size() == 0)
                ans.push_back(i + 1);
            else
                ans.push_back(i - s.top().second);
        }
        s.push({ price[i],
            i });
    }
    return ans;
}
```

Maximum Area of Histogram

```
int MaximumAreaHistogram(vector<int> &heights)
        {
            vector<int> ansr;
            vector<int> ansl;
            stack<pair<int, int>> sr;
            stack<pair<int, int>> sl;
            int n = heights.size();
            for (int i = 0; i < n; i++)
            {
                if (sl.size() == 0)
                    ansl.push_back(-1);
                if (sl.size() > 0 and sl.top().first < heights[i])
                    ansl.push_back(sl.top().second);
                else if (sl.size() > 0 and sl.top().first >=
heights[i])
                {
                    while (sl.size() > 0 and sl.top().first >=
heights[i])
                        sl.pop();
                    if (sl.size() == 0)
                        ansl.push_back(-1);
                    else
                        ansl.push_back(sl.top().second);
                }
                sl.push({ heights[i],
                    i });
            }

            for (int i = n - 1; i >= 0; i--)
            {
                if (sr.size() == 0)
                    ansr.push_back(n);
                if (sr.size() > 0 and sr.top().first < heights[i])
                    ansr.push_back(sr.top().second);
                else if (sr.size() > 0 and sr.top().first >=
heights[i])
                {
                    while (sr.size() > 0 and sr.top().first >=
heights[i])
                        sr.pop();
                    if (sr.size() == 0)
                        ansr.push_back(n);
                    else
                        ansr.push_back(sr.top().second);
                }
                sr.push({ heights[i],
                    i });
            }
            reverse(ansr.begin(), ansr.end());

            int area = INT_MIN;
            for (int i = 0; i < n; i++)
            {
                area = max(area, heights[i] *((i - ansl[i]) + (ansr[i]
- i) - 1));
            }
            return area;
        }
```

Maximum area of rectangle in Binary Matrix

```
        int max_rectangle_histogram(vector<int> height)
        {
            int n = height.size();
            vector<int> ansl, ansr;
            stack<pair<int, int>> sl, sr;

            for (int i = 0; i < n; i++)
            {
                if (sl.size() == 0)
                    ansl.push_back(-1);
                if (sl.size() > 0 and sl.top().first < height[i])
                    ansl.push_back(sl.top().second);
                else if (sl.size() > 0 and sl.top().first >=
height[i])
                {
                    while (sl.size() > 0 and sl.top().first >=
height[i])
                    {
                        sl.pop();
                    }
                    if (sl.size() == 0)
                        ansl.push_back(-1);
                    else
                        ansl.push_back(sl.top().second);
                }
                sl.push({ height[i],
                    i });
            }

            for (int i = n - 1; i >= 0; i--)
            {
                if (sr.size() == 0)
```

```
                    ansr.push_back(n);
                if (sr.size() > 0 and sr.top().first < height[i])
                    ansr.push_back(sr.top().second);
                else if (sr.size() > 0 and sr.top().first >=
height[i])
                {
                    while (sr.size() > 0 and sr.top().first >=
height[i])
                    {
                        sr.pop();
                    }
                    if (sr.size() == 0)
                        ansr.push_back(n);
                    else
                        ansr.push_back(sr.top().second);
                }
                sr.push({ height[i],
                    i });
            }
            reverse(ansr.begin(), ansr.end());

            int res = 0;
            for (int i = 0; i < n; i++)
            {
                res = max(res, height[i] *((i - ansl[i]) + (ansr[i] -
i) - 1));
            }

            return res;
        }
    int maximalRectangle(vector<vector < char>> &matrix)
    {
        int m = matrix.size(), n = matrix[0].size();
        vector<int> histogram(n, 0);
        int result = 0;
        for (int i = 0; i < matrix.size(); i++)
        {
            for (int j = 0; j < matrix[0].size(); j++)
            {
                if (matrix[i][j] == '1')
                    histogram[j] += 1;
                else
                    histogram[j] = 0;
            }
            result = max(result, max_rectangle_histogram(histogram));
        }
        return result;
    }
```

Rain Water Trapping

```
int trap(vector<int> &height)
        {
            vector<int> ansl;
            vector<int> ansr;
            stack<int> sl;
            stack<int> sr;
            int n = height.size();

            vector<int> res;

            for (int i = 0; i < n; i++)
            {
                if (sl.size() == 0)
                    ansl.push_back(height[i]);
                if (sl.size() > 0 and sl.top() > height[i])
                    ansl.push_back(sl.top());
                else if (sl.size() > 0 and sl.top() <= height[i])
                {
                    while (sl.size() > 0 and sl.top() <= height[i])
                    {
                        sl.pop();
                    }
                    if (sl.size() == 0)
                        ansl.push_back(height[i]);
                    else
                        ansl.push_back(sl.top());
                }
                if (sl.size() > 0 and sl.top() > height[i])
                    continue;
                else
                    sl.push(height[i]);
            }

            for (int i = n - 1; i >= 0; i--)
            {
                if (sr.size() == 0)
                    ansr.push_back(height[i]);
                if (sr.size() > 0 and sr.top() > height[i])
                    ansr.push_back(sr.top());
                else if (sr.size() > 0 and sr.top() <= height[i])
                {
                    while (sr.size() > 0 and sr.top() <= height[i])
                    {
                        sr.pop();
                    }
                    if (sr.size() == 0)
                        ansr.push_back(height[i]);
                    else
```

```
                        ansr.push_back(sr.top());
                }
                if (sr.size() > 0 and sr.top() > height[i])
                    continue;
                else
                    sr.push(height[i]);
            }
            reverse(ansr.begin(), ansr.end());

            int ans = 0;

            for (int i = 0; i < n; i++)
            {
                ans += min((ansl[i] - height[i]), (ansr[i] -
height[i]));
            }
            return ans;
        }
```

Implementing a Min_Stack - with Extra Space

```
class MinStack
{
    public:
        stack<int> s;
    stack<int> ss;

    MinStack()
    {}

    void push(int val)
    {
        s.push(val);

        if (ss.size() == 0)
        {
            ss.push(val);
        }
        else if (ss.top() >= val)
        {
            ss.push(val);
        }
    }

    void pop()
    {
        if (ss.top() == s.top())
            ss.pop();
        s.pop();
    }
    int top()
    {
        int a = s.top();
          // cout<<a;
        return a;
    }

    int getMin()
    {
        int a = ss.top();
        return a;
    }
};
```

Implementing a Min_Stack - without Extra Space

```
class MinStack {
public:
    stack<pair<int,int>>s;

    MinStack() {}

    void push(int val) {
        if(s.size()>0 and val<=s.top().second)
        {
            s.push({val,val});
        }
        else if(s.size()==0)
        {
            s.push({val,val});
        }
        else
        {
            s.push({val,s.top().second});
        }
    }

    void pop() {
        s.pop();
    }

    int top() {
        return s.top().first;
    }

    int getMin() {
        return s.top().second;
    }
};
```

Implementing Stack using Heap

```
// C++ program to implement a stack using
// Priority queue(min heap)
#include<bits/stdc++.h>
using namespace std;

typedef pair<int, int> pi;

// User defined stack class
class Stack{

      // cnt is used to keep track of the number of
      //elements in the stack and also serves as key
      //for the priority queue.
      int cnt;
      priority_queue<pair<int, int> > pq;
public:
      Stack():cnt(0){}
      void push(int n);
      void pop();
      int top();
      bool isEmpty();
};

// push function increases cnt by 1 and
// inserts this cnt with the original value.
void Stack::push(int n){
      cnt++;
      pq.push(pi(cnt, n));
}

// pops element and reduces count.
void Stack::pop(){
      if(pq.empty()){ cout<<"Nothing to pop!!!";}
      cnt--;
      pq.pop();
}

// returns the top element in the stack using
// cnt as key to determine top(highest priority),
// default comparator for pairs works fine in this case
int Stack::top(){
      pi temp=pq.top();
      return temp.second;
}

// return true if stack is empty
bool Stack::isEmpty(){
      return pq.empty();
}

// Driver code
int main()
{
	Stack* s=new Stack();
	s->push(1);
	s->push(2);
	s->push(3);
	while(!s->isEmpty()){
		cout<<s->top()<<endl;
		s->pop();
	}
}
```

Celebrity Problem

A celebrity is a person who is known to all but does not know anyone at a party. If you go to a party of N people, find if there is a celebrity in the party or not.

A square NxN matrix M[][] is used to represent people at the party such that if an element of row i and column j is set to 1 it means ith person knows jth person. Here M[i][i] will always be 0.

Note: Follow 0 based indexing.

```
// C++ program to find celebrity - GRAPH APPROACH
#include <bits/stdc++.h>
#include <list>
using namespace std;

// Max # of persons in the party
#define N 8

// Person with 2 is celebrity
bool MATRIX[N][N] = {{0, 0, 1, 0},
                     {0, 0, 1, 0},
                     {0, 0, 0, 0},
                     {0, 0, 1, 0}};

bool knows(int a, int b)
{
	return MATRIX[a][b];
}

// Returns -1 if celebrity
// is not present. If present,
// returns id (value from 0 to n-1).
int findCelebrity(int n)
{
	//the graph needs not be constructed
	//as the edges can be found by
	//using knows function

	//degree array;
	int indegree[n]={0},outdegree[n]={0};

	//query for all edges
	for(int i=0; i<n; i++)
	{
		for(int j=0; j<n; j++)
		{
			int x = knows(i,j);

			//set the degrees
			outdegree[i]+=x;
			indegree[j]+=x;
		}
	}

	//find a person with indegree n-1
	//and out degree 0
	for(int i=0; i<n; i++)
	if(indegree[i] == n-1 && outdegree[i] == 0)
		return i;

	return -1;
}

// Driver code
int main()
{
	int n = 4;
	int id = findCelebrity(n);
	id == -1 ? cout << "No celebrity" :
			cout << "Celebrity ID " << id;
	return 0;
}
```

Time Complexity: O(n2).
Space Complexity: O(n).

```
// C++ program to find celebrity - USING RECURSION
#include <bits/stdc++.h>
#include <list>
using namespace std;

// Max # of persons in the party
#define N 8
```

```
// Person with 2 is celebrity
bool MATRIX[N][N] = { { 0, 0, 1, 0 },
                      { 0, 0, 1, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 1, 0 } };

bool knows(int a, int b) { return MATRIX[a][b]; }

// Returns -1 if a 'potential celebrity'
// is not present. If present,
// returns id (value from 0 to n-1).
int findPotentialCelebrity(int n)
{
     // base case - when n reaches 0 , returns -1
     // since n represents the number of people,
     // 0 people implies no celebrity(= -1)
     if (n == 0)
          return -1;

     // find the celebrity with n-1
     // persons
     int id = findPotentialCelebrity(n - 1);

     // if there are no celebrities
     if (id == -1)
          return n - 1;

     // if the id knows the nth person
     // then the id cannot be a celebrity, but nth person
     // could be one
     else if (knows(id, n - 1)) {
          return n - 1;
     }
     // if the nth person knows the id,
     // then the nth person cannot be a celebrity and the id
     // could be one
     else if (knows(n - 1, id)) {
          return id;
     }

     // if there is no celebrity
     return -1;
}

// Returns -1 if celebrity
// is not present. If present,
// returns id (value from 0 to n-1).
// a wrapper over findCelebrity
int Celebrity(int n)
```

```
{
      // find the celebrity
      int id = findPotentialCelebrity(n);

      // check if the celebrity found
      // is really the celebrity
      if (id == -1)
            return id;
      else {
            int c1 = 0, c2 = 0;

            // check the id is really the
            // celebrity
            for (int i = 0; i < n; i++)
                  if (i != id) {
                        c1 += knows(id, i);
                        c2 += knows(i, id);
                  }

            // if the person is known to
            // everyone.
            if (c1 == 0 && c2 == n - 1)
                  return id;

            return -1;
      }
}

// Driver code
int main()
{
      int n = 4;
      int id = Celebrity(n);
      id == -1 ? cout << "No celebrity"
                  : cout << "Celebrity ID " << id;
      return 0;
}

Time Complexity: O(n).
Space Complexity: O(1).
```

Longest Valid Paranthesis

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int n = s.length(), longest = 0;
        stack<int> st;
        for (int i = 0; i < n; i++) {
            if (s[i] == '(') st.push(i);
            else {
                if (!st.empty()) {
                    if (s[st.top()] == '(') st.pop();
                    else st.push(i);
                }
                else st.push(i);
            }
        }
        if (st.empty()) longest = n;
        else {
            int a = n, b = 0;
            while (!st.empty()) {
                b = st.top(); st.pop();
                longest = max(longest, a-b-1);
                a = b;
            }
            longest = max(longest, a);
        }
        return longest;
    }
};
```

Iterative TOH

The tower of Hanoi is a famous puzzle where we have three rods and N disks. The objective of the puzzle is to move the entire stack to another rod. You are given the number of discs N. Initially, these discs are in the rod 1. You need to print all the steps of discs movement so that all the discs reach the 3rd rod. Also, you need to find the total moves.

Note: The discs are arranged such that the top disc is numbered 1 and the bottom-most disc is numbered N. Also, all the discs have different sizes and a bigger disc cannot be put on the top of a smaller disc. Refer the provided link to get a better clarity about the puzzle.

Input:

N = 2

Output:

move disk 1 from rod 1 to rod 2

move disk 2 from rod 1 to rod 3

move disk 1 from rod 2 to rod 3

3

Explanation: For N=2 , steps will be

as follows in the example and total

3 steps will be taken.

```
// C++ recursive function to
// solve tower of hanoi puzzle
#include <bits/stdc++.h>
using namespace std;

void towerOfHanoi(int n, char from_rod,
                                        char to_rod, char aux_rod)
{
        if (n == 0)
        {
                return;
        }
        towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
        cout << "Move disk " << n << " from rod " << from_rod <<
                                                        " to rod " << to_rod <<
endl;
        towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

// Driver code
int main()
{
        int n = 4; // Number of disks
        towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
        return 0;
}

Time complexity: O(2^n)
Space complexity: O(n)

// C++ Program for Iterative Tower of Hanoi
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

// A structure to represent a stack
struct Stack
{
unsigned capacity;
int top;
int *array;
};

// function to create a stack of given capacity.
struct Stack* createStack(unsigned capacity)
{
        struct Stack* stack =
                (struct Stack*) malloc(sizeof(struct Stack));
        stack -> capacity = capacity;
        stack -> top = -1;
```

```
      stack -> array =
            (int*) malloc(stack -> capacity * sizeof(int));
      return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
return (stack->top == stack->capacity - 1);
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
return (stack->top == -1);
}

// Function to add an item to stack. It increases
// top by 1
void push(struct Stack *stack, int item)
{
      if (isFull(stack))
            return;
      stack -> array[++stack -> top] = item;
}

// Function to remove an item from stack. It
// decreases top by 1
int pop(struct Stack* stack)
{
      if (isEmpty(stack))
            return INT_MIN;
      return stack -> array[stack -> top--];
}

//Function to show the movement of disks
void moveDisk(char fromPeg, char toPeg, int disk)
{
      cout <<"Move the disk " << disk <<" from " << fromPeg << " to "<<
toPeg << endl;
}

// Function to implement legal movement between
// two poles
void moveDisksBetweenTwoPoles(struct Stack *src,
                  struct Stack *dest, char s, char d)
{
      int pole1TopDisk = pop(src);
      int pole2TopDisk = pop(dest);
```

```
	// When pole 1 is empty
	if (pole1TopDisk == INT_MIN)
	{
		push(src, pole2TopDisk);
		moveDisk(d, s, pole2TopDisk);
	}

	// When pole2 pole is empty
	else if (pole2TopDisk == INT_MIN)
	{
		push(dest, pole1TopDisk);
		moveDisk(s, d, pole1TopDisk);
	}

	// When top disk of pole1 > top disk of pole2
	else if (pole1TopDisk > pole2TopDisk)
	{
		push(src, pole1TopDisk);
		push(src, pole2TopDisk);
		moveDisk(d, s, pole2TopDisk);
	}

	// When top disk of pole1 < top disk of pole2
	else
	{
		push(dest, pole2TopDisk);
		push(dest, pole1TopDisk);
		moveDisk(s, d, pole1TopDisk);
	}
}

//Function to implement TOH puzzle
void tohIterative(int num_of_disks, struct Stack
                *src, struct Stack *aux,
                struct Stack *dest)
{
	int i, total_num_of_moves;
	char s = 'S', d = 'D', a = 'A';

	//If number of disks is even, then interchange
	//destination pole and auxiliary pole
	if (num_of_disks % 2 == 0)
	{
		char temp = d;
		d = a;
		a = temp;
	}
	total_num_of_moves = pow(2, num_of_disks) - 1;
```

```
       //Larger disks will be pushed first
       for (i = num_of_disks; i >= 1; i--)
             push(src, i);

       for (i = 1; i <= total_num_of_moves; i++)
       {
             if (i % 3 == 1)
             moveDisksBetweenTwoPoles(src, dest, s, d);

             else if (i % 3 == 2)
             moveDisksBetweenTwoPoles(src, aux, s, a);

             else if (i % 3 == 0)
             moveDisksBetweenTwoPoles(aux, dest, a, d);
       }
}

// Driver Program
int main()
{

       // Input: number of disks
       unsigned num_of_disks = 3;

       struct Stack *src, *dest, *aux;

       // Create three stacks of size 'num_of_disks'
       // to hold the disks
       src = createStack(num_of_disks);
       aux = createStack(num_of_disks);
       dest = createStack(num_of_disks);

       tohIterative(num_of_disks, src, aux, dest);
       return 0;
}

Time Complexity: O(n)
Auxiliary Space: O(n)
```

Minimum element in stack - O(1) space

```
stack<long long int>s;

          long long int min = 0;

     void push(int val) {

          if(s.size()==0)
          {
              s.push(val);

              min = val;
          }

          else if(val>=min)
          {
              s.push(val);
          }

          else
          {
              long long int p = 2*(long long)val-min;

              s.push(p);

              min = val;
          }
      }

     void pop() {
          if(s.size()==0) return;

          else
          {
              if(s.top()>=min)
              {
                  s.pop();
              }

              else
              {
                  int ov = min;

                  min = 2*min-s.top();

                  s.pop();
              }
          }
     }

     int top() {
          if(s.size()==0) return -1;

          else
          {
              if(s.top()>=min)
              {
                  return s.top();
              }
              else
              {
                  return min;
              }
          }
      }

      int getMin() {
          return min;
      }
```

Implement Stack using Array

```
#include<bits/stdc++.h>

using namespace std;
class Stack {
  int size;
  int * arr;
  int top;
  public:
    Stack() {
      top = -1;
      size = 1000;
      arr = new int[size];
    }
  void push(int x) {
    top++;
    arr[top] = x;
  }
  int pop() {
    int x = arr[top];
    top--;
    return x;
  }
  int Top() {
    return arr[top];
  }
  int Size() {
    return top + 1;
  }
};
int main() {

  Stack s;
  s.push(6);
  s.push(3);
  s.push(7);
  cout << "Top of stack is before deleting any element " << s.Top() <<
endl;
```

```
  cout << "Size of stack before deleting any element " << s.Size() <<
endl;
  cout << "The element deleted is " << s.pop() << endl;
  cout << "Size of stack after deleting an element " << s.Size() <<
endl;
  cout << "Top of stack after deleting an element " << s.Top() <<
endl;
  return 0;
}
```

Implement Queue Using Array

```
#include<bits/stdc++.h>

using namespace std;
class Queue {
  int * arr;
  int start, end, currSize, maxSize;
  public:
    Queue() {
      arr = new int[16];
      start = -1;
      end = -1;
      currSize = 0;
    }

  Queue(int maxSize) {
    ( * this).maxSize = maxSize;
    arr = new int[maxSize];
    start = -1;
    end = -1;
    currSize = 0;
  }
  void push(int newElement) {
    if (currSize == maxSize) {
      cout << "Queue is full\nExiting..." << endl;
      exit(1);
    }
    if (end == -1) {
      start = 0;
      end = 0;
    } else
      end = (end + 1) % maxSize;
    arr[end] = newElement;
    cout << "The element pushed is " << newElement << endl;
    currSize++;
  }
  int pop() {
    if (start == -1) {
      cout << "Queue Empty\nExiting..." << endl;
    }
    int popped = arr[start];
    if (currSize == 1) {
      start = -1;
      end = -1;
    } else
      start = (start + 1) % maxSize;
    currSize--;
    return popped;
  }
  int top() {
    if (start == -1) {
      cout << "Queue is Empty" << endl;
      exit(1);
    }
    return arr[start];
  }
  int size() {
    return currSize;
  }

};

int main() {
  Queue q(6);
  q.push(4);
  q.push(14);
  q.push(24);
  q.push(34);
  cout << "The peek of the queue before deleting any element " <<
q.top() << endl;
  cout << "The size of the queue before deletion " << q.size() <<
endl;
  cout << "The first element to be deleted " << q.pop() << endl;
  cout << "The peek of the queue after deleting an element " <<
q.top() << endl;
  cout << "The size of the queue after deleting an element " <<
q.size() << endl;

  return 0;
}
```

Implement Stack using single Queue

```
#include<bits/stdc++.h>

using namespace std;

class Stack {
  queue < int > q;
  public:
    void Push(int x) {
      int s = q.size();
      q.push(x);
      for (int i = 0; i < s; i++) {

        q.push(q.front());
        q.pop();
      }
    }
  int Pop() {
    int n = q.front();
    q.pop();
    return n;
  }
  int Top() {
    return q.front();
  }
  int Size() {
    return q.size();
  }
};

int main() {
  Stack s;
  s.Push(3);
  s.Push(2);
  s.Push(4);
  s.Push(1);
  cout << "Top of the stack: " << s.Top() << endl;
  cout << "Size of the stack before removing element: " << s.Size() <<
endl;
  cout << "The deleted element is: " << s.Pop() << endl;
  cout << "Top of the stack after removing element: " << s.Top() <<
endl;
  cout << "Size of the stack after removing element: " << s.Size();

}
```

Implement Queue using Stack

```
#include <bits/stdc++.h>

using namespace std;

struct Queue {
  stack < int > input, output;

  // Push elements in queue
  void Push(int data) {
    // Pop out all elements from the stack input
    while (!input.empty()) {
      output.push(input.top());
      input.pop();
    }
    // Insert the desired element in the stack input
    cout << "The element pushed is " << data << endl;
    input.push(data);
    // Pop out elements from the stack output and push them into the
stack input
    while (!output.empty()) {
      input.push(output.top());
      output.pop();
    }
  }
  // Pop the element from the Queue
  int Pop() {
    if (input.empty()) {
      cout << "Stack is empty";
      exit(0);
    }
    int val = input.top();
    input.pop();
    return val;
  }
  // Return the Topmost element from the Queue
  int Top() {
    if (input.empty()) {
      cout << "Stack is empty";
      exit(0);
    }
    return input.top();
  }
  // Return the size of the Queue
  int size() {
    return input.size();
  }
};
int main() {
  Queue q;
  q.Push(3);
  q.Push(4);
  cout << "The element poped is " << q.Pop() << endl;
  q.Push(5);
  cout << "The top of the queue is " << q.Top() << endl;
  cout << "The size of the queue is " << q.size() << endl;
}

Time Complexity: O(N)
Space Complexity: O(2N)

#include <bits/stdc++.h>
```

```
using namespace std;

class MyQueue {
  public:
    stack < int > input, output;
  /** Initialize your data structure here. */
  MyQueue() {

  }

  /** Push element x to the back of queue. */
  void push(int x) {
    cout << "The element pushed is " << x << endl;
    input.push(x);
  }

  /** Removes the element from in front of queue and returns that
element. */
  int pop() {
    // shift input to output
    if (output.empty())
      while (input.size())
        output.push(input.top()), input.pop();

    int x = output.top();
    output.pop();
    return x;
  }

  /** Get the front element. */
  int top() {
    // shift input to output
    if (output.empty())
      while (input.size())
        output.push(input.top()), input.pop();
    return output.top();
  }

  int size() {
    return (output.size() + input.size());
  }

};
int main() {
  MyQueue q;
  q.push(3);
  q.push(4);
  cout << "The element poped is " << q.pop() << endl;
  q.push(5);
```

```
   cout << "The top of the queue is " << q.top() << endl;
   cout << "The size of the queue is " << q.size() << endl;

}

Time Complexity: O(1)
Space Complexity: O(2N)
```

Check for Balanced Parentheses

```
#include<bits/stdc++.h>
using namespace std;
bool isValid(string s) {
        stack<char>st;
        for(auto it: s) {
            if(it=='(' || it=='{' || it == '[') st.push(it);
            else {
                if(st.size() == 0) return false;
                char ch = st.top();
                st.pop();
                if((it == ')' and ch == '(') or  (it == ']' and ch ==
'[') or (it == '}' and ch == '{')) continue;
                else return false;
            }
        }
        return st.empty();
    }
int main()
{
    string s="()[{}()]";
    if(isValid(s))
    cout<<"True"<<endl;
    else
    cout<<"False"<<endl;
}
```

Sort a Stack

```
// C++ program to sort a stack using recursion
#include <iostream>
using namespace std;

// Stack is represented using linked list
struct stack {
     int data;
     struct stack* next;
};

// Utility function to initialize stack
void initStack(struct stack** s) { *s = NULL; }
```

```
// Utility function to check if stack is empty
int isEmpty(struct stack* s)
{
	if (s == NULL)
		return 1;
	return 0;
}

// Utility function to push an item to stack
void push(struct stack** s, int x)
{
	struct stack* p = (struct stack*)malloc(sizeof(*p));

	if (p == NULL) {
		fprintf(stderr, "Memory allocation failed.\n");
		return;
	}

	p->data = x;
	p->next = *s;
	*s = p;
}

// Utility function to remove an item from stack
int pop(struct stack** s)
{
	int x;
	struct stack* temp;

	x = (*s)->data;
	temp = *s;
	(*s) = (*s)->next;
	free(temp);

	return x;
}

// Function to find top item
int top(struct stack* s) { return (s->data); }

// Recursive function to insert an item x in sorted way
void sortedInsert(struct stack** s, int x)
{
	// Base case: Either stack is empty or newly inserted
	// item is greater than top (more than all existing)
	if (isEmpty(*s) or x > top(*s)) {
		push(s, x);
		return;
	}
	// If top is greater, remove the top item and recur
	int temp = pop(s);
	sortedInsert(s, x);

	// Put back the top item removed earlier
	push(s, temp);
}

// Function to sort stack
void sortStack(struct stack** s)
{
	// If stack is not empty
	if (!isEmpty(*s)) {
		// Remove the top item
		int x = pop(s);

		// Sort remaining stack
		sortStack(s);

		// Push the top item back in sorted stack
		sortedInsert(s, x);
	}
}

// Utility function to print contents of stack
void printStack(struct stack* s)
{
	while (s) {
		cout << s->data << " ";
		s = s->next;
	}
	cout << "\n";
}

// Driver code
int main(void)
{
	struct stack* top;

	initStack(&top);
	push(&top, 30);
	push(&top, -5);
	push(&top, 18);
	push(&top, 14);
	push(&top, -3);

	cout << "Stack elements before sorting:\n";
	printStack(top);
```

```
      sortStack(&top);
      cout << "\n";

      cout << "Stack elements after sorting:\n";
      printStack(top);

      return 0;
}
```

```
Time Complexity: O(n2)
Auxiliary Space: O(N)
```

Implement LRU Cache

Problem Statement: “Design a data structure that follows the constraints of Least Recently Used (LRU) cache”.

Implement the LRUCache class:

1. LRUCache(int capacity) we need to initialize the LRU cache with positive size capacity.
2. int get(int key) returns the value of the key if the key exists, otherwise return -1.
3. Void put(int key,int value), Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache.if the number of keys exceeds the capacity from this operation, evict the least recently used key.

The functions get and put must each run in O(1) average time complexity.

```
class LRUCache {
  public:
    class node {
      public:
        int key;
      int val;
      node * next;
      node * prev;
      node(int _key, int _val) {
        key = _key;
        val = _val;
      }
    };

  node * head = new node(-1, -1);
  node * tail = new node(-1, -1);

  int cap;
  unordered_map < int, node * > m;

  LRUCache(int capacity) {
    cap = capacity;
    head -> next = tail;
    tail -> prev = head;
  }

  void addnode(node * newnode) {
    node * temp = head -> next;
    newnode -> next = temp;
    newnode -> prev = head;
    head -> next = newnode;
    temp -> prev = newnode;
  }

  void deletenode(node * delnode) {
    node * delprev = delnode -> prev;
    node * delnext = delnode -> next;
    delprev -> next = delnext;
    delnext -> prev = delprev;
  }

  int get(int key_) {
    if (m.find(key_) != m.end()) {
      node * resnode = m[key_];
      int res = resnode -> val;
      m.erase(key_);
      deletenode(resnode);
      addnode(resnode);
      m[key_] = head -> next;
      return res;
    }

    return -1;
  }

  void put(int key_, int value) {
    if (m.find(key_) != m.end()) {
      node * existingnode = m[key_];
      m.erase(key_);
      deletenode(existingnode);
    }
    if (m.size() == cap) {
      m.erase(tail -> prev -> key);
      deletenode(tail -> prev);
    }

    addnode(new node(key_, value));
    m[key_] = head -> next;
  }
};
```

LFU Cache

Implement the LFUCache class:

1. LFUCache(int capacity) Initializes the object with the capacity of the data structure.
2. int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
3. void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the least frequently used key before inserting a new item. For this problem, when there is a tie (i.e., two or more keys with the same frequency), the least recently used key would be invalidated.

To determine the least frequently used key, a use counter is maintained for each key in the cache. The key with the smallest use counter is the least frequently used key.

When a key is first inserted into the cache, its use counter is set to 1 (due to the put operation). The use counter for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

```
class LFUCache {
    int cap;
    int size;
    int minFreq;
    unordered_map<int, pair<int, int>> m; //key to {value,freq};
    unordered_map<int, list<int>::iterator> mIter; //key to list
iterator;
    unordered_map<int, list<int>> fm; //freq to key list;
public:
    LFUCache(int capacity) {
        cap=capacity;
        size=0;
    }

    int get(int key) {
        if(m.count(key)==0) return -1;

        fm[m[key].second].erase(mIter[key]);
        m[key].second++;
        fm[m[key].second].push_back(key);
        mIter[key]=--fm[m[key].second].end();

        if(fm[minFreq].size()==0 )
              minFreq++;

        return m[key].first;
    }
    void set(int key, int value) {
        if(cap<=0) return;

        int storedValue=get(key);
        if(storedValue!=-1)
        {
            m[key].first=value;
            return;
        }

        if(size>=cap )
        {
            m.erase( fm[minFreq].front() );
            mIter.erase( fm[minFreq].front() );
            fm[minFreq].pop_front();
            size--;
        }

        m[key]={value, 1};
        fm[1].push_back(key);
        mIter[key]=--fm[1].end();
        minFreq=1;
        size++;
    }
};
```

Rotten Oranges : Min time to rot all oranges : BFS

Problem Statement: You will be given an m x n grid, where each cell has the following values :

1. 2 – represents a rotten orange
2. 1 – represents a Fresh orange
3. 0 – represents an Empty Cell

Every minute, if a Fresh Orange is adjacent to a Rotten Orange in 4-direction (upward, downwards, right, and left) it becomes Rotten.

Return the minimum number of minutes required such that none of the cells has a Fresh Orange. If it's not possible, return -1.

```
#include<bits/stdc++.h>
using namespace std;
    int orangesRotting(vector<vector<int>>& grid) {
        if(grid.empty()) return 0;
        int m = grid.size(), n = grid[0].size(), days = 0, tot = 0,
cnt = 0;
        queue<pair<int, int>> rotten;
        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                if(grid[i][j] != 0) tot++;
                if(grid[i][j] == 2) rotten.push({i, j});
            }
        }

        int dx[4] = {0, 0, 1, -1};
        int dy[4] = {1, -1, 0, 0};

        while(!rotten.empty()){
            int k = rotten.size();
            cnt += k;
            while(k--){
                int x = rotten.front().first, y =
rotten.front().second;
                rotten.pop();
                for(int i = 0; i < 4; ++i){
                    int nx = x + dx[i], ny = y + dy[i];
                    if(nx < 0 || ny < 0 || nx >= m || ny >= n ||
grid[nx][ny] != 1) continue;
                    grid[nx][ny] = 2;
                    rotten.push({nx, ny});
                }
            }
            if(!rotten.empty()) days++;
        }

        return tot == cnt ? days : -1;
    }

    int main()
    {
        vector<vector<int>> v{ {2,1,1} , {1,1,0} , {0,1,1} } ;
        int rotting = orangesRotting(v);
        cout<<"Minimum Number of Minutes Required "<<rotting<<endl;

    }

Time Complexity: O ( n x n ) x 4
Space Complexity: O ( n x n )
```

DYNAMIC PROGRAMMING

0-1 KNAPSACK

knapsack_recursive.cpp

```
#include <bits/stdc++.h>

using namespace std;

int Knapsack(int wt[], int val[], int W, int n) {
      // base case
      if (n == 0 || W == 0)
            return 0;

      // recursive cases
      if (wt[n - 1] <= W) {
            return max(val[n - 1] + Knapsack(wt, val, W - wt[n - 1], n
- 1),
                        Knapsack(wt, val, W, n - 1));
      }
      else if (wt[n - 1] > W)
            return Knapsack(wt, val, W, n - 1);
      else
            return -1; // to avoid warning
}

signed main() {
      int n; cin >> n; // number of items
      int val[n], wt[n]; // values and wts array
      for (int i = 0; i < n; i++)
            cin >> wt[i];
      for (int i = 0; i < n; i++)
            cin >> val[i];
      int W; cin >> W; // Knappsack capacity

      cout << Knapsack(wt, val, W, n) << endl;
      return 0;
}
```

knapsack_memoization.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int D = 1000; // DP - matrix dimension

int t[D][D]; // DP matrix

int Knapsack(int wt[], int val[], int W, int n) {
      // base case
```

```
	if (n == 0 || W == 0)
		return 0;

	// if already calculated
	if (t[n][W] != -1)
		return t[n][W];
	// else calculate
	else {
		if (wt[n - 1] <= W)
			t[n][W] = max(val[n - 1] + Knapsack(wt, val, W - wt[n
- 1], n - 1),
			                         Knapsack(wt, val, W, n - 1));
		else if (wt[n - 1] > W)
			t[n][W] = Knapsack(wt, val, W, n - 1);

		return t[n][W];
	}
}

signed main() {
	int n; cin >> n; // number of items
	int val[n], wt[n]; // values and wts array
	for (int i = 0; i < n; i++)
		cin >> wt[i];
	for (int i = 0; i < n; i++)
		cin >> val[i];
	int W; cin >> W; // capacity

	// matrix initialization
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= W; j++)
			t[i][j] = -1;

	cout << Knapsack(wt, val, W, n) << endl;
	return 0;
}
```

knapsack_top_down_dp.cpp

```
#include <bits/stdc++.h>
using namespace std;

int Knapsack(int wt[], int val[], int W, int n) {
	int t[n + 1][W + 1]; // DP matrix

	for (int i = 0; i <= n; i++) {
		for (int j = 0; j <= W; j++) {
			if (i == 0 || j == 0) // base case
				t[i][j] = 0;
			else if (wt[i - 1] <= j) { // current wt can fit in
bag
                        int val1 = val[i - 1] + t[i - 1][j - wt[i - 1]];
// take current wt
                        int val2 = t[i - 1][j]; // skip current wt
                        t[i][j] = max(val1, val2);
                  }
                  else if (wt[i - 1] > j) // current wt doesn't fit in
bag
                        t[i][j] = t[i - 1][j];
            }
      }

      return t[n][W];
}

signed main() {
      int n; cin >> n; // number of items
      int val[n], wt[n]; // values and wts array
      for (int i = 0; i < n; i++)
            cin >> wt[i];
      for (int i = 0; i < n; i++)
            cin >> val[i];
      int W; cin >> W; // capacity

      cout << Knapsack(wt, val, W, n) << endl;
      return 0;
}
```

subset_sum_problem_dp.cpp

```
// Subset Sum problem

#include <bits/stdc++.h>
using namespace std;

bool isSubsetPoss(int arr[], int n, int sum) {
      bool t[n + 1][sum + 1]; // DP - matrix
      // initialization
      for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= sum; j++) {
                  if (i == 0)
                        t[i][j] = false;
                  if (j == 0)
                        t[i][j] = true;
            }
      }

      for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                  if (arr[i - 1] <= j)
```

```
					t[i][j] = t[i - 1][j - arr[i - 1]] || t[i - 1]
[j];
				else
					t[i][j] = t[i - 1][j];
			}
	}

	return t[n][sum];
}

signed main() {
	int n; cin >> n;
	int arr[n];
	for (int i = 0; i < n; i++)
		cin >> arr[i];
	int sum; cin >> sum;

	isSubsetPoss(arr, n, sum) ? cout << "Yes\n" : cout << "No\n";
	return 0;
}
```

equal_sum_partition_problem.cpp

```
// Equal sum Partition Problem

#include <bits/stdc++.h>
using namespace std;

bool isSubsetPoss(int arr[], int n, int sum) {
	bool t[n + 1][sum + 1]; // DP - matrix
	// initialization
	for (int i = 0; i <= n; i++) {
		for (int j = 0; j <= sum; j++) {
			if (i == 0)
				t[i][j] = false;
			if (j == 0)
				t[i][j] = true;
		}
	}


	for (int i = 1; i <= n; i++) {
		for (int j = 1; j <= sum; j++) {
			if (arr[i - 1] <= j)
				t[i][j] = t[i - 1][j - arr[i - 1]] || t[i - 1]
[j];
			else
				t[i][j] = t[i - 1][j];
		}
	}
	return t[n][sum];
}

bool EqualSumPartitionPossible(int arr[], int n) {
	int sum = 0; // sum of all elements of arr
	for (int i = 0; i < n; i++)
		sum += arr[i];

	if (sum % 2 != 0) // if sum is odd --> not possible to make equal
partitions
		return false;

	return isSubsetPoss(arr, n, sum / 2);
}

signed main() {
	int n; cin >> n;
	int arr[n];
	for (int i = 0; i < n; i++)
		cin >> arr[i];

	EqualSumPartitionPossible(arr, n) ? cout << "YES\n" : cout <<
"NO\n";
	return 0;
}
```

count_of_subsets_with_given_sum.cpp

```
// Count of Subsets with given Sum

#include <bits/stdc++.h>
using namespace std;

int CountSubsets(int arr[], int n, int sum) {
	int t[n + 1][sum + 1]; // DP - matrix
	// initialization
	for (int i = 0; i <= n; i++) {
		for (int j = 0; j <= sum; j++) {
			if (i == 0)
				t[i][j] = 0;
			if (j == 0)
				t[i][j] = 1;
		}
	}

	for (int i = 1; i <= n; i++) {
		for (int j = 1; j <= sum; j++) {
			if (arr[i - 1] <= j)
				t[i][j] = t[i - 1][j - arr[i - 1]] + t[i - 1]
[j];
                    else
                          t[i][j] = t[i - 1][j];
              }
        }

        return t[n][sum];
}

signed main() {
        int n; cin >> n;
        int arr[n];
        for (int i = 0; i < n; i++)
              cin >> arr[i];
        int sum; cin >> sum;

        cout << CountSubsets(arr, n, sum) << endl;
        return 0;
}
```

Min Subset Sum Difference

```
// Min Subset Sum Difference

#include <bits/stdc++.h>
using namespace std;

vector<int> isSubsetPoss(int arr[], int n, int sum) {
        bool t[n + 1][sum + 1]; // DP - matrix
        // initialization
        for (int i = 0; i <= n; i++) {
              for (int j = 0; j <= sum; j++) {
                    if (i == 0)
                          t[i][j] = false;
                    if (j == 0)
                          t[i][j] = true;
              }
        }

        for (int i = 1; i <= n; i++) {
              for (int j = 1; j <= sum; j++) {
                    if (arr[i - 1] <= j)
                          t[i][j] = t[i - 1][j - arr[i - 1]] || t[i - 1]
[j];
                    else
                          t[i][j] = t[i - 1][j];
              }
        }

        vector<int> v; // contains all subset sums possible with n
elements
        for (int j = 0; j <= sum; j++)
              if (t[n][j] == true)
                    v.push_back(j);

        return v;
}

int MinSubsetSumDiff(int arr[], int n) {
        int range = 0;
        for (int i = 0; i < n; i++)
              range += arr[i];

        vector<int> v = isSubsetPoss(arr, n, range);
        int mn = INT_MAX;
        for (int i = 0; i < v.size(); i++)
              mn = min(mn, abs(range - 2 * v[i]));

        return mn;
}

signed main() {
        int n; cin >> n;
        int arr[n];
        for (int i = 0; i < n; i++)
              cin >> arr[i];

        cout << MinSubsetSumDiff(arr, n) << endl;
        return 0;
}
```

count_of_subset_with_given_diff.cpp

```
// Count of Subsets with given Sum

#include <bits/stdc++.h>
using namespace std;

int CountSubsetsWithSum(int arr[], int n, int sum) {
      int t[n + 1][sum + 1]; // DP - matrix
      // initialization
      for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= sum; j++) {
                  if (i == 0)
                        t[i][j] = 0;
                  if (j == 0)
                        t[i][j] = 1;
            }
      }
      for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                  if (arr[i - 1] <= j)
                        t[i][j] = t[i - 1][j - arr[i - 1]] + t[i - 1]
[j];
                  else
                        t[i][j] = t[i - 1][j];
            }
      }

      return t[n][sum];
}

int CountSubsetsWithDiff(int arr[], int n, int diff) {
      int sumOfArray = 0;
      for (int i = 0; i < n; i++)
            sumOfArray += arr[i];

      if ((sumOfArray + diff) % 2 != 0)
            return 0;
      else
            return CountSubsetsWithSum(arr, n, (sumOfArray + diff) /
2);
}

signed main() {
      int n; cin >> n;
      int arr[n];
      for (int i = 0; i < n; i++)
            cin >> arr[i];
      int diff; cin >> diff;

      cout << CountSubsetsWithDiff(arr, n, diff) << endl;
      return 0;
}
```

target sum

```
// Target Sum

#include <bits/stdc++.h>
using namespace std;

int CountSubsetsWithSum(int arr[], int n, int sum) {
      int t[n + 1][sum + 1]; // DP - matrix
      // initialization
      t[0][0] = 1;
      int k = 1;
      for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= sum; j++) {
                if (i == 0 && j > 0)
                    t[i][j] = 0;
                if (j == 0 && i > 0) {
                    if (arr[i - 1] == 0) {
                        t[i][j] = pow(2, k);
                        k++;
                    }
                    else
                        t[i][j] = t[i - 1][j];
                }
            }
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                if (arr[i - 1] <= j)
                    t[i][j] = t[i - 1][j - arr[i - 1]] + t[i - 1]
[j];
                else
                    t[i][j] = t[i - 1][j];
            }
        }

        return t[n][sum];
}

int TargetSum(int arr[], int n, int diff) {
        int sumOfArray = 0;
        for (int i = 0; i < n; i++)
            sumOfArray += arr[i];

        if ((sumOfArray + diff) % 2 != 0)
            return 0;
        else
            return CountSubsetsWithSum(arr, n, (sumOfArray + diff) /
2);
}

signed main() {
        int n; cin >> n;
        int arr[n];
        for (int i = 0; i < n; i++)
            cin >> arr[i];
        int sum; cin >> sum;

        cout << TargetSum(arr, n, sum) << endl;
        return 0;
}
```

UNBOUNDED KNAPSACK

rod_cutting_problem.cpp

```
// Rod Cutting Problem

#include <bits/stdc++.h>
using namespace std;

int getMaxProfit(int length[], int price[], int n, int L) {
	int dp[n + 1][L + 1];
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= L; j++)
			if (j == 0 || i == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++) {
		for (int j = 1; j <= L; j++) {
			if (length[i - 1] <= j) {
				dp[i][j] = max(dp[i - 1][j],
				               price[i - 1] + dp[i][j - length[i
- 1]]);
			}
			else
				dp[i][j] = dp[i - 1][j];
		}
	}

	return dp[n][L];
}

signed main() {
	int n; cin >> n;
	int length[n], price[n];
	for (int i = 0; i < n; i++)
		cin >> length[i];
	for (int i = 0; i < n; i++)
		cin >> price[i];
	int L; cin >> L;

	cout << getMaxProfit(length, price, n, L) << endl;
	return 0;
}
```

coin_change_max_ways.cpp

```
// Unbounded Knapsack -> Coin Change - I (max number of ways)

#include <bits/stdc++.h>
using namespace std;

int getMaxNumberOfWays(int coins[], int n, int sum) {
	int t[n + 1][sum + 1];
	// initialization
	for (int i = 0; i <= n; i++) {
		for (int j = 0; j <= sum; j++) {
			if (i == 0)
				t[i][j] = 0;
			if (j == 0)
				t[i][j] = 1;
		}
	}

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= sum; j++)
			if (coins[i - 1] <= j)
				t[i][j] = t[i - 1][j] + t[i][j - coins[i - 1]];
			else
				t[i][j] = t[i - 1][j];

	return t[n][sum];
}

signed main() {
	int n; cin >> n;
	int coins[n];
	for (int i = 0; i < n; i++)
		cin >> coins[i];
	int sum; cin >> sum;

	cout << getMaxNumberOfWays(coins, n, sum) << endl;
	return 0;
}
```

coin_change_min_coins.cpp

```
// Unbounded Knapsack -> Coin Change - II (min number of coins)

#include <bits/stdc++.h>
using namespace std;
#define INF INT_MAX-1

int getMinNumberOfCoins(int coins[], int n, int sum) {
	int t[n + 1][sum + 1];
	// initialization
	for (int i = 0; i <= n; i++) {
		for (int j = 0; j <= sum; j++) {
			if (j == 0)
				t[i][j] = 0;
			if (i == 0)
				t[i][j] = INF;
			if (i == 1) {
```

```
                    if (j % coins[i - 1] == 0)
                        t[i][j] = j / coins[i - 1];
                    else
                        t[i][j] = INF;
                }
            }
        }

        t[0][0] = 0;

        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= sum; j++)
                if (coins[i - 1] <= j)
                    t[i][j] = min(t[i - 1][j], 1 + t[i][j - coins[i
- 1]]);
                else
                    t[i][j] = t[i - 1][j];

        return t[n][sum];
}

signed main() {
        int n; cin >> n;
        int coins[n];
        for (int i = 0; i < n; i++)
            cin >> coins[i];
        int sum; cin >> sum;

        cout << getMinNumberOfCoins(coins, n, sum) << endl;
        return 0;
}
```

LCS - LONGEST COMMON SUBSEQUENCE

LCS_recursive.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
        // base case
        if (n == 0 || m == 0)
            return 0;

        // when last character is same
        if (X[n - 1] == Y[m - 1])
            return 1 + LCS(X, Y, n - 1, m - 1);
        // when last character is not same -> pick max
        else
            return max(LCS(X, Y, n - 1, m), LCS(X, Y, n, m - 1));
}

signed main() {
	string X, Y; cin >> X >> Y;
	int n = X.length(), m = Y.length();

	cout << LCS(X, Y, n, m) << endl;
	return 0;
}
```

LCS_memoization.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int D = 1001;
int dp[D][D];

int LCS(string X, string Y, int n, int m) {
	// base case
	if (n == 0 || m == 0)
		dp[n][m] = 0;

	if (dp[n][m] != -1)
		return dp[n][m];

	// when last character is same
	if (X[n - 1] == Y[m - 1])
		dp[n][m] = 1 + LCS(X, Y, n - 1, m - 1);
	// when last character is not same -> pick max
	else
		dp[n][m] = max(LCS(X, Y, n - 1, m), LCS(X, Y, n, m - 1));

	return dp[n][m];
}

signed main() {
	string X, Y; cin >> X >> Y;
	int n = X.length(), m = Y.length();

	memset(dp, -1, sizeof(dp));

	cout << LCS(X, Y, n, m) << endl;
	return 0;
}
```

LCS_bottom_up_dp.cpp

```
#include <bits/stdc++.h>
using namespace std;
```

```
int LCS(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	// base case of recursion --> for initialization of dp - matrix
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1]) // when last character is 
same
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else // when last character is not same -> pick max
				dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

	return dp[n][m];
}

signed main() {
	string X, Y; cin >> X >> Y;
	int n = X.length(), m = Y.length();

	cout << LCS(X, Y, n, m) << endl;
	return 0;
}
```

LCSubstring.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCSubstr(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1])
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else
				dp[i][j] = 0;

	int mx = INT_MIN;
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			mx = max(mx, dp[i][j]);

	return mx;
}

signed main() {
	string X, Y; cin >> X >> Y;
	int n = X.length(), m = Y.length();

	cout << LCSubstr(X, Y, n, m) << endl;
	return 0;
}
```

print_LCS.cpp

```
#include <bits/stdc++.h>
using namespace std;

string LCS(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	// base case of recursion --> for initialization of dp - matrix
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1]) // when last character is
same
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else // when last character is not same -> pick max
				dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

	int i = n, j = m;
	string lcs = "";
	while (i > 0 && j > 0) {
		if (X[i - 1] == Y[j - 1]) {
			lcs += X[i - 1];
			i--, j--;
		}
		else {
			if (dp[i][j - 1] > dp[i - 1][j])
				j--;
			else
				i--;
		}
	}
	reverse(lcs.begin(), lcs.end());

	return lcs;
}

signed main() {
	string X, Y; cin >> X >> Y;
	int n = X.length(), m = Y.length();

	cout << LCS(X, Y, n, m) << endl;
	return 0;
}
```

SCS.cpp

```
// Shortest Common Supersequence
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	// base case of recursion --> for initialization of dp - matrix
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1]) // when last character is
same
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else // when last character is not same -> pick max
				dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

	return dp[n][m];
}

int SCS(string X, string Y, int n, int m) {
	return m + n - LCS(X, Y, n, m);
}

signed main() {
	string X, Y; cin >> X >> Y;
	int n = X.length(), m = Y.length();

	cout << SCS(X, Y, n, m) << endl;
	return 0;
}
```

min_insertion_del_to_convert_a_to_b.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	// base case of recursion --> for initialization of dp - matrix
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1]) // when last character is
same
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else // when last character is not same -> pick max
				dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

	return dp[n][m];
}

int MinInsertDel(string X, string Y, int n, int m) {
	int lcs_len = LCS(X, Y, n, m);
	return m + n - 2 * lcs_len;
}

signed main() {
	string X, Y; cin >> X >> Y;
	int n = X.length(), m = Y.length();

	cout << MinInsertDel(X, Y, n, m) << endl;
	return 0;
}
```

longest_pallin_subseq.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	// base case of recursion --> for initialization of dp - matrix
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1]) // when last character is
same
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else // when last character is not same -> pick max
				dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

	return dp[n][m];
}

int LPS(string X, int n) {
	string rev_X = X;
	reverse(rev_X.begin(), rev_X.end());
	return LCS(X, rev_X, n, n);
}

signed main() {
	string X, Y; cin >> X;
	int n = X.length();

	cout << LPS(X, n) << endl;
	return 0;
}
```

min_del_to_make_pallindrome.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	// base case of recursion --> for initialization of dp - matrix
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1]) // when last character is
same
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else // when last character is not same -> pick max
```

```
                        dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

      return dp[n][m];
}

int LPS(string X, int n) {
      string rev_X = X;
      reverse(rev_X.begin(), rev_X.end());
      return LCS(X, rev_X, n, n);
}

int MinDelForPallindrome(string X, int n) {
      return n - LPS(X, n);
}

signed main() {
      string X, Y; cin >> X;
      int n = X.length();

      cout << MinDelForPallindrome(X, n) << endl;
      return 0;
}
```

print_SCS.cpp

```
// Shortest Common Supersequence
#include <bits/stdc++.h>
using namespace std;

string SCS(string X, string Y, int n, int m) {
      int dp[n + 1][m + 1]; // DP - matrix

      // base case of recursion --> for initialization of dp - matrix
      for (int i = 0; i <= n; i++)
            for (int j = 0; j <= m; j++)
                  if (i == 0 || j == 0)
                        dp[i][j] = 0;

      for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m; j++)
                  if (X[i - 1] == Y[j - 1]) // when last character is
same
                        dp[i][j] = 1 + dp[i - 1][j - 1];
                  else // when last character is not same -> pick max
                        dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

      int i = n, j = m;
      string scs = "";
      while (i > 0 && j > 0) {
            if (X[i - 1] == Y[j - 1]) {
                scs += X[i - 1];
                i--, j--;
            }
            else if (dp[i][j - 1] > dp[i - 1][j]) {
                scs += Y[j - 1];
                j--;
            }
            else {
                scs += X[i - 1];
                i--;
            }
        }

        while (i > 0) {
            scs += X[i - 1];
            i--;
        }

        while (j > 0) {
            scs += Y[j - 1];
            j--;
        }

        reverse(scs.begin(), scs.end());

        return scs;
}

signed main() {
        string X, Y; cin >> X >> Y;
        int n = X.length(), m = Y.length();

        cout << SCS(X, Y, n, m) << endl;
        return 0;
}
```

longest_repeating_subseq.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
        int dp[n + 1][m + 1]; // DP - matrix

        for (int i = 0; i <= n; i++)
            for (int j = 0; j <= m; j++)
                if (i == 0 || j == 0)
                    dp[i][j] = 0;

        for (int i = 1; i <= n; i++)
```

```
            for (int j = 1; j <= m; j++)
                if (X[i - 1] == Y[j - 1] && i != j)
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                else
                    dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

        return dp[n][m];
}

signed main() {
        string X; cin >> X;
        int n = X.length();

        cout << LCS(X, X, n, n) << endl;
        return 0;
}
```

sequence_patttern_matching.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
        int dp[n + 1][m + 1]; // DP - matrix

        // base case of recursion --> for initialization of dp - matrix
        for (int i = 0; i <= n; i++)
            for (int j = 0; j <= m; j++)
                if (i == 0 || j == 0)
                    dp[i][j] = 0;

        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m; j++)
                if (X[i - 1] == Y[j - 1])
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                else
                    dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

        return dp[n][m];
}

bool SeqPatternMatching(string X, string Y, int n, int m) {
        return LCS(X, Y, n, m) == min(n, m);
}

signed main() {
        string X, Y; cin >> X >> Y;
        int n = X.length(), m = Y.length();

        SeqPatternMatching(X, Y, n, m) ? "YES\n" : "NO\n";
```

```
	return 0;
}
```

min_insertion_to_make_string_pallindrome.cpp

```
#include <bits/stdc++.h>
using namespace std;

int LCS(string X, string Y, int n, int m) {
	int dp[n + 1][m + 1]; // DP - matrix

	// base case of recursion --> for initialization of dp - matrix
	for (int i = 0; i <= n; i++)
		for (int j = 0; j <= m; j++)
			if (i == 0 || j == 0)
				dp[i][j] = 0;

	for (int i = 1; i <= n; i++)
		for (int j = 1; j <= m; j++)
			if (X[i - 1] == Y[j - 1]) // when last character is
same
				dp[i][j] = 1 + dp[i - 1][j - 1];
			else // when last character is not same -> pick max
				dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

	return dp[n][m];
}

int LPS(string X, int n) {
	string rev_X = X;
	reverse(rev_X.begin(), rev_X.end());
	return LCS(X, rev_X, n, n);
}

int MinInsertForPallindrome(string X, int n) {
	return n - LPS(X, n);
}

signed main() {
	string X, Y; cin >> X;
	int n = X.length();

	cout << MinInsertForPallindrome(X, n) << endl;
	return 0;
}
```

MATRIX CHAIN MULTIPLICATION - MCM

MCM_recursive.cpp

```
#include <bits/stdc++.h>
using namespace std;

int Solve(int arr[], int i, int j) {
	if (i >= j)
		return 0;

	int ans = INT_MAX;
	for (int k = i; k <= j - 1; k++) {
		int temp_ans = Solve(arr, i, k) + Solve(arr, k + 1, j) +
arr[i - 1] * arr[k] * arr[j];
		ans = min(ans, temp_ans);
	}

	return ans;
}

signed main() {
	int n; cin >> n;
	int arr[n];
	for (int i = 0; i < n; i++)
		cin >> arr[i];

	cout << Solve(arr, 1, n - 1) << endl;
	return 0;
}
```

MCM_memoization.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int D = 1000;
int t[D][D];

int Solve(int arr[], int i, int j) {
	if (i >= j) {
		t[i][j] = 0;
		return 0;
	}

	if (t[i][j] != -1)
		return t[i][j];

	int ans = INT_MAX;
	for (int k = i; k <= j - 1; k++) {
		int temp_ans = Solve(arr, i, k) + Solve(arr, k + 1, j) +
arr[i - 1] * arr[k] * arr[j];
		ans = min(ans, temp_ans);
	}
	return t[i][j] = ans;
}

signed main() {
	int n; cin >> n;
	int arr[n];
	for (int i = 0; i < n; i++)
		cin >> arr[i];

	memset(t, -1, sizeof(t));

	cout << Solve(arr, 1, n - 1) << endl;
	return 0;
}
```

pallindrome_partitioning_recursive.cpp

```
#include <bits/stdc++.h>
using namespace std;

bool isPallindrome(string X, int i, int j) {
	while (i <= j) {
		if (X[i] != X[j])
			return false;
		i++, j--;
	}

	return true;
}

int Solve(string X, int i, int j) {
	if (i >= j || isPallindrome(X, i, j))
		return 0;

	int ans = INT_MAX;
	for (int k = i; k < j; k++) {
		int temp_ans = Solve(X, i, k) + Solve(X, k + 1, j) + 1;
		ans = min(ans, temp_ans);
	}

	return ans;
}

int main() {
	string X; cin >> X;

	cout << Solve(X, 0, X.length() - 1) << endl;
```

```
    return 0;
}
```

pallindrome_partitioning_memoization.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int D = 1001;
int t[D][D];

bool isPallindrome(string X, int i, int j) {
    while (i <= j) {
        if (X[i] != X[j])
            return false;
        i++, j--;
    }

    return true;
}

int Solve(string X, int i, int j) {
    if (i >= j || isPallindrome(X, i, j)) {
        t[i][j] = 0;
        return 0;
    }

    if (t[i][j] != -1)
        return t[i][j];

    int ans = INT_MAX;
    for (int k = i; k < j; k++) {
        int temp_ans = Solve(X, i, k) + Solve(X, k + 1, j) + 1;
        ans = min(ans, temp_ans);
    }

    return t[i][j] = ans;
}

int main() {
    string X; cin >> X;

    memset(t, -1, sizeof(t));

    cout << Solve(X, 0, X.length() - 1) << endl;
    return 0;
}
```

pallindrome_partitioning_memoized_optimization.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

const int D = 1001;
int t[D][D];

bool isPallindrome(string X, int i, int j) {
	while (i <= j) {
		if (X[i] != X[j])
			return false;
		i++, j--;
	}

	return true;
}

int Solve(string X, int i, int j) {
	if (t[i][j] != -1)
		return t[i][j];

	if (i >= j || isPallindrome(X, i, j)) {
		t[i][j] = 0;
		return 0;
	}

	int ans = INT_MAX;
	for (int k = i; k < j; k++) {
		int left, right;
		if (t[i][k] == -1)
			left = Solve(X, i, k);
		else
			left = t[i][k];

		if (t[k + 1][j] == -1)
			right = Solve(X, k + 1, j);
		else
			right = t[k + 1][j];

		int temp_ans = left + right + 1;
		ans = min(ans, temp_ans);
	}

	return t[i][j] = ans;
}

int main() {
	string X; cin >> X;

	memset(t, -1, sizeof(t));
	cout << Solve(X, 0, X.length() - 1) << endl;
	return 0;
}
```

evaluate_expression_to_true.cpp

```
#include <bits/stdc++.h>
using namespace std;

int Solve(string X, int i, int j, bool isTrue) {
	if (i >= j) {
		if (isTrue)
			return X[i] == 'T';
		else
			return X[i] == 'F';
	}

	int ans = 0;
	for (int k = i + 1; k < j; k += 2) {
		int l_T = Solve(X, i, k - 1, true);
		int l_F = Solve(X, i, k - 1, false);
		int r_T = Solve(X, k + 1, j, true);
		int r_F = Solve(X, k + 1, j, false);

		if (X[k] == '|') {
			if (isTrue == true)
				ans += l_T * r_T + l_T * r_F + l_F * r_T;
			else
				ans += l_F * r_F;
		}
		else if (X[k] == '&') {
			if (isTrue == true)
				ans += l_T * r_T;
			else
				ans += l_T * r_F + l_F * r_T + l_F * r_F;
		}
		else if (X[k] == '^') {
			if (isTrue == true)
				ans += l_T * r_F + l_F * r_T;
			else
				ans += l_T * r_T + l_F * r_F;
		}

	}

	return ans;
}

signed main() {
      string X; cin >> X;
      cout << Solve(X, 0, X.length() - 1, true) << endl;
      return 0;
}
```

evaluate_expression_to_true_memoization_using_map.cpp

```
#include <bits/stdc++.h>
using namespace std;

unordered_map<string, int> ump;

int Solve(string X, int i, int j, bool isTrue) {
      string key = to_string(i) + " " + to_string(j) + " " + (isTrue ?
"T" : "F");

      if (ump.find(key) != ump.end())
            return ump[key];

      if (i >= j) {
            if (isTrue)
                  ump[key] = X[i] == 'T';
            else
                  ump[key] = X[i] == 'F';
            return ump[key];
      }

      int ans = 0;
      for (int k = i + 1; k < j; k += 2) {
            int l_T = Solve(X, i, k - 1, true);
            int l_F = Solve(X, i, k - 1, false);
            int r_T = Solve(X, k + 1, j, true);
            int r_F = Solve(X, k + 1, j, false);

            if (X[k] == '|') {
                  if (isTrue == true)
                        ans += l_T * r_T + l_T * r_F + l_F * r_T;
                  else
                        ans += l_F * r_F;
            }
            else if (X[k] == '&') {
                  if (isTrue == true)
                        ans += l_T * r_T;
                  else
                        ans += l_T * r_F + l_F * r_T + l_F * r_F;
            }
            else if (X[k] == '^') {
                  if (isTrue == true)
                        ans += l_T * r_F + l_F * r_T;
                  else
```

```
                    ans += l_T * r_T + l_F * r_F;
            }

        }

        return ump[key] = ans;
}

signed main() {
        string X; cin >> X;
        ump.clear();
        cout << Solve(X, 0, X.length() - 1, true) << endl;
        return 0;
}
```

evaluate_expression_to_true_memoization_using_3d_array.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int D = 1001;
int dp[2][D][D];

int Solve(string X, int i, int j, bool isTrue) {
        if (i >= j) {
                if (isTrue)
                        dp[1][i][j] = X[i] == 'T';
                else
                        dp[0][i][j] = X[i] == 'F';
                return dp[isTrue][i][j];
        }

        if (dp[isTrue][i][j] != -1)
                return dp[isTrue][i][j];

        int ans = 0;
        for (int k = i + 1; k < j; k += 2) {
                int l_T = Solve(X, i, k - 1, true);
                int l_F = Solve(X, i, k - 1, false);
                int r_T = Solve(X, k + 1, j, true);
                int r_F = Solve(X, k + 1, j, false);

                if (X[k] == '|') {
                        if (isTrue == true)
                                ans += l_T * r_T + l_T * r_F + l_F * r_T;
                        else
                                ans += l_F * r_F;
                }
                else if (X[k] == '&') {
                        if (isTrue == true)
```

```
                    ans += l_T * r_T;
                else
                    ans += l_T * r_F + l_F * r_T + l_F * r_F;
            }
            else if (X[k] == '^') {
                if (isTrue == true)
                    ans += l_T * r_F + l_F * r_T;
                else
                    ans += l_T * r_T + l_F * r_F;
            }

        }

        dp[isTrue][i][j] = ans;

        return ans;
}

signed main() {
        string X; cin >> X;

        memset(dp[0], -1, sizeof(dp[0]));
        memset(dp[1], -1, sizeof(dp[1]));

        cout << Solve(X, 0, X.length() - 1, true) << endl;
        return 0;
}
```

scramble_strings_recursive.cpp

```
#include <bits/stdc++.h>
using namespace std;

bool Solve(string X, string Y) {
        if (X.compare(Y) == 0)
            return true;
        if (X.length() <= 1)
            return false;

        int n = X.length();
        int flag = false;
        for (int i = 1; i <= n - 1; i++) {
            if ((Solve(X.substr(0, i), Y.substr(n - i, i)) &&
Solve(X.substr(i), Y.substr(0, n - i))) ||
                    (Solve(X.substr(0, i), Y.substr(0, i)) &&
Solve(X.substr(i), Y.substr(i)))) {
                flag = true;
                break;
            }
        }
```

```
	return flag;
}

int main() {
	string X, Y; cin >> X >> Y;

	if (X.length() != Y.length())
		cout << "No\n";
	else
		Solve(X, Y) ? cout << "Yes\n" : cout << "No\n";
	return 0;
}
```

scramble_strings_memoization.cpp

```
#include <bits/stdc++.h>
using namespace std;

unordered_map<string, int> ump;

bool Solve(string X, string Y) {
	string key = X + " " + Y;
	if (ump.find(key) != ump.end())
		return ump[key];

	if (X.compare(Y) == 0) {
		ump[key] = true;
		return true;
	}
	if (X.length() <= 1) {
		ump[key] = false;
		return false;
	}

	int n = X.length();
	int flag = false;
	for (int i = 1; i <= n - 1; i++) {
		if ((Solve(X.substr(0, i), Y.substr(n - i, i)) &&
Solve(X.substr(i), Y.substr(0, n - i))) ||
		         (Solve(X.substr(0, i), Y.substr(0, i)) &&
Solve(X.substr(i), Y.substr(i)))) {
			flag = true;
			break;
		}
	}

	return ump[key] = flag;
}
```

```
int main() {
      string X, Y; cin >> X >> Y;

      ump.clear();

      if (X.length() != Y.length())
            cout << "No\n";
      else
            Solve(X, Y) ? cout << "Yes\n" : cout << "No\n";
      return 0;
}
```

egg_dropping_problem_recursive.cpp

```
#include <bits/stdc++.h>
using namespace std;

int Solve(int eggs, int floors) {
      if (eggs == 1)
            return floors;
      if (floors == 0 || floors == 1)
            return floors;

      int mn = INT_MAX;
      for (int k = 1; k <= floors; k++) {
            int temp_ans = 1 + max(Solve(eggs - 1, k - 1), Solve(eggs,
floors - k));
            mn = min(mn, temp_ans);
      }

      return mn;
}

signed main() {
      int eggs, floors;
      cin >> eggs >> floors;

      cout << Solve(eggs, floors) << endl;
      return 0;
}
```

egg_dropping_problem_memoization.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int D = 101;
int t[D][D];

int Solve(int eggs, int floors) {
     if (t[eggs][floors] != -1)
           return t[eggs][floors];

     if (eggs == 1 || floors == 0 || floors == 1) {
           t[eggs][floors] = floors;
           return floors;
     }

     int mn = INT_MAX;
     for (int k = 1; k <= floors; k++) {
           int temp_ans = 1 + max(Solve(eggs - 1, k - 1), Solve(eggs, 
floors - k));
           mn = min(mn, temp_ans);
     }

     return t[eggs][floors] = mn;
}

signed main() {
     int eggs, floors;
     cin >> eggs >> floors;

     memset(t, -1, sizeof(t));

     cout << Solve(eggs, floors) << endl;
     return 0;
}
```

egg_dropping_problem_memoized_optimization.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int D = 101;
int dp[D][D];

int Solve(int eggs, int floors) {
      if (dp[eggs][floors] != -1)
            return dp[eggs][floors];

      if (eggs == 1 || floors == 0 || floors == 1) {
            dp[eggs][floors] = floors;
            return floors;
      }

      int mn = INT_MAX;
      for (int k = 1; k <= floors; k++) {
            int top, bottom;
            if (dp[eggs - 1][k - 1] != -1)
                  top = dp[eggs - 1][k - 1];
            else {
                  top = Solve(eggs - 1, k - 1);
                  dp[eggs - 1][k - 1] = top;
            }

            if (dp[eggs][floors - k] != -1)
                  bottom = dp[eggs][floors - k];
            else {
                  bottom = Solve(eggs, floors - k);
                  dp[eggs][floors - k] = bottom;
            }
            int temp_ans = 1 + max(top, bottom);
            mn = min(mn, temp_ans);
      }

      return dp[eggs][floors] = mn;
}

signed main() {
      int eggs, floors;
      cin >> eggs >> floors;

      memset(dp, -1, sizeof(dp));

      cout << Solve(eggs, floors) << endl;
      return 0;
}
```

egg_dropping_problem_binary-search.cpp

```
class Solution {
    int dp[107][10007];
public:
    int Solve(int eggs, int floors) {
        if (dp[eggs][floors] != -1)
            return dp[eggs][floors];

        if (eggs == 1 || floors == 0 || floors == 1) {
            dp[eggs][floors] = floors;
            return floors;
        }

        int ans = floors;
        int low = 1, high = floors;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            int bottom = Solve(eggs - 1, mid - 1); // egg broke
            int top = Solve(eggs, floors - mid); // egg doesn't broke
            int temp = 1 + max(top, bottom); // max for worst case
            if (bottom < top) // go upward for worst case
                low = mid + 1;
            else // go downward for worst case
                high = mid - 1;

            ans = min(ans, temp);
        }

        return dp[eggs][floors] = ans;
    }
    int superEggDrop(int K, int N) {
        memset(dp, -1, sizeof(dp));
        return Solve(K, N);
    }
};
```

DYNAMIC PROGRAMING - EXTRAS

Climbing Stairs

Given a number of stairs. Starting from the 0th stair we need to climb to the “Nth” stair. At a time we can climb either one or two steps. We need to return the total number of distinct ways to reach from 0th to Nth stair.

```
#include <bits/stdc++.h>

using  namespace  std;

int main() {

  int n=3;
  vector<int> dp(n+1,-1);

  dp[0]= 1;
  dp[1]= 1;

  for(int i=2; i<=n; i++){
      dp[i] = dp[i-1]+ dp[i-2];
  }
  cout<<dp[n];
  return 0;
}
```

```
#include <bits/stdc++.h>

using  namespace  std;

int main() {

  int n=3;

  int prev2 = 1;
  int prev = 1;

  for(int i=2; i<=n; i++){
      int cur_i = prev2+ prev;
      prev2 = prev;
      prev= cur_i;
  }
  cout<<prev;
  return 0;
}
```

Frog Jump

Given a number of stairs and a frog, the frog wants to climb from the 0th stair to the (N-1)th stair. At a time the frog can climb either one or two steps. A height[N] array is also given. Whenever the frog jumps from a stair i to stair j, the energy consumed in the jump is abs(height[i]- height[j]), where abs() means the absolute difference. We need to return the minimum energy that can be used by the frog to jump from stair 0 to stair N-1.

```
#include <bits/stdc++.h>

using namespace std;

int solve(int ind, vector<int>& height, vector<int>& dp){
    if(ind==0) return 0;
    if(dp[ind]!=-1) return dp[ind];
    int jumpTwo = INT_MAX;
    int jumpOne= solve(ind-1, height,dp)+ abs(height[ind]-height[ind-
1]);
    if(ind>1)
        jumpTwo = solve(ind-2, height,dp)+ abs(height[ind]-height[ind-
2]);

    return dp[ind]=min(jumpOne, jumpTwo);
}


int main() {

  vector<int> height{30,10,60 , 10 , 60 , 50};
  int n=height.size();
  vector<int> dp(n,-1);
  cout<<solve(n-1,height,dp);
}
```

```
#include <bits/stdc++.h>

using  namespace  std;

int main() {

  vector<int> height{30,10,60,10,60,50};
  int n=height.size();
  vector<int> dp(n,-1);
  dp[0]=0;
  for(int ind=1;ind<n;ind++){
      int jumpTwo = INT_MAX;
        int jumpOne= dp[ind-1] + abs(height[ind]-height[ind-1]);
        if(ind>1)
            jumpTwo = dp[ind-2] + abs(height[ind]-height[ind-2]);

        dp[ind]=min(jumpOne, jumpTwo);
  }
  cout<<dp[n-1];
}
```

Frog Jump with k Distances

This is a follow-up question to “Frog Jump” discussed in the previous article. In the previous question, the frog was allowed to jump either one or two steps at a time. In this question, the frog is allowed to jump up to ‘K’ steps at a time. If K=4, the frog can jump 1,2,3, or 4 steps at every index.

```
#include <bits/stdc++.h>

using namespace std;

int solveUtil(int ind, vector<int>& height, vector<int>& dp, int k){
    if(ind==0) return 0;
    if(dp[ind]!=-1) return dp[ind];

    int mmSteps = INT_MAX;

    for(int j=1;j<=k;j++){
        if(ind-j>=0){
      int jump = solveUtil(ind-j, height, dp, k)+ abs(height[ind]-
height[ind-j]);
            mmSteps= min(jump, mmSteps);
        }
    }
    return dp[ind]= mmSteps;

}

int solve(int n, vector<int>& height , int k){
    vector<int> dp(n,-1);
    return solveUtil(n-1, height, dp, k);
}

int main() {

  vector<int> height{30,10,60 , 10 , 60 , 50};
  int n=height.size();
  int k=2;
  vector<int> dp(n,-1);
  cout<<solve(n,height,k);
}
```

```
#include <bits/stdc++.h>

using namespace std;

int solveUtil(int n, vector<int>& height, vector<int>& dp, int k){
    dp[0]=0;
    for(int i=1;i<n;i++){
        int mmSteps = INT_MAX;
        for(int j=1;j<=k;j++){
            if(i-j>=0){
                int jump = dp[i-j]+ abs(height[i]- height[i-j]);
                mmSteps= min(jump, mmSteps);
            }
        }
        dp[i]= mmSteps;
    }
    return dp[n-1];
}

int solve(int n, vector<int>& height , int k){
    vector<int> dp(n,-1);
    return solveUtil(n, height, dp, k);
}

int main() {

  vector<int> height{30,10,60 , 10 , 60 , 50};
  int n=height.size();
  int k=2;
  vector<int> dp(n,-1);
  cout<<solve(n,height,k);
}
```

Maximum sum of non-adjacent elements

Given an array of ‘N’ positive integers, we need to return the maximum sum of the subsequence such that no two elements of the subsequence are adjacent elements in the array.

Note: A subsequence of an array is a list with elements of the array where some elements are deleted (or not deleted at all) and the elements should be in the same order in the subsequence as in the array.

```
#include <bits/stdc++.h>

using namespace std;

int solveUtil(int ind, vector<int>& arr, vector<int>& dp){

    if(dp[ind]!=-1) return dp[ind];

    if(ind==0) return arr[ind];
    if(ind<0)  return 0;

    int pick= arr[ind]+ solveUtil(ind-2, arr,dp);
    int nonPick = 0 + solveUtil(ind-1, arr, dp);
    return dp[ind]=max(pick, nonPick);
}

int solve(int n, vector<int>& arr){
    vector<int> dp(n,-1);
    return solveUtil(n-1, arr, dp);
}


int main() {

  vector<int> arr{2,1,4,9};
  int n=arr.size();
  cout<<solve(n,arr);

}
```

```
#include <bits/stdc++.h>

using namespace std;

int solveUtil(int n, vector<int>& arr, vector<int>& dp){

    dp[0]= arr[0];

    for(int i=1 ;i<n; i++){
        int pick = arr[i];
        if(i>1)
            pick += dp[i-2];
        int nonPick = 0+ dp[i-1];

        dp[i]= max(pick, nonPick);
    }


    return dp[n-1];
}

int solve(int n, vector<int>& arr){
    vector<int> dp(n,-1);
    return solveUtil(n, arr, dp);
}


int main() {

  vector<int> arr{2,1,4,9};
  int n=arr.size();
  cout<<solve(n,arr);
}
```

House Robber

A thief needs to rob money in a street. The houses in the street are arranged in a circular manner. Therefore the first and the last house are adjacent to each other. The security system in the street is such that if adjacent houses are robbed, the police will get notified.

Given an array of integers “Arr” which represents money at each house, we need to return the maximum amount of money that the thief can rob without alerting the police.

```
#include <bits/stdc++.h>

using namespace std;

long long int solve(vector<int>& arr){
    int n = arr.size();
    long long int prev = arr[0];
    long long int prev2 =0;

    for(int i=1; i<n; i++){
        long long int pick = arr[i];
        if(i>1)
            pick += prev2;
        int long long nonPick = 0 + prev;

        long long int cur_i = max(pick, nonPick);
        prev2 = prev;
        prev= cur_i;

    }
    return prev;
}

long long int robStreet(int n, vector<int> &arr){
    vector<int> arr1;
    vector<int> arr2;

    if(n==1)
       return arr[0];

    for(int i=0; i<n; i++){

        if(i!=0) arr1.push_back(arr[i]);
        if(i!=n-1) arr2.push_back(arr[i]);
    }

    long long int ans1 = solve(arr1);
    long long int ans2 = solve(arr2);
    return max(ans1,ans2);
}


int main() {

  vector<int> arr{1,5,1,2,6};
  int n=arr.size();
  cout<<robStreet(n,arr);
}
```

Ninja’s Training

A Ninja has an ‘N’ Day training schedule. He has to perform one of these three activities (Running, Fighting Practice, or Learning New Moves) each day. There are merit points associated with performing an activity each day. The same activity can’t be performed on two consecutive days. We need to find the maximum merit points the ninja can attain in N Days.

We are given a 2D Array POINTS of size ‘N*3’ which tells us the merit point of specific activity on that particular day. Our task is to calculate the maximum number of merit points that the ninja can earn.

```
#include <bits/stdc++.h>

using namespace std;

int f(int day, int last, vector<vector<int>>
&points,vector<vector<int>> &dp) {

  if (dp[day][last] != -1) return dp[day][last];

  if (day == 0) {
    int maxi = 0;
    for (int i = 0; i <= 2; i++) {
      if (i != last)
        maxi = max(maxi, points[0][i]);
    }
    return dp[day][last] = maxi;
  }

  int maxi = 0;
  for (int i = 0; i <= 2; i++) {
    if (i != last) {
      int activity = points[day][i] + f(day - 1, i, points, dp);
      maxi = max(maxi, activity);
    }
  }

  return dp[day][last] = maxi;
}

int ninjaTraining(int n, vector < vector < int > > & points) {

  vector < vector < int > > dp(n, vector < int > (4, -1));
  return f(n - 1, 3, points, dp);
}

int main() {

  vector < vector < int > > points = {{10,40,70},
                                      {20,50,80},
                                      {30,60,90}};

  int n = points.size();
  cout << ninjaTraining(n, points);
}


#include <bits/stdc++.h>

using namespace std;

int ninjaTraining(int n, vector < vector < int > > & points) {

  vector < vector < int > > dp(n, vector < int > (4, 0));

  dp[0][0] = max(points[0][1], points[0][2]);
  dp[0][1] = max(points[0][0], points[0][2]);
  dp[0][2] = max(points[0][0], points[0][1]);
  dp[0][3] = max(points[0][0], max(points[0][1], points[0][2]));

  for (int day = 1; day < n; day++) {
    for (int last = 0; last < 4; last++) {
      dp[day][last] = 0;
      for (int task = 0; task <= 2; task++) {
        if (task != last) {
          int activity = points[day][task] + dp[day - 1][task];
          dp[day][last] = max(dp[day][last], activity);
        }
      }
    }

  }

  return dp[n - 1][3];
```

```
}

int main() {

  vector<vector<int> > points = {{10,40,70},
                                 {20,50,80},
                                 {30,60,90}};
  int n = points.size();
  cout << ninjaTraining(n, points);
}

#include <bits/stdc++.h>

using namespace std;

int ninjaTraining(int n, vector < vector < int > > & points) {

  vector < int > prev(4, 0);

  prev[0] = max(points[0][1], points[0][2]);
  prev[1] = max(points[0][0], points[0][2]);
  prev[2] = max(points[0][0], points[0][1]);
  prev[3] = max(points[0][0], max(points[0][1], points[0][2]));

  for (int day = 1; day < n; day++) {

    vector < int > temp(4, 0);
    for (int last = 0; last < 4; last++) {
      temp[last] = 0;
      for (int task = 0; task <= 2; task++) {
        if (task != last) {
          temp[last] = max(temp[last], points[day][task] +
prev[task]);
        }
      }
    }

    prev = temp;

  }

  return prev[3];
}

int main() {


  vector<vector<int> > points = {{10,40,70},
                                 {20,50,80},
                                                    {30,60,90}};

    int n = points.size();
    cout << ninjaTraining(n, points);
}
```

DP on Grids

Given two values M and N, which represent a matrix[M][N]. We need to find the total unique paths from the top-left cell (matrix[0][0]) to the rightmost cell (matrix[M-1][N-1]).

At any cell we are allowed to move in only two directions:- bottom and right.

```
#include <bits/stdc++.h>

using namespace std;

int countWaysUtil(int i, int j, vector<vector<int> > &dp) {
  if(i==0 && j == 0)
    return 1;
  if(i<0 || j<0)
    return 0;
  if(dp[i][j]!=-1) return dp[i][j];

  int up = countWaysUtil(i-1,j,dp);
  int left = countWaysUtil(i,j-1,dp);

  return dp[i][j] = up+left;

}

int countWays(int m, int n){
    vector<vector<int> > dp(m,vector<int>(n,-1));
    return countWaysUtil(m-1,n-1,dp);

}

int main() {

  int m=3;
  int n=2;

  cout<<countWays(m,n);
}

Time Complexity: O(M*N)
Space Complexity: O((N-1)+(M-1)) + O(M*N)

#include <bits/stdc++.h>
```

```
using namespace std;

int countWaysUtil(int m, int n, vector<vector<int> > &dp) {
  for(int i=0; i<m ;i++){
      for(int j=0; j<n; j++){

          //base condition
          if(i==0 && j==0){
              dp[i][j]=1;
              continue;
          }

          int up=0;
          int left = 0;

          if(i>0)
            up = dp[i-1][j];
          if(j>0)
            left = dp[i][j-1];

          dp[i][j] = up+left;
      }
  }

  return dp[m-1][n-1];


}

int countWays(int m, int n){
    vector<vector<int> > dp(m,vector<int>(n,-1));
    return countWaysUtil(m,n,dp);

}

int main() {

  int m=3;
  int n=2;

  cout<<countWays(m,n);
}

Time Complexity: O(M*N)
Space Complexity: O(M*N)

#include <bits/stdc++.h>

using namespace std;
```

```
int countWays(int m, int n){
    vector<int> prev(n,0);
    for(int i=0; i<m; i++){
        vector<int> temp(n,0);
        for(int j=0; j<n; j++){
            if(i==0 && j==0){
                temp[j]=1;
                continue;
            }

            int up=0;
            int left =0;

            if(i>0)
                up = prev[j];
            if(j>0)
                left = temp[j-1];

            temp[j] = up + left;
        }
        prev = temp;
    }

    return prev[n-1];

}

int main() {

  int m=3;
  int n=2;

  cout<<countWays(m,n);
}

Time Complexity: O(M*N)
Space Complexity: O(N)
```

Grid Unique Paths 2

We are given an "N*M" Maze. The maze contains some obstacles. A cell is 'blockage' in the maze if its value is -1. 0 represents non-blockage. There is no path possible through a blocked cell.

We need to count the total number of unique paths from the top-left corner of the maze to the bottom-right corner. At every cell, we can move either down or towards the right.

```
#include <bits/stdc++.h>

using namespace std;
```

```
int mazeObstaclesUtil(int n, int m, vector<vector<int>>
&maze,vector<vector<int>>
 &dp)
{
  for(int i=0; i<n ;i++){
      for(int j=0; j<m; j++){

          //base conditions
          if(i>0 && j>0 && maze[i][j]==-1){
            dp[i][j]=0;
            continue;
          }
          if(i==0 && j==0){
              dp[i][j]=1;
              continue;
          }

          int up=0;
          int left = 0;

          if(i>0)
            up = dp[i-1][j];
          if(j>0)
            left = dp[i][j-1];

          dp[i][j] = up+left;
      }
  }

  return dp[n-1][m-1];


}

int mazeObstacles(int n, int m, vector<vector<int> > &maze){
    vector<vector<int> > dp(n,vector<int>(m,-1));
    return mazeObstaclesUtil(n,m,maze,dp);

}

int main() {

  vector<vector<int> > maze{{0,0,0},
                            {0,-1,0},
                            {0,0,0}};

  int n = maze.size();
  int m = maze[0].size();
  cout<<mazeObstacles(n,m,maze);
}
```

Minimum Path Sum In a Grid

We are given an "N*M" matrix of integers. We need to find a path from the top-left corner to the bottom-right corner of the matrix, such that there is a minimum cost past that we select.

At every cell, we can move in only two directions: right and bottom. The cost of a path is given as the sum of values of cells of the given matrix.

```
#include <bits/stdc++.h>

using namespace std;

int minSumPathUtil(int i, int j,vector<vector<int>>
&matrix,vector<vector<int>> &dp)
{
  if(i==0 && j == 0)
    return matrix[0][0];
  if(i<0 || j<0)
    return 1e9;
  if(dp[i][j]!=-1) return dp[i][j];

  int up = matrix[i][j]+minSumPathUtil(i-1,j,matrix,dp);
  int left = matrix[i][j]+minSumPathUtil(i,j-1,matrix,dp);

  return dp[i][j] = min(up,left);

}

int minSumPath(int n, int m, vector<vector<int> > &matrix){
    vector<vector<int> > dp(n,vector<int>(m,-1));
    return minSumPathUtil(n-1,m-1,matrix,dp);

}

int main() {

  vector<vector<int> > matrix{{5,9,6},
                              {11,5,2}};

  int n = matrix.size();
  int m = matrix[0].size();

  cout<<minSumPath(n,m,matrix);
}
```

```
Time Complexity: O(N*M)
Space Complexity: O((M-1)+(N-1)) + O(N*M)

#include <bits/stdc++.h>

using namespace std;

int minSumPath(int n, int m, vector<vector<int> > &matrix){
    vector<vector<int> > dp(n,vector<int>(m,0));
    for(int i=0; i<n ; i++){
        for(int j=0; j<m; j++){
            if(i==0 && j==0) dp[i][j] = matrix[i][j];
            else{

                int up = matrix[i][j];
                if(i>0) up += dp[i-1][j];
                else up += 1e9;

                int left = matrix[i][j];
                if(j>0) left+=dp[i][j-1];
                else left += 1e9;

                dp[i][j] = min(up,left);
            }
        }
    }

    return dp[n-1][m-1];

}

int main() {

  vector<vector<int> > matrix{{5,9,6},
                              {11,5,2}};

  int n = matrix.size();
  int m = matrix[0].size();

  cout<<minSumPath(n,m,matrix);
}

Time Complexity: O(N*M)
Space Complexity: O(N*M)

#include <bits/stdc++.h>

using namespace std;

int minSumPath(int n, int m, vector<vector<int> > &matrix){
      vector<int> prev(m,0);
    for(int i=0; i<n ; i++){
        vector<int> temp(m,0);
        for(int j=0; j<m; j++){
            if(i==0 && j==0) temp[j] = matrix[i][j];
            else{

                int up = matrix[i][j];
                if(i>0) up += prev[j];
                else up += 1e9;

                int left = matrix[i][j];
                if(j>0) left+=temp[j-1];
                else left += 1e9;

                temp[j] = min(up,left);
            }
        }
        prev=temp;
    }

    return prev[m-1];

}



int main() {

  vector<vector<int> > matrix{{5,9,6},
                              {11,5,2}};

  int n = matrix.size();
  int m = matrix[0].size();

  cout<<minSumPath(n,m,matrix);
}

Time Complexity: O(M*N)
Space Complexity: O(N)
```

Minimum path sum in Triangular Grid

We are given a Triangular matrix. We need to find the minimum path sum from the first row to the last row.

At every cell we can move in only two directions: either to the bottom cell (↓) or to the bottom-right cell(↘)

```
#include <bits/stdc++.h>

using namespace std;

int minimumPathSumUtil(int i, int j, vector<vector<int> >
&triangle,int n,
vector<vector<int> > &dp) {

  if(dp[i][j]!=-1)
  return dp[i][j];

  if(i==n-1) return triangle[i][j];

  int down = triangle[i][j]+minimumPathSumUtil(i+1,j,triangle,n,dp);
  int diagonal = triangle[i][j]
+minimumPathSumUtil(i+1,j+1,triangle,n,dp);

  return dp[i][j] = min(down, diagonal);

}

int minimumPathSum(vector<vector<int> > &triangle, int n){
    vector<vector<int> > dp(n,vector<int>(n,-1));
    return minimumPathSumUtil(0,0,triangle,n,dp);

}

int main() {

  vector<vector<int> > triangle{{1},
                                {2,3},
                                {3,6,7},
                                {8,9,6,10}};

  int n = triangle.size();

  cout<<minimumPathSum(triangle,n);
}
```

```
Time Complexity: O(N*N)
Space Complexity: O(N) + O(N*N)
```

```
#include <bits/stdc++.h>

using namespace std;

int minimumPathSum(vector<vector<int> > &triangle, int n){
    vector<vector<int> > dp(n,vector<int>(n,0));

    for(int j=0;j<n;j++){
        dp[n-1][j] = triangle[n-1][j];
    }

    for(int i=n-2; i>=0; i--){
        for(int j=i; j>=0; j--){

            int down = triangle[i][j]+dp[i+1][j];
            int diagonal = triangle[i][j]+dp[i+1][j+1];

            dp[i][j] = min(down, diagonal);
        }
    }

    return dp[0][0];

}

int main() {

  vector<vector<int> > triangle{{1},
                               {2,3},
                               {3,6,7},
                               {8,9,6,10}};

  int n = triangle.size();

  cout<<minimumPathSum(triangle,n);
}
```

```
Time Complexity: O(N*N)
Space Complexity: O(N*N)
```

```
#include <bits/stdc++.h>

using namespace std;

int minimumPathSum(vector<vector<int> > &triangle, int n){
    vector<int> front(n,0), cur(n,0);

    for(int j=0;j<n;j++){
        front[j] = triangle[n-1][j];
    }

    for(int i=n-2; i>=0; i--){
        for(int j=i; j>=0; j--){

            int down = triangle[i][j]+front[j];
            int diagonal = triangle[i][j]+front[j+1];

            cur[j] = min(down, diagonal);
        }
        front=cur;
    }

    return front[0];

}

int main() {

  vector<vector<int> > triangle{{1},
                               {2,3},
                               {3,6,7},
                               {8,9,6,10}};

  int n = triangle.size();

  cout<<minimumPathSum(triangle,n);
}
```

```
Time Complexity: O(N*N)
Space Complexity: O(N)
```

Minimum/Maximum Falling Path Sum

We are given an ‘N*M’ matrix. We need to find the maximum path sum from any cell of the first row to any cell of the last row.

At every cell we can move in three directions: to the bottom cell (↓), to the bottom-right cell(↘), or to the bottom-left cell(↙).

```
#include <bits/stdc++.h>

using namespace std;

int getMaxUtil(int i, int j, int m, vector<vector<int>> &matrix,
vector<vector<int> > &dp){

    // Base Conditions
    if(j<0 || j>=m)
        return -1e9;
    if(i==0)
        return matrix[0][j];

    if(dp[i][j]!=-1) return dp[i][j];

    int up = matrix[i][j] + getMaxUtil(i-1,j,m,matrix,dp);
    int leftDiagonal = matrix[i][j] + getMaxUtil(i-1,j-1,m,matrix,dp);
    int rightDiagonal = matrix[i][j] + getMaxUtil(i-
1,j+1,m,matrix,dp);
```

```
    return dp[i][j]= max(up,max(leftDiagonal,rightDiagonal));

}

int getMaxPathSum(vector<vector<int> > &matrix){

    int n = matrix.size();
    int m = matrix[0].size();

    vector<vector<int>> dp(n,vector<int>(m,-1));

    int maxi = INT_MIN;

    for(int j=0; j<m;j++){
        int ans = getMaxUtil(n-1,j,m,matrix,dp);
        maxi = max(maxi,ans);
    }

    return maxi;
}

int main() {

  vector<vector<int> > matrix{{1,2,10,4},
                              {100,3,2,1},
                              {1,1,20,2},
                              {1,2,2,1}};

  cout<<getMaxPathSum(matrix);
}
```

Time Complexity: O(N*N)
Space Complexity: O(N) + O(N*M)

```
#include <bits/stdc++.h>

using namespace std;

int getMaxPathSum(vector<vector<int> > &matrix){

    int n = matrix.size();
    int m = matrix[0].size();

    vector<vector<int>> dp(n,vector<int>(m,0));

    // Initializing first row - base condition
    for(int j=0; j<m; j++){
        dp[0][j] = matrix[0][j];
    }
    for(int i=1; i<n; i++){
        for(int j=0;j<m;j++){

            int up = matrix[i][j] + dp[i-1][j];

            int leftDiagonal= matrix[i][j];
            if(j-1>=0) leftDiagonal += dp[i-1][j-1];
            else leftDiagonal += -1e9;

            int rightDiagonal = matrix[i][j];
            if(j+1<m) rightDiagonal += dp[i-1][j+1];
            else rightDiagonal += -1e9;

            dp[i][j] = max(up, max(leftDiagonal,rightDiagonal));

        }
    }

    int maxi = INT_MIN;

    for(int j=0; j<m;j++){
        maxi = max(maxi,dp[n-1][j]);
    }

    return maxi;
}

int main() {

  vector<vector<int> > matrix{{1,2,10,4},
                              {100,3,2,1},
                              {1,1,20,2},
                              {1,2,2,1}};

  cout<<getMaxPathSum(matrix);
}
```

Time Complexity: O(N*M)
Space Complexity: O(N*M)

```
#include <bits/stdc++.h>

using namespace std;

int getMaxPathSum(vector<vector<int> > &matrix){

     int n = matrix.size();
    int m = matrix[0].size();
    vector<int> prev(m,0), cur(m,0);

    // Initializing first row - base condition
    for(int j=0; j<m; j++){
        prev[j] = matrix[0][j];
    }

    for(int i=1; i<n; i++){
        for(int j=0;j<m;j++){

            int up = matrix[i][j] + prev[j];

            int leftDiagonal= matrix[i][j];
            if(j-1>=0) leftDiagonal += prev[j-1];
            else leftDiagonal += -1e9;

            int rightDiagonal = matrix[i][j];
            if(j+1<m) rightDiagonal += prev[j+1];
            else rightDiagonal += -1e9;

            cur[j] = max(up, max(leftDiagonal,rightDiagonal));

        }

        prev = cur;
    }

    int maxi = INT_MIN;

    for(int j=0; j<m;j++){
        maxi = max(maxi,prev[j]);
    }

    return maxi;

}

int main() {

  vector<vector<int> > matrix{{1,2,10,4},
                              {100,3,2,1},
                              {1,1,20,2},
                              {1,2,2,1}};

  cout<<getMaxPathSum(matrix);
}
```

Time Complexity: O(N*M)
Space Complexity: O(M)

3-d DP : Ninja and his friends

We are given an 'N*M' matrix. Every cell of the matrix has some chocolates on it, mat[i][j] gives us the number of chocolates. We have two friends 'Alice' and 'Bob'. initially, Alice is standing on the cell(0,0) and Bob is standing on the cell(0, M-1). Both of them can move only to the cells below them in these three directions: to the bottom cell (↓), to the bottom-right cell(↘), or to the bottom-left cell(↙).

When Alica and Bob visit a cell, they take all the chocolates from that cell with them. It can happen that they visit the same cell, in that case, the chocolates need to be considered only once.

They cannot go out of the boundary of the given matrix, we need to return the maximum number of chocolates that Bob and Alice can together collect.

```
#include<bits/stdc++.h>

using namespace std;

int maxChocoUtil(int i, int j1, int j2, int n, int m, vector < vector
< int >>
& grid, vector < vector < vector < int >>> & dp) {
  if (j1 < 0 || j1 >= m || j2 < 0 || j2 >= m)
    return -1e9;

  if (i == n - 1) {
    if (j1 == j2)
      return grid[i][j1];
    else
      return grid[i][j1] + grid[i][j2];
  }

  if (dp[i][j1][j2] != -1)
    return dp[i][j1][j2];

  int maxi = INT_MIN;
  for (int di = -1; di <= 1; di++) {
    for (int dj = -1; dj <= 1; dj++) {
      int ans;
      if (j1 == j2)
        ans = grid[i][j1] + maxChocoUtil(i + 1, j1 + di, j2 + dj, n,
m, grid, dp);
      else
        ans = grid[i][j1] + grid[i][j2] + maxChocoUtil(i + 1, j1 + di,
j2 + dj, n,
        m, grid, dp);
      maxi = max(maxi, ans);
    }
  }
  return dp[i][j1][j2] = maxi;
}

int maximumChocolates(int n, int m, vector < vector < int >> & grid) {

  vector < vector < vector < int >>> dp(n, vector < vector < int >>
(m, vector < int
  > (m, -1)));

  return maxChocoUtil(0, 0, m - 1, n, m, grid, dp);
}

int main() {

   vector<vector<int> > matrix{
      {2,3,1,2},
      {3,4,2,2},
      {5,6,3,5},
  };

  int n = matrix.size();
  int m = matrix[0].size();

  cout << maximumChocolates(n, m, matrix);
}
```

Time Complexity: O(N*M*M) * 9
Space Complexity: O(N) + O(N*M*M)

```
#include<bits/stdc++.h>

using namespace std;

int maximumChocolates(int n, int m, vector < vector < int >> & grid) {
  // Write your code here.
  vector < vector < vector < int >>> dp(n, vector < vector < int >>
(m,
  vector < int > (m, 0)));

  for (int j1 = 0; j1 < m; j1++) {
    for (int j2 = 0; j2 < m; j2++) {
      if (j1 == j2)
        dp[n - 1][j1][j2] = grid[n - 1][j1];
      else
        dp[n - 1][j1][j2] = grid[n - 1][j1] + grid[n - 1][j2];
    }
  }

  //Outer Nested Loops for travering DP Array
  for (int i = n - 2; i >= 0; i--) {
    for (int j1 = 0; j1 < m; j1++) {
      for (int j2 = 0; j2 < m; j2++) {

        int maxi = INT_MIN;

        //Inner nested loops to try out 9 options
        for (int di = -1; di <= 1; di++) {
          for (int dj = -1; dj <= 1; dj++) {

            int ans;

            if (j1 == j2)
              ans = grid[i][j1];
            else
              ans = grid[i][j1] + grid[i][j2];

            if ((j1 + di < 0 || j1 + di >= m) ||
              (j2 + dj < 0 || j2 + dj >= m))

              ans += -1e9;
            else
              ans += dp[i + 1][j1 + di][j2 + dj];

            maxi = max(ans, maxi);
          }
        }
        dp[i][j1][j2] = maxi;
      }
    }
  }

  return dp[0][0][m - 1];

}

int main() {

  vector<vector<int> > matrix{
      {2,3,1,2},
      {3,4,2,2},
      {5,6,3,5},
  };

  int n = matrix.size();
  int m = matrix[0].size();

  cout << maximumChocolates(n, m, matrix);
}
```

```
Time Complexity: O(N*M*M)*9
Space Complexity: O(N*M*M)

#include<bits/stdc++.h>

using namespace std;

int maximumChocolates(int n, int m, vector < vector < int >> & grid) {
  // Write your code here.
  vector < vector < int >> front(m, vector < int > (m, 0)), cur(m,
vector < int >
   (m, 0));

  for (int j1 = 0; j1 < m; j1++) {
    for (int j2 = 0; j2 < m; j2++) {
      if (j1 == j2)
        front[j1][j2] = grid[n - 1][j1];
      else
        front[j1][j2] = grid[n - 1][j1] + grid[n - 1][j2];
    }
  }

  //Outer Nested Loops for travering DP Array
  for (int i = n - 2; i >= 0; i--) {
    for (int j1 = 0; j1 < m; j1++) {
      for (int j2 = 0; j2 < m; j2++) {

        int maxi = INT_MIN;

        //Inner nested loops to try out 9 options
        for (int di = -1; di <= 1; di++) {
          for (int dj = -1; dj <= 1; dj++) {

            int ans;

            if (j1 == j2)
              ans = grid[i][j1];
            else
              ans = grid[i][j1] + grid[i][j2];

            if ((j1 + di < 0 || j1 + di >= m) ||
              (j2 + dj < 0 || j2 + dj >= m))

              ans += -1e9;
            else
              ans += front[j1 + di][j2 + dj];

            maxi = max(ans, maxi);
          }
        }
        cur[j1][j2] = maxi;
      }
    }
    front = cur;
  }

  return front[0][m - 1];

}

int main() {

  vector<vector<int> > matrix{
      {2,3,1,2},
      {3,4,2,2},
      {5,6,3,5},
  };

  int n = matrix.size();
  int m = matrix[0].size();

  cout << maximumChocolates(n, m, matrix);
}
Output:

Time Complexity: O(N*M*M)*9
Space Complexity: O(M*M)
```

Distinct Subsequences

We are given two strings S1 and S2, we want to know how many distinct subsequences of S2 are present in S1.

```
#include <bits/stdc++.h>

using  namespace  std;

int prime = 1e9+7;

int countUtil(string s1, string s2, int ind1, int
ind2,vector<vector<int>>& dp){
    if(ind2<0)
        return 1;
    if(ind1<0)
        return 0;

    if(dp[ind1][ind2]!=-1)
```

```
        return dp[ind1][ind2];

    if(s1[ind1]==s2[ind2]){
        int leaveOne = countUtil(s1,s2,ind1-1,ind2-1,dp);
        int stay = countUtil(s1,s2,ind1-1,ind2,dp);

        return dp[ind1][ind2] = (leaveOne + stay)%prime;
    }

    else{
        return dp[ind1][ind2] = countUtil(s1,s2,ind1-1,ind2,dp);
    }
}

int subsequenceCounting(string &t, string &s, int lt, int ls) {
    // Write your code here.

    vector<vector<int>> dp(lt,vector<int>(ls,-1));
    return countUtil(t,s,lt-1,ls-1,dp);
}


int main() {

  string s1 = "babgbag";
  string s2 = "bag";

  cout << "The Count of Distinct Subsequences is "
  <<subsequenceCounting(s1,s2,s1.size(),s2.size());
}

Time Complexity: O(N*M)
Space Complexity: O(N*M) + O(N+M)
```

```
#include <bits/stdc++.h>

using  namespace  std;

int prime = 1e9+7;


int subsequenceCounting(string &s1, string &s2, int n, int m) {
    // Write your code here.

    vector<vector<int>> dp(n+1,vector<int>(m+1,0));

    for(int i=0;i<n+1;i++){
        dp[i][0]=1;
    }
    for(int i=1;i<m+1;i++){
        dp[0][i]=0;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<m+1;j++){

            if(s1[i-1]==s2[j-1])
                dp[i][j] = (dp[i-1][j-1] + dp[i-1][j])%prime;
            else
                dp[i][j] = dp[i-1][j];
        }
    }


    return dp[n][m];
}


int main() {

  string s1 = "babgbag";
  string s2 = "bag";

  cout << "The Count of Distinct Subsequences is "<<
  subsequenceCounting(s1,s2,s1.size(),s2.size());
}
```

Time Complexity: O(N*M)
Space Complexity: O(N*M)

```
#include <bits/stdc++.h>

using  namespace  std;

int prime = 1e9+7;


int subsequenceCounting(string &s1, string &s2, int n, int m) {
    // Write your code here.

    vector<int> prev(m+1,0);

    prev[0]=1;

    for(int i=1;i<n+1;i++){
        for(int j=m;j>=1;j--){ // Reverse direction

            if(s1[i-1]==s2[j-1])
```

```
                prev[j] = (prev[j-1] + prev[j])%prime;
            else
                prev[j] = prev[j]; //can omit this statemwnt
        }
    }


    return prev[m];
}

int main() {

  string s1 = "babgbag";
  string s2 = "bag";

  cout << "The Count of Distinct Subsequences is "<<
  subsequenceCounting(s1,s2,s1.size(),s2.size());
}

Time Complexity: O(N*M)
Space Complexity: O(M)
```

Edit Distance

We are given two strings 'S1' and 'S2'. We need to convert S1 to S2. The following three operations are allowed:

1. Deletion of a character.
2. Replacement of a character with another one.
3. Insertion of a character.

We have to return the minimum number of operations required to convert S1 to S2 as our answer.

```
#include <bits/stdc++.h>
using namespace std;

int editDistanceUtil(string& S1, string& S2, int i, int j,
vector<vector<int>>& dp){

    if(i<0)
        return j+1;
    if(j<0)
        return i+1;

    if(dp[i][j]!=-1) return dp[i][j];

    if(S1[i]==S2[j])
        return dp[i][j] =  0+editDistanceUtil(S1,S2,i-1,j-1,dp);
    // Minimum of three choices
    else return dp[i][j] = 1+min(editDistanceUtil(S1,S2,i-1,j-1,dp),
    min(editDistanceUtil(S1,S2,i-1,j,dp),editDistanceUtil(S1,S2,i,j-
1,dp)));

}

int editDistance(string& S1, string& S2){

    int n = S1.size();
    int m = S2.size();

    vector<vector<int>> dp(n,vector<int>(m,-1));
    return editDistanceUtil(S1,S2,n-1,m-1,dp);

}

int main() {

  string s1 = "horse";
  string s2 = "ros";

  cout << "The minimum number of operations required is:
"<<editDistance(s1,s2);
}
```

Time Complexity: O(N*M)
Space Complexity: O(N*M) + O(N+M)

```
#include <bits/stdc++.h>

using namespace std;


int editDistance(string& S1, string& S2){

    int n = S1.size();
    int m = S2.size();

    vector<vector<int>> dp(n+1,vector<int>(m+1,0));

    for(int i=0;i<=n;i++){
        dp[i][0] = i;
    }
    for(int j=0;j<=m;j++){
        dp[0][j] = j;
    }

    for(int i=1;i<n+1;i++){
        for(int j=1;j<m+1;j++){
            if(S1[i-1]==S2[j-1])
                dp[i][j] = 0+dp[i-1][j-1];

            else dp[i][j] = 1+min(dp[i-1][j-1],min(dp[i-1][j],dp[i][j-
1]));
        }
    }

    return dp[n][m];

}


int main() {

  string s1 = "horse";
  string s2 = "ros";

  cout << "The minimum number of operations required is:
"<<editDistance(s1,s2);
}
```

```
Time Complexity: O(N*M)
Space Complexity: O(N*M)
```

```
#include <bits/stdc++.h>

using namespace std;


int editDistance(string& S1, string& S2){

    int n = S1.size();
    int m = S2.size();

    vector<int> prev(m+1,0);
    vector<int> cur(m+1,0);

    for(int j=0;j<=m;j++){
        prev[j] = j;
    }

    for(int i=1;i<n+1;i++){
        cur[0]=i;
        for(int j=1;j<m+1;j++){
            if(S1[i-1]==S2[j-1])
                cur[j] = 0+prev[j-1];

            else cur[j] = 1+min(prev[j-1],min(prev[j],cur[j-1]));
        }
        prev = cur;
    }

    return prev[m];

}

int main() {

  string s1 = "horse";
  string s2 = "ros";

  cout << "The minimum number of operations required is: 
"<<editDistance(s1,s2);
}

Time Complexity: O(N*M)
Space Complexity: O(M)
```

Wildcard Matching

We are given two strings ‘S1’ and ‘S2’. String S1 can have the following two special characters:

1. ‘?’ can be matched to a single character of S2.
2. ‘*’ can be matched to any sequence of characters of S2. (sequence can be of length zero or more).

We need to check whether strings S1 and S2 match or not.

```
#include <bits/stdc++.h>

using namespace std;

bool isAllStars(string & S1, int i) {
  for (int j = 0; j <= i; j++) {
    if (S1[j] != '*')
      return false;
  }
  return true;
}

bool wildcardMatchingUtil(string & S1, string & S2, int i, int j,
vector < vector < bool >> & dp) {

  //Base Conditions
  if (i < 0 && j < 0)
    return true;
  if (i < 0 && j >= 0)
    return false;
  if (j < 0 && i >= 0)
    return isAllStars(S1, i);

  if (dp[i][j] != -1) return dp[i][j];

  if (S1[i] == S2[j] || S1[i] == '?')
    return dp[i][j] = wildcardMatchingUtil(S1, S2, i - 1, j - 1, dp);

  else {
    if (S1[i] == '*')
      return wildcardMatchingUtil(S1, S2, i - 1, j, dp) ||
wildcardMatchingUtil(S1, S2, i, j - 1, dp);
    else return false;
  }
}

bool wildcardMatching(string & S1, string & S2) {

  int n = S1.size();
  int m = S2.size();

  vector < vector < bool >> dp(n, vector < bool > (m, -1));
  return wildcardMatchingUtil(S1, S2, n - 1, m - 1, dp);

}

int main() {

  string S1 = "ab*cd";
  string S2 = "abdefcd";

  if (wildcardMatching(S1, S2))
    cout << "String S1 and S2 do match";
  else cout << "String S1 and S2 do not match";
}
```

```
Time Complexity: O(N*M)
Space Complexity: O(N*M) + O(N+M)
```

```
#include <bits/stdc++.h>

using namespace std;

bool isAllStars(string & S1, int i) {

  // S1 is taken in 1-based indexing
  for (int j = 1; j <= i; j++) {
    if (S1[j - 1] != '*')
      return false;
  }
  return true;
}

bool wildcardMatching(string & S1, string & S2) {

  int n = S1.size();
  int m = S2.size();

  vector < vector < bool >> dp(n + 1, vector < bool > (m, false));

  dp[0][0] = true;

  for (int j = 1; j <= m; j++) {
    dp[0][j] = false;
  }
  for (int i = 1; i <= n; i++) {
    dp[i][0] = isAllStars(S1, i);
  }

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {

      if (S1[i - 1] == S2[j - 1] || S1[i - 1] == '?')
        dp[i][j] = dp[i - 1][j - 1];

      else {
        if (S1[i - 1] == '*')
          dp[i][j] = dp[i - 1][j] || dp[i][j - 1];

        else dp[i][j] = false;
      }
    }
  }

  return dp[n][m];

}

int main() {

  string S1 = "ab*cd";
  string S2 = "abdefcd";

  if (wildcardMatching(S1, S2))
    cout << "String S1 and S2 do match";
  else cout << "String S1 and S2 do not match";
}
```

```
Time Complexity: O(N*M)
Space Complexity: O(N*M)

#include <bits/stdc++.h>

using namespace std;

bool isAllStars(string & S1, int i) {

  // S1 is taken in 1-based indexing
  for (int j = 1; j <= i; j++) {
    if (S1[j - 1] != '*')
      return false;
  }
  return true;
}

bool wildcardMatching(string & S1, string & S2) {

  int n = S1.size();
  int m = S2.size();

  vector < bool > prev(m + 1, false);
  vector < bool > cur(m + 1, false);

  prev[0] = true;

  for (int i = 1; i <= n; i++) {
    cur[0] = isAllStars(S1, i);
    for (int j = 1; j <= m; j++) {

      if (S1[i - 1] == S2[j - 1] || S1[i - 1] == '?')
        cur[j] = prev[j - 1];

      else {
        if (S1[i - 1] == '*')
          cur[j] = prev[j] || cur[j - 1];

        else cur[j] = false;
      }
    }
    prev = cur;
  }

  return prev[m];

}

int main() {
  string S1 = "ab*cd";
  string S2 = "abdefcd";

  if (wildcardMatching(S1, S2))
    cout << "String S1 and S2 do match";
  else cout << "String S1 and S2 do not match";
}
```

```
Time Complexity: O(N*M)
Space Complexity: O(M)
```

Stock Buy and Sell |

We are given an array Arr[] of length n. It represents the price of a stock on ‘n’ days. The following guidelines need to be followed:

1. We can buy and sell a stock only once.
2. We can buy and sell the stock on any day but to sell the stock, we need to first buy it on the same or any previous day.

We need to tell the maximum profit one can get by buying and selling this stock.

```
#include <bits/stdc++.h>

using namespace std;

int maximumProfit(vector<int> &Arr){
    // Write your code here.
      int maxProfit = 0;
      int mini = Arr[0];

      for(int i=1;i<Arr.size();i++){
         int curProfit = Arr[i] - mini;
         maxProfit = max(maxProfit,curProfit);
         mini = min(mini,Arr[i]);
         }
      return maxProfit;
}

int main() {

  vector<int> Arr  ={7,1,5,3,6,4};

  cout<<"The maximum profit by selling the stock is 
"<<maximumProfit(Arr);
}
```

Buy and Sell Stock – II

We are given an array Arr[] of length n. It represents the price of a stock on 'n' days. The following guidelines need to be followed:

1. We can buy and sell the stock any number of times.
2. In order to sell the stock, we need to first buy it on the same or any previous day.
3. We can't buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can't sell before buying and can't buy before selling any previously bought stock.

```
#include <bits/stdc++.h>

using namespace std;

long getAns(long *Arr, int ind, int buy, int n, vector<vector<long>>
&dp ){

    if(ind==n) return 0; //base case

    if(dp[ind][buy]!=-1)
        return dp[ind][buy];

    long profit;

    if(buy==0){// We can buy the stock
        profit = max(0+getAns(Arr,ind+1,0,n,dp), -Arr[ind] +
getAns(Arr,ind+1,1,n,dp));
    }

    if(buy==1){// We can sell the stock
        profit = max(0+getAns(Arr,ind+1,1,n,dp), Arr[ind] +
getAns(Arr,ind+1,0,n,dp));
    }

    return dp[ind][buy] = profit;
}


long getMaximumProfit(long *Arr, int n)
{
    //Write your code here

    vector<vector<long>> dp(n,vector<long>(2,-1));

    if(n==0) return 0;
    long ans = getAns(Arr,0,0,n,dp);
    return ans;
}
```

```
int main() {

  int n =6;
  long Arr[n] = {7,1,5,3,6,4};

  cout<<"The maximum profit that can be generated is
"<<getMaximumProfit(Arr, n);
}
```

```
Time Complexity: O(N*2)
Space Complexity: O(N*2) + O(N)
```

```
#include <bits/stdc++.h>

using namespace std;


long getMaximumProfit(long *Arr, int n)
{
    //Write your code here

    vector<vector<long>> dp(n+1,vector<long>(2,-1));

    //base condition
    dp[n][0] = dp[n][1] = 0;

    long profit;

    for(int ind= n-1; ind>=0; ind--){
        for(int buy=0; buy<=1; buy++){
            if(buy==0){// We can buy the stock
                profit = max(0+dp[ind+1][0], -Arr[ind] + dp[ind+1]
[1]);
            }

            if(buy==1){// We can sell the stock
                profit = max(0+dp[ind+1][1], Arr[ind] + dp[ind+1][0]);
            }

            dp[ind][buy]  = profit;
        }
    }
    return dp[0][0];
}

int main() {

  int n =6;
  long Arr[n] = {7,1,5,3,6,4};
  cout<<"The maximum profit that can be generated is
"<<getMaximumProfit(Arr, n);
}
```

Time Complexity: O(N*2)
Space Complexity: O(N*2)

```
#include <bits/stdc++.h>

using namespace std;


long getMaximumProfit(long *Arr, int n)
{
    //Write your code here

    vector<long> ahead (2,0);
    vector<long> cur(2,0);

    //base condition
    ahead[0] = ahead[1] = 0;

    long profit;

    for(int ind= n-1; ind>=0; ind--){
        for(int buy=0; buy<=1; buy++){
            if(buy==0){// We can buy the stock
                profit = max(0+ahead[0], -Arr[ind] + ahead[1]);
            }

            if(buy==1){// We can sell the stock
                profit = max(0+ahead[1], Arr[ind] + ahead[0]);
            }
            cur[buy]  = profit;
        }

        ahead = cur;
    }
    return cur[0];
}

int main() {

  int n =6;
  long Arr[n] = {7,1,5,3,6,4};

  cout<<"The maximum profit that can be generated is
"<<getMaximumProfit(Arr, n);
}
```

```
Time Complexity: O(N*2)
Space Complexity: O(1)
```

Buy and Sell Stock – III

We are given an array Arr[] of length n. It represents the price of a stock on 'n' days. The following guidelines need to be followed:

1. We can buy and sell the stock any number of times.
2. In order to sell the stock, we need to first buy it on the same or any previous day.
3. We can't buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can't sell before buying and can't buy before selling any previously bought stock.
4. We can do at most 2 transactions.

```
#include <bits/stdc++.h>

using namespace std;

int getAns(vector<int>& Arr, int n, int ind, int buy, int cap,
vector<vector<vector<int>>>& dp ){

    if(ind==n || cap==0) return 0; //base case

    if(dp[ind][buy][cap]!=-1)
        return dp[ind][buy][cap];

    int profit;

    if(buy==0){// We can buy the stock
        profit = max(0+getAns(Arr,n,ind+1,0,cap,dp),
                     -Arr[ind] + getAns(Arr,n,ind+1,1,cap,dp));
    }

    if(buy==1){// We can sell the stock
        profit = max(0+getAns(Arr,n,ind+1,1,cap,dp),
                     Arr[ind] + getAns(Arr,n,ind+1,0,cap-1,dp));
    }

    return dp[ind][buy][cap] = profit;
}


int maxProfit(vector<int>& prices, int n)
{
    // Creating a 3d - dp of size [n][2][3]
    vector<vector<vector<int>>> dp(n,
                                    vector<vector<int>>
                                            (2,vector<int>(3,-1)));
    return getAns(prices,n,0,0,2,dp);

}

int main() {

  vector<int> prices = {3,3,5,0,0,3,1,4};
  int n = prices.size();

  cout<<"The maximum profit that can be generated is
"<<maxProfit(prices, n);
}
```

```
Time Complexity: O(N*2*3)
Space Complexity: O(N*2*3) + O(N)
```

```
#include <bits/stdc++.h>

using namespace std;


int maxProfit(vector<int>& Arr, int n)
{
    // Creating a 3d - dp of size [n+1][2][3] initialized to 0
    vector<vector<vector<int>>> dp(n+1,
                                    vector<vector<int>>
                                            (2,vector<int>(3,0)));

    // As dp array is intialized to 0, we have already covered the
base case

    for(int ind = n-1; ind>=0; ind--){
        for(int buy = 0; buy<=1; buy++){
            for(int cap=1; cap<=2; cap++){

                if(buy==0){// We can buy the stock
                    dp[ind][buy][cap] = max(0+dp[ind+1][0][cap],
                                -Arr[ind] + dp[ind+1][1][cap]);
                 }

                if(buy==1){// We can sell the stock
                    dp[ind][buy][cap] = max(0+dp[ind+1][1][cap],
                                Arr[ind] + dp[ind+1][0][cap-1]);
                }
            }
        }
    }


    return dp[0][0][2];
}

int main() {

  vector<int> prices = {3,3,5,0,0,3,1,4};
  int n = prices.size();

  cout<<"The maximum profit that can be generated is
"<<maxProfit(prices, n);
}
```

```
Time Complexity: O(N*2*3)
Space Complexity: O(N*2*3)
```

```
#include <bits/stdc++.h>

using namespace std;


int maxProfit(vector<int>& Arr, int n)
{

    vector<vector<int>> ahead(2,vector<int> (3,0));

    vector<vector<int>> cur(2,vector<int> (3,0));


    for(int ind = n-1; ind>=0; ind--){
        for(int buy = 0; buy<=1; buy++){
            for(int cap=1; cap<=2; cap++){

                if(buy==0){// We can buy the stock
                    cur[buy][cap] = max(0+ahead[0][cap],
                                 -Arr[ind] + ahead[1][cap]);
                 }

                if(buy==1){// We can sell the stock
                    cur[buy][cap] = max(0+ahead[1][cap],
                                 Arr[ind] + ahead[0][cap-1]);
                }
            }
        }
        ahead = cur;
    }

    return ahead[0][2];

}
```

```
int main() {

  vector<int> prices = {3,3,5,0,0,3,1,4};
  int n = prices.size();

  cout<<"The maximum profit that can be generated is
"<<maxProfit(prices, n);
}

Time Complexity: O(N*2*3)
Space Complexity: O(1)
```

Buy and Sell Stock – IV

We are given an array Arr[] of length n. It represents the price of a stock on 'n' days. The following guidelines need to be followed:

1. We can buy and sell the stock any number of times.
2. In order to sell the stock, we need to first buy it on the same or any previous day.
3. We can't buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can't sell before buying and can't buy before selling any previously bought stock.
4. We can do at-most K transactions.

```
#include <bits/stdc++.h>

using namespace std;

int getAns(vector<int>& Arr, int n, int ind, int buy, int cap,
vector<vector<vector<int>>>& dp ){

    if(ind==n || cap==0) return 0; //base case

    if(dp[ind][buy][cap]!=-1)
        return dp[ind][buy][cap];

    int profit;

    if(buy==0){// We can buy the stock
        profit = max(0+getAns(Arr,n,ind+1,0,cap,dp),
                     -Arr[ind] + getAns(Arr,n,ind+1,1,cap,dp));
    }

    if(buy==1){// We can sell the stock
        profit = max(0+getAns(Arr,n,ind+1,1,cap,dp),
                    Arr[ind] + getAns(Arr,n,ind+1,0,cap-1,dp));
    }

    return dp[ind][buy][cap] = profit;
}
```

```
int maximumProfit(vector<int>& prices, int n, int k)
{

    // Creating a 3d - dp of size [n][2][k+1]
    vector<vector<vector<int>>> dp(n,
                                    vector<vector<int>>
                                            (2,vector<int>(k+1,-1)));

    return getAns(prices,n,0,0,k,dp);

}

int main() {

  vector<int> prices = {3,3,5,0,0,3,1,4};
  int n = prices.size();
int k = 2;

  cout<<"The maximum profit that can be generated is
"<<maximumProfit(prices, n,k);
}

Time Complexity: O(N*2*3)
Space Complexity: O(N*2*K) + O(N)

#include <bits/stdc++.h>

using namespace std;


int maximumProfit(vector<int>& Arr, int n, int k)
{
    // Creating a 3d - dp of size [n+1][2][k+1] initialized to 0
    vector<vector<vector<int>>> dp(n+1,
                                    vector<vector<int>>
                                            (2,vector<int>(k+1,0)));

    // As dp array is initialized to 0, we have already covered the
base case

    for(int ind = n-1; ind>=0; ind--){
        for(int buy = 0; buy<=1; buy++){
            for(int cap=1; cap<=k; cap++){

                if(buy==0){// We can buy the stock
                    dp[ind][buy][cap] = max(0+dp[ind+1][0][cap],
                                -Arr[ind] + dp[ind+1][1][cap]);
                 }
                if(buy==1){// We can sell the stock
                    dp[ind][buy][cap] = max(0+dp[ind+1][1][cap],
                                Arr[ind] + dp[ind+1][0][cap-1]);
                }
            }
        }
    }


    return dp[0][0][k];

}

int main() {

  vector<int> prices = {3,3,5,0,0,3,1,4};
  int n = prices.size();
  int k =2;

  cout<<"The maximum profit that can be generated is
"<<maximumProfit(prices, n, k);
}
```

```
Time Complexity: O(N*2*k)
Space Complexity: O(N*2*k)
```

```
#include <bits/stdc++.h>

using namespace std;


int maxProfit(vector<int>& Arr, int n, int k)
{

    vector<vector<int>> ahead(2,vector<int> (k+1,0));

    vector<vector<int>> cur(2,vector<int> (k+1,0));


    for(int ind = n-1; ind>=0; ind--){
        for(int buy = 0; buy<=1; buy++){
            for(int cap=1; cap<=k; cap++){

                if(buy==0){// We can buy the stock
                    cur[buy][cap] = max(0+ahead[0][cap],
                                -Arr[ind] + ahead[1][cap]);
                 }

                if(buy==1){// We can sell the stock
```

```
                    cur[buy][cap] = max(0+ahead[1][cap],
                                Arr[ind] + ahead[0][cap-1]);
                }
            }
        }
        ahead = cur;
    }

    return ahead[0][k];

}

int main() {

  vector<int> prices = {3,3,5,0,0,3,1,4};
  int n = prices.size();
  int k=2;
  cout<<"The maximum profit that can be generated is
"<<maxProfit(prices, n,k);
}

Time Complexity: O(N*2*K)
Space Complexity: O(K)
```

Buy and Sell Stocks With Cooldown

We are given an array Arr[] of length n. It represents the price of a stock on 'n' days. The following guidelines need to be followed:

1. We can buy and sell the stock any number of times.
2. In order to sell the stock, we need to first buy it on the same or any previous day.
3. We can't buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can't sell before buying and can't buy before selling any previously bought stock.
4. We can't buy a stock on the very next day of selling it. This is the cooldown clause.

```
#include <bits/stdc++.h>

using namespace std;

int getAns(vector<int> Arr, int ind, int buy, int n,
vector<vector<int>> &dp ){

    if(ind>=n) return 0; //base case

    if(dp[ind][buy]!=-1)
        return dp[ind][buy];

    int profit;
    if(buy==0){// We can buy the stock
        profit = max(0+getAns(Arr,ind+1,0,n,dp), -Arr[ind]
+getAns(Arr,ind+1,1,n,dp));
    }

    if(buy==1){// We can sell the stock
        profit = max(0+getAns(Arr,ind+1,1,n,dp), Arr[ind]
+getAns(Arr,ind+2,0,n,dp));
    }

    return dp[ind][buy] = profit;
}


int stockProfit(vector<int> &Arr)
{
    int n = Arr.size();
    vector<vector<int>> dp(n,vector<int>(2,-1));

    int ans = getAns(Arr,0,0,n,dp);
    return ans;
}

int main() {

  vector<int> prices {4,9,0,4,10};

  cout<<"The maximum profit that can be generated is
"<<stockProfit(prices);
}
```

Time Complexity: O(N*2)
Space Complexity: O(N*2) + O(N)

```
#include <bits/stdc++.h>

using namespace std;

int stockProfit(vector<int> &Arr)
{
    int n = Arr.size();
    vector<vector<int>> dp(n+2,vector<int>(2,0));

    for(int ind = n-1; ind>=0; ind--){
        for(int buy=0; buy<=1; buy++){
            int profit;

            if(buy==0){// We can buy the stock
                profit = max(0+dp[ind+1][0], -Arr[ind] + dp[ind+1]
[1]);
```

```
            }

            if(buy==1){// We can sell the stock
                profit = max(0+dp[ind+1][1], Arr[ind] + dp[ind+2][0]);
            }

            dp[ind][buy] = profit;
        }
    }

    return dp[0][0];
}

int main() {

  vector<int> prices {4,9,0,4,10};

  cout<<"The maximum profit that can be generated is
"<<stockProfit(prices);
}
```

Time Complexity: O(N*2)
Space Complexity: O(N*2)

```
#include <bits/stdc++.h>

using namespace std;

int stockProfit(vector<int> &Arr)
{
    int n = Arr.size();
    vector<int> cur(2,0);
    vector<int> front1(2,0);
    vector<int> front2(2,0);

    for(int ind = n-1; ind>=0; ind--){
        for(int buy=0; buy<=1; buy++){
            int profit;

            if(buy==0){// We can buy the stock
                profit = max(0+front1[0], -Arr[ind] + front1[1]);
            }

            if(buy==1){// We can sell the stock
                profit = max(0+front1[1], Arr[ind] + front2[0]);
            }

            cur[buy] = profit;
        }
        front2 = front1;
        front1 = cur;

    }

    return cur[0];
}

int main() {

  vector<int> prices {4,9,0,4,10};

  cout<<"The maximum profit that can be generated is
"<<stockProfit(prices);
}

Time Complexity: O(N*2)
Space Complexity: O(1)
```

Buy and Sell Stocks With Transaction Fees

We are given an array Arr[] of length n. It represents the price of a stock on 'n' days. The following guidelines need to be followed:

1. We can buy and sell the stock any number of times.
2. In order to sell the stock, we need to first buy it on the same or any previous day.
3. We can't buy a stock again after buying it once. In other words, we first buy a stock and then sell it. After selling we can buy and sell again. But we can't sell before buying and can't buy before selling any previously bought stock.
4. After every transaction, there is a transaction fee ('fee') associated with it.

```
#include <bits/stdc++.h>

using namespace std;

int getAns(vector<int> &Arr,int ind,int buy,int n,int
fee,vector<vector<int>> &dp){

    if(ind==n) return 0; //base case

    if(dp[ind][buy]!=-1)
        return dp[ind][buy];

    int profit;

    if(buy==0){// We can buy the stock
        profit = max(0+getAns(Arr,ind+1,0,n,fee,dp), -Arr[ind] +
getAns(Arr,ind+1,1,n,fee,dp));
    }
    if(buy==1){// We can sell the stock
        profit = max(0+getAns(Arr,ind+1,1,n,fee,dp), Arr[ind] -fee +
getAns(Arr,ind+1,0,n,fee,dp));
    }

    return dp[ind][buy] = profit;
}


int maximumProfit(int n, int fee, vector<int>& Arr)
{
    //Write your code here

    vector<vector<int>> dp(n,vector<int>(2,-1));

    if(n==0) return 0;
    int ans = getAns(Arr,0,0,n,fee,dp);
    return ans;
}

int main() {

  vector<int> prices = {1,3,2,8,4,9};
  int n = prices.size();
  int fee = 2;

  cout<<"The maximum profit that can be generated is
"<<maximumProfit(n,fee,prices);
}
```

Time Complexity: O(N*2)
Space Complexity: O(N*2) + O(N)

```
#include <bits/stdc++.h>

using namespace std;

int maximumProfit(int n, int fee, vector<int>& Arr)
{
    //Write your code here
    if(n==0) return 0;

    vector<vector<int>> dp(n+1,vector<int>(2,0));

    // base condition is handled by initializing dp array as 0

      for(int ind= n-1; ind>=0; ind--){
        for(int buy=0; buy<=1; buy++){
            int profit;
            if(buy==0){// We can buy the stock
                profit = max(0+dp[ind+1][0], -Arr[ind] + dp[ind+1]
[1]);
            }

            if(buy==1){// We can sell the stock
                profit = max(0+dp[ind+1][1], Arr[ind] -fee + dp[ind+1]
[0]);
            }

            dp[ind][buy]  = profit;
        }
    }

    return dp[0][0];
}

int main() {

  vector<int> prices = {1,3,2,8,4,9};
  int n = prices.size();
  int fee = 2;

  cout<<"The maximum profit that can be generated is
"<<maximumProfit(n,fee,prices);
}

Time Complexity: O(N*2)
Space Complexity: O(N*2)

#include <bits/stdc++.h>

using namespace std;

int maximumProfit(int n, int fee, vector<int>& Arr)
{
    if(n==0) return 0;

    vector<long> ahead (2,0);
    vector<long> cur(2,0);

    //base condition
    ahead[0] = ahead[1] = 0;

    long profit;

    for(int ind= n-1; ind>=0; ind--){
        for(int buy=0; buy<=1; buy++){
            if(buy==0){// We can buy the stock
                profit = max(0+ahead[0], -Arr[ind] + ahead[1]);
            }

            if(buy==1){// We can sell the stock
                profit = max(0+ahead[1], Arr[ind] -fee + ahead[0]);
            }
            cur[buy]  = profit;
        }

        ahead = cur;
    }
    return cur[0];

}

int main() {

  vector<int> prices = {1,3,2,8,4,9};
  int n = prices.size();
  int fee = 2;

  cout<<"The maximum profit that can be generated is
"<<maximumProfit(n,fee,prices);
}

Time Complexity: O(N*2)
Space Complexity: O(1)
```

TRY YOUR SELF (DP)

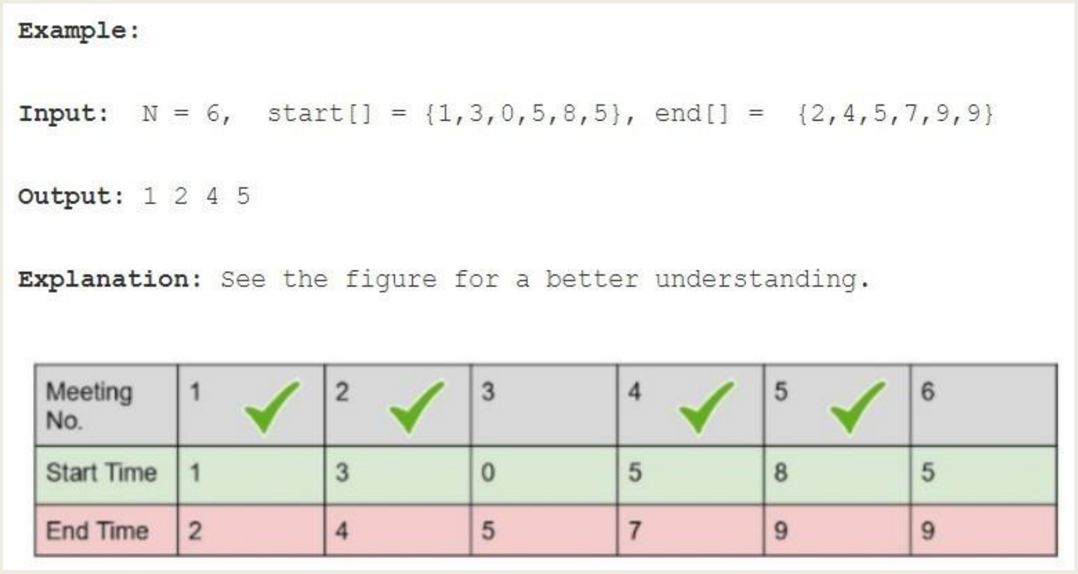
These topics in GFG or LEETCODE

1. Longest Increasing Subsequence
2. Printing Longest Increasing Subsequence
3. Longest Increasing Subsequence
4. Largest Divisible Subset
5. Longest String Chain
6. Longest Bitonic Subsequence
7. Number of Longest Increasing Subsequences
8. Minimum Cost to Cut the Stick
9. Burst Balloons

GREEDY

N meetings in one room

Problem Statement: There is one meeting room in a firm. You are given two arrays, start and end each of size N.For an index 'i', start[i] denotes the starting time of the ith meeting while end[i] will denote the ending time of the ith meeting. Find the maximum number of meetings that can be accommodated if only one meeting can happen in the room at a particular time. Print the order in which these meetings will be performed.



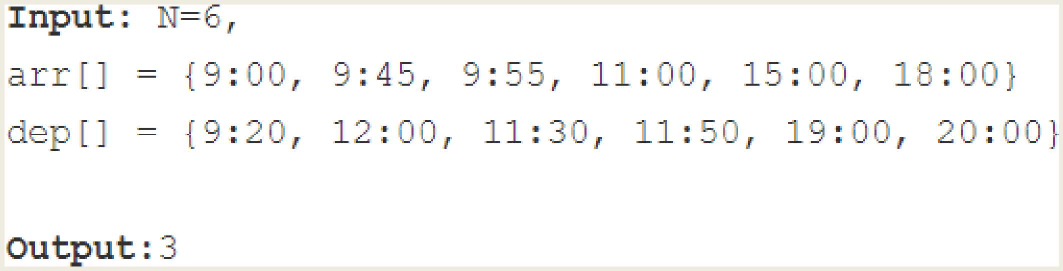
```
Example:

Input:  N = 6,  start[] = {1,3,0,5,8,5}, end[] =  {2,4,5,7,9,9}

Output: 1 2 4 5

Explanation: See the figure for a better understanding.
```

Meeting No.	1 ✓	2 ✓	3	4 ✓	5 ✓	6
Start Time	1	3	0	5	8	5
End Time	2	4	5	7	9	9

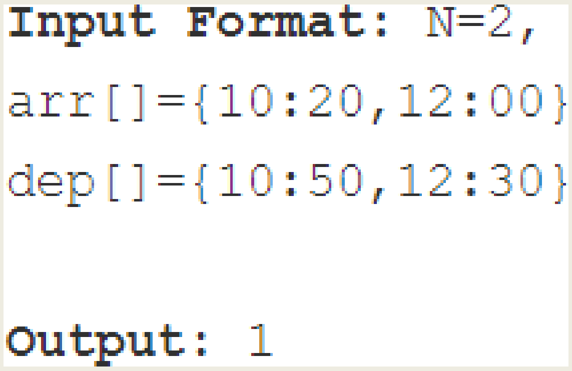
```
#include <bits/stdc++.h>
using namespace std;

struct meeting {
   int start;
   int end;
   int pos;
};

class Solution {
   public:
      bool static comparator(struct meeting m1, meeting m2) {
         if (m1.end < m2.end) return true;
         else if (m1.end > m2.end) return false;
         else if (m1.pos < m2.pos) return true;
         return false;
      }
   void maxMeetings(int s[], int e[], int n) {
      struct meeting meet[n];
      for (int i = 0; i < n; i++) {
         meet[i].start = s[i], meet[i].end = e[i], meet[i].pos = i +
1;
      }
      sort(meet, meet + n, comparator);

      vector < int > answer;

      int limit = meet[0].end;
      answer.push_back(meet[0].pos);

      for (int i = 1; i < n; i++) {
         if (meet[i].start > limit) {
            limit = meet[i].end;
            answer.push_back(meet[i].pos);
         }
      }
      cout<<"The order in which the meetings will be performed is
"<<endl;
      for (int i = 0; i < answer.size(); i++) {
         cout << answer[i] << " ";
      }

   }

};
int main() {
   Solution obj;
   int n = 6;
   int start[] = {1,3,0,5,8,5};
   int end[] = {2,4,5,7,9,9};
   obj.maxMeetings(start, end, n);
   return 0;
}

# Time Complexity: O(n) to iterate through every position and insert
them in a data structure. O(n log n)  to sort the data structure in
ascending order of end time.
# O(n)  to iterate through the positions and check which meeting can
be performed.

# Overall : O(n) +O(n log n) + O(n) ~O(n log n)

# Space Complexity: O(n) since we used an additional data structure
for storing the start time, end time, and meeting no.
```

Minimum number of platforms required for a railway

Problem Statement: We are given two arrays that represent the arrival and departure times of trains that stop at the platform. We need to find the minimum number of platforms needed at the railway station so that no train has to wait.



```
Input: N=6,
arr[] = {9:00, 9:45, 9:55, 11:00, 15:00, 18:00}
dep[] = {9:20, 12:00, 11:30, 11:50, 19:00, 20:00}

Output:3
```



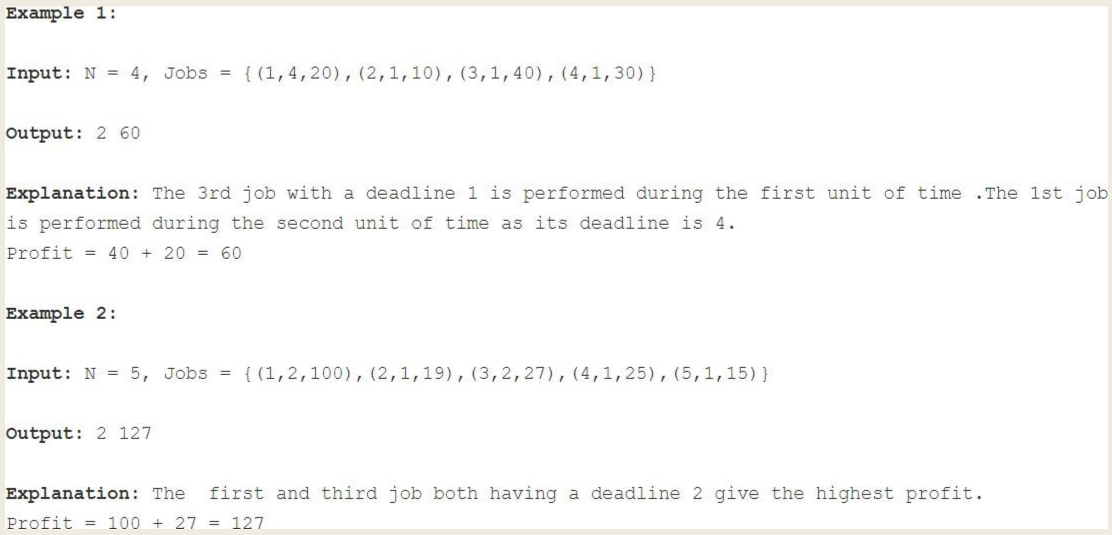
```
Input Format: N=2,
arr[]={10:20,12:00}
dep[]={10:50,12:30}


Output: 1
```

```
// through sorting

#include<bits/stdc++.h>
 using namespace std;

 int countPlatforms(int n,int arr[],int dep[])
 {
    sort(arr,arr+n);
    sort(dep,dep+n);

    int ans=1;
    int count=1;
    int i=1,j=0;
    while(i<n && j<n)
    {
        if(arr[i]<=dep[j]) //one more platform needed
        {
            count++;
            i++;
        }
        else //one platform can be reduced
        {
            count--;
            j++;
        }
        ans=max(ans,count); //updating the value with the current
maximum
```

```
    }
    return ans;
 }

 int main()
 {
    int arr[]={900,945,955,1100,1500,1800};
    int dep[]={920,1200,1130,1150,1900,2000};
    int n=sizeof(dep)/sizeof(dep[0]);
    cout<<"Minimum number of Platforms required
"<<countPlatforms(n,arr,dep)<<endl;
 }
# Output:

# Minimum number of Platforms required 3

# Time Complexity: O(nlogn) Sorting takes O(nlogn) and traversal of
arrays takes O(n) so overall time complexity is O(nlogn).

# Space complexity: O(1)  (No extra space used).
```

Job Sequencing Problem

Problem Statement: You are given a set of N jobs where each job comes with a deadline and profit. The profit can only be earned upon completing the job within its deadline. Find the number of jobs done and the maximum profit that can be obtained. Each job takes a single unit of time and only one job can be performed at a time.



```
Example 1:

Input: N = 4, Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}

Output: 2 60

Explanation: The 3rd job with a deadline 1 is performed during the first unit of time .The 1st job
is performed during the second unit of time as its deadline is 4.
Profit = 40 + 20 = 60

Example 2:

Input: N = 5, Jobs = {(1,2,100),(2,1,19),(3,2,27),(4,1,25),(5,1,15)}

Output: 2 127

Explanation: The  first and third job both having a deadline 2 give the highest profit.
Profit = 100 + 27 = 127
```

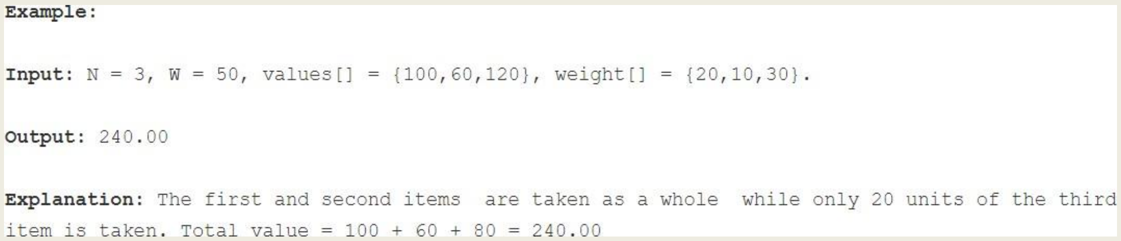
```
#include<bits/stdc++.h>

using namespace std;
// A structure to represent a job
struct Job {
   int id; // Job Id
   int dead; // Deadline of job
   int profit; // Profit if job is over before or on deadline
};
class Solution {
   public:
      bool static comparison(Job a, Job b) {
         return (a.profit > b.profit);
      }
   //Function to find the maximum profit and the number of jobs done
   pair < int, int > JobScheduling(Job arr[], int n) {

      sort(arr, arr + n, comparison);
      int maxi = arr[0].dead;
      for (int i = 1; i < n; i++) {
         maxi = max(maxi, arr[i].dead);
      }

      int slot[maxi + 1];

      for (int i = 0; i <= maxi; i++)
         slot[i] = -1;

      int countJobs = 0, jobProfit = 0;

      for (int i = 0; i < n; i++) {
         for (int j = arr[i].dead; j > 0; j--) {
            if (slot[j] == -1) {
               slot[j] = i;
               countJobs++;
               jobProfit += arr[i].profit;
               break;
            }
         }
      }

      return make_pair(countJobs, jobProfit);
   }
};
int main() {
   int n = 4;
   Job arr[n] = {{1,4,20},{2,1,10},{3,2,40},{4,2,30}};

   Solution ob;
   //function call
   pair < int, int > ans = ob.JobScheduling(arr, n);
   cout << ans.first << " " << ans.second << endl;

   return 0;
```

```
}

# Output: 3 90

# Time Complexity: O(N log N) + O(N*M).

# O(N log N ) for sorting the jobs in decreasing order of profit.
O(N*M) since we are iterating through all N jobs and for every job we
are checking from the last deadline,
# say M deadlines in the worst case.

# Space Complexity: O(M) for an array that keeps track on which day
which job is performed if M is the maximum deadline available.
```

Fractional Knapsack Problem : Greedy Approach

Problem Statement: The weight of N items and their corresponding values are given. We have to put these items in a knapsack of weight W such that the total value obtained is maximized.

Note: We can either take the item as a whole or break it into smaller units.



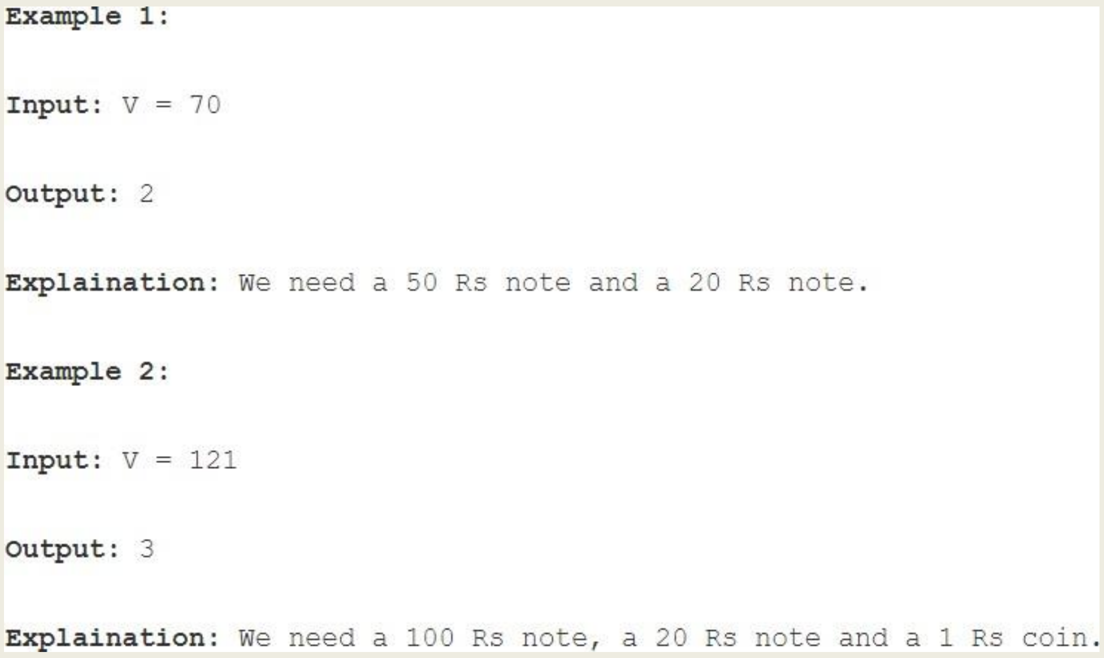
```
Example:

Input: N = 3, W = 50, values[] = {100,60,120}, weight[] = {20,10,30}.

Output: 240.00

Explanation: The first and second items  are taken as a whole  while only 20 units of the third
item is taken. Total value = 100 + 60 + 80 = 240.00
```

```
#include <bits/stdc++.h>

using namespace std;

struct Item {
   int value;
   int weight;
};
class Solution {
   public:
      bool static comp(Item a, Item b) {
         double r1 = (double) a.value / (double) a.weight;
         double r2 = (double) b.value / (double) b.weight;
         return r1 > r2;
      }
   // function to return fractionalweights
   double fractionalKnapsack(int W, Item arr[], int n) {

      sort(arr, arr + n, comp);
```

```
        int curWeight = 0;
        double finalvalue = 0.0;

        for (int i = 0; i < n; i++) {

            if (curWeight + arr[i].weight <= W) {
                curWeight += arr[i].weight;
                finalvalue += arr[i].value;
            } else {
                int remain = W - curWeight;
                finalvalue += (arr[i].value / (double) arr[i].weight) *
 (double) remain;
                break;
            }
        }

        return finalvalue;

    }
};
int main() {
    int n = 3, weight = 50;
    Item arr[n] = { {100,20},{60,10},{120,30} };
    Solution obj;
    double ans = obj.fractionalKnapsack(weight, arr, n);
    cout << "The maximum value is " << setprecision(2) << fixed << ans;
    return 0;
}

# Output:

# The maximum value is 240.00

# Time Complexity: O(n log n + n). O(n log n) to sort the items and
O(n) to iterate through all the items for calculating the answer.

# Space Complexity: O(1), no additional data structure has been used.
```

Find minimum number of coins

Problem Statement: Given a value V, if we want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency, i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change.



Example 1:

Input: V = 70

Output: 2

Explaination: We need a 50 Rs note and a 20 Rs note.

Example 2:

Input: V = 121

Output: 3

Explaination: We need a 100 Rs note, a 20 Rs note and a 1 Rs coin.

```
#include<bits/stdc++.h>


using namespace std;
int main() {
  int V = 49;
  vector < int > ans;
  int coins[] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
  int n = 9;
  for (int i = n - 1; i >= 0; i--) {
    while (V >= coins[i]) {
      V -= coins[i];
      ans.push_back(coins[i]);
    }
  }
  cout<<"The minimum number of coins is "<<ans.size()<<endl;
  cout<<"The coins are "<<endl;
  for (int i = 0; i < ans.size(); i++) {
    cout << ans[i] << " ";
  }

  return 0;
}

# Output:

# The minimum number of coins is 5
# The coins are
# 20 20 5 2 2
# Time Complexity:O(V)

# Space Complexity:O(1)
```

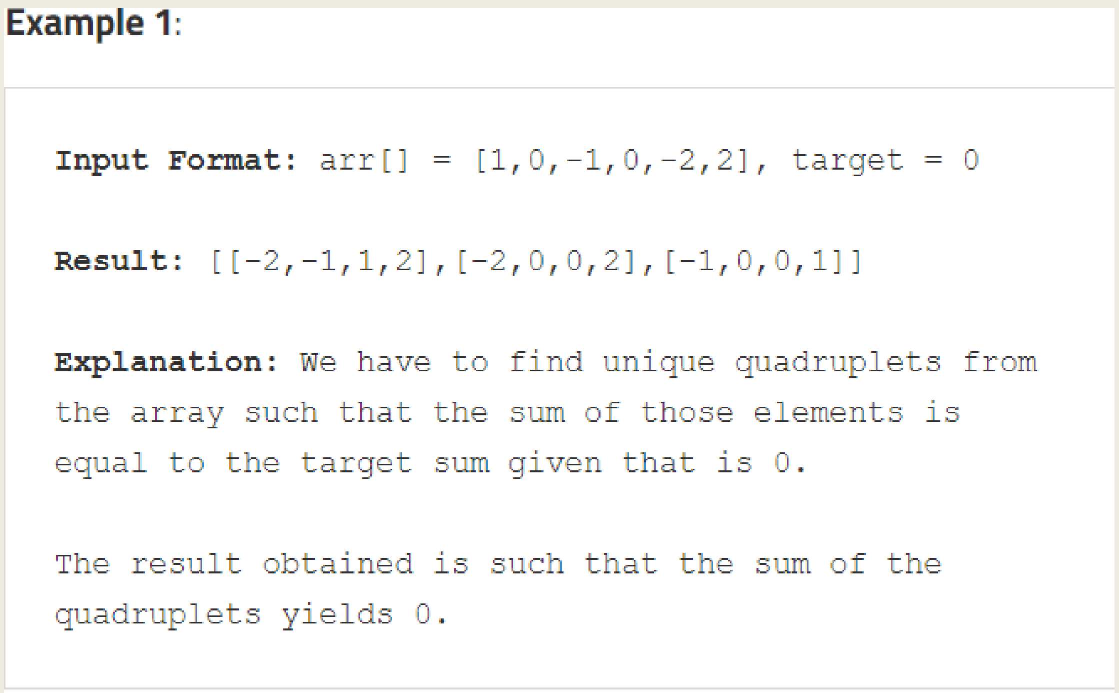
4 Sum | Find Quads that add up to a target value

Problem Statement: Given an array of N integers, your task is to find unique quads that add up to give a target value. In short, you need to return an array of all the unique quadruplets [arr[a], arr[b], arr[c], arr[d]] such that their sum is equal to a given target. _____

Pre-req: Binary Search and 2-sum problem

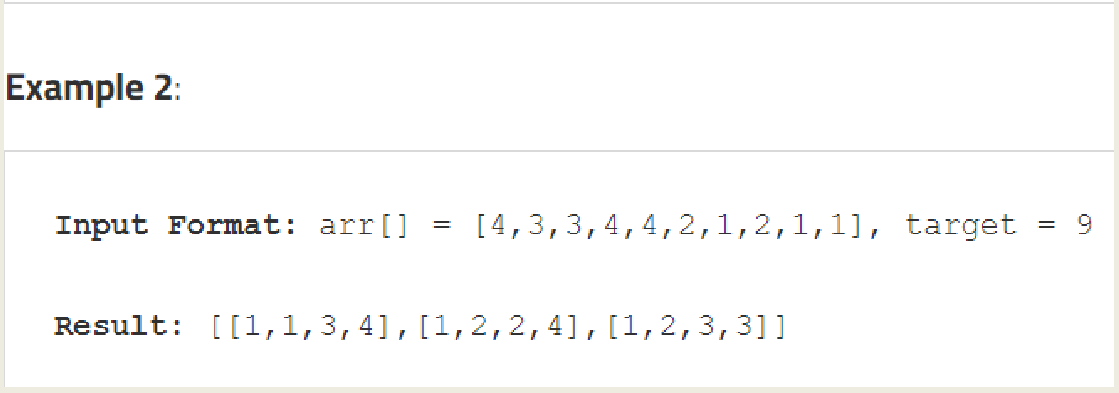
Note:

0 <= a, b, c, d < n ; a, b, c, and d are distinct. ; arr[a] + arr[b] + arr[c] + arr[d] == target



Example 1:

```
Input Format: arr[] = [1,0,-1,0,-2,2], target = 0

Result: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

Explanation: We have to find unique quadruplets from
the array such that the sum of those elements is
equal to the target sum given that is 0.

The result obtained is such that the sum of the
quadruplets yields 0.
```



Example 2:

```
Input Format: arr[] = [4,3,3,4,4,2,1,2,1,1], target = 9

Result: [[1,1,3,4],[1,2,2,4],[1,2,3,3]]
```

```
// using 3-pointers and Binary Search

#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        int n = nums.size();

        sort(nums.begin(),nums.end());

       set<vector<int>> sv;
        for(int i=0;i<n;i++)
        {
            for(int j=i+1;j<n;j++)
            {

                for(int k=j+1;k<n;k++)
                {

                   int x = (long long)target -
                           (long long)nums[i]-
                           (long long)nums[j]-(long long)nums[k];

                        if(binary_search(nums.begin()
+k+1,nums.end(),x)){
                            vector<int> v;
                            v.push_back(nums[i]);
                            v.push_back(nums[j]);
                            v.push_back(nums[k]);
                            v.push_back(x);
                            sort(v.begin(),v.end());
                            sv.insert(v);
                        }
                }
            }
        }
        vector<vector<int>> res(sv.begin(),sv.end());
        return res;
    }
};
int main()
{
    Solution obj;
    vector<int> v{1,0,-1,0,-2,2};

    vector<vector<int>> sum=obj.fourSum(v,0);
    cout<<"The unique quadruplets are"<<endl;
    for (int i = 0; i < sum.size(); i++) {
        for (int j = 0; j < sum[i].size(); j++)
```

```
            cout << sum[i][j] << " ";
        cout << endl;
    }
}
Output:

The unique quadruplets are
-2 -1 1 2
-2 0 0 2
-1 0 0 1

# Time Complexity: O(N log N + N³ logN)
# Reason: Sorting the array takes N log N and three nested loops will
be taking N³ and for binary search, it takes another log N.
# Space Complexity: O(M * 4), where M is the number of quads
```

```
// Optimised Approach

# Sort the array, and then fix two pointers, so the remaining sum will
be (target – (nums[i] + nums[j])).
# Now we can fix two-pointers, one front, and another back, and easily
use a two-pointer to find the remaining two numbers of the quad.
# In order to avoid duplications, we jump the duplicates, because
taking duplicates will result in repeating quads.
# We can easily jump duplicates, by skipping the same elements by
running a loop.


#include<bits/stdc++.h>
using namespace std;
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& num, int target) {
         vector<vector<int> > res;

        if (num.empty())
            return res;
        int n = num.size();
        sort(num.begin(),num.end());

        for (int i = 0; i < n; i++) {

            int target_3 = target - num[i];

            for (int j = i + 1; j < n; j++) {
                int target_2 = target_3 - num[j];

                int front = j + 1;
                int back = n - 1;

                while(front < back) {

                    int two_sum = num[front] + num[back];

                    if (two_sum < target_2) front++;

                    else if (two_sum > target_2) back--;

                    else {

                        vector<int> quadruplet(4, 0);
                        quadruplet[0] = num[i];
                        quadruplet[1] = num[j];
                        quadruplet[2] = num[front];
                        quadruplet[3] = num[back];
                        res.push_back(quadruplet);

                        // Processing the duplicates of number 3
                        while (front < back && num[front] ==
quadruplet[2]) ++front;

                        // Processing the duplicates of number 4
                        while (front < back && num[back] ==
quadruplet[3]) --back;

                    }
                }

                // Processing the duplicates of number 2
                while(j + 1 < n && num[j + 1] == num[j]) ++j;
            }

            // Processing the duplicates of number 1
            while (i + 1 < n && num[i + 1] == num[i]) ++i;

        }

        return res;
    }
};

int main()
{
    Solution obj;
    vector<int> v{1,0,-1,0,-2,2};

    vector<vector<int>> sum=obj.fourSum(v,0);
    cout<<"The unique quadruplets are"<<endl;
    for (int i = 0; i < sum.size(); i++) {
        for (int j = 0; j < sum[i].size(); j++)
            cout << sum[i][j] << " ";
        cout << endl;
    }
}
```

SEGMENT TREE

```
int getMid(int s, int e) { return s + (e -s)/2; }

//......................................min query segment tree .........................................

int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{

	if (qs <= ss && qe >= se)
		return st[index];

	if (se < qs || ss > qe)
		return INT_MAX;

	int mid = getMid(ss, se);
	return min(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
				RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}

int RMQ(int *st, int n, int qs, int qe)
{

	if (qs < 0 || qe > n-1 || qs > qe)
	{
//		cout<<"Invalid Input";
		return 0;
	}
	return RMQUtil(st, 0, n-1, qs, qe, 0);
}


int constructSTUtil(vector<int>arr, int ss, int se,
								int *st, int si)
{

	if (ss == se)
	{
		st[si] = arr[ss];
		return arr[ss];
	}

	int mid = getMid(ss, se);
	st[si] = min(constructSTUtil(arr, ss, mid, st, si*2+1),
constructSTUtil(arr, mid+1, se, st, si*2+2));
	return st[si];
}


int *constructST(vector<int>arr, int n)
{
	int x = (int)(ceil(log2(n)));
	int max_size = 2*(int)pow(2, x) - 1;
	int *st = new int[max_size];
	constructSTUtil(arr, 0, n-1, st, 0);
	return st;
}


//..........................................max query segment tree ...................................

int RMQUtil1(int *st, int ss, int se, int qs, int qe, int index)
{

	if (qs <= ss && qe >= se)
		return st[index];

	if (se < qs || ss > qe)
		return INT_MAX;

	int mid = getMid(ss, se);
	return max(RMQUtil1(st, ss, mid, qs, qe, 2*index+1),
				RMQUtil1(st, mid+1, se, qs, qe, 2*index+2));
}

int RMQ1(int *st, int n, int qs, int qe)
{

	if (qs < 0 || qe > n-1 || qs > qe)
	{
//		cout<<"Invalid Input";
		return 0;
	}

	return RMQUtil1(st, 0, n-1, qs, qe, 0);
}


int constructSTUtil1(vector<int>arr, int ss, int se,
								int *st, int si)
{

	if (ss == se)
	{
		st[si] = arr[ss];
		return arr[ss];
	}


	int mid = getMid(ss, se);
```

```
      st[si] = max(constructSTUtil1(arr, ss, mid, st, si*2+1),
                              constructSTUtil1(arr, mid+1, se, st,
si*2+2));
      return st[si];
}


int *constructST1(vector<int>arr, int n)
{

      int x = (int)(ceil(log2(n)));

      int max_size = 2*(int)pow(2, x) - 1;

      int *st = new int[max_size];

      constructSTUtil1(arr, 0, n-1, st, 0);

      return st;
}

int main()
{
    vector<int> arr={1,5,2,7,9};
    int *st= constructST(arr, n);

    # RMQ(st, n, lower_index, upper_index) for min st
    # RMQ1(st, n, lower_index, upper_index) for min st

}
```

SLIDING WINDOW

FIXED WINDOW

Maximum sum subarray of size K

```
//Time Complexity : O(n)
//Auxiliary Space : O(1)
class Solution{
  public:
int maximumSumSubarray(int K, vector<int> &Arr , int N){
  int i=0;
  int j=0;
  int sum=0;
  int mx=INT_MIN;
  while (j<N){
    sum = sum + Arr[j]; // do calculation to reduse tc
    if (j-i+1 < K) j++; // increament j upto when the size of the size
of window is not equal to required size
    else if ((j-i+1) == K) // when sindow size hit to the required
window size
    {
      mx = max(mx,sum); // selecting ans from the candidates
      sum = sum - Arr[i]; // start removing from the first
      i++;
      j++;
    }
  }
  return mx;
}
};
```

1st negative number of every window of size K

```
/*
Time Complexity: O(n)
Auxiliary Space: O(k)
*/
vector<long long> printFirstNegativeInteger(long long int A[],long
long int N, long long int K) {
 long long i;
 long long j;
 vector<long long> ans;
 list<long long> l;
  while (j<N){
    if (A[j]<0)
      l.push_back(A[j]);
    if (j-i+1<K) j++;
    else if ((j-i+1)==K)
    {
      if (l.size()==0)
```

```
        ans.push_back(0);
      else
        ans.push_back(l.front());
      if(A[i]<0)
        l.pop_front();
      i++;
      j++;
    }
  }
  return ans;
}
```

Count occurrence of Anagrams

all the permutations of a string pat are anagrams of the string pat, and we need to find whether the anagrams of pat is present in string txt.

in short, anagram of a string is

1. length of string before = length of string after
2. frequency of each letters before = frequency of each letters after

```
class Solution{
  public:

  int search(string pat, string txt) {
  unordered_map <char, int=""> m;
  for(auto i: pat)
  m[i]++;

  int k = pat.size();
  int count = m.size();
  int ans = 0;
  int i = 0, j = 0;

  while(j < txt.size()) {

  if(m.find(txt[j]) != m.end()) {
  m[txt[j]]--;

  if(m[txt[j]] == 0)
  count--;
  }

  if(j - i + 1 < k) j++;

  else if(j - i + 1 == k) {
  if(count == 0)
  ans++;
  if(m.find(txt[i]) != m.end()) {
  m[txt[i]]++;

  if(m[txt[i]] == 1)
  count++;
  }

  i++; j++;
  }
  }

  return ans;
  }
};
```

Maximum of all subarrays of size K

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
       vector<int> ans;
        list<int> l;
        int i=0;
        int j=0;

        if (k>nums.size()) // edge case
        {
            ans.push_back(*max_element(l.begin(),l.end()));
            return ans;
        }

        while (j<nums.size())
        {
            while(l.size()>0 && l.back() <nums[j])
            {
                l.pop_back();
            }
            l.push_back(nums[j]);
            if ((j-i+1)<k)
                j++;
            else if (j-i+1==k)
            {
                ans.push_back(l.front());
                if (l.front()==nums[i])
                    l.pop_front();
                i++;
                j++;
            }

        }
        return ans;
    }
};
```

VARIABLE WINDOW

Largest subarray of sum k

```
#include<bits/stdc++.h>

using namespace std;

int longestSubArrWithSumK_Optimal(int arr[], int n, int k) {
  int start = 0, end = -1, sum = 0, maxLength = 0;
  while (start < n) {
    while ((end + 1 < n) && (sum + arr[end + 1] <= k))
      sum += arr[++end];

    if (sum == k)
      maxLength = max(maxLength, (end - start + 1));

    sum -= arr[start];
    start++;
  }
  return maxLength;
}

int main() {

  int arr[] = {7,1,6,0};
  int n = sizeof(arr) / sizeof(arr[0]), k = 7;

  cout << "Length of the longest subarray with sum K is " <<
  longestSubArrWithSumK_Optimal(arr, n, k);

  return 0;
}
```

Longest substring with K unique characters

Input: S = "ababcbcca", K = 2

Output: 5

Explanation: The longest substring with at most 2 distinct characters is "bcbcc" with characters 'b' and 'c'.

Example 2:

Input: S = abcde, K = 1

Output: 1

Explanation: Since all the characters of the strings are different, the substring length with K distinct characters depends on K itself.

```
int KDistinct(string input, int K)
{
      // define a hash table to store the frequencies
      // of characters in the window
      unordered_map<char, int> table;

      // variable to mark start and end of a window
      int start = 0, end = 0;

      // define a variable to store the result
      int longest = 0;

      // one by one add elements to the window
      for (end = 0; end < input.size(); end++)
      {
            // the new character
            char newCharacter = input[end];

            // add the new character in the hash table
            table[newCharacter]++;

            // check if number of distinct characters in window is
            // more than K
            while (table.size() > K)
            {
                  char startCharacter = input[start];
                  // shrink the window
                  start++;
                  table[startCharacter]--;

                  if (table[startCharacter] == 0)
                  {
                        table.erase(startCharacter);
                  }
            }
            longest = max(longest, end - start + 1);
      }
      return longest;
}

int main()
{
      string input = "ababcbcca";
      int K = 2;
      cout << KDistinct(input, K) << endl;
      return 0;
 }
```

Longest substring with/without repeating characters

```
# Longest substring without repeating characters
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> mp;
        int i=0;
        int j=0;
        int mx=INT_MIN;
        if (s.size()==0) return 0;
        while(j<s.size())
        {
            mp[s[j]]++;
            if (mp.size()==j-i+1)
            {
                mx=max(mx,j-i+1);
            }
            else if (mp.size()<j-i+1)
            {
                while(mp.size()<j-i+1)
                {
                    mp[s[i]]--;
                    if (mp[s[i]]==0)
                    {
                        mp.erase(s[i]);
                    }
                    i++;
                }

            }
            j++;
        }
        return mx;
    }
};
```

THANK YOU