C++ Programs

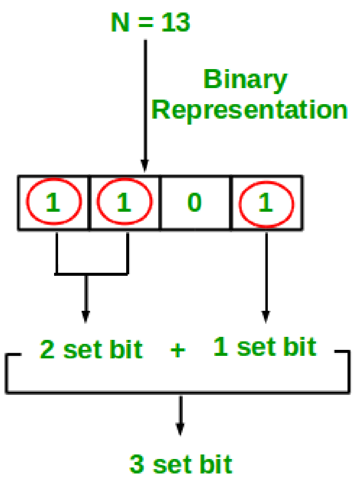
Contents

- ## Bit Manipulation Algorithms

❖ Number of mismatching bits in the binary representation of two integers

```
// compute number of different bits
int solve(int A, int B)
{
    int XOR = A ^ B;
    // Check for 1's in the binary form using
    int count = 0;
    while (XOR) {
        XOR = XOR & (XOR - 1);
        count++;
    }
    return count;
}
```

❖ Count set bits in an integer



```
unsigned int countSetBits(unsigned int n)
{
    unsigned int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}
```

- ## Linked list

❖ Count nodes of linked list

```
int getCount(struct Node* head)
{
    int count = 0;
    struct Node* temp = head;
    if (temp == NULL)
        return 0;
    else
    {
        while (temp != NULL)
        {
            count++;
            temp = temp->next;
        }
    }
    return count;
}
```

❖ Linked List Length Even or Odd?

```
// Function should return 0 is length is even else return 1
int isLengthEvenOrOdd(struct Node* head)
{
    //Code here
    int count = 0;
    struct Node* temp = head;
    if (temp == NULL)
        return 0;
    else
    {
        while (temp != NULL)
        {
            count++;
            temp = temp->next;
        }
    }
    return (count % 2);
}
```

❖ Inserting a node

```
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

❖ Deleting a node

```
void deleteNode(Node** head_ref, int key)
{
    // Store head node
    Node* temp = *head_ref;
    Node* prev = NULL;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next; // Changed head
        delete temp;            // free old head
        return;
    }

    // Else Search for the key to be deleted, keep track of the previous node as we need to
change 'prev->next' */
    else
    {
        while (temp != NULL && temp->data != key)
        {
            prev = temp;
            temp = temp->next;
        }

        // If key was not present in linked list
        if (temp == NULL)
            return;

        // Unlink the node from linked list
        prev->next = temp->next;
        // Free memory
        delete temp;
    }
}
```

❖ Delete a Linked List node at a given position

```
void deleteNode(Node** head_ref, int position)
{
    // If linked list is empty
    if (*head_ref == NULL)
        return;

    // Store head node
    Node* temp = *head_ref;

    // If head needs to be removed
    if (position == 0)
    {
        // Change head
        *head_ref = temp->next;

        // Free old head
        free(temp);
        return;
    }
    // Find previous node of the node to be deleted
    for (int i = 0; temp != NULL && i < position - 1; i++)
        temp = temp->next;

    // If position is more than number of nodes
    if (temp == NULL || temp->next == NULL)
        return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    Node* next = temp->next->next;

    // Unlink the node from linked list
    free(temp->next); // Free memory

    // Unlink the deleted node from list
    temp->next = next;
}
```

❖ Reverse singly linked list

```
void reverseList()
{
    CNode* search_Node = start;
    CNode* rev_List = nullptr;
    CNode* last_Node;
    while (search_Node != nullptr)
    {
        last_Node = rev_List;
        rev_List = search_Node;
        search_Node = search_Node->next;
        rev_List->next = last_Node;
    }
    start = rev_List;
}
```

❖ Remove duplicates from singly linked list

```
void removeDuplicates(CNode* start)
{
    CNode* ptr1, * ptr2, * dup;
    ptr1 = start;
    while (ptr1 != nullptr ptr1->next != nullptr)
    {
        ptr2 = ptr1;
        while (ptr2->next != nullptr)
        {
            if (ptr1->data == ptr2->next->data)
            {
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;
                delete dup;
            }
            else // Tricky
            {
                ptr2 = ptr2->next;
            }
        }
        ptr1 = ptr1->next;
    }
}
```

❖ Find the middle of a given linked list

```
void printMiddle(class Node* head) {
    struct Node* slow_ptr = head;
    struct Node* fast_ptr = head;

    if (head != NULL)
    {
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;
            slow_ptr = slow_ptr->next;
        }
        cout << "The middle element is [" << slow_ptr->data << "]" << endl;
    }
}
```

❖ Linked list using smart pointer

```
class CNode
{
public:
    CNode(int data) :m_data(data) {}
    int m_data;
    shared_ptr<CNode> next;
```

```cpp
};
shared_ptr<CNode> head_ptr;

shared_ptr<CNode> createNode(int data)
{
    shared_ptr<CNode> newNode = make_shared<CNode>(data);
    return newNode;
}

void insert(int data)
{
    if (head_ptr.get() == nullptr)
    {
        head_ptr = createNode(data);
    }
    else
    {
        shared_ptr<CNode> temp = head_ptr;
        while (temp.get()->next)
        {
            temp = temp->next;
        }
        temp->next = createNode(data);
    }
}

void deleteNode (int data)
{
    if (head_ptr.get() && head_ptr.get()->m_data == data)
    {
        shared_ptr<CNode> del = head_ptr;
        head_ptr = head_ptr.get()->next;
        del.reset();
    }
    else
    {
        shared_ptr<CNode> temp = head_ptr;
        while (temp.get()->next)
        {
            if (temp.get()->next.get()->m_data == data)
            {
                shared_ptr<CNode> del = temp.get()->next;
                temp->next = del->next;
                del.reset();
                break;
            }
            temp = temp->next;
        }
    }
}

bool searchItem(int data)
{
    shared_ptr<CNode> temp = head_ptr;
    while (temp.get())
    {
        if (temp.get()->m_data == data)
            return true;
        temp = temp->next;
    }
    return false;
}

void printList(shared_ptr<CNode> head)
{
    shared_ptr<CNode> temp = head;
    while (temp.get())
    {
        cout << temp.get()->m_data << endl;
        temp = temp->next;
    }
}

int main()
{
    insert(10);
    insert(30);
    insert(20);
    insert(40);
    bool found = searchItem(20);
    if (found)
        deleteNode(20);
    printList(head_ptr);
    return 0;
}
```

❖ Function to check if a singly linked list is palindrome

This method takes O(n) time and O(1) extra space.
1) Get the middle of the linked list.
2) Reverse the second half of the linked list.
3) Check if the first half and second half are identical.
4) Construct the original linked list by reversing the second half again and attaching it back to the first half.

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

    ListNode* temp = head;
    // Iterate to count odd/even
    int n = 0;
    while (temp != NULL)
    {
        temp = temp->next;
        n++;
    }
    temp = head;
    // Find the mid element
    ListNode* head_mid = find_middle(temp, n);
    // Reverse the second half linked-list
    ListNode* head_rev = reverse_link(head_mid);
    // Verify first half and second half of linked-list are equivalent
    while (head_rev != NULL)
    {

        if (head->val != head_rev->val)
            return false;

        head_rev = head_rev->next;
        head = head->next;
    }
    return true;
}
```

❖ Detect and Remove Loop in a Linked List

```
void detectAndRemoveLoop(Node* head)
{
    // If list is empty or has only one node
    // without loop
    if (head == NULL || head->next == NULL)
        return;

    Node* slow = head, * fast = head;

    // Move slow and fast 1 and 2 steps ahead respectively.
    slow = slow->next;
    fast = fast->next->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next) {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* If loop exists */
    if (slow == fast)
    {
        slow = head;

        // this check is needed when slow and fast both meet at the head of the LL
          // eg: 1->2->3->4->5 and then 5->next = 1 i.e the head of the LL
        if (slow == fast) {
            while (fast->next != slow) fast = fast->next;
        }
        else {
            while (slow->next != fast->next) {
                slow = slow->next;
                fast = fast->next;
            }
        }

        /* since fast->next is the looping point */
        fast->next = NULL; /* remove loop */
    }
}
```

❖ Best Possible algorithm to check if two linked lists are merging at any point? If so, where?

Sol: Walk those two lists parallel by one element, add each element to Set of visited nodes (can be hash map, or simple set, you only need to check if you visited that node before). At each step check if you visited that node (if yes, then it's merging point), and add it to set of nodes if you visit it first time. Another version (as pointed by @reinier) is to walk only first list, store its nodes in Set and then only check second list against that Set. First approach is faster when your lists merge early, as you don't need to store all nodes from first list. Second is better at worst case, where both list don't merge at all, since it didn't store nodes from second list in Set

• Segregate programs

❖ Function to segregate 0s and 1s

```
void segregate0and1(int arr[], int n)
{
    int count = 0; // Counts the no of zeros in arr
    for (int i = 0; i < n; i++) {
        if (arr[i] == 0)
            count++;
    }

    // Loop fills the arr with 0 until count
    for (int i = 0; i < count; i++)
        arr[i] = 0;

    // Loop fills remaining arr space with 1
    for (int i = count; i < n; i++)
        arr[i] = 1;
}
```

❖ Function to segregate 0s , 1s and 2

```
void sort012(int a[], int arr_size)
{
    int lo = 0;
    int hi = arr_size - 1;
    int mid = 0;

    // Iterate till all the elements are sorted
    while (mid <= hi) {
        switch (a[mid]) {

            // If the element is 0
        case 0:
            swap(a[lo++], a[mid++]);
            break;

            // If the element is 1 .
        case 1:
            mid++;
            break;

            // If the element is 2
        case 2:
            swap(a[mid], a[hi--]);
            break;
        }
    }
}
```

❖ Segregate Even and Odd numbers

```
void segregateEvenOdd(int arr[], int size)
{
    /* Initialize left and right indexes */
    int left = 0, right = size - 1;
    while (left < right)
    {
        /* Increment left index while we see 0 at left */
        while (arr[left] % 2 == 0 && left < right)
            left++;

        /* Decrement right index while we see 1 at right */
        while (arr[right] % 2 == 1 && left < right)
            right--;

        if (left < right)
        {
            /* Swap arr[left] and arr[right]*/
            swap(&arr[left], &arr[right]);
            left++;
            right--;
        }
    }
}
```

❖ Segregate even and odd nodes in a Linked List

```
void segregateEvenOdd(Node** head_ref)
{
    Node* end = *head_ref;
    Node* prev = NULL;
    Node* curr = *head_ref;

    /* Get pointer to the last node */
    while (end->next != NULL)
        end = end->next;

    Node* new_end = end;

    /* Consider all odd nodes before the first even node and move then after end */
    while (curr->data % 2 != 0 && curr != end)
    {
        new_end->next = curr;
        curr = curr->next;
        new_end->next->next = NULL;
        new_end = new_end->next;
    }

    // 10->8->17->17->15
    /* Do following steps only if there is any even node */
    if (curr->data % 2 == 0)
    {
        /* Change the head pointer to point to first even node */
        *head_ref = curr;

        /* now current points to the first even node */
        while (curr != end)
        {
            if ((curr->data) % 2 == 0)
            {
                prev = curr;
                curr = curr->next;
            }
            else
            {
                /* break the link between prev and current */
                prev->next = curr->next;

                /* Make next of curr as NULL */
                curr->next = NULL;

                /* Move curr to end */
                new_end->next = curr;

                /* make curr as new end of list */
                new_end = curr;

                /* Update current pointer to next of the moved node */
                curr = prev->next;
            }
        }
    }
    /* We must have prev set before executing lines following this statement */
    else prev = curr;

    /* If there are more than 1 odd nodes and end of original list is odd then
    move this node to end to maintain same order of odd numbers in modified list */
    if (new_end != end && (end->data) % 2 != 0)
    {
        prev->next = end->next;
        end->next = NULL;
        new_end->next = end;
    }
    return;
}
```

❖ Move all zeros to end of the array

```
void pushZerosToEnd(int arr[], int n)
{
    int count = 0;  // Count of non-zero elements

    for (int i = 0; i < n; i++)
        if (arr[i] != 0)
            arr[count++] = arr[i]; // here count is
                                   // incremented

    // Now all non-zero elements have been shifted to front and 'count'
    // is set as index of first 0. Make all elements 0 from count to end.
    while (count < n)
        arr[count++] = 0;
}
```

❖ Function to print the second largest elements

```
void print2largest(int arr[], int arr_size)
{
    int i, first, second;

    /* There should be atleast two elements */
    if (arr_size < 2) {
        printf(" Invalid Input ");
        return;
    }

    first = second = INT_MIN;
    for (i = 0; i < arr_size; i++) {
        /* If current element is greater than first then update both first and second */
        if (arr[i] > first) {
            second = first;
            first = arr[i];
        }

        /* If arr[i] is in between first and second then update second. */
        else if (arr[i] > second && arr[i] != first)
            second = arr[i];
    }
    if (second == INT_MIN)
        printf("There is no second largest element\n");
    else
        printf("The second largest element is %dn", second);
}
```

❖ K-th Smallest/Largest Element in an Unsorted Array using Priority Queue

```
int kthLargeElement(int arr[], int size, int k)
{
```

```cpp
    if (size < k)
        return -1;
    priority_queue<int, std::vector<int>, std::greater<int>> pq;
    // For kth smallest use std::less

    for (int i = 0; i < size; ++i) {
        // Insert elements into the priority queue
        pq.push(arr[i]);

        // If size of the priority queue exceeds k
        if (pq.size() > k) {
            pq.pop();
        }
    }
    return pq.top();
}
```

- ## Stack & Queue

❖ Implement Stack using Queues

```cpp
class Stack {
    // Two inbuilt queues
    queue<int> q1, q2;

    // To maintain current number of elements
    int curr_size;

public:
    Stack()
    {
        curr_size = 0;
    }

    void push(int x)
    {
        curr_size++;

        // Push x first in empty q2
        q2.push(x);

        // Push all the remaining elements in q1 to q2.
        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }

        // swap the names of two queues
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
    }

    void pop()
    {
        // if no elements are there in q1
        if (q1.empty())
            return;
        q1.pop();
        curr_size--;
    }

    int top()
    {
        if (q1.empty())
            return -1;
        return q1.front();
    }

    int size()
    {
        return curr_size;
    }
};
```

❖ Implement a stack using single queue

```
// Push operation
void Stack::push(int val)
{
    //  Get previous size of queue
    int s = q.size();

    // Push current element
    q.push(val);

    // Pop (or Dequeue) all previous elements and put them after current
    // element
    for (int i = 0; i < s; i++)
    {
        // this will add front element into rear of queue
        q.push(q.front());

        // this will delete front element
        q.pop();
    }
}

// Removes the top element
void Stack::pop()
{
    if (q.empty())
        cout << "No elements\n";
    else
        q.pop();
}

// Returns top of stack
int  Stack::top()
{
    return (q.empty()) ? -1 : q.front();
}

// Returns true if Stack is empty else false
bool Stack::empty()
{
    return (q.empty());
}
```

❖ Queue using Stacks (People tech)

```
struct Queue {
    stack<int> s1, s2;
    void enQueue(int x)
    {
        // Move all elements from s1 to s2
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }

        // Push item into s1
        s1.push(x);

        // Push everything back to s1
        while (!s2.empty()) {
            s1.push(s2.top());
            s2.pop();
        }
    }

    // Dequeue an item from the queue
    int deQueue()
    {
        // if first stack is empty
        if (s1.empty()) {
            cout << "Q is Empty";
            exit(0);
        }

        // Return top of s1
        int x = s1.top();
        s1.pop();
        return x;
    }
};
```

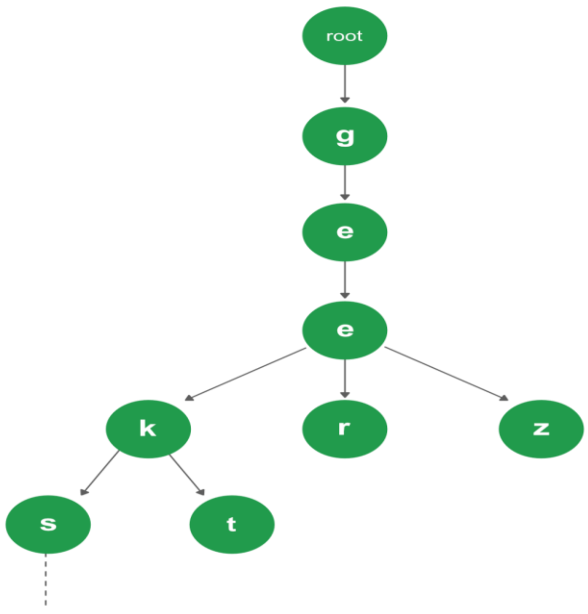
❖ Design a stack that supports getMin() in O(1) time and O(1) extra space

```
struct MyStack
{
    stack<int> s;
    int minEle;
    void getMin()
    {
        if (s.empty())
            cout << "Stack is empty\n";

        // variable minEle stores the minimum element in the stack.
        else
            cout << "Minimum Element in the stack is: "
            << minEle << "\n";
    }

    // Prints top element of MyStack
    void peek()
    {
        if (s.empty())
        {
            cout << "Stack is empty ";
            return;
        }

        int t = s.top(); // Top element.

        cout << "Top Most Element is: ";

        // If t < minEle means minEle stores value of t.
        (t < minEle) ? cout << minEle : cout << t;
    }

    // Remove the top element from MyStack
    void pop()
    {
        if (s.empty())
        {
            cout << "Stack is empty\n";
            return;
        }
        cout << "Top Most Element Removed: ";
        int t = s.top();
        s.pop();
        // Minimum will change as the minimum element of the stack is being removed.
        if (t < minEle)
        {
            cout << minEle << "\n";
            minEle = 2 * minEle - t;
        }

        else
            cout << t << "\n";
    }
    // Removes top element from MyStack
    void push(int x)
    {
        // Insert new number into the stack
        if (s.empty())
        {
            minEle = x;
            s.push(x);
            cout << "Number Inserted: " << x << "\n";
            return;
        }
        // If new number is less than minEle
        else if (x < minEle)
        {
            s.push(2 * x - minEle);
            minEle = x;
        }
        else
            s.push(x);

        cout << "Number Inserted: " << x << "\n";
    }
};
```

- ## Trie Data Structure

Tries is a tree that stores strings. The maximum number of children of a node is equal to the size of the alphabet. Trie supports search, insert and delete operations in O(L) time where **L** is the length of the key.

Trie is an efficient information re**Trie**val data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in O(M) time. However the **penalty is on Trie storage requirements**



Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node.

```
struct TrieNode
{
     struct TrieNode *children[ALPHABET_SIZE];
     // isEndOfWord is true if the node
     // represents end of a word
     bool isEndOfWord;
};
```

- **Hashing**
  In hashing, we convert the key to a small value and the value is used to index data. Hashing supports search, insert and delete operations in O(L) time on average.

- **Why Trie? :-**
- With Trie, we can insert and find strings in *O(L)* time where *L* represent the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in open addressing and separate chaining)
- Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing.
- We can efficiently do prefix search (or auto-complete) with Trie.

- **Issues with Trie :-**
  The main disadvantage of tries is that they **need a lot of memory for storing the strings**. For each node we have too many node pointers(equal to number of characters of the alphabet)

❖ Trie | (Insert and Search)

```
// C++ implementation of search and insert operations on Trie
#include <bits/stdc++.h>
using namespace std;

const int ALPHABET_SIZE = 26;
```

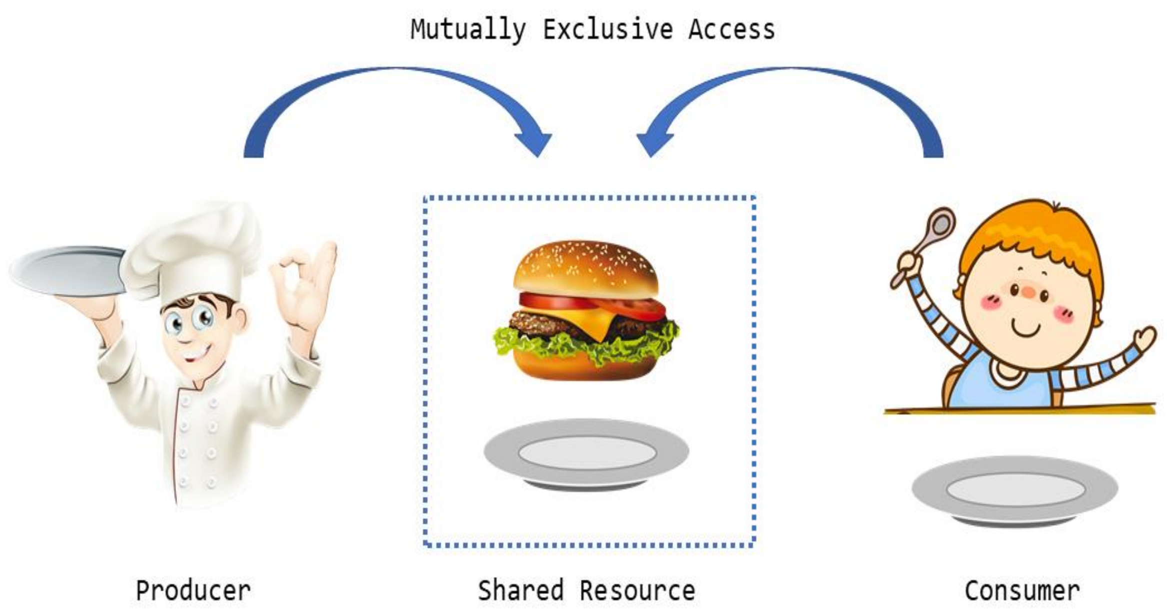
```cpp
// trie node
struct TrieNode
{
    struct TrieNode* children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode* getNode(void)
{
    struct TrieNode* pNode = new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}
// If not present, inserts key into trie If the key is prefix of trie node,
//just marks leaf node
void insert(struct TrieNode* root, string key)
{
    struct TrieNode* pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}
// Returns true if key presents in trie, else false
bool search(struct TrieNode* root, string key)
{
    struct TrieNode* pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl->isEndOfWord);
}
// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    string keys[] = { "the", "a", "there",
                     "answer", "any", "by",
                      "bye", "their" };
    int n = sizeof(keys) / sizeof(keys[0]);
    struct TrieNode* root = getNode();

    // Construct trie
    for (int i = 0; i < n; i++)
        insert(root, keys[i]);

    // Search for different keys
    search(root, "the") ? cout << "Yes\n" :
        cout << "No\n";
    search(root, "these") ? cout << "Yes\n" :
        cout << "No\n";
    return 0;
}
```

❖ Trie | (Delete)

```
// Recursive function to delete a key from given Trie
TrieNode* remove(TrieNode* root, string key, int depth = 0)
{
    // If tree is empty
    if (!root)
        return NULL;

    // If last character of key is being processed
    if (depth == key.size()) {

        // This node is no more end of word after
        // removal of given key
        if (root->isEndOfWord)
            root->isEndOfWord = false;

        // If given is not prefix of any other word
        if (isEmpty(root)) {
            delete (root);
            root = NULL;
        }

        return root;
    }

    // If not last character, recur for the child
    // obtained using ASCII value
    int index = key[depth] - 'a';
    root->children[index] =
        remove(root->children[index], key, depth + 1);

    // If root does not have any child (its only child got
    // deleted), and it is not end of another word.
    if (isEmpty(root) && root->isEndOfWord == false) {
        delete (root);
        root = NULL;
    }

    return root;
}
```

- **Multithreading**

❖ Producer-Consumer Problem Using Mutex in C++



The critical point is the producer and the consumer both need to access the shared resource (buffer) without knowing what the other is doing. We must handle the synchronization properly so that **no data provided by the producer is lost** (producer generates the next product before consumer accept**) or duplicated** (producer fails to update before consumer consume obtain next one).

❖ Program

e.g.1

```
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <thread>
#include <chrono>

std::vector vec;
std::mutex m;
std::condition_variable cv;
const int bufferSize = 10;

void consumer()
{
    while (true)
    {
        std::unique_lock<std::mutex> ul(m);
        cv.wait(ul, [] {return vec.size() > 0; })
        int val = vec.back();
        vec.pop_back();
        cout << "Consumed" << val;
       bufferSize--;
        ul.unlock();
        cv.notifyOne();
    }
}
void producer(int val)
```

```
{
    while (val > 0)
    {
        std::unique_lock<std::mutex> ul(m);
        cv.wait(ul, [] {return vec.size() < bufferSize; })
        vec.push_back(val);
        cout << "Produced" << val;
       bufferSize++;
        ul.unlock();
        cv.notifyOne();
    }
}

int main()
{
    tread t1(consumer, 1);
    tread t2(producer, 2);
    t1.join();
    t2.join();
    return 0;
}
```

e.g.2

```
std::mutex g_mutex;
bool g_ready = false;
int g_data = 0;


int produceData() {
    int randomNumber = rand() % 1000;
    std::cout << "produce data: " << randomNumber << "\n";
    return randomNumber;
}

void consumeData(int data) { std::cout << "receive data: " << data << "\n"; }

// consumer thread function
void consumer() {
    while (true) {
        while (!g_ready) {
            // sleep for 1 second
            std::this_thread::sleep_for(std::chrono::seconds(1));
        }
        std::unique_lock<std::mutex> ul(g_mutex);
        consumeData(g_data);
        g_ready = false;
    }
}

// producer thread function
void producer() {
    while (true) {
        std::unique_lock<std::mutex> ul(g_mutex);
        g_data = produceData();
        g_ready = true;
        ul.unlock();
        while (g_ready) {
            // sleep for 1 second
            std::this_thread::sleep_for(std::chrono::seconds(1));
        }
    }
}

void consumerThread() { consumer(); }

void producerThread() { producer(); }
```

```
int main() {
    std::thread t1(consumerThread);
    std::thread t2(producerThread);
    t1.join();
    t2.join();
    return 0;
}
```

❖ Print numbers in sequence using thread synchronization

The problem is to synchronize n number of threads using thread. The idea is take thread count and print 1 in first thread, print 2 in second thread, print 3 in third thread, ….. print n in nth thread and again repeat from thread 1 infinitely.

```
unsigned int printThis = 0;
#define maxNum 20
mutex mPrint;
condition_variable cv;
void printNum(int threadID)
{
    while (true)
    {
        if (printThis >= maxNum)
            printThis = 0;
        unique_lock<mutex> lock1(mPrint);
        cv.wait(lock1, [&, threadID] {return (threadID == printThis); });

        cout << "Thread " << ++threadID << " is printing:";
        cout <<  ++printThis << endl;
        lock1.unlock();
        cv.notify_all();
    }
}


int main()
{
    thread t[maxNum];
    for (int i = 0; i < maxNum; i++)
    {
        t[i] = thread(printNum, i);
    }
    for (int i = 0; i < maxNum; i++)
    {
        t[i].join();
    }
}
```

Multi-threaded program with 3 threads that prints sequence of numbers

e.g

```
Thread 1 is printing:1
Thread 2 is printing:2
Thread 3 is printing:3
Thread 4 is printing:4
Thread 1 is printing:5
Thread 2 is printing:6
```

```
unsigned int printThis = 0;
#define maxNum 100
mutex mPrint;
```

```
condition_variable cv;
void printNum(int threadID)
{
    while (printThis <= maxNum)
    {
        unique_lock<mutex> lock1(mPrint);
        cv.wait(lock1, [=] {return ((printThis) % 4 == threadID); });
        cout << "Thread " << threadID+1 << " is printing:";
        cout <<  ++printThis << endl;
        lock1.unlock();
        cv.notify_all();
    }
}


int main()
{
    thread t[4];
    for (int i = 0; i < 4; i++)
    {
        t[i] = thread(printNum, i);
    }
    for (int i = 0; i < 4; i++)
    {
        t[i].join();
    }
}
```

❖ Non-Repeating Elements of a given array using multithreaded program

Given an array **arr[]** of size **N** and an integer **T** representing the count of threads, the task is to find all non-repeating array elements using multithreading.

```
map<int, int> elements;
mutex mut1;
void threadFunc(int arr[], int index)
{
    unique_lock<mutex> lock(mut1);
    int start = 3 * index;
    int end = start +3 ;
    for (int i = start; i < end; i++)
    {
        if (elements.find(arr[i]) == elements.end())
            elements.emplace(arr[i], 1);
        else
            elements[arr[i]] += 1;
    }

}
void getNonRepeatingElements(int arr[], int len)
{
    int numOfThreads = len / 3;
    vector<thread> threads;
    for (int i = 0; i < numOfThreads; i++)
    {
        threads.emplace_back(thread(threadFunc, arr, i));
    }
    for (auto& thread : threads)
    {
        thread.join();
    }
    cout << "Non repeating elements are:" << endl;;
    for (auto& itr : elements)
    {
        if (itr.second == 1)
            cout << itr.first << endl;
    }
}


int main()
{
    int arr[] = { 2, 5, 8, 3, 2, 5, 6, 3, 2 };
    getNonRepeatingElements(arr, 9);
}

O/P:
Non repeating elements are:
2
3
5
```

❖ Maximum in a 2D matrix using Multi-threading in C++

```
typedef vector< vector<int> > Matrix;
int maxElement;
mutex mut1;

void threadFunc(Matrix matrix, int cols, int threadIdx)
{
    unique_lock<mutex> lock(mut1);
    int rowIdx = threadIdx;
    for (int j = 0; j < cols; j++)
    {
        if (matrix[rowIdx][j] > maxElement)
            maxElement = matrix[rowIdx][j];
    }
}

void getMaxElements(Matrix matrix, int rows, int cols)
{
    int numOfThreads = rows;
    vector<thread> threads;
    for (int i = 0; i < numOfThreads; i++)
    {
        threads.emplace_back(thread(threadFunc, matrix, cols, i));
    }
    for (auto& thread : threads)
    {
        thread.join();
    }
}

int main()
{
    Matrix matrix = { {1, 5, 3, 6}, {22, 80, 4, 34},
            {4, 45, 67, 3}, {99, 3, 23, 3} };
    getMaxElements(matrix,4 , 4);
    cout << "Max element of matrix is:" << maxElement;

}
```

❖ Sharing a queue among three threads

Share a queue among three threads A, B, C as per given norms :

- Thread A generates random integers and pushes them into a shared queue.
- Threads B and C compete with each other to grab an integer from the queue.
- The threads B and C compute the sum of integers that they have grabbed from the queue.
- Compare the sums as computed by B and C. The greatest is the winner.

```
queue<int> threadQueue;
mutex mut;
condition_variable cv;
long threadASum = 0;
long threadBSum = 0;

auto produceData = []
{
    int numGen = 0;
    while (numGen < 10000)
    {
        unique_lock<mutex> lock(mut);
        int randNum = rand() % 100;
        threadQueue.push(randNum);
        numGen++;
    }
};

auto consumeData = [](int threadID)
{
    while (!threadQueue.empty())
    {
        unique_lock<mutex> lock(mut);
//Note wait_for here. Might be one thread is waiting here will wait for max 1 second here.
        cv.wait_for(lock, std::chrono::seconds(1), [] { return !threadQueue.empty(); });
        if (!threadQueue.empty())
        {
            int randNum = threadQueue.front();
            threadQueue.pop();
            if (threadID == 1)
            {
                threadASum += randNum;
            }
            else if (threadID == 2)
                threadBSum += randNum;
        }
        lock.unlock();
        cv.notify_one();
    }
};

int main()
{
    thread t1(produceData);
    thread t2(consumeData, 1);
    thread t3(consumeData,2);

    t1.join();
    t2.join();
    t3.join();

    if (threadASum < threadBSum)
        cout << "Thread B is winner.";
    else if (threadASum > threadBSum)
        cout << "Thread A is winner.";
    else
        cout << "Thread A & B bboth has same score.";
    return 0;
}
```

- **Dynamic programming**

❖ Stock Buy Sell to Maximize Profit

```
int maxProfit(int[] arr, int len)
{
    int maxProf = 0;
    if (len > 0)
    {
        int minStock = arr[0];
        for (int cnt = 0; cnt < len; cnt++)
        {
            if (a[0] < minStock)
                minStock = arr[i];
            int profit = arr[i] - minStock;
            if (profit > maxProf)
                maxProf = profit;
        }
    }
    return maxProf;
}
```

❖ Stock Buy Sell to Maximize Profit in any number of transation

```
int maxProfit(const int* arr, int n1) {
    int maxProfit = 0;
    if (n1 > 0)
    {
        for (int i = 1; i < n1; i++)
        {
            if (arr[i] > arr[i - 1])
                maxProfit += (arr[i] - arr[i - 1]);
        }
    }
    return maxProfit;
}
```

❖ Stock Buy Sell to Maximize Profit in K transaction

```
int maxProfit(int price[], int n, int k)
{
    // table to store results of subproblems
    // profit[t][i] stores maximum profit using atmost
    // t transactions up to day i (including day i)
    int profit[k + 1][n + 1];

    // For day 0, you can't earn money
    // irrespective of how many times you trade
    for (int i = 0; i <= k; i++)
        profit[i][0] = 0;

    // profit is 0 if we don't do any transaction
    // (i.e. k =0)
    for (int j = 0; j <= n; j++)
        profit[0][j] = 0;

    // fill the table in bottom-up fashion
    for (int i = 1; i <= k; i++) {
        int maxDiff = INT_MIN;
        for (int j = 1; j < n; j++) {
            maxDiff = max(maxDiff,
                profit[i - 1][j - 1] - price[j - 1]);
            profit[i][j] = max(profit[i][j - 1],
                price[j] + maxDiff);
        }
    }

    return profit[k][n - 1];
}
```

The time complexity of the above solution is **O(kn)** and space complexity is **O(nk).**

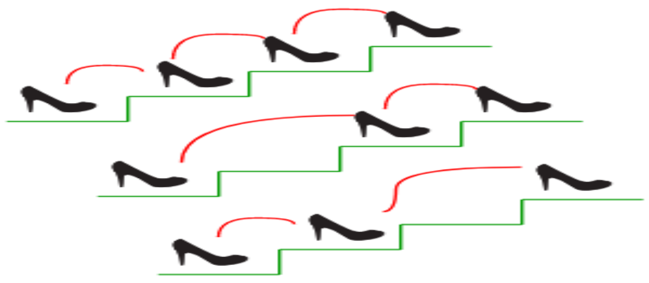
Optimization :

If K is K >= N/2 then we can use the Maximize Profit in any number of transation algorithm.

❖ Find minimum number of coins that make a given value

```
int minCoins(int coins[], int m, int V)
{
    // base case
    if (V == 0) return 0;

    // Initialize result
    int res = INT_MAX;

    // Try every coin that has smaller value than V
    for (int i = 0; i < m; i++)
    {
        if (coins[i] <= V)
        {
            int sub_res = minCoins(coins, m, V - coins[i]);

            // Check for INT_MAX to avoid overflow and see if
            // result can minimized
            if (sub_res != INT_MAX && sub_res + 1 < res)
                res = sub_res + 1;
        }
    }
    return res;
}
```

❖ Count ways to reach the nth stair using step 1, 2 or 3 (Microsoft)



```
int numOfWaysToReachAllSteps(int numSteps)
{
    if (numSteps == 1)
        return 1;
    if (numSteps == 2)
        return 3;
    // For 1 & 2 steps at a time.
    return numOfWaysToReachAllSteps(numSteps - 1) + numOfWaysToReachAllSteps(numSteps - 2);
    // For 1 , 2 & 3 steps at a time.
    /*return numOfWaysToReachAllSteps(numSteps - 1)+numOfWaysToReachAllSteps(numSteps - 2) +
        numOfWaysToReachAllSteps(numSteps - 3);*/
}
```

- **Custom Classes**

❖ Custom String class (Credit Suisse)

```
class Mystring {
    Mystring::Mystring()
        : str{ nullptr }
    {
        str = new char[1];
        str[0] = '\0';
    }

    Mystring::Mystring(char* val)
    {
        if (val == nullptr) {
            str = new char[1];
            str[0] = '\0';
        }

        else {

            str = new char[strlen(val) + 1];

            // Copy character of val[] using strcpy
            strcpy(str, val);

            cout << "The string passed is: " << str << endl;
        }
    }
    // Function to illustrate Copy Constructor
    Mystring::Mystring(const Mystring& source)
    {
        str = new char[strlen(source.str) + 1];
        strcpy(str, source.str);
    }
    // Function to illustrate Move Constructor
    Mystring::Mystring(Mystring&& source)
    {
        str = source.str;
        source.str = nullptr;
    }
};
```

❖ Custom shared pointer class (Credit Suisse)

```
template <typename T>

class Shared_ptr
{
public:
    // Constructor
    explicit Shared_ptr(T* ptr = nullptr)
    {
        m_ptr = ptr;
        if (ptr)
        {
            m_counter++; // Increament count on referencing to new ptr
        }
    }

    // Copy constructor
    Shared_ptr(Shared_ptr<T>& sp)
    {
        m_ptr = sp.m_ptr;    // Note No deep copy.
        m_counter = sp.m_counter;
        m_counter++;
    }

    // Reference count
    unsigned int use_count()
    {
        return m_counter;
    }

    // Get the pointer
    T* get()
    {
        return m_ptr;
    }

    // Overload * operator
    T& operator*()
    {
        return *m_ptr;
    }

    // Overload -> operator
    T* operator->()
    {
        return m_ptr;
    }

    // Destructor
    ~Shared_ptr()
    {
        // If counter of the pointer is zero then we can delete te raw pointer.
        m_counter--;
        if (m_counter == 0)
            delete m_ptr;
    }



private:
    // Reference counter
    int m_counter;

    // Shared pointer
    T* m_ptr;
};

int main()
{
    // ptr1 pointing to an integer.
    Shared_ptr<int> ptr1(new int(151));
    *ptr1 = 100;
    cout << " ptr1's value now: " << *ptr1 << endl;
    cout << *ptr1;

    {
        // ptr2 pointing to same integer which ptr1 is pointing to
         // Shared pointer reference counter should have increased now to 2.
        Shared_ptr<int> ptr2 = ptr1;
        cout << "--- Shared pointers ptr1, ptr2 ---\n";
        cout << *ptr1;
        cout << *ptr2;

        {
            // ptr3 pointing to same integer which ptr1 and ptr2 are pointing to.
             // Shared pointer reference counter should have increased now to 3.
            Shared_ptr<int> ptr3(ptr2);
            cout << "--- Shared pointers ptr1, ptr2, ptr3 "
                "---\n";
            cout << *ptr1;
            cout << *ptr2;
            cout << *ptr3;
        }

        // ptr3 is out of scope.It would have been destructed.
        // So shared pointer reference counter should have decreased now to 2.
        cout << "--- Shared pointers ptr1, ptr2 ---\n";
        cout << *ptr1;
        cout << *ptr2;
    }

    // ptr2 is out of scope.It would have been destructed.
    // So shared pointer reference counter should have decreased now to 1.
    cout << "--- Shared pointers ptr1 ---\n";
    cout << *ptr1;

    return 0;
}
```

❖ Custom unique pointer class

```
template <class T>
class my_unique_ptr
{
private:
       T* ptr = nullptr;

public:
       my_unique_ptr() : ptr(nullptr) // default constructor
       {
       }

       my_unique_ptr(T* ptr) : ptr(ptr)
       {
       }

       my_unique_ptr(const my_unique_ptr& obj) = delete; // copy constructor is deleted
       my_unique_ptr& operator=(const my_unique_ptr& obj) = delete; // copy assignment is
deleted

       my_unique_ptr(my_unique_ptr&& dyingObj) // move constructor
       {
              // Transfer ownership of the memory pointed by dyingObj to this object
              this->ptr = dyingObj.ptr;
              dyingObj.ptr = nullptr;
       }

       void operator=(my_unique_ptr&& dyingObj) // move assignment
       {
              __cleanup__(); // cleanup any existing data

              // Transfer ownership of the memory pointed by dyingObj to this object
```

```cpp
            this->ptr = dyingObj.ptr;
            dyingObj.ptr = nullptr;
        }

        T* operator->() // deferencing arrow operator
        {
            return this->ptr;
        }

        T& operator*()
        {
            return *(this->ptr);
        }

        ~my_unique_ptr() // destructor
        {
            __cleanup__();
        }

private:
        void __cleanup__()
        {
            if (ptr != nullptr)
                delete ptr;
        }
};

int main()
{
        // creates a my_unique_ptr object holding a 'Box' object
        my_unique_ptr<int> box1(new int(5));
        my_unique_ptr<int> box2 = std::move(box1); // Move constructor gets called.
        my_unique_ptr<int> box3;// default constructor gets called.
        box3 = std::move(box2);// Move assignment gets called.
        return 0;
}
```

❖ Custom vector class

```cpp
template <typename T>
class vectorClass
{
    // arr is the integer pointer
    // which stores the address of our vector
    T* m_arr;
    // capacity of the vector
    int m_capacity;
    // currently present in the vector
    int m_size;

public:
    // Default constructor to initialise an initial capacity of 1 element
    vectorClass()
    {
        m_arr = new T[1];
        m_capacity = 1;
        m_size = 0;
    }

    // Function to add an element at the last
    void push(T data)
    {
        // We need to double the capacity
        if (m_size == m_capacity) {
            T* temp = new T[2 * m_capacity];
            // copying old array elements to new array
            for (int i = 0; i < m_capacity; i++) {
                temp[i] = m_arr[i];
            }

            // deleting previous array
            delete[] m_arr;
            m_capacity *= 2;
            m_arr = temp;
        }

        // Inserting data
        m_arr[m_size] = data;
        m_size++;
    }

    // function to add element at any index
    void push(T data, int index)
    {

        // if index is equal to capacity then this
        // function is same as push defined above
        if (index == m_capacity)
            push(data);
        else
            m_arr[index] = data;
    }

    // function to extract element at any index
    T get(int index)
    {
        // if index is within the range
        if (index < size)
            return m_arr[index];
    }

    // function to delete last element
    void pop() { m_size--; }

    // function to get size of the vector
    int size() { return m_size; }

    // function to get capacity of the vector
    int getcapacity() { return m_capacity; }

    // function to print array elements
    void print()
    {
        for (int i = 0; i < m_size; i++) {
            cout << m_arr[i] << " ";
        }
        cout << endl;
    }
};
```

❖ Custom stack class

```
#define SIZE 10
template<typename T>
class MyStack
{
    T stack[SIZE];
    int top = -1;
public:
    MyStack(){}
    void push(T val)
    {
        if (top < SIZE)
        {
            top++;
            stack[top] = val;
        }
    }
    T pop()
    {
        T ret = stack[top];
        top--;
        return ret;

    }
    bool isEmpty()
    {
        if (top < 1)
            return true;
        return false;
    }
    bool isFull()
    {
        if (top == (SIZE -1))
            return true;
        return false;
    }
    void printStack()
    {
        cout << "stack is: " << endl;
        for (auto itr = 0; itr < top; itr++)
        {
            cout << stack[itr] << endl;
        }
    }
};

int main()
{
    MyStack<int> stack;
    stack.push(5); stack.push(7); stack.push(3); stack.push(1);
    stack.push(2); stack.push(8); stack.push(6); stack.push(9);
    stack.printStack();
    stack.pop();
    stack.pop();
    stack.printStack();

    return 0;
}
```

❖ Custom queue class

```
template<typename T>
class MyQueqe
{
private:
    int size = 0;
    T queue[10];

public:
    MyQueqe() {}
    void enque(T val)
    {
        if (!isFull())
        {
            queue[size] = val;
            size++;
        }
        else
            cout << "Queue is full..";
    }

    T deque()
    {
        if (!isEmpty())
        {
            T val = queue[0];
            return val;
        }
        else
            cout << "Queue is empty..";
    }
    bool isFull(){ return (size >= 10); }
    bool isEmpty() { return (size == 0); }
    void printQueue()
    {
        for (int i = 0; i < size; i++)
        {
            cout << queue[i] << endl;
        }
    }
};
```

❖ Custom Hash Table

```
// template for generic type
template <typename K, typename V>

// Our own Hashmap class
class HashMap {
    // hash element array
    HashNode<K, V>** arr;
    int capacity;
    // current size
    int size;
    // dummy node
    HashNode<K, V>* dummy;

public:
    HashMap()
    {
        // Initial capacity of hash array
        capacity = 20;
        size = 0;
        arr = new HashNode<K, V>*[capacity];

        // Initialise all elements of array as NULL
        for (int i = 0; i < capacity; i++)
            arr[i] = NULL;

        // dummy node with value and key -1
        dummy = new HashNode<K, V>(-1, -1);
    }
    // This implements hash function to find index
    // for a key
    int hashCode(K key)
    {
        return key % capacity;
    }
```

```
// Function to add key value pair
void insertNode(K key, V value)
{
    HashNode<K, V>* temp = new HashNode<K, V>(key, value);

    // Apply hash function to find index for given key
    int hashIndex = hashCode(key);

    // find next free space
    while (arr[hashIndex] != NULL
        && arr[hashIndex]->key != key
        && arr[hashIndex]->key != -1) {
        hashIndex++;
        hashIndex %= capacity;
    }

    // if new node to be inserted
    // increase the current size
    if (arr[hashIndex] == NULL
        || arr[hashIndex]->key == -1)
        size++;
    arr[hashIndex] = temp;
}

// Function to delete a key value pair
V deleteNode(int key)
{
    // Apply hash function
    // to find index for given key
    int hashIndex = hashCode(key);

    // finding the node with given key
    while (arr[hashIndex] != NULL) {
        // if node found
        if (arr[hashIndex]->key == key) {
            HashNode<K, V>* temp = arr[hashIndex];

            // Insert dummy node here for further use
            arr[hashIndex] = dummy;

            // Reduce size
            size--;
            return temp->value;
        }
        hashIndex++;
        hashIndex %= capacity;
    }

    // If not found return null
    return NULL;
}

// Function to search the value for a given key
V get(int key)
{
    // Apply hash function to find index for given key
    int hashIndex = hashCode(key);
    int counter = 0;

    // finding the node with given key
    while (arr[hashIndex] != NULL) { // int counter =0; // BUG!

        if (counter++ > capacity) // to avoid infinite loop
            return NULL;
            // if node found return its value
            if (arr[hashIndex]->key == key)
                return arr[hashIndex]->value;
            hashIndex++;
            hashIndex %= capacity;
        }
        // If not found return null
        return NULL;
    }

    // Return current size
    int sizeofMap()
    {
        return size;
    }

    // Return true if size is 0
    bool isEmpty()
    {
        return size == 0;
    }

    // Function to display the stored key value pairs
    void display()
    {
        for (int i = 0; i < capacity; i++) {
            if (arr[i] != NULL && arr[i]->key != -1)
                cout << "key = " << arr[i]->key
                << "  value = "
                << arr[i]->value << endl;
        }
    }
};

// Driver method to test map class
int main()
{
    HashMap<int, int>* h = new HashMap<int, int>;
    h->insertNode(1, 1);
    h->insertNode(2, 2);
    h->insertNode(2, 3);
    h->display();
    cout << h->sizeofMap() << endl;
    cout << h->deleteNode(2) << endl;
    cout << h->sizeofMap() << endl;
    cout << h->isEmpty() << endl;
    cout << h->get(2);

    return 0;
}
```

❖ Convert a normal BST to Balanced BST

Input:

```
    30
   /
  20
 /
```

```
  10
```

Output:

```
    20
   /  \
 10    30
```

Input:

```
        4
       /
      3
     /
    2
   /
  1
```

Output:

```
     3          3         2
   /  \       /  \      /  \
  1    4   OR  2    4  OR  1    3   OR ..
   \         /                \
    2       1                  4
```

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

struct Node
{
    int data;
    Node* left, * right;
};

/* This function traverse the skewed binary tree and
   stores its nodes pointers in vector nodes[] */
void storeBSTNodes(Node* root, vector<Node*>& nodes)
{
    // Base case
    if (root == NULL)
        return;

    // Store nodes in Inorder (which is sorted order for BST)
    storeBSTNodes(root->left, nodes);
    nodes.push_back(root);
    storeBSTNodes(root->right, nodes);
}

/* Recursive function to construct binary tree */
Node* buildTreeUtil(vector<Node*>& nodes, int start,
    int end)
{
    // base case
    if (start > end)
```

```
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end) / 2;
    Node* root = nodes[mid];

    /* Using index in Inorder traversal, construct
       left and right subtress */
    root->left = buildTreeUtil(nodes, start, mid - 1);
    root->right = buildTreeUtil(nodes, mid + 1, end);

    return root;
}

// This functions converts an unbalanced BST to a balanced BST
Node* buildTree(Node* root)
{
    // Store nodes of given BST in sorted order
    vector<Node*> nodes;
    storeBSTNodes(root, nodes);

    // Constructs BST from nodes[]
    int n = nodes.size();
    return buildTreeUtil(nodes, 0, n - 1);
}

// Utility function to create a new node
Node* newNode(int data)
{
    Node* node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* Function to do preorder traversal of tree */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver program
int main()
{
    /* Constructed skewed binary tree is
                10
               /
              8
             /
            7
           /
          6
         /
        5   */

    Node* root = newNode(10);
    root->left = newNode(8);
    root->left->left = newNode(7);
    root->left->left->left = newNode(6);
    root->left->left->left->left = newNode(5);
    root = buildTree(root);

    printf("Preorder traversal of balanced "
        "BST is : \n");
    preOrder(root);

    return 0;
}
```

❖ Simulating final class in C++

```
class Final;  // The class to be made final

class MakeFinal // used to make the Final class final
{
private:
    MakeFinal() { cout << "MakFinal constructor" << endl; }
    friend class Final;
};

class Final : virtual MakeFinal
{
public:
    Final() { cout << "Final constructor" << endl; }
};

class Derived : public Final // Compiler error
{
public:
    Derived() { cout << "Derived constructor" << endl; }
};

int main(int argc, char* argv[])
{
    Derived d;
    return 0;
}
O/P:Compilation error
```

In the above example, *Derived*'s constructor directly invokes *MakeFinal's* constructor, and the constructor of *MakeFinal* is private, therefore we get the compilation error.

• Operator overloading

❖ [] Operator overloading

```
class Point {
    int arr[2];
public:
    Point(int x, int y) {
        arr[0] = x;
        arr[1] = y;
    }
    int& operator[] (int pos)
    {
        if (pos == 0)
            return arr[0];
        else if (pos == 1)
            return arr[1];
        else
            cout << "Out of bound";
    }
    void print()
    {
        cout << "Point: (" << arr[0] << ", " << arr[1] << ")";
    }
};

int main()
{
    Point p1(2, 3);
    p1[0] = 5;
    p1[1] = 10;
    p1.print();
    p1[5] = 2; // This is out of bound

    return 0;
}
o/p : Point(5, 10)
       "Out of bound";
```

❖ () Operator overloading

```
class Distance {
private:
    int feet;              // 0 to infinite
    int inches;            // 0 to 12

public:
     // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    // overload function call
    Distance operator()(int a, int b, int c) {
        Distance D;

        // just put random calculation
        D.feet = a + c + 10;
        D.inches = b + c + 100;
        return D;
    }

    // method to display distance
    void displayDistance() {
        cout << "F: " << feet << " I:" << inches << endl;
    }
};


// Driver code
int main()
{
    Distance D1(11, 10), D2;

    cout << "First Distance : ";
    D1.displayDistance();

    D2 = D1(10, 10, 10); // invoke operator()
    cout << "Second Distance :";
    D2.displayDistance();

    return 0;
}
```

- **Tree**

❖ Search a given key in a given BST

```
node* search(node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == nullptr || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

❖ Insersion in BST

```
BST* BST::Insert(BST* root, int value)
{
    if (!root)
    {
        // Insert the first node, if root is NULL.
        return new BST(value);
    }

    // Insert data.
    if (value > root->data)
    {
        // If the 'value' to be inserted is greater than 'root' node data.
        root->right = Insert(root->right, value);
    }
    else
    {
        // If the 'value' to be inserted is greater than 'root' node data.
        root->left = Insert(root->left, value);
    }

    // Return 'root' node, after insertion.
    return root;
}
```

❖ Traversal in BST

```
void BST::Inorder(BST* root)
{
    if (!root) {
        return;
    }
    Inorder(root->left);
    cout << root->data << endl;
    Inorder(root->right);
}
```

❖ Delete node in BST

```
/* Given a non-empty binary search tree, return the node
with minimum key value found in that tree. Note that the
entire tree does not need to be searched. */
struct node* minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function
deletes the key and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's
    // key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's
    // key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node to be deleted
    else {
        // node has no child
        if (root->left == NULL and root->right == NULL)
       {    delete root;
            return NULL;
       }

        // node with only one child or no child
        else if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

❖ Convert BST to Doubly linked list

```
CNode* m_head = nullptr;
CNode* m_prev = nullptr;
void convertToDList(CNode* root)
{
    if (root == nullptr)
        return;

    convertToDList(root->left);

    if (m_prev == nullptr)
        m_head = root;
    else
    {
        root->left = m_prev;
        m_prev->right = root;
    }

    m_prev = root;
    convertToDList(root->right);
}

void printDList(CNode* head)
{
    if (head == nullptr)
        return;
    CNode* node = head;
    cout << " List is:" << endl;
    while (node != nullptr)
    {
        cout << node->m_data << endl;
        node = node->right;
    }
}


int main()
{
    CNode* m_root = nullptr;
    m_root = insert(20, m_root);
    insert(10, m_root);
    insert(25, m_root);
    insert(5, m_root);
    insert(30, m_root);
    insert(33, m_root);
    insert(15, m_root);
    inorder(m_root);
    convertToDList(m_root);
    printDList(m_head);

    return 0;
}
```

❖ check if a binary tree is BST or not

```
int isBST(struct node* node)
```

```
{
    if (node == NULL)
        return 1;

    /* false if left is > than node */
    if (node->left != NULL && node->left->data > node->data)
        return 0;

    /* false if right is < than node */
    if (node->right != NULL && node->right->data < node->data)
        return 0;

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return 0;

    /* passing all that, it's a BST */
    return 1;
}
```

❖ Find the Maximum Depth or Height of a Tree

```
int maxDepth(CNode* node)
{
    if (node == NULL)
        return -1;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return(lDepth + 1);
        else return(rDepth + 1);
    }
}
```

• Sorting

❖ Bubble Sort

```
void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
```

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.
Best Case Time Complexity: O(n). Best case occurs when array is already sorted.
Auxiliary Space: O(1)

❖ Selection Sort

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n - 1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Time Complexity: O(n^2) as there are two nested loops.
Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

• Mathematical programs

❖ Square root of an integer

```
// Returns floor of square root of x
int floorSqrt(int x)
{
    // Base cases
    if (x == 0 || x == 1)
        return x;

    // Starting from 1, try all numbers until
    // i*i is greater than or equal to x.
    int i = 1, result = 1;
    while (result <= x)
    {
        i++;
        result = i * i;
    }
    return i - 1;
}
```

❖ Check prime number

```
// function check whether a number is prime or not
bool isPrime(int n)
{
    // Corner case
    if (n <= 1)
        return false;

    // Check from 2 to square root of n
    for (int i = 2; i <= sqrt(n); i++)
        if (n % i == 0)
            return false;

    return true;
}
```

❖ Get sum of digits

```
int getSum(int n)
{
    int sum = 0;
    while (n != 0) {
        sum = sum + n % 10;
        n = n / 10;
    }
    return sum;
}
```

❖ Maximum of four numbers without using conditional or bitwise operator

Given four numbers, print the maximum of the 4 entered numbers

```
Input : 4 8 6 5

Output : 8
```

```
int maxOfFour(int w, int x, int y, int z)
{
    int a[2];
    a[0] = w, a[1] = x;

    // b is 0 if w is less than or equal to x, else it is non-zero.
    bool b = (a[0] - a[1] + abs(a[0] - a[1]));

    // After below operation, a[0] is maximum of w and x.
    swap(a[0], a[!b]);

    // After below three steps, a[0] is maximum of w, x and y.
    a[1] = y;
    b = (a[0] - a[1] + abs(a[0] - a[1]));
    swap(a[0], a[!b]);

    // After below three steps, a[0] is maximum of w, x, y and z.
    a[1] = z;
    b = (a[0] - a[1] + abs(a[0] - a[1]));
    swap(a[0], a[!b]);

    return a[0];
}
```

❖ Sieve of Eratosthenes (Check all prime numbers under number n)

Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number.

```
void SieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize all entries it as true.
    // A value in prime[i] will finally be false if i is Not a prime, else true.
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (int p = 2; p * p <= n; p++)
    {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true)
        {
            // Update all multiples of p greater than or equal to the square of it numbers
            // which are multiple of p and are less than p^2 are already been marked.
             for  (int  i = p * p; i <= n; i += p)
                    prime[i] = false;


        }
    }

    // Print all prime numbers
    for (int p = 2; p <= n; p++)
        if (prime[p])
            cout << p << " ";
}
```

❖ Greatest common factor

```
int gcd(int a, int b)
{
    // Base case
    if (b == 0)
        return a;
    // Recursively calculate GCD
    return gcd(b, a % b);
}
```

❖ Power of number

```
int power(int x, int y)
{
    // If x^0 return 1
    if (y == 0)
        return 1;
    // If we need to find of 0^y
    if (x == 0)
        return 0;
    // For all other cases
    return x * power(x, y - 1);
}
```

- ## Search

❖ Linear search

```
int LinearSearch(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

The **time complexity** of the above algorithm is O(n).

❖ Binary search

```
int binarySearch(int arr[], int left, int right, int x)
{
    if (right >= left) {
        int mid = left + (right - left) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, left, mid - 1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid + 1, right, x);
    }

    // We reach here when element is not present in array
    return -1;
}
```

Time Complexity:

The time complexity of Binary Search is **O(Log n).**

❖ Find position of an element in a sorted array of infinite numbers

```
int findPos(int arr[], int key)
{
    int l = 0, h = 1;
    int val = arr[0];

    // Find h to do binary search
    while (arr[h] < key)
    {
        l = h;         // store previous high
        h = 2 * h;       // double high index
    }

    // At this point we have updated low and high indices, Thus use binary search between
them
    return binarySearch(arr, l, h, key);
}
```

Time Complexity:

The time complexity of this exponential Search is **O(Log n).**

❖ Find duplicates in O(n) time and O(1) extra space

Given an array of n elements that contains elements from 0 to n-1, with any of these numbers appearing any number of times. Find these repeating numbers in O(n) and using only constant memory space.

```
Input : n = 7 and array[] = {1, 2, 3, 6, 3, 6, 1}
Output: 1, 3, 6
```

```
void printRepeating(int arr[], int size)
{
    cout << "The repeating elements are:" << endl;
    for (int i = 0; i < size; i++)
    {
        if (arr[abs(arr[i])] >= 0)
            arr[arr[i]] = -arr[abs(arr[i])];
        else
            cout << abs(arr[i]) << endl;
    }
}

int main()
{
    int arr[] = { 1, 2, 3, 1, 3, 6, 6 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printRepeating(arr, arr_size);

    return 0;
}
```

- **String**

❖ String concat

```
void stringconcat(char* str1, char* str2) {
    while (*str1 != '\0') {
        str1++;
    }

    while (*str2 != '\0') {
        *str1 = *str2;
        str1++;
        str2++;
    }
}
```

❖ Palindrom of string using recursion

```
bool isPalRec(char str[],
    int s, int e)
{
    // If there is only one character
    if (s == e)
        return true;
    // If first and last characters do not match
    if (str[s] != str[e])
        return false;
    // If there are more than two characters, check if
    // middle substring is also palindrome or not.
    if (s < e + 1)
        return isPalRec(str, s + 1, e - 1);

    return true;
}
```

❖ Recursive program to generate power set of string

Given a set represented as a string, write a recursive code to print all subsets of it.

Input : set = "abc"

Output : "". "a", "b", "c", "ab", "ac", "bc", "abc"

```
void powerSet(string str, int index = 0,
    string curr = "")
{
    int n = str.length();

    // base case
    if (index == n) {
        cout << curr << endl;
        return;
    }
    // Two cases for every character (i) We consider the character as part of current subset
    // (ii) We do not consider current character as part of current subset
    powerSet(str, index + 1, curr + str[index]);
    powerSet(str, index + 1, curr);
}
```

Time Comlpexity : O(2^n)

❖ Program to calculate unique substring of string

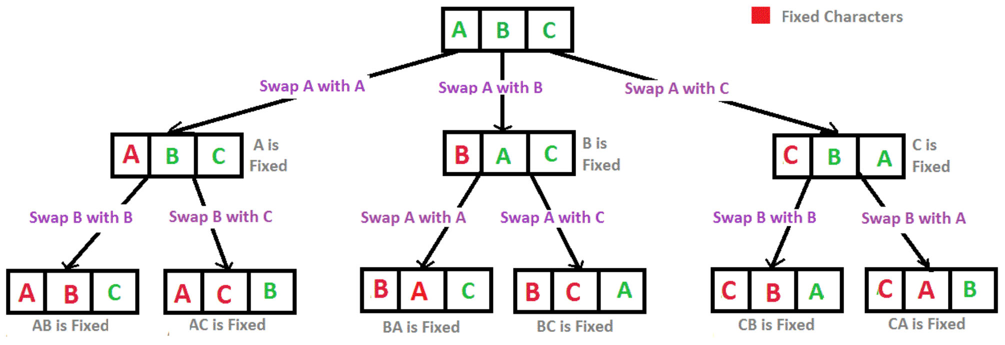
Input: bacdc

O/P

```
bacdc

bacd

bac

ba

b

acdc

acd

ac

a

cdc

cd

c

dc

d
```

```
// No c two times


long substringCalculator(string s) {

    set<string> subset;
    long numDistSubStrings = 0;
    string tempStr = s;
    int count = tempStr.length();
    for (int i = 0; i < count; i++)
    {
        int orglen = s.length();
        tempStr = s;
        while (!tempStr.empty())
        {
            int strlen = tempStr.length();

            if (strlen > 0 && subset.find(tempStr) == subset.end())
            {
                numDistSubStrings++;
            }
            if(strlen == 1)
                subset.emplace(s.substr(0, 1));

            tempStr = tempStr.substr(0, strlen - 1);
        }

        s = s.substr(1, orglen - 1);
    }
    return numDistSubStrings;
}
```

❖ Write a program to print all permutations of a given string



Recursion Tree for Permutations of String "ABC"

```
void permute(string a, int l, int r)
{
    // Base case
    if (l == r)
        cout << a << endl;
    else
        // Permutations made
        for (int i = l; i <= r; i++)
        {
            // Swapping done
            swap(a[l], a[i]);
            // Recursion called
            permute(a, l + 1, r);
            //backtrack
            swap(a[l], a[i]);
        }
    }
}
```

❖ Frequency of a substring in a string

```
int countFreq(string& pat, string& txt)
{
    int M = pat.length();
    int N = txt.length();
    int res = 0;

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        /* For current index i, check for pattern match */
        int j;
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        if (j == M)
        {
            res++;
            j = 0;
        }
    }
    return res;
}

int main()
{
    string txt = "dhimanman";
    string pat = "man";
    cout << countFreq(pat, txt);
    return 0;
}
```

```
O/P: 2
```

Time Complexity: O(M * N)

Alternative using find of string using string find.

```
int main()
{
    string s("hello hello");
    int count = 0;
    size_t nPos = s.find("hello", 0); // first occurrence
    while (nPos != string::npos)
    {
        count++;
        nPos = s.find("hello", nPos + 1);
    }

    cout << count;
};
```

Alternative using multithreading.

```
string mystr = "abchabcufgaaabcijf";
int mystrLen = mystr.length();
string pat = "abc";
int patCnt = 0;

void checkPattern(int loc)
{
    int j = 0;
    for(int i = loc; i < mystrLen; i++)
    {
        if (mystr[i] != pat[j])
            break;
        else
            j++;
        if (j == pat.length()-1)
            patCnt++;
    }

}

int main()
{
    std::thread myThreads[18];

    for (int i = 0; i < 18; i++) {
        myThreads[i] = std::thread(checkPattern, i);
    }
    cout << "Pattern count is :" << patCnt;
    for (int i = 0; i < 18; i++) {
        myThreads[i].join();
    }
}
```

❖ Print reverse of a string using recursion

```
1.
void reverse(string str)
{
    if (str.size() == 0)
    {
        return;
    }
    reverse(str.substr(1));
    cout << str[0];
}
```

```
2.

void reverseStr(string& str)
{
    int n = str.length();

    // Swap character starting from two
    // corners
    for (int i = 0, j = n - 1; i < j; i++, j--)
        swap(str[i], str[j]);
}
```

❖ Check for Balanced Brackets in an expression

Given an expression string exp, write a program to examine whether the pairs and the orders of "{", "}", "(", ")", "[", "]" are correct in exp.

***Input**: exp = "[()]{}{[()()]()}"*
***Output**: Balanced*
***Input**: exp = "[(])"*
***Output**: Not Balanced*

```
bool areBracketsBalanced(string expr)
{
    stack<char> s;
    char x;
    // Traversing the Expression
    for (int i = 0; i < expr.length(); i++)
    {
        if (expr[i] == '(' || expr[i] == '['
            || expr[i] == '{')
        {
            // Push the element in the stack
            s.push(expr[i]);
            continue;
        }

        // IF current current character is not opening bracket, then it must be closing. So
stack
        // cannot be empty at this point.
        if (s.empty())
            return false;

        switch (expr[i]) {
        case ')':
            // Store the top element in a
            x = s.top();
            s.pop();
            if (x == '{' || x == '[')
                return false;
            break;
        case '}':
            // Store the top element in b
            x = s.top();
            s.pop();
            if (x == '(' || x == '[')
                return false;
            break;
        case ']':
            // Store the top element in c
            x = s.top();
            s.pop();
            if (x == '(' || x == '{')
                return false;
            break;
        }
    }
    // Check Empty Stack
    return (s.empty());
}
int main()
{
    string expr = "{()}[]";

    // Function call
    if (areBracketsBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
}
```

```
O/P: Balanced
```

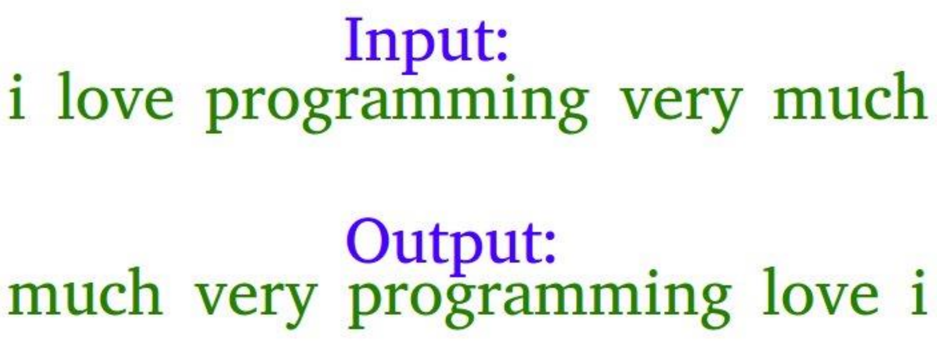
❖ Find maximum number that can be formed using digits of a given number

```
int printMaxNum(int num)
{
    // hashed array to store count of digits
    int count[10] = { 0 };
    // Converting given number to string
    string str = to_string(num);

    // Updating the count array
    for (int i = 0; i < str.length(); i++)
        count[str[i] - '0']++;

    // result is to store the final number
    int result = 0, multiplier = 1;
    // Traversing the count array
    // to calculate the maximum number
    for (int i = 0; i <= 9; i++)
    {
        while (count[i] > 0)
        {
            result = result + (i * multiplier);
            count[i]--;
            multiplier = multiplier * 10;
        }
    }
    // return the result
    return result;
}

// Driver program to test above function
int main()
{
    int num = 38293367;
    cout << printMaxNum(num);
    return 0;
}
```

O/P: 98763332

Time Complexity: O(N), where N is the number of digits in the given number.

❖ Reverse words in a given string


Input:
i love programming very much

Output:
much very programming love i

```
void reverseWords(string s)
{
    // temporary vector to store all words
    vector<string> tmp;
    string str = "";
    for (int i = 0; i < s.length(); i++)
    {
        // Check if we encounter space push word(str) to vector and make str NULL
        if (s[i] == ' ')
        {
            tmp.push_back(str);
            str = "";
        }
        // Else add character to str to form current word
        else
            str += s[i];
    }
    // Last word remaining,add it to vector
    tmp.push_back(str);

    // Now print from last to first in vector
    int i;
    for (i = tmp.size() - 1; i > 0; i--)
        cout << tmp[i] << " ";
    // Last word remaining,print it
    cout << tmp[0] << endl;
}
```

❖ Word machine with capability add, duplicate, pop, sum and subtract from the stack

```
std::stack<unsigned long int> mystack;

int doOperation(vector<string>& vecIn)
{
    for (int i = 0; i < vecIn.size(); i++)
    {
        if (vecIn[i] == "DUP")
        {
            int value = mystack.top();
            mystack.push(value);
        }
        else if (vecIn[i] == "POP")
        {
            mystack.pop();
        }
        else if (vecIn[i] == "+")
        {
            if (mystack.empty())
                return -1;
            unsigned long int value1 = mystack.top();
            mystack.pop();
            if (mystack.empty())
                return -1;
            unsigned long int value2 = mystack.top();
            if (1048576 > value1 + value2)
            {
                mystack.pop();
                mystack.push(value1 + value2);
            }
            else
            {
                return -1;
            }
        }
        else if (vecIn[i] == "-")
        {
            if (mystack.empty())
                return -1;
            unsigned long int value1 = mystack.top();
            mystack.pop();
            if (mystack.empty())
                return -1;
            unsigned long int value2 = mystack.top();
            mystack.pop();
            mystack.push(value1 - value2);
        }
        else if (stoi(vecIn[i]))
        {
            int value = stoi(vecIn[i]);
            mystack.push(value);
        }

    }
    return mystack.top();
}


int solution(string& S) {
    if (S.length() == 0) {
        return -1;
    }

    vector<string> vect; // Array to hold tuples
    string str = "";
    for (int i = 0; i < S.length(); i++)
    {
        // Check for space
        if (S[i] == ' ')
        {
            vect.push_back(str);
            str = "";
        }
        // Else add character to str to form current word
        else
            str += S[i];
    }
    vect.push_back(str);

    return doOperation(vect);

}
int main()
{
    string str = "1048575 DUP +";
    int ret = solution(str);

}
```

- **Transform the string**

❖ Transform the string

Given a string s, change the string s according to the rules provided below:

- Delete all the vowels from the string.
- Insert **#** in front of all the consonants.
- Change the case of all the letters of the string.

```
string change_case(string a)
{
    int l = a.length();

    for (int i = 0; i < l; i++)
```

```cpp
    {
        // If character is lowercase change to uppercase
        if (a[i] >= 'a' && a[i] <= 'z')
            a[i] = (char)(65 +
                (int)(a[i] - 'a'));

        // If character is uppercase change to lowercase
        else if (a[i] >= 'A' && a[i] <= 'Z')
            a[i] = (char)(97 +
                (int)(a[i] - 'A'));
    }
    return a;
}

// Function to delete vowels
string delete_vowels(string a)
{
    string temp = "";
    int l = a.length();
    for (int i = 0; i < l; i++)
    {
        //If character is consonant
        if (a[i] != 'a' && a[i] != 'e' &&
            a[i] != 'i' && a[i] != 'o' &&
            a[i] != 'u' && a[i] != 'A' &&
            a[i] != 'E' && a[i] != 'O' &&
            a[i] != 'U' && a[i] != 'I')
            temp += a[i];
    }
    return temp;
}

// Function to insert "#"
string insert_hash(string a)
{
    string temp = "";
    int l = a.length();

    for (int i = 0; i < l; i++)
    {
        // If character is not special
        if ((a[i] >= 'a' && a[i] <= 'z') ||
            (a[i] >= 'A' && a[i] <= 'Z'))
            temp = temp + '#' + a[i];
        else
            temp = temp + a[i];
    }
    return temp;
}

// Function to transform string
void transformSting(string a)
{
    string b = delete_vowels(a);
    string c = change_case(b);
    string d = insert_hash(c);
    cout << d;
}

// Driver function
int main()
{
    string a = "SunshinE!!";
    // Calling function
    transformSting(a);
    return 0;
}
```

❖ Transform One String to Another using Minimum Number of Given Operation

Given two strings A and B, the task is to convert A to B if possible. The only operation allowed is to put any character from A and insert it at front. Find if it's possible to convert the string. If yes, then output minimum no. of operations required for transformation.

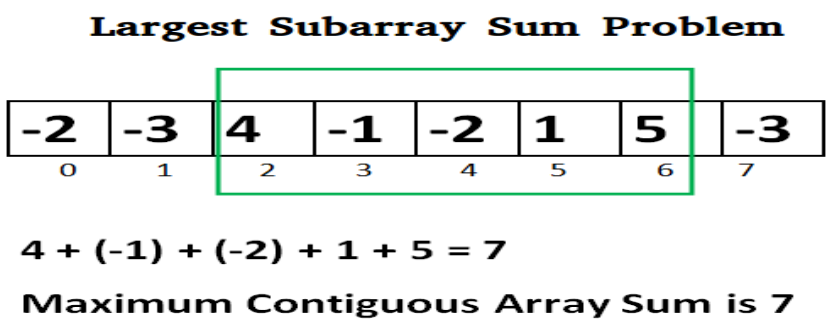
```
Input:  A = "ABD", B = "BAD"

Output: 1

Explanation: Pick B and insert it at front.


Input:  A = "EACBD", B = "EABCD"

Output: 3

Explanation: Pick B and insert at front, EACBD => BEACD

             Pick A and insert at front, BEACD => ABECD

             Pick E and insert at front, ABECD => EABCD
```

```
int minOps(string& A, string& B)
{
    int m = A.length(), n = B.length();

    // This parts checks whether conversion is
    // possible or not
    if (n != m)
        return -1;
    int count[256];
    memset(count, 0, sizeof(count));
    for (int i = 0; i < n; i++)   // count characters in A
        count[B[i]]++;
    for (int i = 0; i < n; i++)   // subtract count for
        count[A[i]]--;         // every character in B
    for (int i = 0; i < 256; i++)   // Check if all counts become 0
        if (count[i])
            return -1;

    // This part calculates the number of operations required
    int res = 0;
    for (int i = n - 1, j = n - 1; i >= 0; )
    {
        // If there is a mismatch, then keep incrementing
        // result 'res' until B[j] is not found in A[0..i]
        while (i >= 0 && A[i] != B[j])
        {
            i--;
            res++;
        }

        // If A[i] and B[j] match
        if (i >= 0)
        {
            i--;
            j--;
        }
    }
    return res;
}

// Driver program
int main()
{
    string A = "EACBD";
    string B = "EABCD";
    cout << "Minimum number of operations "
        "required is " << minOps(A, B);
    return 0;
}
```

- **Array**

❖ Largest Sum Contiguous Subarray

Write an efficient program to find the sum of contiguous subarray within a one-dimensional array of numbers that has the largest sum.


Largest Subarray Sum Problem
-2
-3
4
-1
-2
1
5
-3
0
1
2
3
4
5
6
7
4 + (-1) + (-2) + 1 + 5 = 7
Maximum Contiguous Array Sum is 7

```
int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0;
    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;

        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = { -2, -3, 4, -1, -2, 1, 5, -3 };
    int n = sizeof(a) / sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    cout << "Maximum contiguous sum is " << max_sum;
    return 0;
}
```

❖ Find subarray with given sum

Given an unsorted array of nonnegative integers, find a continuous subarray which adds to a given number.

```
Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
Output: Sum found between indexes 2 and 4
Sum of elements between indices
2 and 4 is 20 + 3 + 10 = 33
```

```
bool subArraySum(int arr[], int count, int sum)
{
    int currSum = 0;
    int start = 0;  // If we want start index of the subarray.
    int end = 0; // If we want end index of the subarray.
    std::unordered_map<int, int> sumTOIdxMap;
    for (int i = 0; i < count; i++)
    {
        if (currSum == sum)
        {
            start = 0;
            end = i;
            return true;
        }

        if (sumTOIdxMap.find(currSum - sum) != sumTOIdxMap.end())
        {
            start = sumTOIdxMap[currSum - sum] +1;
            end = i;
            return true;
        }
        else
        {
            currSum += arr[i];
            sumTOIdxMap.emplace(currSum, i);
        }
    }
    return false;
}


int main()
{
    int arr[] = { 4, 7,-3, 10, 4, 6, -15, 5, 10, 2 };
    if (subArraySum(arr, 10, 15))
        cout << "Subarray with sum of 15 is present";
    else
        cout << "Subarray with sum of 15 is not present";

    return 0;
}
```

❖ Count Distinct Elements in every Window of size k

```
void countDistinctElement(int arr[], int size, int k)
{
    std::map<int, int> windowMap;
    for (int i = 0; i < k; i++)
    {
        if(windowMap.find(arr[i]) == windowMap.end())
            windowMap.emplace(arr[i], 1);
        else
        {
            windowMap[arr[i]] += 1;
        }
    }
    cout << "Distinct Element: " << windowMap.size() << endl;
    for (int i = k; i < size; i++)
    {
        if (windowMap[arr[i-k]] == 1)
            windowMap.erase(arr[i - k]);
        else
            windowMap[arr[i-k]] -= 1;

        if (windowMap.find(arr[i]) == windowMap.end())
            windowMap.emplace(arr[i], 1);
        else
        {
            windowMap[arr[i]] += 1;
        }
        cout << "Distinct Element: " << windowMap.size() << endl;
    }

}

int main()
{
    int arr[] = { 1, 2, 3, 1, 3, 6, 7, 5, 6, 2 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    countDistinctElement(arr, arr_size, 4);
    return 0;
}
```

O/P:

```
Distinct Element: 3

Distinct Element: 3

Distinct Element: 3

Distinct Element: 4

Distinct Element: 4

Distinct Element: 3

Distinct Element: 4
```

❖ Merge two sorted arrays

```
void mergeArrays(int arr1[], int arr2[], int n1,
    int n2, int arr3[])
{
    int i = 0, j = 0, k = 0;

    // Traverse both array
    while (i < n1 && j < n2)
    {
        // Check if current element of first array is smaller than current element
        // of second array. If yes, store first array element and increment first array
        // index. Otherwise do same with second array
        if (arr1[i] < arr2[j])
            arr3[k++] = arr1[i++];
        else
            arr3[k++] = arr2[j++];
    }

    // Store remaining elements of first array
    while (i < n1)
```

```
        arr3[k++] = arr1[i++];

    // Store remaining elements of second array
    while (j < n2)
        arr3[k++] = arr2[j++];
}
```

- ## Missalaneous

❖ Count all possible paths from top left to bottom right of a mXn matrix

```
int numberOfPaths(int m, int n)
{
    // If either given row number is first or given column number is first
    if (m == 1 || n == 1)
        return 1;

    // If diagonal movements are allowed then the last addition is required.
    return numberOfPaths(m - 1, n) + numberOfPaths(m, n - 1);
}
```

❖ Given a sorted array and a number x, find the pair in array whose sum is closest to x

```
Input: arr[] = {10, 22, 28, 29, 30, 40}, x = 54

Output: 22 and 30
```

```
// Prints the pair with sum closest to x
void printClosest(int arr[], int n, int x)
{
    int res_l, res_r;  // To store indexes of result pair

    // Initialize left and right indexes and difference between
    // pair sum and x
    int l = 0, r = n - 1, diff = INT_MAX;

    // While there are elements between l and r
    while (r > l)
    {
        // Check if this pair is closer than the closest pair so far
        if (abs(arr[l] + arr[r] - x) < diff)
        {
            res_l = l;
            res_r = r;
            diff = abs(arr[l] + arr[r] - x);
        }

        // If this pair has more sum, move to smaller values.
        if (arr[l] + arr[r] > x)
            r--;
        else // Move to larger values
            l++;
    }

    cout << " The closest pair is " << arr[res_l] << " and " << arr[res_r];
}

// Driver program to test above functions
int main()
{
    int arr[] = { 10, 22, 28, 29, 30, 40 }, x = 54;
    int n = sizeof(arr) / sizeof(arr[0]);
    printClosest(arr, n, x);
    return 0;
}
```

❖ Unique paths in a Grid with Obstacles

```
int uniquePathsWithObstacles(vector<vector<int> >& A)
{
    int r = A.size();
    int c = A[0].size();
    if (A[0][0] == 1)
        return 0;
    A[0][0] = 1;

    // first row all are '1' until obstacle
    for (int j = 1; j < c; j++) {

        if (A[0][j] == 0) {
            A[0][j] = A[0][j - 1];
        }
        else
            // No ways to reach at this index
            A[0][j] = 0;
    }

    // first column all are '1' until obstacle
    for (int i = 1; i < r; i++) {

        if (A[i][0] == 0) {
            A[i][0] = A[i - 1][0];
        }
        else
            // No ways to reach at this index
            A[i][0] = 0;
    }
    for (int i = 1; i < r; i++) {

        for (int j = 1; j < c; j++) {
            // If current cell has no obstacle
            if (A[i][j] == 0) {

                A[i][j] = A[i - 1][j] + A[i][j - 1];
            }
            else {
                // No ways to reach at this index
                A[i][j] = 0;
            }
        }
    }
    return A[r - 1][c - 1];
}
```
Time Complexity: O(m*n)
Auxiliary Space: O(1)

```
int main()
{
    std::vector<std::vector<int>> vec = { {0,0,0},{0,1,0},{0,0,0} };
    cout << uniquePathsWithObstacles(vec);

}
```

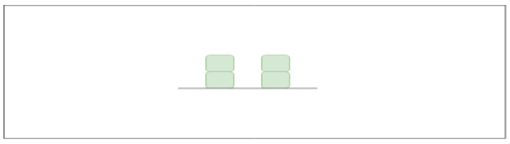
❖ Find majority element in an array with time complexity O(n*2)

```
int findMajority(int[] arr)
{
    int ansIndex = 0;
    int count;
    for (int i = 0; i < arr.length(); i++)
    {
        if (arr[i] == arr[ansIndex])
            count++;
        else
            count--;
        if (count == 0)
            ansIndex = i;
    }
    //Check now is the arr[ansIndex] is majority element.
    count = 0;
    for (int i = 0; i < arr.length(); i++)
    {
        if (arr[i] == arr[ansIndex])
            count++;
    }
    if (count > arr.length() / 2)
        return arr[ansIndex];
}
```

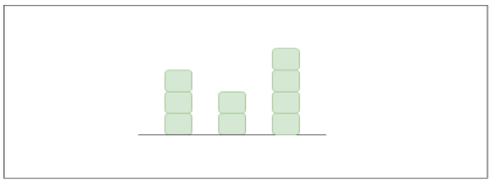
❖ Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

```
Input: arr[]   = {2, 0, 2}
Output: 2
Explanation:
The structure is like below
```



```
Input: arr[]   = {3, 0, 2, 0, 4}
Output: 7
```

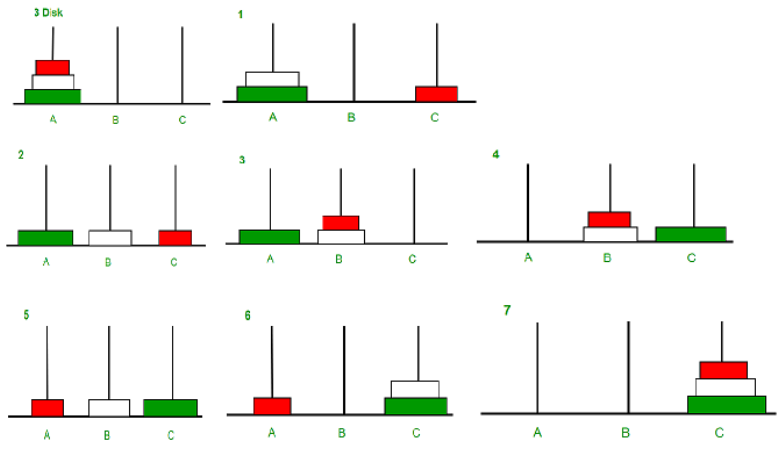


```
int findWater(int arr[], int n)
{
    // left[i] contains height of tallest bar to the left of i'th bar including itself
    int left[n];

    // Right [i] contains height of tallest bar to the right of ith bar including itself
    int right[n];

    // Initialize result
    int water = 0;

    // Fill left array
    left[0] = arr[0];
    for (int i = 1; i < n; i++)
        left[i] = max(left[i - 1], arr[i]);

    // Fill right array
    right[n - 1] = arr[n - 1];
    for (int i = n - 2; i >= 0; i--)
        right[i] = max(right[i + 1], arr[i]);

// Calculate the accumulated water element by element consider the amount of water on i'th
bar, the amount of water accumulated on this particular bar will be equal to min(left[i],
right[i]) - arr[i] .
    for (int i = 0; i < n; i++)
        water += min(left[i], right[i]) - arr[i];

    return water;
}
```

❖ Program for Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.



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

- ## Window Sliding Technique

❖ **Maximum sum in a subarray of size k.**

```
Given an array of integers of size ‘n’.
Our aim is to calculate the maximum sum of ‘k’
consecutive elements in the array.

Input  : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}
         k = 4
Output : 39
```

```
// Returns maximum sum in a subarray of size k.
int maxSum(int arr[], int n, int k)
{
    // Initialize result
    int max_sum = INT_MIN;

    // Consider all blocks starting with i.
    for (int i = 0; i < n - k + 1; i++) {
        int current_sum = 0;
        for (int j = 0; j < k; j++)
            current_sum = current_sum + arr[i + j];

        // Update result if required.
        max_sum = max(current_sum, max_sum);
    }

    return max_sum;
}
```

❖ **Find the smallest window in a string containing all characters of another string**

```
S = "ADOBECODEBANC"
T = "ABC"
```

Minimum window is "BANC"

```
#include <algorithm>

string Minimum_Window(string A, string B)
{
    int map[256] = { 0 };
    int count = 0;
     // creating map
    for (int i = 0; i < B.length(); i++) {
        if (map[B[i]] == 0)
            count++;
        map[B[i]]++;
    }
    int minLength = INT_MAX;
    int start_Idx = 0;
    int itr = 0;
    for (int i = 0; i < A.length(); i++) {
        map[A[i]]--;
        if (map[A[i]] == 0)
            count--;

        if (count == 0)
        {
            while (count == 0)
            {
                if (minLength > i - itr + 1) {
                    minLength = min(minLength, i - itr + 1);
                    start_Idx = itr;
                }

                map[A[itr]]++;
                if (map[A[itr]] > 0)
                    count++;

                itr++;
            }
        }

    }
    if (minLength != INT_MAX)
        return A.substr(start_Idx, minLength);
    else
        return "";
}
```

❖ Smallest subarray with sum greater than a given value (Microsoft)

```
int minSubArray(int arr[], int size)
{
    int minLen = INT_MAX;
    int sumOfSubArr = 0;
    int start = 0;
    int k = 7;
    for (int i = 0; i < size; i++)
    {
        sumOfSubArr += arr[i];
        if (sumOfSubArr >= k)
        {
            while (sumOfSubArr >= k)
            {
                if (minLen > i - start + 1)
                    minLen = i - start + 1;
                sumOfSubArr -= arr[start];
                start++;

            }
        }
    }
    if (minLen != INT_MAX)
        return minLen;
    else
        return 0;
}


int main() {

    int arr[] = { 2,1,5,2,3,2 };
    cout << minSubArray(arr, 6);
}
```

❖ Maximize profit possible by selling M products such that profit of a product is the number of products left of that supplier (Morgan Stanley)

Input: *arr[] = {4, 6}, M = 4*
Output: *19 (6,5,4,4)*

```
long maximumProfit(vector<int> inventory, long order) {

    /***** Simple Version ****/
    priority_queue<int, std::vector<int>, std::less<int>> maxInventories;
    for (auto v : inventory)
    {
        maxInventories.emplace(v);
    }

    // Iterate the maxProfit now for max order number
    long max_profit = 0;
    for (int cnt = 0; cnt < order; cnt++)
    {
        // Get top element
        int val = maxInventories.top();
        max_profit += val;
        maxInventories.pop();
        maxInventories.push(val - 1);
    }
    return max_profit;
}
```

❖ Find next greater number with same set of digits (Microsoft)

```
vector<int> nextBigNumber(vector<int> arr, int count)
{
    // If number of digits is 1 then just return the vector
    if (count == 1)
        return arr;

    int i = 0;
    for (i = count - 1; i > 0; i--) {
        if (arr[i] > arr[i - 1])
            break;
    }

    // If there is a possibility of a next greater element
    if (i != 0)
    {
        for (int j = count - 1; j >= i; j--) {
            if (arr[i - 1] < arr[j]) {
                // Swap the found smallest digit i.e. arr[j]
                // with arr[i-1]
                std::swap(arr[i - 1], arr[j]);
                break;
            }
        }
    }
    // Reverse the digits after (i-1) because the digits after (i-1) are in decreasing order
and
```

```
    // thus we will get the smallest element possible from these digits
    std::reverse(arr.begin() + i, arr.end());
    return arr;
}
int main() {


    int n = 6;
    vector<int> arr{ 5,3,4,9,7,6 };
    vector<int> res;
    res = nextBigNumber(arr, n);
    for (int i = 0; i < res.size(); i++) {
        cout << res[i] << " ";
    }
}
```

Graphs

```
class Graph
{
    std::map<int, bool> visNodes;
    std::map<int, std::list<int>> nodes;
public:
    Graph() {}
    void addEdge(int from, int to)
    {
        if (nodes.find(from) != nodes.end())
        {
            nodes[from].push_back(to);
        }
        else
        {
            std::list<int> lst;
            lst.push_back(to);
            nodes.emplace(make_pair(from, lst));
        }

    }
    void DFS(int val)
    {
        visNodes[val] = true;
        cout << val << " ";

        auto lst = nodes[val];
        for (auto item : lst)
        {
            if(!visNodes[item])
                DFS(item);
        }
    }

    void BFS(int val)
    {
        visNodes[val] = true;
        queue<int> q;
        q.push(val);
        while (!q.empty())
        {
            int val = q.front();
            cout << val << "  ";
            q.pop();

            auto lst = nodes[val];
            for (auto item : lst)
            {
                if (!visNodes[item])
                {
                    q.push(item);
                    visNodes[item] = true;
                }

            }
        }
    }
};


int main()
{
    Graph g1;
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(2, 0);
    g1.addEdge(2, 3);
    g1.addEdge(3, 3);
    g1.BFS(2);
    g1.DFS(2);
}
```

Global Compliance Engine:

Description: Compliance engine is a product that validates order(to be processed by Exchanges) , against the compliance rules defined by the exchange. It basically does the limit checking on orders and generates status like OK/BREACHED depending upon order details. Once the order is successfully validated by the engine, the order is sent to the respective exchange for processing. The compliance engine also maintains positions for applying daily checks on orders.