**Stacks & Queues**

# 1. Implement Stack from Scratch

Following example shows how to implement stack by creating user defined push() method for entering elements and pop() method for retrieving elements from the stack.

public class MyStack {

private int maxSize;

private long[] stackArray;

private int top;

public MyStack(int s) {

maxSize = s;

stackArray = new long[maxSize];

top = -1;

}

public void push(long j) {

stackArray[++top] = j;

}

public long pop() {

return stackArray[top--];

}

public long peek() {

return stackArray[top];

}

public boolean isEmpty() {

return (top == -1);

}

public boolean isFull() {

return (top == maxSize - 1);

}

public static void main(String[] args) {

MyStack theStack = new MyStack(10);

theStack.push(10);

theStack.push(20);

theStack.push(30);

theStack.push(40);

theStack.push(50);

while (!theStack.isEmpty()) {

long value = theStack.pop();

System.out.print(value);

System.out.print(" ");

}

System.out.println("");

}

}

Result

The above code sample will produce the following result.

50 40 30 20 10

# 2. Implement Queue from Scratch

[Queue](http://en.wikipedia.org/wiki/Queue_%28data_structure%29)is a linear structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO).  A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.   
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.  
**Operations on Queue:**   
Mainly the following four basic operations are performed on queue:  
**Enqueue:**Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.   
**Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.   
**Front:**Get the front item from queue.   
**Rear:** Get the last item from queue.   
 

**Applications of Queue:**   
Queue is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n**F**irst **O**ut order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property of Queue makes it also useful in following kind of scenarios.  
**1)** When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.   
**2)**When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.  
See [this](http://introcs.cs.princeton.edu/43stack/)for more detailed applications of Queue and Stack.  
**Array implementation Of Queue**   
For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in circular manner.

// CPP program for array

// implementation of queue

#include <bits/stdc++.h>

using namespace std;

// A structure to represent a queue

class Queue {

public:

int front, rear, size;

unsigned capacity;

int\* array;

};

// function to create a queue

// of given capacity.

// It initializes size of queue as 0

Queue\* createQueue(unsigned capacity)

{

Queue\* queue = new Queue();

queue->capacity = capacity;

queue->front = queue->size = 0;

// This is important, see the enqueue

queue->rear = capacity - 1;

queue->array = new int[queue->capacity];

return queue;

}

// Queue is full when size

// becomes equal to the capacity

int isFull(Queue\* queue)

{

return (queue->size == queue->capacity);

}

// Queue is empty when size is 0

int isEmpty(Queue\* queue)

{

return (queue->size == 0);

}

// Function to add an item to the queue.

// It changes rear and size

void enqueue(Queue\* queue, int item)

{

if (isFull(queue))

return;

queue->rear = (queue->rear + 1)

% queue->capacity;

queue->array[queue->rear] = item;

queue->size = queue->size + 1;

cout << item << " enqueued to queue\n";

}

// Function to remove an item from queue.

// It changes front and size

int dequeue(Queue\* queue)

{

if (isEmpty(queue))

return INT\_MIN;

int item = queue->array[queue->front];

queue->front = (queue->front + 1)

% queue->capacity;

queue->size = queue->size - 1;

return item;

}

// Function to get front of queue

int front(Queue\* queue)

{

if (isEmpty(queue))

return INT\_MIN;

return queue->array[queue->front];

}

// Function to get rear of queue

int rear(Queue\* queue)

{

if (isEmpty(queue))

return INT\_MIN;

return queue->array[queue->rear];

}

// Driver code

int main()

{

Queue\* queue = createQueue(1000);

enqueue(queue, 10);

enqueue(queue, 20);

enqueue(queue, 30);

enqueue(queue, 40);

cout << dequeue(queue)

<< " dequeued from queue\n";

cout << "Front item is "

<< front(queue) << endl;

cout << "Rear item is "

<< rear(queue) << endl;

return 0;

}

**Output:**

10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

10 dequeued from queue

Front item is 20

Rear item is 40

**Complexity Analysis:**

* **Time Complexity:**

**Operations** **Complexity**

Enque(insertion) O(1)

Deque(deletion) O(1)

Front(Get front) O(1)

Rear(Get Rear) O(1)

* **Auxiliary Space:** O(N).   
  N is the size of array for storing elements.

Pros of Array Implementation:

1. Easy to implement.

Cons of Array Implementation:

1. Static Data Structure, fixed size.
2. If the queue has a large number of enqueue and dequeue operations, at some point (in case of linear increment of front and rear indexes) we may not be able to insert elements in the queue even if the queue is empty (this problem is avoided by using circular queue).

# 3. Implement 2 stack in an array

Create a data structure *twoStacks*that represents two stacks. Implementation of *twoStacks*should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.  
push1(int x) –> pushes x to first stack   
push2(int x) –> pushes x to second stack  
pop1() –> pops an element from first stack and return the popped element   
pop2() –> pops an element from second stack and return the popped element  
Implementation of *twoStack*should be space efficient.

## Solution:

**Method 1 (Divide the space in two halves)**   
A simple way to implement two stacks is to divide the array in two halves and assign the half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.   
The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

#include <iostream>

#include <stdlib.h>

using namespace std;

class twoStacks {

int\* arr;

int size;

int top1, top2;

public:

// Constructor

twoStacks(int n)

{

size = n;

arr = new int[n];

top1 = n / 2 + 1;

top2 = n / 2;

}

// Method to push an element x to stack1

void push1(int x)

{

// There is at least one empty

// space for new element

if (top1 > 0) {

top1--;

arr[top1] = x;

}

else {

cout << "Stack Overflow"

<< " By element :" << x << endl;

return;

}

}

// Method to push an element

// x to stack2

void push2(int x)

{

// There is at least one empty

// space for new element

if (top2 < size - 1) {

top2++;

arr[top2] = x;

}

else {

cout << "Stack Overflow"

<< " By element :" << x << endl;

return;

}

}

// Method to pop an element from first stack

int pop1()

{

if (top1 <= size / 2) {

int x = arr[top1];

top1++;

return x;

}

else {

cout << "Stack UnderFlow";

exit(1);

}

}

// Method to pop an element

// from second stack

int pop2()

{

if (top2 >= size / 2 + 1) {

int x = arr[top2];

top2--;

return x;

}

else {

cout << "Stack UnderFlow";

exit(1);

}

}

};

/\* Driver program to test twStacks class \*/

int main()

{

twoStacks ts(5);

ts.push1(5);

ts.push2(10);

ts.push2(15);

ts.push1(11);

ts.push2(7);

cout << "Popped element from stack1 is "

<< " : " << ts.pop1()

<< endl;

ts.push2(40);

cout << "\nPopped element from stack2 is "

<< ": " << ts.pop2()

<< endl;

return 0;

}

**Output:**

Stack Overflow By element :7

Popped element from stack1 is : 11

Stack Overflow By element :40

Popped element from stack2 is : 15

**Complexity Analysis:**

* **Time Complexity:**
  + **Push operation :**O(1)
  + **Pop operation :**O(1)
* **Auxiliary Space:** O(N).   
  Use of array to implement stack so. It is not the space-optimised method as explained above.

**Method 2 (A space efficient implementation)**   
This method efficiently utilizes the available space. It doesn’t cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This check is highlighted in the below code.

#include <iostream>

#include <stdlib.h>

using namespace std;

class twoStacks {

int\* arr;

int size;

int top1, top2;

public:

twoStacks(int n) // constructor

{

size = n;

arr = new int[n];

top1 = -1;

top2 = size;

}

// Method to push an element x to stack1

void push1(int x)

{

// There is at least one empty space for new element

if (top1 < top2 - 1) {

top1++;

arr[top1] = x;

}

else {

cout << "Stack Overflow";

exit(1);

}

}

// Method to push an element x to stack2

void push2(int x)

{

// There is at least one empty

// space for new element

if (top1 < top2 - 1) {

top2--;

arr[top2] = x;

}

else {

cout << "Stack Overflow";

exit(1);

}

}

// Method to pop an element from first stack

int pop1()

{

if (top1 >= 0) {

int x = arr[top1];

top1--;

return x;

}

else {

cout << "Stack UnderFlow";

exit(1);

}

}

// Method to pop an element from second stack

int pop2()

{

if (top2 < size) {

int x = arr[top2];

top2++;

return x;

}

else {

cout << "Stack UnderFlow";

exit(1);

}

}

};

/\* Driver program to test twStacks class \*/

int main()

{

twoStacks ts(5);

ts.push1(5);

ts.push2(10);

ts.push2(15);

ts.push1(11);

ts.push2(7);

cout << "Popped element from stack1 is "

<< ts.pop1();

ts.push2(40);

cout << "\nPopped element from stack2 is "

<< ts.pop2();

return 0;

}

**Output:**

Popped element from stack1 is 11

Popped element from stack2 is 40

**Complexity Analysis:**

* **Time Complexity:**
  + **Push operation :**O(1)
  + **Pop operation :**O(1)
* **Auxiliary Space :**O(N).   
  Use of array to implement stack so it is a space-optimized method.

# 276. Design a stack with operations on middle element

How to implement a stack which will support following operations in**O(1) time complexity**?   
1) push() which adds an element to the top of stack.   
2) pop() which removes an element from top of stack.   
3) findMiddle() which will return middle element of the stack.   
4) deleteMiddle() which will delete the middle element.   
Push and pop are standard stack operations.   
The important question is, whether to use a linked list or array for implementation of stack?

## Solution:

Please note that, we need to find and delete middle element. Deleting an element from middle is not O(1) for array. Also, we may need to move the middle pointer up when we push an element and move down when we pop(). In singly linked list, moving middle pointer in both directions is not possible.   
The idea is to use Doubly Linked List (DLL). We can delete middle element in O(1) time by maintaining mid pointer. We can move mid pointer in both directions using previous and next pointers.   
Following is implementation of push(), pop() and findMiddle() operations. Implementation of deleteMiddle() is left as an exercise. If there are even elements in stack, findMiddle() returns the second middle element. For example, if stack contains {1, 2, 3, 4}, then findMiddle() would return 3.

/\* C++ Program to implement a stack

that supports findMiddle() and

deleteMiddle in O(1) time \*/

#include <bits/stdc++.h>

using namespace std;

/\* A Doubly Linked List Node \*/

class DLLNode {

public:

DLLNode\* prev;

int data;

DLLNode\* next;

};

/\* Representation of the stack data structure

that supports findMiddle() in O(1) time.

The Stack is implemented using Doubly Linked List.

It maintains pointer to head node, pointer to

middle node and count of nodes \*/

class myStack {

public:

DLLNode\* head;

DLLNode\* mid;

int count;

};

/\* Function to create the stack data structure \*/

myStack\* createMyStack()

{

myStack\* ms = new myStack();

ms->count = 0;

return ms;

};

/\* Function to push an element to the stack \*/

void push(myStack\* ms, int new\_data)

{

/\* allocate DLLNode and put in data \*/

DLLNode\* new\_DLLNode = new DLLNode();

new\_DLLNode->data = new\_data;

/\* Since we are adding at the beginning,

prev is always NULL \*/

new\_DLLNode->prev = NULL;

/\* link the old list off the new DLLNode \*/

new\_DLLNode->next = ms->head;

/\* Increment count of items in stack \*/

ms->count += 1;

/\* Change mid pointer in two cases

1) Linked List is empty

2) Number of nodes in linked list is odd \*/

if (ms->count == 1) {

ms->mid = new\_DLLNode;

}

else {

ms->head->prev = new\_DLLNode;

if (!(ms->count

& 1)) // Update mid if ms->count is even

ms->mid = ms->mid->prev;

}

/\* move head to point to the new DLLNode \*/

ms->head = new\_DLLNode;

}

/\* Function to pop an element from stack \*/

int pop(myStack\* ms)

{

/\* Stack underflow \*/

if (ms->count == 0) {

cout << "Stack is empty\n";

return -1;

}

DLLNode\* head = ms->head;

int item = head->data;

ms->head = head->next;

// If linked list doesn't

// become empty, update prev

// of new head as NULL

if (ms->head != NULL)

ms->head->prev = NULL;

ms->count -= 1;

// update the mid pointer when

// we have odd number of

// elements in the stack, i,e

// move down the mid pointer.

if ((ms->count) & 1)

ms->mid = ms->mid->next;

free(head);

return item;

}

// Function for finding middle of the stack

int findMiddle(myStack\* ms)

{

if (ms->count == 0) {

cout << "Stack is empty now\n";

return -1;

}

return ms->mid->data;

}

// Function for deleting middle of the stack

int deletemiddle(myStack\* ms) // IMPROVED BY Sohaib Ayub

{

int temp=ms->mid->data;

if(ms->mid->prev)

ms->mid->prev->next=ms->mid->next;

if(ms->mid->next)

ms->mid->next->prev=ms->mid->prev;

delete ms->mid;

ms->count--;

ms->mid = ms->mid->next; //So that mid does not contain garbage value

return temp;

}

**Output**

Item popped is 77

Item popped is 66

Item popped is 55

Middle Element is 33

Deleted Middle Element is 33

Middle Element is 22

// Driver code

int main()

{

/\* Let us create a stack using push() operation\*/

myStack\* ms = createMyStack();

push(ms, 11);

push(ms, 22);

push(ms, 33);

push(ms, 44);

push(ms, 55);

push(ms, 66);

push(ms, 77);

cout << "Item popped is " << pop(ms) << endl;

cout << "Item popped is " << pop(ms) << endl;

cout << "Item popped is " << pop(ms) << endl;

cout << "Middle Element is " << findMiddle(ms) << endl;

cout << "Deleted Middle Element is "<<deletemiddle(ms)<<endl;

cout << "Middle Element is " << findMiddle(ms) << endl;

return 0;

}

# 277. Implement "N" stacks in an Array

Create a data structure kStacks that represents k stacks. Implementation of kStacks should use only one array, i.e., k stacks should use the same array for storing elements. Following functions must be supported by kStacks.

push(int x, int sn) –> pushes x to stack number ‘sn’ where sn is from 0 to k-1pop(int sn) –> pops an element from stack number ‘sn’ where sn is from 0 to k-1

## Solution:

**Method 1 (Divide the array in slots of size n/k)**  
A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use arr[0] to arr[n/k-1] for first stack, and arr[n/k] to arr[2n/k-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.

**Method 2 (A space efficient implementation)**  
The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two *integer*arrays as extra space.

Following are the two extra arrays are used:  
***1) top[]:***This is of size k and stores indexes of top elements in all stacks.  
***2) next[]:*** This is of size n and stores indexes of next item for the items in array arr[]. Here arr[] is actual array that stores k stacks.  
Together with k stacks, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable ‘free’.

All entries in top[] are initialized as -1 to indicate that all stacks are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, ‘free’ is initialized as 0.

Following is implementation of the above idea.

// A C++ program to demonstrate implementation of k stacks in a single

// array in time and space efficient way

#include<bits/stdc++.h>

using namespace std;

// A C++ class to represent k stacks in a single array of size n

class kStacks

{

int \*arr; // Array of size n to store actual content to be stored in stacks

int \*top; // Array of size k to store indexes of top elements of stacks

int \*next; // Array of size n to store next entry in all stacks

// and free list

int n, k;

int free; // To store beginning index of free list

public:

//constructor to create k stacks in an array of size n

kStacks(int k, int n);

// A utility function to check if there is space available

bool isFull() { return (free == -1); }

// To push an item in stack number 'sn' where sn is from 0 to k-1

void push(int item, int sn);

// To pop an from stack number 'sn' where sn is from 0 to k-1

int pop(int sn);

// To check whether stack number 'sn' is empty or not

bool isEmpty(int sn) { return (top[sn] == -1); }

};

//constructor to create k stacks in an array of size n

kStacks::kStacks(int k1, int n1)

{

// Initialize n and k, and allocate memory for all arrays

k = k1, n = n1;

arr = new int[n];

top = new int[k];

next = new int[n];

// Initialize all stacks as empty

for (int i = 0; i < k; i++)

top[i] = -1;

// Initialize all spaces as free

free = 0;

for (int i=0; i<n-1; i++)

next[i] = i+1;

next[n-1] = -1; // -1 is used to indicate end of free list

}

// To push an item in stack number 'sn' where sn is from 0 to k-1

void kStacks::push(int item, int sn)

{

// Overflow check

if (isFull())

{

cout << "\nStack Overflow\n";

return;

}

int i = free; // Store index of first free slot

// Update index of free slot to index of next slot in free list

free = next[i];

// Update next of top and then top for stack number 'sn'

next[i] = top[sn];

top[sn] = i;

// Put the item in array

arr[i] = item;

}

// To pop an from stack number 'sn' where sn is from 0 to k-1

int kStacks::pop(int sn)

{

// Underflow check

if (isEmpty(sn))

{

cout << "\nStack Underflow\n";

return INT\_MAX;

}

// Find index of top item in stack number 'sn'

int i = top[sn];

top[sn] = next[i]; // Change top to store next of previous top

// Attach the previous top to the beginning of free list

next[i] = free;

free = i;

// Return the previous top item

return arr[i];

}

/\* Driver program to test twStacks class \*/

int main()

{

// Let us create 3 stacks in an array of size 10

int k = 3, n = 10;

kStacks ks(k, n);

// Let us put some items in stack number 2

ks.push(15, 2);

ks.push(45, 2);

// Let us put some items in stack number 1

ks.push(17, 1);

ks.push(49, 1);

ks.push(39, 1);

// Let us put some items in stack number 0

ks.push(11, 0);

ks.push(9, 0);

ks.push(7, 0);

cout << "Popped element from stack 2 is " << ks.pop(2) << endl;

cout << "Popped element from stack 1 is " << ks.pop(1) << endl;

cout << "Popped element from stack 0 is " << ks.pop(0) << endl;

return 0;

}

**Output:**

Popped element from stack 2 is 45

Popped element from stack 1 is 39

Popped element from stack 0 is 7

Time complexities of operations push() and pop() is O(1).

The best part of above implementation is, if there is a slot available in stack, then an item can be pushed in any of the stacks, i.e., no wastage of space.

# 278. Check the expression has valid or Balanced parenthesis or not.

Given an expression string **x**. Examine whether the pairs and the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp.  
For example, the function should return 'true' for exp = “[()]{}{[()()]()}” and 'false' for exp = “[(])”.

**Example 1:**

**Input**:

{([])}

**Output**:

true

**Explanation**:

{ ( [ ] ) }. Same colored brackets can form

balaced pairs, with 0 number of

unbalanced bracket.

**Example 2:**

**Input**:

()

**Output**:

true

**Explanation**:

(). Same bracket can form balanced pairs,

and here only 1 type of bracket is

present and in balanced way.

**Example 3:**

**Input**:

([]

**Output**:

false

**Explanation**:

([]. Here square bracket is balanced but

the small bracket is not balanced and

Hence , the output will be unbalanced.

**Your Task:**  
This is a **function**problem. You only need to complete the function **ispar()**that takes a **string**as a **parameter**and returns a boolean value **true**if **brackets**are **balanced**else **returns false**. The **printing**is done **automatically**by the **driver code**.  
  
**Expected Time Complexity**: O(|x|)  
**Expected Auixilliary Space**: O(|x|)  
  
**Constraints:**  
1 ≤ |x| ≤ 32000

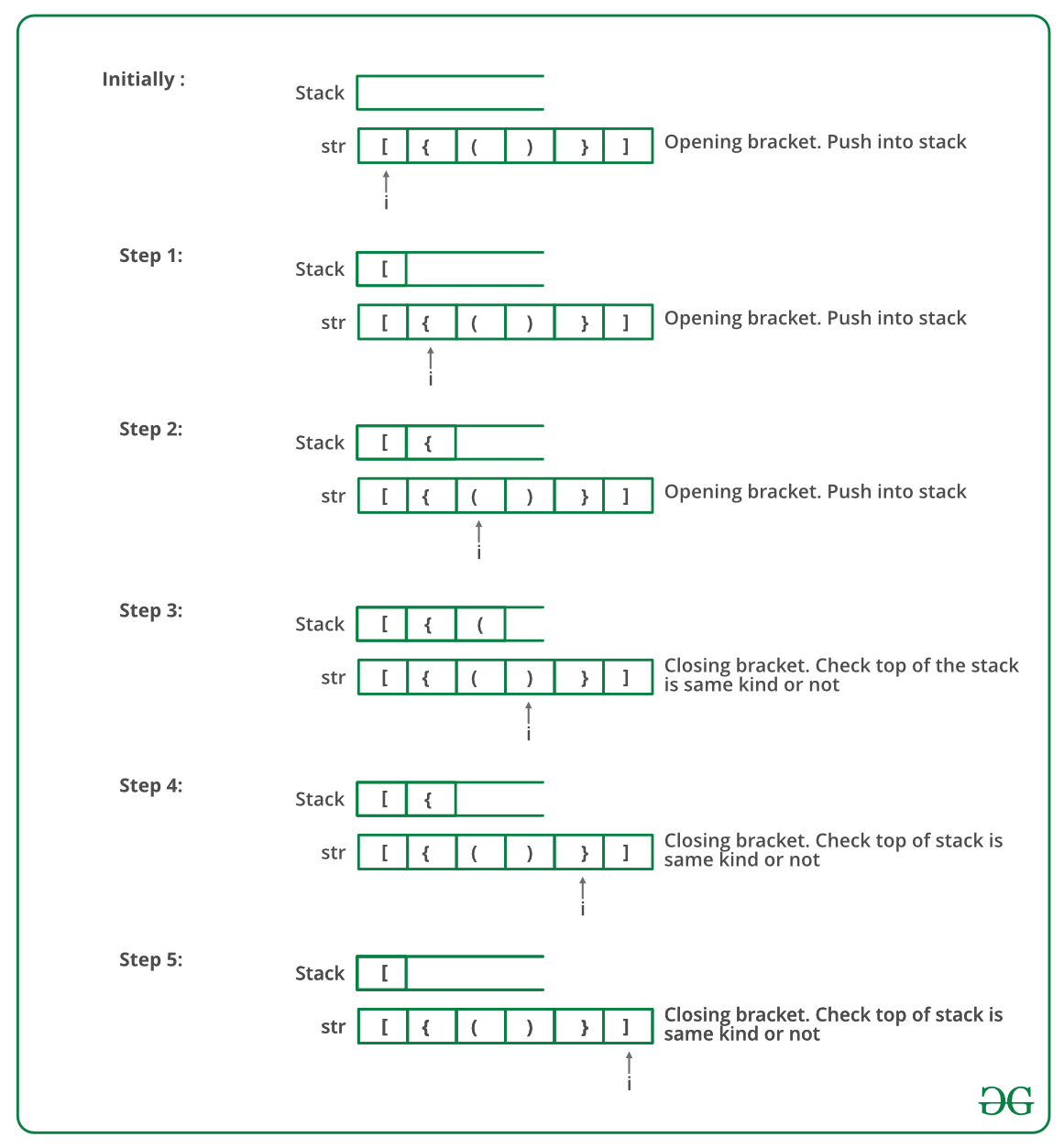
**Note**: The drive code prints "balanced" if function return true, otherwise it prints "not balanced".

## Solution:

**Algorithm:**

* Declare a character [stack](http://www.geeksforgeeks.org/stack-data-structure/) S.
* Now traverse the expression string exp.
  1. If the current character is a starting bracket (**‘(‘ or ‘{‘ or ‘[‘**) then push it to stack.
  2. If the current character is a closing bracket (**‘)’ or ‘}’ or ‘]’**) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.
* After complete traversal, if there is some starting bracket left in stack then “not balanced”

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

// CPP program to check for balanced brackets.

#include <bits/stdc++.h>

using namespace std;

// function to check if brackets are 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());

}

// Driver code

int main()

{

string expr = "{()}[]";

// Function call

if (areBracketsBalanced(expr))

cout << "Balanced";

else

cout << "Not Balanced";

return 0;

}

**Output**

Balanced

**Time Complexity:** O(n)   
**Auxiliary Space:** O(n) for stack.

**My code:**

bool ispar(string x)

{

// Your code here

stack<char> st;

int n = x.size();

for(int i=0;i<n;i++){

if(x[i]=='{' || x[i]=='[' || x[i]=='(')

st.push(x[i]);

else{

if(st.empty())

return false;

if( (st.top()=='{' && x[i]!='}') || (st.top()=='[' && x[i]!=']') || (st.top()=='(' && x[i]!=')') )

return false;

st.pop();

}

}

if(!st.empty())

return false;

return true;

}

279. [Reverse a String using Stack](https://practice.geeksforgeeks.org/problems/reverse-a-string-using-stack/1)

[Reverse a String using Stack](https://practice.geeksforgeeks.org/problems/reverse-a-string-using-stack/1)

# 279. Reverse a String using Stack

You are given a string **S**, the task is to reverse the string using stack.

**Example 1:**

**Input:** S="GeeksforGeeks"

**Output:** skeeGrofskeeG

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function**reverse()** which takes the string **S**as an input parameter and returns the reversed string.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 ≤ length of the string ≤ 100

## Solution:

Following is simple algorithm to reverse a string using stack.

1) Create an empty stack.

2) One by one push all characters of string to stack.

3) One by one pop all characters from stack and put

them back to string.

Following programs implements above algorithm.

// C++ program to reverse a string using stack

#include <bits/stdc++.h>

using namespace std;

// A structure to represent a stack

class Stack

{

public:

int top;

unsigned capacity;

char\* array;

};

// function to create a stack of given

// capacity. It initializes size of stack as 0

Stack\* createStack(unsigned capacity)

{

Stack\* stack = new Stack();

stack->capacity = capacity;

stack->top = -1;

stack->array = new char[(stack->capacity \* sizeof(char))];

return stack;

}

// Stack is full when top is equal to the last index

int isFull(Stack\* stack)

{ return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1

int isEmpty(Stack\* stack)

{ return stack->top == -1; }

// Function to add an item to stack.

// It increases top by 1

void push(Stack\* stack, char item)

{

if (isFull(stack))

return;

stack->array[++stack->top] = item;

}

// Function to remove an item from stack.

// It decreases top by 1

char pop(Stack\* stack)

{

if (isEmpty(stack))

return -1;

return stack->array[stack->top--];

}

// A stack based function to reverse a string

void reverse(char str[])

{

// Create a stack of capacity

//equal to length of string

int n = strlen(str);

Stack\* stack = createStack(n);

// Push all characters of string to stack

int i;

for (i = 0; i < n; i++)

push(stack, str[i]);

// Pop all characters of string and

// put them back to str

for (i = 0; i < n; i++)

str[i] = pop(stack);

}

// Driver code

int main()

{

char str[] = "GeeksQuiz";

reverse(str);

cout << "Reversed string is " << str;

return 0;

}

**Output:**

Reversed string is ziuQskeeG

**Time Complexity:**O(n) where n is number of characters in stack.   
**Auxiliary Space:**O(n) for stack.

A string can also be reversed without using any auxiliary space. Following C, Java, C# and Python programs to implement reverse without using stack.

// C++ program to reverse a string without using stack

#include <bits/stdc++.h>

using namespace std;

// A utility function to swap two characters

void swap(char \*a, char \*b)

{

char temp = \*a;

\*a = \*b;

\*b = temp;

}

// A stack based function to reverse a string

void reverse(char str[])

{

// get size of string

int n = strlen(str), i;

for (i = 0; i < n/2; i++)

swap(&str[i], &str[n-i-1]);

}

// Driver program to test above functions

int main()

{

char str[] = "abc";

reverse(str);

cout<<"Reversed string is "<< str;

return 0;

}

**Output:**

Reversed string is cba

# 280. Design a Stack that supports getMin() in O(1) time and O(1) extra space.

Design a data-structure**SpecialStack**that supports all the stack operations like **push()**, **pop()**,**isEmpty()**, **isFull()** and an additional operation **getMin()** which should return **minimum**element from the SpecialStack. Your task is to **complete all the functions**, using stack data-Structure.

**Example 1:**

**Input:**

Stack: 18 19 29 15 16

**Output:** 15

**Explanation:**

The minimum element of the stack is 15.

**Your Task:**  
Since this is a function problem, you don't need to take inputs. You just have to complete 5 functions, **push()**which takes the stack and an integer x as input and pushes it into the stack; **pop()**which takes the stack as input and pops out the topmost element from the stack; **isEmpty()**which takes the stack as input and returns true/false depending upon whether the stack is empty or not; **isFull()**which takes the stack and the size of the stack as input and returns true/false depending upon whether the stack is full or not (depending upon the  
given size); **getMin()**which takes the stack as input and returns the minimum element of the stack.   
**Note:** The output of the code will be the value returned by **getMin()**function.

**Constraints:**  
1 ≤ N ≤ 104

## Solution:

Use two stacks: one to store actual stack elements and the other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of the auxiliary stack is always the minimum. Let us see how push() and pop() operations work.

**Push(int x) // inserts an element x to Special Stack**  
1) push x to the first stack (the stack with actual elements)   
2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.   
…..a) If x is smaller than y then push x to the auxiliary stack.   
…..b) If x is greater than y then push y to the auxiliary stack.  
**int Pop() // removes an element from Special Stack and return the removed element**  
1) pop the top element from the auxiliary stack.   
2) pop the top element from the actual stack and return it.  
Step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.  
**int getMin() // returns the minimum element from Special Stack**  
1) Return the top element of the auxiliary stack.

We can see that **all the above operations are O(1)**.   
Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15, and 16 are inserted to the SpecialStack.

When we insert 18, both stacks change to following.

Actual Stack

18 <--- top

Auxiliary Stack

18 <---- top

When 19 is inserted, both stacks change to following.

Actual Stack

19 <--- top

18

Auxiliary Stack

18 <---- top

18

When 29 is inserted, both stacks change to following.

Actual Stack

29 <--- top

19

18

Auxiliary Stack

18 <---- top

18

18

When 15 is inserted, both stacks change to following.

Actual Stack

15 <--- top

29

19

18

Auxiliary Stack

15 <---- top

18

18

18

When 16 is inserted, both stacks change to following.

Actual Stack

16 <--- top

15

29

19

18

Auxiliary Stack

15 <---- top

15

18

18

18

The following is the implementation for SpecialStack class. In the below implementation, SpecialStack inherits from Stack and has one Stack object *min* which works as an auxiliary stack.

#include <iostream>

#include <stdlib.h>

using namespace std;

/\* A simple stack class with

basic stack funtionalities \*/

class Stack {

private:

static const int max = 100;

int arr[max];

int top;

public:

Stack() { top = -1; }

bool isEmpty();

bool isFull();

int pop();

void push(int x);

};

/\* Stack's member method to check

if the stack is empty \*/

bool Stack::isEmpty()

{

if (top == -1)

return true;

return false;

}

/\* Stack's member method to check

if the stack is full \*/

bool Stack::isFull()

{

if (top == max - 1)

return true;

return false;

}

/\* Stack's member method to remove

an element from it \*/

int Stack::pop()

{

if (isEmpty()) {

cout << "Stack Underflow";

abort();

}

int x = arr[top];

top--;

return x;

}

/\* Stack's member method to insert

an element to it \*/

void Stack::push(int x)

{

if (isFull()) {

cout << "Stack Overflow";

abort();

}

top++;

arr[top] = x;

}

/\* A class that supports all the stack

operations and one additional

operation getMin() that returns the

minimum element from stack at

any time. This class inherits from

the stack class and uses an

auxiliary stack that holds minimum

elements \*/

class SpecialStack : public Stack {

Stack min;

public:

int pop();

void push(int x);

int getMin();

};

/\* SpecialStack's member method to insert

an element to it. This method

makes sure that the min stack is also

updated with appropriate minimum

values \*/

void SpecialStack::push(int x)

{

if (isEmpty() == true) {

Stack::push(x);

min.push(x);

}

else {

Stack::push(x);

int y = min.pop();

min.push(y);

if (x < y)

min.push(x);

else

min.push(y);

}

}

/\* SpecialStack's member method to

remove an element from it. This method

removes top element from min stack also. \*/

int SpecialStack::pop()

{

int x = Stack::pop();

min.pop();

return x;

}

/\* SpecialStack's member method to get

minimum element from it. \*/

int SpecialStack::getMin()

{

int x = min.pop();

min.push(x);

return x;

}

/\* Driver program to test SpecialStack

methods \*/

int main()

{

SpecialStack s;

s.push(10);

s.push(20);

s.push(30);

cout << s.getMin() << endl;

s.push(5);

cout << s.getMin();

return 0;

}

**Output**

10

5

**Complexity Analysis:**

* **Time Complexity:**
  1. **For insert operation: O(1)** (As insertion ‘push’ in a stack takes constant time)
  2. **For delete operation: O(1)** (As deletion ‘pop’ in a stack takes constant time)
  3. **For ‘Get Min’ operation: O(1)** (As we have used an auxiliary stack which has it’s top as the minimum element)
* **Auxiliary Space:** O(n).   
  Use of auxiliary stack for storing values.

**Space Optimized Version**   
The above approach can be optimized. We can limit the number of elements in the auxiliary stack. We can push only when the incoming element of the main stack is smaller than or equal to the top of the auxiliary stack. Similarly during pop, if the pop-off element equal to the top of the auxiliary stack, remove the top element of the auxiliary stack. Following is the modified implementation of push() and pop().

/\* SpecialStack's member method to

insert an element to it. This method

makes sure that the min stack is

also updated with appropriate minimum

values \*/

void SpecialStack::push(int x)

{

if (isEmpty() == true) {

Stack::push(x);

min.push(x);

}

else {

Stack::push(x);

int y = min.pop();

min.push(y);

/\* push only when the incoming element

of main stack is smaller

than or equal to top of auxiliary stack \*/

if (x <= y)

min.push(x);

}

}

/\* SpecialStack's member method to

remove an element from it. This method

removes top element from min stack also. \*/

int SpecialStack::pop()

{

int x = Stack::pop();

int y = min.pop();

/\* Push the popped element y back

only if it is not equal to x \*/

if (y != x)

min.push(y);

return x;

}

**Complexity Analysis:**

* **Time Complexity:**
  1. **For Insert operation: O(1)** (As insertion ‘push’ in a stack takes constant time)
  2. **For Delete operation: O(1)** (As deletion ‘pop’ in a stack takes constant time)
  3. **For ‘Get Min’ operation: O(1)** (As we have used an auxiliary which has it’s top as the minimum element)
* **Auxiliary Space:** O(n).   
  The complexity in the worst case is the same as above but in other cases, it will take slightly less space than the above approach as repetition is neglected.

**Further optimized O(1) time complexity and O(1) space complexity solution :**  
The above approach can be optimized further and the solution can be made to work in O(1) time complexity and O(1) space complexity. The idea is to store min element found till current insertion) along with all the elements as a reminder of a DUMMY\_VALUE, and the actual element as a multiple of the DUMMY\_VALUE.  
For example, while pushing an element ‘e’ into the stack, store it as **(e \* DUMMY\_VALUE + minFoundSoFar)**, this way we know what was the minimum value present in the stack at the time ‘e’ was being inserted.

To pop the actual value just return e/DUMMY\_VALUE and set the new minimum as **(minFoundSoFar % DUMMY\_VALUE)**.

**Note: Following method will fail if we try to insert DUMMY\_VALUE in the stack, so we have to make our selection of DUMMY\_VALUE carefully.**  
Let’s say the following elements are being inserted in the stack – 3 2 6 1 8 5

**d** is dummy value.

**s** is wrapper stack

**top** is top element of the stack

min is the minimum value at that instant when the elements were inserted/removed

The following steps shows the current state of the above variables at any instant –

1. *s.push(3);  
   min=3****//updated min as stack here is empty*** *s = {3\*d + 3}  
   top = (3\*d + 3)/d = 3*
2. *s.push(2);  
   min = 2****//updated min as min > current element*** *s = {3\*d + 3****->****2\*d + 2}  
   top = (2\*d + 2)/d = 2*
3. *s.push(6);  
   min = 2  
   s = {3\*d + 3****->****2\*d + 2****->****6\*d + 2}  
   top = (6\*d + 2)/d = 6*
4. *s.push(1);  
   min = 1****//updated min as min > current element*** *s = {3\*d + 3-> 2\*d + 2-> 6\*d + 2 -> 1\*d + 1}  
   top = (1\*d + 1)/d = 1*
5. *s.push(8);  
   min = 1  
   s = {3\*d + 3****->****2\*d + 2****->****6\*d + 2****->****1\*d + 1****->****8\*d + 1}  
   top = (8\*d + 1)/d = 8*
6. *s.push(5);  
   min = 1  
   s = {3\*d + 3****->****2\*d + 2****->****6\*d + 2****->****1\*d + 1****->****8\*d + 1****->****5\*d + 1}  
   top = (5\*d + 1)/d = 5*
7. *s.pop();  
   s = {3\*d + 3 -> 2\*d + 2 -> 6\*d + 2 -> 1\*d + 1 -> 8\*d + 1 -> 5\*d + 1}  
   top = (5\*d + 1)/d = 5  
   min = (8\*d + 1)%d = 1****// min is always remainder of the second top element in stack.***
8. *s.pop();  
   s = {3\*d + 3 -> 2\*d + 2-> 6\*d + 2 -> 1\*d + 1 -> 8\*d + 1}  
   top = (8\*d + 1)/d = 8  
   min = (1\*d + 1)%d = 1*
9. *s.pop()  
   s = {3\*d + 3 -> 2\*d + 2-> 6\*d + 2 -> 1\*d + 1}   
   top = (1\*d + 1)/d = 1  
   min = (6\*d + 2)%d = 2*
10. *s.pop()  
    s = {3\*d + 3-> 2\*d + 2-> 6\*d + 2}  
    top = (6\*d + 2)/d = 6  
    min = (2\*d + 2)%d = 2*
11. *s.pop()  
    s = {3\*d + 3-> 2\*d + 2}  
    top = (2\*d + 2)/d = 2  
    min = (3\*d + 3)%d = 3*
12. *s.pop()  
    s = {3\*d + 3}  
    top = (3\*d + 3)/d = 3  
    min  = -1 // since stack is now empty*

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

/\* A special stack having peek() pop() and

\* push() along with additional getMin() that

\* returns minimum value in a stack without

\* using extra space and all operations in O(1)

\* time.. ???? \*/

class SpecialStack

{

// Sentinel value for min

int min = -1;

// DEMO\_VALUE

static const int demoVal = 9999;

stack<int> st;

public:

void getMin()

{

cout << "min is: " << min << endl;

}

void push(int val)

{

// If stack is empty OR current element

// is less than min, update min.

if (st.empty() || val < min)

{

min = val;

}

// Encode the current value with

// demoVal, combine with min and

// insert into stack

st.push(val \* demoVal + min);

cout << "pushed: " << val << endl;

}

int pop()

{

// if stack is empty return -1;

if ( st.empty() ) {

cout << "stack underflow" << endl ;

return -1;

}

int val = st.top();

st.pop();

// If stack is empty, there would

// be no min value present, so

// make min as -1

if (!st.empty())

min = st.top() % demoVal;

else

min = -1;

cout << "popped: " << val / demoVal << endl;

// Decode actual value from

// encoded value

return val / demoVal;

}

int peek()

{

// Decode actual value

// from encoded value

return st.top() / demoVal;

}

};

// Driver Code

int main()

{

SpecialStack s;

vector<int> arr = { 3, 2, 6, 1, 8, 5, 5, 5, 5 };

for(int i = 0; i < arr.size(); i++)

{

s.push(arr[i]);

s.getMin();

}

cout << endl;

for(int i = 0; i < arr.size(); i++)

{

s.pop();

s.getMin();

}

return 0;

}

**Output**

pushed: 3

min is: 3

pushed: 2

min is: 2

pushed: 6

min is: 2

pushed: 1

min is: 1

pushed: 8

min is: 1

pushed: 5

min is: 1

pushed: 5

min is: 1

pushed: 5

min is: 1

pushed: 5

min is: 1

popped: 5

min is: 1

popped: 5

min is: 1

popped: 5

min is: 1

popped: 5

min is: 1

popped: 8

min is: 1

popped: 1

min is: 2

popped: 6

min is: 2

popped: 2

min is: 3

popped: 3

min is: -1

**Complexity Analysis:**

**For push() operation:** O(1) (As insertion ‘push’ in a stack takes constant time)  
**For pop() operation:** O(1) (As pop operation in a stack takes constant time)

**For ‘Get Min’ operation:** O(1) (As we have maintained min variable throughout the code)

**Auxiliary Space:** O(1). No extra space is used.

In this article, a new approach is discussed that supports minimum with O(1) extra space. We define a variable **minEle** that stores the current minimum element in the stack. Now the interesting part is, how to handle the case when minimum element is removed. To handle this, we push “2x – minEle” into the stack instead of x so that previous minimum element can be retrieved using current minEle and its value stored in stack. Below are detailed steps and explanation of working.  
**Push(x)** : Inserts x at the top of stack. 

* If stack is empty, insert x into the stack and make minEle equal to x.
* If stack is not empty, compare x with minEle. Two cases arise:
  + If x is greater than or equal to minEle, simply insert x.
  + If x is less than minEle, insert (2\*x – minEle) into the stack and make minEle equal to x. For example, let previous minEle was 3. Now we want to insert 2. We update minEle as 2 and insert 2\*2 – 3 = 1 into the stack.

**Pop() :**Removes an element from top of stack.

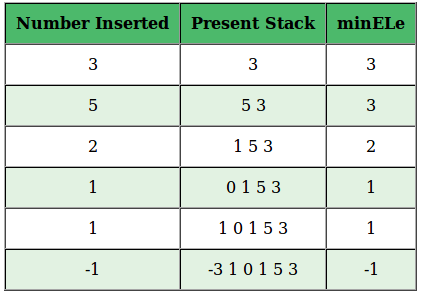
* Remove element from top. Let the removed element be y. Two cases arise:
  + If y is greater than or equal to minEle, the minimum element in the stack is still minEle.
  + If y is less than minEle, the minimum element now becomes (2\*minEle – y), so update (minEle = 2\*minEle – y). This is where we retrieve previous minimum from current minimum and its value in stack. For example, let the element to be removed be 1 and minEle be 2. We remove 1 and update minEle as 2\*2 – 1 = 3.

**Important Points:** 

* Stack doesn’t hold actual value of an element if it is minimum so far.
* Actual minimum element is always stored in minEle

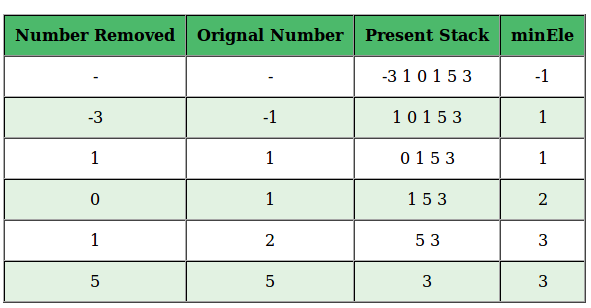
**Illustration** 

**Push(x)** 



* Number to be Inserted: 3, Stack is empty, so insert 3 into stack and minEle = 3.
* Number to be Inserted: 5, Stack is not empty, 5> minEle, insert 5 into stack and minEle = 3.
* Number to be Inserted: 2, Stack is not empty, 2< minEle, insert (2\*2-3 = 1) into stack and minEle = 2.
* Number to be Inserted: 1, Stack is not empty, 1< minEle, insert (2\*1-2 = 0) into stack and minEle = 1.
* Number to be Inserted: 1, Stack is not empty, 1 = minEle, insert 1 into stack and minEle = 1.
* Number to be Inserted: -1, Stack is not empty, -1 < minEle, insert (2\*-1 – 1 = -3) into stack and minEle = -1.

**Pop()** 



* Initially the minimum element minEle in the stack is -1.
* Number removed: -3, Since -3 is less than the minimum element the original number being removed is minEle which is -1, and the new minEle = 2\*-1 – (-3) = 1
* Number removed: 1, 1 == minEle, so number removed is 1 and minEle is still equal to 1.
* Number removed: 0, 0< minEle, original number is minEle which is 1 and new minEle = 2\*1 – 0 = 2.
* Number removed: 1, 1< minEle, original number is minEle which is 2 and new minEle = 2\*2 – 1 = 3.
* Number removed: 5, 5> minEle, original number is 5 and minEle is still 3

// C++ program to implement a stack that supports

// getMinimum() in O(1) time and O(1) extra space.

#include <bits/stdc++.h>

using namespace std;

// A user defined stack that supports getMin() in

// addition to push() and pop()

struct MyStack

{

stack<int> s;

int minEle;

// Prints minimum element of MyStack

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";

}

};

// Driver Code

int main()

{

MyStack s;

s.push(3);

s.push(5);

s.getMin();

s.push(2);

s.push(1);

s.getMin();

s.pop();

s.getMin();

s.pop();

s.peek();

return 0;

}

**Output**

Number Inserted: 3

Number Inserted: 5

Minimum Element in the stack is: 3

Number Inserted: 2

Number Inserted: 1

Minimum Element in the stack is: 1

Top Most Element Removed: 1

Minimum Element in the stack is: 2

Top Most Element Removed: 2

Top Most Element is: 5

**How does this approach work?**   
When element to be inserted is less than minEle, we insert “2x – minEle”. The important thing to notes is, 2x – minEle will always be less than x (proved below), i.e., new minEle and while popping out this element we will see that something unusual has happened as the popped element is less than the minEle. So we will be updating minEle.

How 2\*x - minEle is less than x in push()?

x < minEle which means x - minEle < 0

// Adding x on both sides

x - minEle + x < 0 + x

2\*x - minEle < x

We can conclude 2\*x - minEle < new minEle

While popping out, if we find the element(y) less than the current minEle, we find the new minEle = 2\*minEle – y.

How previous minimum element, prevMinEle is, 2\*minEle - y

in pop() is y the popped element?

// We pushed y as 2x - prevMinEle. Here

// prevMinEle is minEle before y was inserted

y = 2\*x - prevMinEle

// Value of minEle was made equal to x

minEle = x .

new minEle = 2 \* minEle - y

= 2\*x - (2\*x - prevMinEle)

= prevMinEle // This is what we wanted

**Another Approach:**

create a class node which has two variables val and min.  val will store the actual value that we are going to insert in stack ,where as min will store the min value so far seen upto that node. look into the code for better understanding.

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.util.\*;

class MinStack {

Stack<Node> s;

class Node{

int val;

int min;

public Node(int val,int min){

this.val=val;

this.min=min;

}

}

/\*\* initialize your data structure here. \*/

public MinStack() {

this.s=new Stack<Node>();

}

public void push(int x) {

if(s.isEmpty()){

this.s.push(new Node(x,x));

}else{

int min=Math.min(this.s.peek().min,x);

this.s.push(new Node(x,min));

}

}

public int pop() {

return this.s.pop().val;

}

public int top() {

return this.s.peek().val;

}

public int getMin() {

return this.s.peek().min;

}

}

class GFG {

public static void main (String[] args) {

MinStack s=new MinStack();

s.push(-1);

s.push(10);

s.push(-4);

s.push(0);

System.out.println(s.getMin());

System.out.println(s.pop());

System.out.println(s.pop());

System.out.println(s.getMin());

}

}

//time O(1);

//it takes o(n) space since every node has to remember min value

**Output**

-4

0

-4

-1

# 281. Find the next Greater element

Given an array **arr[ ]** of size **N** having distinct elements, the task is to find the next greater element for each element of the array in order of their appearance in the array.  
Next greater element of an element in the array is the nearest element on the right which is greater than the current element.  
If there does not exist next greater of current element, then next greater element for current element is -1. For example, next greater of the last element is always -1.

**Example 1:**

**Input**:

N = 4, arr[] = [1 3 2 4]

**Output**:

3 4 4 -1

**Explanation**:

In the array, the next larger element

to 1 is 3 , 3 is 4 , 2 is 4 and for 4 ?

since it doesn't exist, it is -1.

**Example 2:**

**Input**:

N = 5, arr[] [6 8 0 1 3]

**Output**:

8 -1 1 3 -1

**Explanation**:

In the array, the next larger element to

6 is 8, for 8 there is no larger elements

hence it is -1, for 0 it is 1 , for 1 it

is 3 and then for 3 there is no larger

element on right and hence -1.

**Your Task:**  
This is a **function**problem. You only need to complete the function **nextLargerElement()**that takes list of integers **arr[ ]**and**N** as input parametersand returns list of integers of length N denoting the next greater elements for all the corresponding elements in the input array.

**Expected Time Complexity** : O(N)  
**Expected Auxilliary Space** : O(N)

**Constraints:**  
1 ≤ N ≤ 106  
1 ≤ Ai ≤ 1018

## Solution:

**Method 1 (Simple)**   
Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by the outer loop. If a greater element is found then that element is printed as next, otherwise, -1 is printed.

Below is the implementation of the above approach:

// Simple C++ program to print

// next greater elements in a

// given array

#include<iostream>

using namespace std;

/\* prints element and NGE pair

for all elements of arr[] of size n \*/

void printNGE(int arr[], int n)

{

int next, i, j;

for (i = 0; i < n; i++)

{

next = -1;

for (j = i + 1; j < n; j++)

{

if (arr[i] < arr[j])

{

next = arr[j];

break;

}

}

cout << arr[i] << " -- "

<< next << endl;

}

}

// Driver Code

int main()

{

int arr[] = {11, 13, 21, 3};

int n = sizeof(arr)/sizeof(arr[0]);

printNGE(arr, n);

return 0;

}

**Output**

11 -- 13

13 -- 21

21 -- -1

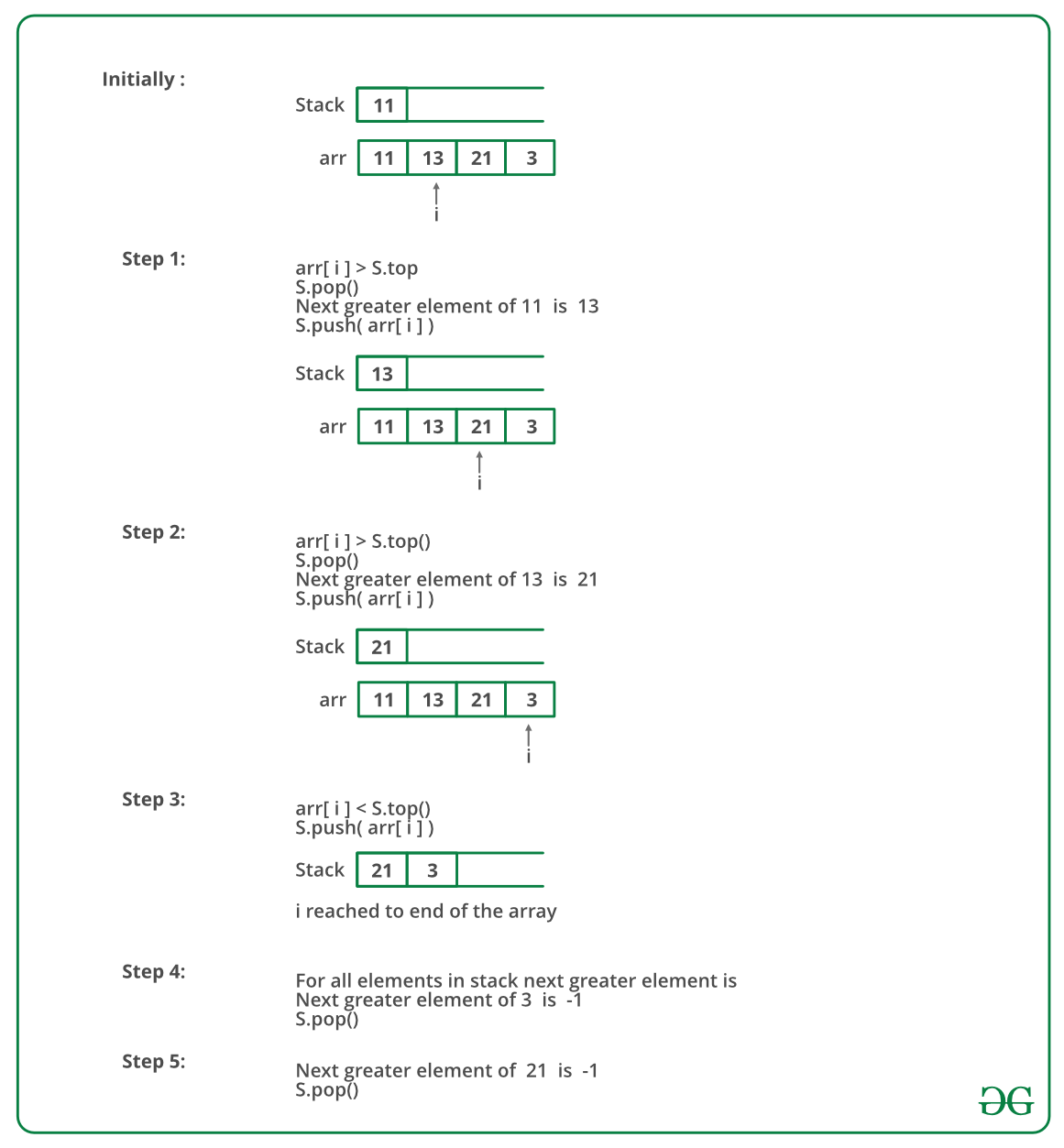
3 -- -1

***Time Complexity:****O(N2)*  
***Auxiliary Space:****O(1)*

**Method 2 (Using Stack)**

* Push the first element to stack.
* Pick rest of the elements one by one and follow the following steps in loop.
  1. Mark the current element as *next*.
  2. If stack is not empty, compare top element of stack with *next*.
  3. If next is greater than the top element, Pop element from stack. *next*is the next greater element for the popped element.
  4. Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements.
* Finally, push the next in the stack.
* After the loop in step 2 is over, pop all the elements from the stack and print -1 as the next element for them.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

// A Stack based C++ program to find next

// greater element for all array elements.

#include <bits/stdc++.h>

using namespace std;

/\* prints element and NGE pair for all

elements of arr[] of size n \*/

void printNGE(int arr[], int n)

{

stack<int> s;

/\* push the first element to stack \*/

s.push(arr[0]);

// iterate for rest of the elements

for (int i = 1; i < n; i++)

{

if (s.empty()) {

s.push(arr[i]);

continue;

}

/\* if stack is not empty, then

pop an element from stack.

If the popped element is smaller

than next, then

a) print the pair

b) keep popping while elements are

smaller and stack is not empty \*/

while (s.empty() == false

&& s.top() < arr[i])

{

cout << s.top()

<< " --> " << arr[i] << endl;

s.pop();

}

/\* push next to stack so that we can find

next greater for it \*/

s.push(arr[i]);

}

/\* After iterating over the loop, the remaining

elements in stack do not have the next greater

element, so print -1 for them \*/

while (s.empty() == false) {

cout << s.top() << " --> " << -1 << endl;

s.pop();

}

}

/\* Driver code \*/

int main()

{

int arr[] = { 11, 13, 21, 3 };

int n = sizeof(arr) / sizeof(arr[0]);

printNGE(arr, n);

return 0;

}

**Output**

11 --> 13

13 --> 21

3 --> -1

21 --> -1

***Time Complexity:****O(N)*  
***Auxiliary Space:****O(N)*

The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.

1. Initially pushed to the stack.
2. Popped from the stack when next element is being processed.
3. Pushed back to the stack because the next element is smaller.
4. Popped from the stack in step 3 of the algorithm.

**How to get elements in the same order as input?**

The above approach may not produce output elements in the same order as the input. To achieve the same order, we can traverse the same in reverse order

Below is the implementation of the above approach:

// A Stack based C++ program to find next

// greater element for all array elements

// in same order as input.

#include <bits/stdc++.h>

using namespace std;

/\* prints element and res pair for all

elements of arr[] of size n \*/

void printNGE(int arr[], int n)

{

stack<int> s;

int res[n];

for (int i = n - 1; i >= 0; i--) {

/\* if stack is not empty, then

pop an element from stack.

If the popped element is smaller

than next, then

a) print the pair

b) keep popping while elements are

smaller and stack is not empty \*/

if (!s.empty()) {

while (!s.empty() && s.top() <= arr[i]) {

s.pop();

}

}

res[i] = s.empty() ? -1 : s.top();

s.push(arr[i]);

}

for (int i = 0; i < n; i++)

cout << arr[i] << " --> " << res[i] << endl;

}

// Driver Code

int main()

{

int arr[] = { 11, 13, 21, 3 };

int n = sizeof(arr) / sizeof(arr[0]);

// Function call

printNGE(arr, n);

return 0;

}

**Output**

11 ---> 13

13 ---> 21

21 ---> -1

3 ---> -1

***Time Complexity:****O(N)*  
***Auxiliary Space:****O(N)*

**Method 3:**

1. This is same as above method but the elements are pushed and popped only once into the stack. The array is changed in place. The array elements are pushed into the stack until it finds a greatest element in the right of array. In other words the elements are popped from stack when top of the stack value is smaller in the current array element.

2. Once all the elements are processed in the array but stack is not empty. The left out elements in the stack doesn’t encounter any greatest element . So pop the element from stack and change it’s index value as -1 in the array.

# Python3 code

class Solution:

def nextLargerElement(self,arr,n):

#code here

s=[]

for i in range(len(arr)):

while s and s[-1].get("value") < arr[i]:

d = s.pop()

arr[d.get("ind")] = arr[i]

s.append({"value": arr[i], "ind": i})

while s:

d = s.pop()

arr[d.get("ind")] = -1

return arr

if \_\_name\_\_ == "\_\_main\_\_":

print(Solution().nextLargerElement([6,8,0,1,3],5))

**Output**

[8, -1, 1, 3, -1]

***Time Complexity:****O(N)*  
***Auxiliary Space:****O(N)*

**My code:**

vector<long long> nextLargerElement(vector<long long> arr, int n){

vector<long long> res(n,-1);

stack<int> st;

st.push(0);

for(int i=1;i<n;i++){

while(!st.empty() && arr[i]>arr[st.top()]){

res[st.top()] = arr[i];

st.pop();

}

st.push(i);

}

return res;

}

# 282. The celebrity Problem

A celebrity is a person who is known to all but does not know anyone at a party. If you go to a party of N people, find if there is a celebrity in the party or not.  
A square NxN matrix M[][] is used to represent people at the party such that if an element of row i and column j  is set to 1 it means ith person knows jth person. Here M[i][i] will always be 0.  
**Note:** Follow 0 based indexing.

**Example 1:**

**Input:**

N = 3

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

{0 0 0},

{0 1 0}}

**Output:** 1

**Explanation:** 0th and 2nd person both

know 1. Therefore, 1 is the celebrity.

**Example 2:**

**Input:**

N = 2

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

{1 0}}

**Output:** -1

**Explanation:** The two people at the party both

know each other. None of them is a celebrity.

**Your Task:**  
You don't need to read input or print anything. Complete the function **celebrity()** which takes the matrix M and its size N as input parameters and returns the index of the celebrity. If no such celebrity is present, return -1.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
2 <= N <= 3000  
0 <= M[][] <= 1

## Solution:

**Method 1:** This uses Graph to arrive at the particular solution.

**Approach:**   
Model the solution using graphs. Initialize indegree and outdegree of every vertex as 0. If A knows B, draw a directed edge from A to B, increase indegree of B and outdegree of A by 1. Construct all possible edges of the graph for every possible pair [i, j]. There are NC2 pairs. If a celebrity is present in the party, there will be one sink node in the graph with outdegree of zero and indegree of N-1.

**Algorithm:**

1. Create two arrays *indegree* and *outdegree*, to store the indegree and outdegree
2. Run a nested loop, the outer loop from 0 to n and inner loop from 0 to n.
3. For every pair i, j check if i knows j then increase the outdegree of i and indegree of j
4. For every pair i, j check if j knows i then increase the outdegree of j and indegree of i
5. Run a loop from 0 to n and find the id where the indegree is n-1 and outdegree is 0

**Implementation:**

// C++ program to find celebrity

#include <bits/stdc++.h>

#include <list>

using namespace std;

// Max # of persons in the party

#define N 8

// Person with 2 is celebrity

bool MATRIX[N][N] = {{0, 0, 1, 0},

{0, 0, 1, 0},

{0, 0, 0, 0},

{0, 0, 1, 0}};

bool knows(int a, int b)

{

return MATRIX[a][b];

}

// Returns -1 if celebrity

// is not present. If present,

// returns id (value from 0 to n-1).

int findCelebrity(int n)

{

//the graph needs not be constructed

//as the edges can be found by

//using knows function

//degree array;

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

//query for all edges

for(int i=0; i<n; i++)

{

for(int j=0; j<n; j++)

{

int x = knows(i,j);

//set the degrees

outdegree[i]+=x;

indegree[j]+=x;

}

}

//find a person with indegree n-1

//and out degree 0

for(int i=0; i<n; i++)

if(indegree[i] == n-1 && outdegree[i] == 0)

return i;

return -1;

}

// Driver code

int main()

{

int n = 4;

int id = findCelebrity(n);

id == -1 ? cout << "No celebrity" :

cout << "Celebrity ID " << id;

return 0;

}

**Output :**

Celebrity ID 2

**Complexity Analysis:**

* **Time Complexity:** O(n2).   
  A nested loop is run traversing the array, SO the time complexity is O(n2)
* **Space Complexity:** O(n).   
  Since extra space of size n is required.

**Approach :**   
The problem can be solved using recursion. Say, if the ‘potential celebrity’ of N-1 persons is known, can the solution to N be found from it? A potential celebrity is one who is the only one left after eliminating n-1 people. n-1 people are eliminated with the following strategy:

* If A knows B, then A cannot be a celebrity. But B could be.
* Else If B knows A, then B cannot be a celebrity. But A could be.

The above-mentioned approach use **Recursion** to find the potential celebrity among n persons, recursively calls n-1 persons, till the base case of 0 persons is reached. For 0 persons -1 is returned indicating that there are no possible celebrities since there are 0 people. In the ith stage of recursion, the ith person and (i-1)th person are compared to check if anyone of them knows the other. And using the above logic (in the bullet points) the potential celebrity is returned to the (i+1)th stage.

Once the recursive function returns an id. We check if this id does not know anybody else, but all others know this id. If this is true, then this id will be the celebrity.

**Algorithm :**

1. Create a recursive function that takes an integer n.
2. Check the base case, if the value of n is 0 then return -1.
3. Call the recursive function and get the ID of the potential celebrity from the first n-1 elements.
4. If the id is -1 then assign n as the potential celebrity and return the value.
5. If the potential  celebrity of first n-1 elements knows n-1 then return n-1, (0 based indexing)
6. If the celebrity of first n-1 elements does not know n-1 then return id of celebrity of n-1 elements, (0 based indexing)
7. Else return -1
8. Create a wrapper function and check whether the id returned by the function is really the celebrity or not.

**Implementation:**

// C++ program to find celebrity

#include <bits/stdc++.h>

#include <list>

using namespace std;

// Max # of persons in the party

#define N 8

// Person with 2 is celebrity

bool MATRIX[N][N] = { { 0, 0, 1, 0 },

{ 0, 0, 1, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 1, 0 } };

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

// Returns -1 if a 'potential celebrity'

// is not present. If present,

// returns id (value from 0 to n-1).

int findPotentialCelebrity(int n)

{

// base case - when n reaches 0 , returns -1

// since n represents the number of people,

// 0 people implies no celebrity(= -1)

if (n == 0)

return -1;

// find the celebrity with n-1

// persons

int id = findPotentialCelebrity(n - 1);

// if there are no celebrities

if (id == -1)

return n - 1;

// if the id knows the nth person

// then the id cannot be a celebrity, but nth person

// could be one

else if (knows(id, n - 1)) {

return n - 1;

}

// if the nth person knows the id,

// then the nth person cannot be a celebrity and the id

// could be one

else if (knows(n - 1, id)) {

return id;

}

// if there is no celebrity

return -1;

}

// Returns -1 if celebrity

// is not present. If present,

// returns id (value from 0 to n-1).

// a wrapper over findCelebrity

int Celebrity(int n)

{

// find the celebrity

int id = findPotentialCelebrity(n);

// check if the celebrity found

// is really the celebrity

if (id == -1)

return id;

else {

int c1 = 0, c2 = 0;

// check the id is really the

// celebrity

for (int i = 0; i < n; i++)

if (i != id) {

c1 += knows(id, i);

c2 += knows(i, id);

}

// if the person is known to

// everyone.

if (c1 == 0 && c2 == n - 1)

return id;

return -1;

}

}

// Driver code

int main()

{

int n = 4;

int id = Celebrity(n);

id == -1 ? cout << "No celebrity"

: cout << "Celebrity ID " << id;

return 0;

}

**Output :**

Celebrity ID 2

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  The recursive function is called n times, so the time complexity is O(n).
* **Space Complexity:** O(1).   
  As no extra space is required.

**Approach:** There are some observations based on elimination technique (Refer *Polya’s How to Solve It* book).

* If A knows B, then A can’t be a celebrity. Discard A, and *B may be celebrity*.
* If A doesn’t know B, then B can’t be a celebrity. Discard B, and *A may be celebrity*.
* Repeat above two steps till there is only one person.
* Ensure the remained person is a celebrity. (What is the need of this step?)

**Algorithm:**

1. Create a stack and push all the id’s in the stack.
2. Run a loop while there are more than 1 element in the stack.
3. Pop top two element from the stack (represent them as A and B)
4. If A knows B, then A can’t be a celebrity and push B in stack. Else if A doesn’t know B, then B can’t be a celebrity push A in stack.
5. Assign the remaining element in the stack as the celebrity.
6. Run a loop from 0 to n-1 and find the count of persons who knows the celebrity and the number of people whom the celebrity knows. if the count of persons who knows the celebrity is n-1 and the count of people whom the celebrity knows is 0 then return the id of celebrity else return -1.

**Implementation:**

// C++ program to find celebrity

#include <bits/stdc++.h>

#include <list>

using namespace std;

// Max # of persons in the party

#define N 8

// Person with 2 is celebrity

bool MATRIX[N][N] = {{0, 0, 1, 0},

{0, 0, 1, 0},

{0, 0, 0, 0},

{0, 0, 1, 0}};

bool knows(int a, int b)

{

return MATRIX[a][b];

}

// Returns -1 if celebrity

// is not present. If present,

// returns id (value from 0 to n-1).

int findCelebrity(int n)

{

// Handle trivial

// case of size = 2

stack<int> s;

// Celebrity

int C;

// Push everybody to stack

for (int i = 0; i < n; i++)

s.push(i);

// Extract top 2

// Find a potential celebrity

while (s.size() > 1)

{ int A = s.top();

s.pop();

int B = s.top();

s.pop();

if (knows(A, B))

{

s.push(B);

}

else

{

s.push(A);

}

}

// If there are only two people

// and there is no

// potential candicate

if(s.empty())

return -1;

// Potential candidate?

C = s.top();

s.pop();

// Check if C is actually

// a celebrity or not

for (int i = 0; i < n; i++)

{

// If any person doesn't

// know 'C' or 'C' doesn't

// know any person, return -1

if ( (i != C) &&

(knows(C, i) ||

!knows(i, C)) )

return -1;

}

return C;

}

// Driver code

int main()

{

int n = 4;

int id = findCelebrity(n);

id == -1 ? cout << "No celebrity" :

cout << "Celebrity ID " << id;

return 0;

}

**Output :**

Celebrity ID 2

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  The total number of comparisons 3(N-1), so the time complexity is O(n).
* **Space Complexity:** O(n).   
  n extra space is needed to store the stack.

**Optimal Approach:** The idea is to use two pointers, one from start and one from the end. Assume the start person is A, and the end person is B. If A knows B, then A must not be the celebrity. Else, B must not be the celebrity. At the end of the loop, only one index will be left as a celebrity. Go through each person again and check whether this is the celebrity.   
The **Two Pointer approach** can be used where two pointers can be assigned, one at the start and the other at the end, and the elements can be compared and the search space can be reduced. 

**Algorithm :**

1. Create two indices i and j, where i = 0 and j = n-1
2. Run a loop until i is less than j.
3. Check if i knows j, then i can’t be a celebrity. so increment i, i.e. i++
4. Else j cannot be a celebrity, so decrement j, i.e. j–
5. Assign i as the celebrity candidate
6. Now at last check that whether the candidate is actually a celebrity by re-running a loop from 0 to n-1  and constantly checking that if the candidate knows a person or if there is a candidate who does not know the candidate, then we should return -1. else at the end of the loop, we can be sure that the candidate is actually a celebrity.

**Implementation:**

// C++ program to find celebrity

// in the given Matrix of people

#include<bits/stdc++.h>

using namespace std;

#define N 4

int celebrity(int M[N][N], int n)

{

// This function returns the celebrity

// index 0-based (if any)

int i = 0, j = n - 1;

while (i < j) {

if (M[j][i] == 1) // j knows i

j--;

else // j doesnt know i so i cant be celebrity

i++;

}

// i points to our celebrity candidate

int candidate = i;

// Now, all that is left is to check that whether

// the candidate is actually a celebrity i.e: he is

// known by everyone but he knows no one

for (i = 0; i < n; i++) {

if (i != candidate) {

if (M[i][candidate] == 0

|| M[candidate][i] == 1)

return -1;

}

}

// if we reach here this means that the candidate

// is really a celebrity

return candidate;

}

int main()

{

int M[N][N] = { { 0, 0, 1, 0 },

{ 0, 0, 1, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 1, 0 } };

int celebIdx = celebrity(M, 4);

if (celebIdx == -1)

cout<<("No celebrity found!");

else {

cout<<

"0-based celebrity index is : " << celebIdx;

}

return 0;

}

**Output :**

0-based celebrity index is : 2

**Complexity Analysis:**

* **Time Complexity:** O(n)
* **Space Complexity:** O(1) No extra space is required.

# 283. Arithmetic Expression evaluation

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between ( A + B )\*C and A + ( B \* C ) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

**1. Polish notation (prefix notation) –**   
It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e.,

+AB

**2. Reverse Polish notation(postfix notation) –**   
It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e., 

AB+

Stack-organized computers are better suited for post-fix notation than the traditional infix notation. Thus, the infix notation must be converted to the postfix notation. The conversion from infix notation to postfix notation must take into consideration the operational hierarchy.

There are 3 levels of precedence for 5 binary operators as given below: 

Highest: Exponentiation (^)

Next highest: Multiplication (\*) and division (/)

Lowest: Addition (+) and Subtraction (-)

**For example –** 

Infix notation: (A-B)\*[C/(D+E)+F]

Post-fix notation: AB- CDE +/F +\*

Here, we first perform the arithmetic inside the parentheses (A-B) and (D+E). The division of C/(D+E) must be done prior to the addition with F. After that multiply the two terms inside the parentheses and bracket.

Now we need to calculate the value of these arithmetic operations by using a stack.

The procedure for getting the result is: 

1. Convert the expression in Reverse Polish notation( post-fix notation).
2. Push the operands into the stack in the order they appear.
3. When any operator encounters then pop two topmost operands for executing the operation.
4. After execution push the result obtained into the stack.
5. After the complete execution of expression, the final result remains on the top of the stack.

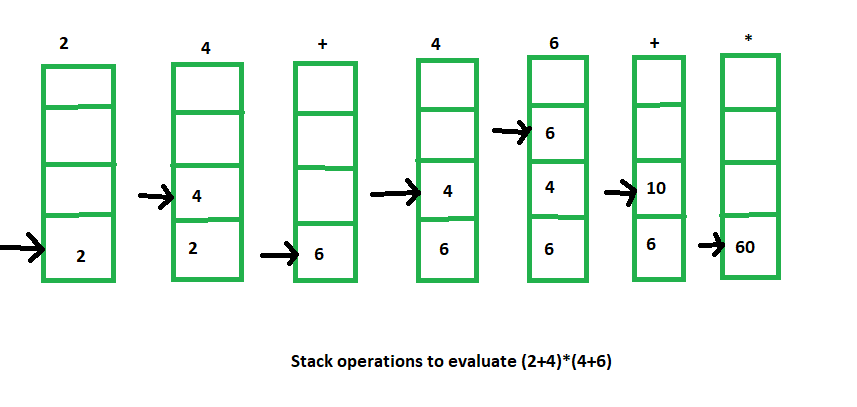
**For example –** 

Infix notation: (2+4) \* (4+6)

Post-fix notation: 2 4 + 4 6 + \*

Result: 60

The stack operations for this expression evaluation is shown below:



# 284. Evaluation of Postfix expression

Given string **S** representing a postfix expression, the task is to evaluate the expression and find the final value. Operators will only include the basic arithmetic operators like **\*, /, + and -**.

**Example 1:**

**Input**: S = "231\*+9-"

**Output**: -4

**Explanation**:

After solving the given expression,

we have -4 as result.

**Example 2:**

**Input**: S = "123+\*8-"

**Output**: -3

**Explanation**:

After solving the given postfix

expression, we have -3 as result.

**Your Task:**  
You do not need to read input or print anything. Complete the function**evaluatePostfixExpression()**that takes the string S denoting the expression as input parameter and returnsthe evaluated value.

**Expected Time Complexity**: O(|S|)  
**Expected Auixilliary Space**: O(|S|)

**Constraints:**  
1 ≤ |S| ≤ 105

0 ≤ |Si|≤ 9 (And given operators)

## Solution:

Following is an algorithm for evaluation postfix expressions.   
1) Create a stack to store operands (or values).   
2) Scan the given expression and do the following for every scanned element.   
…..a) If the element is a number, push it into the stack   
…..b) If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack   
3) When the expression is ended, the number in the stack is the final answer

**Example:**   
Let the given expression be “2 3 1 \* + 9 -“. We scan all elements one by one.   
1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’   
2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)   
3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’   
4) Scan ‘\*’, it’s an operator, pop two operands from stack, apply the \* operator on operands, we get 3\*1 which results in 3. We push the result ‘3’ to stack. The stack now becomes ‘2 3’.   
5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result ‘5’ to stack. The stack now becomes ‘5’.   
6) Scan ‘9’, it’s a number, we push it to the stack. The stack now becomes ‘5 9’.   
7) Scan ‘-‘, it’s an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result ‘-4’ to the stack. The stack now becomes ‘-4’.   
8) There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).

Below is the implementation of the above algorithm.

// C++ program to evaluate value of a postfix expression

#include <iostream>

#include <string.h>

using namespace std;

// Stack type

struct Stack

{

int top;

unsigned capacity;

int\* array;

};

// Stack Operations

struct Stack\* createStack( unsigned capacity )

{

struct Stack\* stack = (struct Stack\*) malloc(sizeof(struct Stack));

if (!stack) return NULL;

stack->top = -1;

stack->capacity = capacity;

stack->array = (int\*) malloc(stack->capacity \* sizeof(int));

if (!stack->array) return NULL;

return stack;

}

int isEmpty(struct Stack\* stack)

{

return stack->top == -1 ;

}

char peek(struct Stack\* stack)

{

return stack->array[stack->top];

}

char pop(struct Stack\* stack)

{

if (!isEmpty(stack))

return stack->array[stack->top--] ;

return '$';

}

void push(struct Stack\* stack, char op)

{

stack->array[++stack->top] = op;

}

// The main function that returns value of a given postfix expression

int evaluatePostfix(char\* exp)

{

// Create a stack of capacity equal to expression size

struct Stack\* stack = createStack(strlen(exp));

int i;

// See if stack was created successfully

if (!stack) return -1;

// Scan all characters one by one

for (i = 0; exp[i]; ++i)

{

// If the scanned character is an operand (number here),

// push it to the stack.

if (isdigit(exp[i]))

push(stack, exp[i] - '0');

// If the scanned character is an operator, pop two

// elements from stack apply the operator

else

{

int val1 = pop(stack);

int val2 = pop(stack);

switch (exp[i])

{

case '+': push(stack, val2 + val1); break;

case '-': push(stack, val2 - val1); break;

case '\*': push(stack, val2 \* val1); break;

case '/': push(stack, val2/val1); break;

}

}

}

return pop(stack);

}

// Driver program to test above functions

int main()

{

char exp[] = "231\*+9-";

cout<<"postfix evaluation: "<< evaluatePostfix(exp);

return 0;

}

**Output:**

postfix evaluation: -4

The time complexity of the evaluation algorithm is O(n) where n is a number of characters in the input expression.

There are the following limitations of the above implementation.   
1) It supports only 4 binary operators ‘+’, ‘\*’, ‘-‘ and ‘/’. It can be extended for more operators by adding more switch cases.   
2) The allowed operands are only single-digit operands. The program can be extended for multiple digits by adding a separator-like space between all elements (operators and operands) of the given expression. 

Below given is the extended program which allows operands to have multiple digits.

// C++ program to evaluate value of a postfix

// expression having multiple digit operands

#include <bits/stdc++.h>

using namespace std;

// Stack type

class Stack

{

public:

int top;

unsigned capacity;

int\* array;

};

// Stack Operations

Stack\* createStack( unsigned capacity )

{

Stack\* stack = new Stack();

if (!stack) return NULL;

stack->top = -1;

stack->capacity = capacity;

stack->array = new int[(stack->capacity \* sizeof(int))];

if (!stack->array) return NULL;

return stack;

}

int isEmpty(Stack\* stack)

{

return stack->top == -1 ;

}

int peek(Stack\* stack)

{

return stack->array[stack->top];

}

int pop(Stack\* stack)

{

if (!isEmpty(stack))

return stack->array[stack->top--] ;

return '$';

}

void push(Stack\* stack,int op)

{

stack->array[++stack->top] = op;

}

// The main function that returns value

// of a given postfix expression

int evaluatePostfix(char\* exp)

{

// Create a stack of capacity equal to expression size

Stack\* stack = createStack(strlen(exp));

int i;

// See if stack was created successfully

if (!stack) return -1;

// Scan all characters one by one

for (i = 0; exp[i]; ++i)

{

//if the character is blank space then continue

if(exp[i] == ' ')continue;

// If the scanned character is an

// operand (number here),extract the full number

// Push it to the stack.

else if (isdigit(exp[i]))

{

int num=0;

//extract full number

while(isdigit(exp[i]))

{

num = num \* 10 + (int)(exp[i] - '0');

i++;

}

i--;

//push the element in the stack

push(stack,num);

}

// If the scanned character is an operator, pop two

// elements from stack apply the operator

else

{

int val1 = pop(stack);

int val2 = pop(stack);

switch (exp[i])

{

case '+': push(stack, val2 + val1); break;

case '-': push(stack, val2 - val1); break;

case '\*': push(stack, val2 \* val1); break;

case '/': push(stack, val2/val1); break;

}

}

}

return pop(stack);

}

// Driver code

int main()

{

char exp[] = "100 200 + 2 / 5 \* 7 +";

cout << evaluatePostfix(exp);

return 0;

}

**Output :**

757

# 285. Implement a method to insert an element at its bottom without using any other data structure.

Given a [stack](https://www.geeksforgeeks.org/stack-data-structure/) **S** and an integer **N**, the task is to insert **N** at the bottom of the stack.

**Examples:**

***Input:****N = 7  
S = 1 <- (Top)  
      2  
     3  
     4  
     5****Output:****1 2 3 4 5 7*

***Input:****N = 17  
S = 1 <- (Top)  
     12  
     34  
     47  
     15****Output:****1 12 34 47 15 17*

## Solution:

Instead of using a temporary stack, the *implicit stack* can be used through [recursion](http://www.geeksforgeeks.org/recursion/). Follow the steps below to solve the problem:

1. Define a [recursion function](https://www.geeksforgeeks.org/recursive-functions/) that accepts the stack **S** and an integer as parameters and returns a stack.
2. Base case to be considered is if the [stack is empty](https://www.geeksforgeeks.org/stack-empty-and-stack-size-in-c-stl/). For this scenario, [push **N** into the stack](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/) and return it.
3. Otherwise, remove the [top element of **S**](https://www.geeksforgeeks.org/stack-top-c-stl/)and store it in a variable, say **X**.
4. Recurse again using the new stack
5. Push **X** into **S**.

Below is the implementation of the above approach:

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Recursive function to use implicit stack

// to insert an element at the bottom of stack

stack<int> recur(stack<int> S, int N)

{

// If stack is empty

if (S.empty())

S.push(N);

else {

// Stores the top element

int X = S.top();

// Pop the top element

S.pop();

// Recurse with remaining elements

S = recur(S, N);

// Push the previous

// top element again

S.push(X);

}

return S;

}

// Function to insert an element

// at the bottom of stack

void insertToBottom(

stack<int> S, int N)

{

// Recursively insert

// N at the bottom of S

S = recur(S, N);

// Print the stack S

while (!S.empty()) {

cout << S.top() << " ";

S.pop();

}

}

// Driver Code

int main()

{

// Input

stack<int> S;

S.push(5);

S.push(4);

S.push(3);

S.push(2);

S.push(1);

int N = 7;

insertToBottom(S, N);

return 0;

}

**Output:**

1 2 3 4 5 7

***Time Complexity:****O(N), where N is the total number of elements in the* *stack.*  
***Auxiliary Space:****O(N)*

**If it is allowed to use a temporary stack:**

**Naive Approach:** The simplest approach would be to create another stack. Follow the steps below to solve the problem:

1. [Initialize a stack](https://www.geeksforgeeks.org/stack-in-cpp-stl/), say **temp**.
2. Keep popping from the given stack **S** and [pushing the popped elements into **temp**](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/), until the stack **S** becomes empty.
3. [Push **N** into the stack **S**](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/).
4. Now, keep popping from the stack**temp**and push the popped elements into the stack **S**, until the [stack **temp** becomes empty](https://www.geeksforgeeks.org/stack-empty-and-stack-size-in-c-stl/).

Below is the implementation of the above approach:

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to insert an element

// at the bottom of a given stack

void insertToBottom(stack<int> S, int N)

{

// Temporary stack

stack<int> temp;

// Iterate until S becomes empty

while (!S.empty()) {

// Push the top element of S

// into the stack temp

temp.push(S.top());

// Pop the top element of S

S.pop();

}

// Push N into the stack S

S.push(N);

// Iterate until temp becomes empty

while (!temp.empty()) {

// Push the top element of

// temp into the stack S

S.push(temp.top());

// Pop the top element of temp

temp.pop();

}

// Print the elements of S

while (!S.empty()) {

cout << S.top() << " ";

S.pop();

}

}

// Driver Code

int main()

{

// Input

stack<int> S;

S.push(5);

S.push(4);

S.push(3);

S.push(2);

S.push(1);

int N = 7;

insertToBottom(S, N);

return 0;

}

**Output:**

1 2 3 4 5 7

***Time Complexity:****O(N)*  
***Auxiliary Space:****O(N)*

# 286. Reverse a stack using recursion

Write a program to reverse a stack using recursion. You are not allowed to use loop constructs like while, for..etc, and you can only use the following ADT functions on Stack S:   
isEmpty(S)   
push(S)   
pop(S)

## Solution:

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one at the bottom of the stack.   
For example, let the input stack be

1 <-- top

2

3

4

First 4 is inserted at the bottom.

4 <-- top

Then 3 is inserted at the bottom

4 <-- top

3

Then 2 is inserted at the bottom

4 <-- top

3

2

Then 1 is inserted at the bottom

4 <-- top

3

2

1

So we need a function that inserts at the bottom of a stack using the above given basic stack function.

**void insertAtBottom(():**First pops all stack items and stores the popped item in function call stack using recursion. And when stack becomes empty, pushes new item and all items stored in call stack.

**void reverse():** This function mainly uses insertAtBottom() to pop all items one by one and insert the popped items at the bottom.

// C++ code to reverse a

// stack using recursion

#include<bits/stdc++.h>

using namespace std;

// using std::stack for

// stack implementation

stack<char> st;

// initializing a string to store

// result of reversed stack

string ns;

// Below is a recursive function

// that inserts an element

// at the bottom of a stack.

char insert\_at\_bottom(char x)

{

if(st.size() == 0)

st.push(x);

else

{

// All items are held in Function Call

// Stack until we reach end of the stack

// When the stack becomes empty, the

// st.size() becomes 0, the above if

// part is executed and the item is

// inserted at the bottom

char a = st.top();

st.pop();

insert\_at\_bottom(x);

// push allthe items held in

// Function Call Stack

// once the item is inserted

// at the bottom

st.push(a);

}

}

// Below is the function that

// reverses the given stack using

// insert\_at\_bottom()

char reverse()

{

if(st.size()>0)

{

// Hold all items in Function

// Call Stack until we

// reach end of the stack

char x = st.top();

st.pop();

reverse();

// Insert all the items held

// in Function Call Stack

// one by one from the bottom

// to top. Every item is

// inserted at the bottom

insert\_at\_bottom(x);

}

}

// Driver Code

int main()

{

// push elements into

// the stack

st.push('1');

st.push('2');

st.push('3');

st.push('4');

cout<<"Original Stack"<<endl;

// print the elements

// of original stack

cout<<"1"<<" "<<"2"<<" "

<<"3"<<" "<<"4"

<<endl;

// function to reverse

// the stack

reverse();

cout<<"Reversed Stack"

<<endl;

// storing values of reversed

// stack into a string for display

while(!st.empty())

{

char p=st.top();

st.pop();

ns+=p;

}

//display of reversed stack

cout<<ns[3]<<" "<<ns[2]<<" "

<<ns[1]<<" "<<ns[0]<<endl;

return 0;

}

**Output:**

Original Stack

1 2 3 4

Reversed Stack

4 3 2 1

**Time Complexity**: This approach takes the worst time complexity of O(n^2),

**Space Complexity**: This approach takes the space complexity of O(n) for stack of function calls.

**Note:** Without stack, we can easily do it using another temporary stack.

# 287. Sort a Stack

Given a stack, the task is to sort it such that the top of the stack has the greatest element.

**Example 1:**

**Input:**

Stack: 3 2 1

**Output:** 3 2 1

**Example 2:**

**Input:**

Stack: 11 2 32 3 41

**Output:** 41 32 11 3 2

**Your Task:**  
You don't have to read input or print anything. Your task is to complete the function **sort()**which sorts the elements present in the given stack. (The sorted stack is printed by the driver's code by popping the elements of the stack.)

**Expected Time Complexity**: O(N\*N)  
**Expected Auxilliary Space**: O(N) recursive.

**Constraints:**  
1<=N<=100  
  
**Note:**The **Input/Ouput** format and **Example** given are used for system's internal purpose, and should be used by a user for **Expected Output** only. As it is a function problem, hence a user should not read any input from stdin/console. The task is to complete the function specified, and not to write the full code.

## Solution:

**Sort a stack using recursion**

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one in sorted order. Here sorted order is important.

**Algorithm**   
We can use below algorithm to sort stack elements:

sortStack(stack S)

if stack is not empty:

temp = pop(S);

sortStack(S);

sortedInsert(S, temp);

Below algorithm is to insert element is sorted order:

sortedInsert(Stack S, element)

if stack is empty OR element > top element

push(S, elem)

else

temp = pop(S)

sortedInsert(S, element)

push(S, temp)

**Illustration:**

Let given stack be

-3 <-- top of the stack

14

18

-5

30

Let us illustrate sorting of stack using above example:  
First pop all the elements from the stack and store popped element in variable ‘temp’. After poping all the elements function’s stack frame will look like:

temp = -3 --> stack frame #1

temp = 14 --> stack frame #2

temp = 18 --> stack frame #3

temp = -5 --> stack frame #4

temp = 30 --> stack frame #5

Now stack is empty and ‘insert\_in\_sorted\_order()’ function is called and it inserts 30 (from stack frame #5) at the bottom of the stack. Now stack looks like below:

**30** <-- top of the stack

Now next element i.e. -5 (from stack frame #4) is picked. Since -5 < 30, -5 is inserted at the bottom of stack. Now stack becomes:

30 <-- top of the stack

-5

Next 18 (from stack frame #3) is picked. Since 18 < 30, 18 is inserted below 30. Now stack becomes:

30 <-- top of the stack

**18**

-5

Next 14 (from stack frame #2) is picked. Since 14 < 30 and 14 < 18, it is inserted below 18. Now stack becomes:

30 <-- top of the stack

18

**14**

-5

Now -3 (from stack frame #1) is picked, as -3 < 30 and -3 < 18 and -3 < 14, it is inserted below 14. Now stack becomes:

30 <-- top of the stack

18

14

**-3**

-5

**Implementation:**

Below is the implementation of the above algorithm.

// C++ program to sort a stack using recursion

#include <iostream>

using namespace std;

// Stack is represented using linked list

struct stack {

int data;

struct stack\* next;

};

// Utility function to initialize stack

void initStack(struct stack\*\* s) { \*s = NULL; }

// Utility function to check if stack is empty

int isEmpty(struct stack\* s)

{

if (s == NULL)

return 1;

return 0;

}

// Utility function to push an item to stack

void push(struct stack\*\* s, int x)

{

struct stack\* p = (struct stack\*)malloc(sizeof(\*p));

if (p == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

return;

}

p->data = x;

p->next = \*s;

\*s = p;

}

// Utility function to remove an item from stack

int pop(struct stack\*\* s)

{

int x;

struct stack\* temp;

x = (\*s)->data;

temp = \*s;

(\*s) = (\*s)->next;

free(temp);

return x;

}

// Function to find top item

int top(struct stack\* s) { return (s->data); }

// Recursive function to insert an item x in sorted way

void sortedInsert(struct stack\*\* s, int x)

{

// Base case: Either stack is empty or newly inserted

// item is greater than top (more than all existing)

if (isEmpty(\*s) or x > top(\*s)) {

push(s, x);

return;

}

// If top is greater, remove the top item and recur

int temp = pop(s);

sortedInsert(s, x);

// Put back the top item removed earlier

push(s, temp);

}

// Function to sort stack

void sortStack(struct stack\*\* s)

{

// If stack is not empty

if (!isEmpty(\*s)) {

// Remove the top item

int x = pop(s);

// Sort remaining stack

sortStack(s);

// Push the top item back in sorted stack

sortedInsert(s, x);

}

}

// Utility function to print contents of stack

void printStack(struct stack\* s)

{

while (s) {

cout << s->data << " ";

s = s->next;

}

cout << "\n";

}

// Driver code

int main(void)

{

struct stack\* top;

initStack(&top);

push(&top, 30);

push(&top, -5);

push(&top, 18);

push(&top, 14);

push(&top, -3);

cout << "Stack elements before sorting:\n";

printStack(top);

sortStack(&top);

cout << "\n";

cout << "Stack elements after sorting:\n";

printStack(top);

return 0;

}

**Output:**

Stack elements before sorting:

-3 14 18 -5 30

Stack elements after sorting:

30 18 14 -3 -5

**Complexity Analysis:**

* **Time Complexity:** O(n2).   
  In the worst case for every **sortstack()**, sortedinsert() is called for ‘N’ times recursively for putting element to the right place
* **Auxiliary Space:**O(N)  
  Use of stack data structure for storing values

**Sort a stack using a temporary stack**

We follow this algorithm.

1. Create a temporary stack say **tmpStack**.
2. While input stack is NOT empty do this:
   * Pop an element from input stack call it **temp**
   * while temporary stack is NOT empty and top of temporary stack is greater than temp,   
     pop from temporary stack and push it to the input stack
   * push **temp** in temporary stack
3. The sorted numbers are in tmpStack

Here is a dry run of the above pseudo code.

input: [34, 3, 31, 98, 92, 23]

Element taken out: 23

input: [34, 3, 31, 98, 92]

tmpStack: [23]

Element taken out: 92

input: [34, 3, 31, 98]

tmpStack: [23, 92]

Element taken out: 98

input: [34, 3, 31]

tmpStack: [23, 92, 98]

Element taken out: 31

input: [34, 3, 98, 92]

tmpStack: [23, 31]

Element taken out: 92

input: [34, 3, 98]

tmpStack: [23, 31, 92]

Element taken out: 98

input: [34, 3]

tmpStack: [23, 31, 92, 98]

Element taken out: 3

input: [34, 98, 92, 31, 23]

tmpStack: [3]

Element taken out: 23

input: [34, 98, 92, 31]

tmpStack: [3, 23]

Element taken out: 31

input: [34, 98, 92]

tmpStack: [3, 23, 31]

Element taken out: 92

input: [34, 98]

tmpStack: [3, 23, 31, 92]

Element taken out: 98

input: [34]

tmpStack: [3, 23, 31, 92, 98]

Element taken out: 34

input: [98, 92]

tmpStack: [3, 23, 31, 34]

Element taken out: 92

input: [98]

tmpStack: [3, 23, 31, 34, 92]

Element taken out: 98

input: []

tmpStack: [3, 23, 31, 34, 92, 98]

final sorted list: [3, 23, 31, 34, 92, 98]

// C++ program to sort a stack using an

// auxiliary stack.

#include <bits/stdc++.h>

using namespace std;

// This function return the sorted stack

stack<int> sortStack(stack<int> &input)

{

stack<int> tmpStack;

while (!input.empty())

{

// pop out the first element

int tmp = input.top();

input.pop();

// while temporary stack is not empty and top

// of stack is greater than temp

while (!tmpStack.empty() && tmpStack.top() > tmp)

{

// pop from temporary stack and push

// it to the input stack

input.push(tmpStack.top());

tmpStack.pop();

}

// push temp in temporary of stack

tmpStack.push(tmp);

}

return tmpStack;

}

// main function

int main()

{

stack<int> input;

input.push(34);

input.push(3);

input.push(31);

input.push(98);

input.push(92);

input.push(23);

// This is the temporary stack

stack<int> tmpStack = sortStack(input);

cout << "Sorted numbers are:\n";

while (!tmpStack.empty())

{

cout << tmpStack.top()<< " ";

tmpStack.pop();

}

}

**Output:**

Sorted numbers are:

98 92 34 31 23 3

# 288. Merge Overlapping Intervals

Given a set of time intervals in any order, merge all overlapping intervals into one and output the result which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity.   
For example, let the given set of intervals be {{1,3}, {2,4}, {5,7}, {6,8}}. The intervals {1,3} and {2,4} overlap with each other, so they should be merged and become {1, 4}. Similarly, {5, 7} and {6, 8} should be merged and become {5, 8}

Write a function that produces the set of merged intervals for the given set of intervals.

## Solution:

A **simple approach** is to start from the first interval and compare it with all other intervals for overlapping, if it overlaps with any other interval, then remove the other interval from the list and merge the other into the first interval. Repeat the same steps for remaining intervals after first. This approach cannot be implemented in better than O(n^2) time.  
An **efficient approach** is to first sort the intervals according to the starting time. Once we have the sorted intervals, we can combine all intervals in a linear traversal. The idea is, in sorted array of intervals, if interval[i] doesn’t overlap with interval[i-1], then interval[i+1] cannot overlap with interval[i-1] because starting time of interval[i+1] must be greater than or equal to interval[i]. Following is the detailed step by step algorithm.

Sort the intervals based on increasing order of

starting time.

**2.** Push the first interval on to a stack.

**3.** For each interval do the following

**a.** If the current interval does not overlap with the stack

top, push it.

**b.** If the current interval overlaps with stack top and ending

time of current interval is more than that of stack top,

update stack top with the ending time of current interval.

**4.** At the end stack contains the merged intervals.

Below is an implementation of the above approach.

// A C++ program for merging overlapping intervals

#include<bits/stdc++.h>

using namespace std;

// An interval has start time and end time

struct Interval

{

int start, end;

};

// Compares two intervals according to their staring time.

// This is needed for sorting the intervals using library

// function std::sort(). See http://goo.gl/iGspV

bool compareInterval(Interval i1, Interval i2)

{

return (i1.start < i2.start);

}

// The main function that takes a set of intervals, merges

// overlapping intervals and prints the result

void mergeIntervals(Interval arr[], int n)

{

// Test if the given set has at least one interval

if (n <= 0)

return;

// Create an empty stack of intervals

stack<Interval> s;

// sort the intervals in increasing order of start time

sort(arr, arr+n, compareInterval);

// push the first interval to stack

s.push(arr[0]);

// Start from the next interval and merge if necessary

for (int i = 1 ; i < n; i++)

{

// get interval from stack top

Interval top = s.top();

// if current interval is not overlapping with stack top,

// push it to the stack

if (top.end < arr[i].start)

s.push(arr[i]);

// Otherwise update the ending time of top if ending of current

// interval is more

else if (top.end < arr[i].end)

{

top.end = arr[i].end;

s.pop();

s.push(top);

}

}

// Print contents of stack

cout << "\n The Merged Intervals are: ";

while (!s.empty())

{

Interval t = s.top();

cout << "[" << t.start << "," << t.end << "] ";

s.pop();

}

return;

}

// Driver program

int main()

{

Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };

int n = sizeof(arr)/sizeof(arr[0]);

mergeIntervals(arr, n);

return 0;

}

Output:

The Merged Intervals are: [1,9]

Time complexity of the method is O(nLogn) which is for sorting. Once the array of intervals is sorted, merging takes linear time.  
**A O(n Log n) and O(1) Extra Space Solution**   
The above solution requires O(n) extra space for the stack. We can avoid the use of extra space by doing merge operations in-place. Below are detailed steps.

1) Sort all intervals in increasing order of start time.

2) Traverse sorted intervals starting from first interval,

do following for every interval.

a) If current interval is not first interval and it

overlaps with previous interval, then merge it with

previous interval. Keep doing it while the interval

overlaps with the previous one.

b) Else add current interval to output list of intervals.

Note that if intervals are sorted by decreasing order of start times, we can quickly check if intervals overlap or not by comparing the start time of the previous interval with the end time of the current interval.  
Below is the implementation of the above algorithm.

// C++ program to merge overlapping Intervals in

// O(n Log n) time and O(1) extra space.

#include<bits/stdc++.h>

using namespace std;

// An Interval

struct Interval

{

int s, e;

};

// Function used in sort

bool mycomp(Interval a, Interval b)

{ return a.s < b.s; }

void mergeIntervals(Interval arr[], int n)

{

// Sort Intervals in increasing order of

// start time

sort(arr, arr+n, mycomp);

int index = 0; // Stores index of last element

// in output array (modified arr[])

// Traverse all input Intervals

for (int i=1; i<n; i++)

{

// If this is not first Interval and overlaps

// with the previous one

if (arr[index].e >= arr[i].s)

{

// Merge previous and current Intervals

arr[index].e = max(arr[index].e, arr[i].e);

arr[index].s = min(arr[index].s, arr[i].s);

}

else {

index++;

arr[index] = arr[i];

}

}

// Now arr[0..index-1] stores the merged Intervals

cout << "\n The Merged Intervals are: ";

for (int i = 0; i <= index; i++)

cout << "[" << arr[i].s << ", " << arr[i].e << "] ";

}

// Driver program

int main()

{

Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };

int n = sizeof(arr)/sizeof(arr[0]);

mergeIntervals(arr, n);

return 0;

}

**Output:**

The Merged Intervals are: [1,9]

# 289. Largest rectangular Area in Histogram

Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have the same width and the width is**1 unit**, there will be **N** bars height of each bar will be given by the array **arr**.

**Example 1:**

**Input:**

N = 7

arr[] = {6,2,5,4,5,1,6}

**Output:** 12

**Explanation:**



**Example 2:**

**Input:**

N = 8

arr[] = {7 2 8 9 1 3 6 5}

**Output:** 16

**Explanation:** Maximum size of the histogram

will be 8  and there will be 2 consecutive

histogram. And hence the area of the

histogram will be 8x2 = 16.

**Your Task:**  
The task is to complete the function **getMaxArea**() which takes the array arr[] and its size N as inputs and finds the largest rectangular area possible and **returns** the answer.

**Expected Time Complxity** : O(N)  
**Expected Auxilliary Space** : O(N)

**Constraints:**  
1 ≤ N ≤ 106  
1 ≤ arr[i] ≤ 1012

## Solution:

For every bar 'x', we calculate the area with 'x' as the smallest bar in the rectangle. If we calculate such area for every bar 'x' and find the maximum of all areas, our task is done. How to calculate area with 'x' as smallest bar? We need to know index of the first smaller (smaller than 'x') bar on left of 'x' and index of first smaller bar on right of 'x'. Let us call these indexes as 'left index' and 'right index' respectively.   
We traverse all bars from left to right, maintain a stack of bars. Every bar is pushed to stack once. A bar is popped from stack when a bar of smaller height is seen. When a bar is popped, we calculate the area with the popped bar as smallest bar. How do we get left and right indexes of the popped bar - the current index tells us the 'right index' and index of previous item in stack is the 'left index'. Following is the complete algorithm.  
**1)**Create an empty stack.  
**2)**Start from first bar, and do following for every bar 'hist[i]' where 'i' varies from 0 to n-1.   
......**a)** If stack is empty or hist[i] is higher than the bar at top of stack, then push 'i' to stack.   
......**b)** If this bar is smaller than the top of stack, then keep removing the top of stack while top of the stack is greater. Let the removed bar be hist[tp]. Calculate area of rectangle with hist[tp] as smallest bar. For hist[tp], the 'left index' is previous (previous to tp) item in stack and 'right index' is 'i' (current index).  
**3)** If the stack is not empty, then one by one remove all bars from stack and do step 2.b for every removed bar.

Following is implementation of the above algorithm.

// C++ program to find maximum rectangular area in

// linear time

#include<bits/stdc++.h>

using namespace std;

// The main function to find the maximum rectangular

// area under given histogram with n bars

int getMaxArea(int hist[], int n)

{

// Create an empty stack. The stack holds indexes

// of hist[] array. The bars stored in stack are

// always in increasing order of their heights.

stack<int> s;

int max\_area = 0; // Initialize max area

int tp; // To store top of stack

int area\_with\_top; // To store area with top bar

// as the smallest bar

// Run through all bars of given histogram

int i = 0;

while (i < n)

{

// If this bar is higher than the bar on top

// stack, push it to stack

if (s.empty() || hist[s.top()] <= hist[i])

s.push(i++);

// If this bar is lower than top of stack,

// then calculate area of rectangle with stack

// top as the smallest (or minimum height) bar.

// 'i' is 'right index' for the top and element

// before top in stack is 'left index'

else

{

tp = s.top(); // store the top index

s.pop(); // pop the top

// Calculate the area with hist[tp] stack

// as smallest bar

area\_with\_top = hist[tp] \* (s.empty() ? i :

i - s.top() - 1);

// update max area, if needed

if (max\_area < area\_with\_top)

max\_area = area\_with\_top;

}

}

// Now pop the remaining bars from stack and calculate

// area with every popped bar as the smallest bar

while (s.empty() == false)

{

tp = s.top();

s.pop();

area\_with\_top = hist[tp] \* (s.empty() ? i :

i - s.top() - 1);

if (max\_area < area\_with\_top)

max\_area = area\_with\_top;

}

return max\_area;

}

// Driver program to test above function

int main()

{

int hist[] = {6, 2, 5, 4, 5, 1, 6};

int n = sizeof(hist)/sizeof(hist[0]);

cout << "Maximum area is " << getMaxArea(hist, n);

return 0;

}

**Output**

Maximum area is 12

**Time Complexity:**Since every bar is pushed and popped only once, the time complexity of this method is O(n).

**Space Complexity :**O(n)

**Another Efficient Approach :**By finding next smaller element and previous smaller element for every element in **O(n) time complexity and O(n) auxiliary space** .  
**Step 1 :**First we will take two arrays **left\_smaller[]**and **right\_smaller[]**and initialize it with -1 and n respectively.

**Step 2 :** For every element we will store the index of previous smaller and next smaller element in left\_smaller[] and right\_smaller[] arrays respectively.

                (It will take O(n) time)

**Step 3 :** Now for every element we will calculate area by taking this ith element as the smallest in the range left\_smaller[i] and right\_smaller[i] and multiplying it with the difference of left\_smaller[i] and right\_smaller[i].

**Step 4 :** We can find the maximum of all the area calculated in step 3 to get the desired maximum area.

#include <bits/stdc++.h>

using namespace std;

//Function to find largest rectangular area possible in a given histogram.

int getMaxArea(int arr[], int n)

{

// Your code here

//we create an empty stack here.

stack<int> s;

//we push -1 to the stack because for some elements there will be no previous

//smaller element in the array and we can store -1 as the index for previous smaller.

s.push(-1);

int area = arr[0];

int i = 0;

//We declare left\_smaller and right\_smaller array of size n and initialize them with -1 and n as their default value.

//left\_smaller[i] will store the index of previous smaller element for ith element of the array.

//right\_smaller[i] will store the index of next smaller element for ith element of the array.

vector<int> left\_smaller(n, -1), right\_smaller(n, n);

while(i<n){

while(!s.empty()&&s.top()!=-1&&arr[s.top()]>arr[i]){

//if the current element is smaller than element with index stored on the

//top of stack then, we pop the top element and store the current element index

//as the right\_smaller for the poped element.

right\_smaller[s.top()] = i;

s.pop();

}

if(i>0&&arr[i]==arr[i-1]){

//we use this condition to avoid the unnecessary loop to find the left\_smaller.

//since the previous element is same as current element, the left\_smaller will always be the same for both.

left\_smaller[i] = left\_smaller[i-1];

}else{

//Element with the index stored on the top of the stack is always smaller than the current element.

//Therefore the left\_smaller[i] will always be s.top().

left\_smaller[i] = s.top();

}

s.push(i);

i++;

}

for(int j = 0; j<n; j++){

//here we find area with every element as the smallest element in their range and compare it with the previous area.

// in this way we get our max Area form this.

area = max(area, arr[j]\*(right\_smaller[j]-left\_smaller[j]-1));

}

return area;

}

int main()

{

int hist[] = {6, 2, 5, 4, 5, 1, 6};

int n = sizeof(hist)/sizeof(hist[0]);

cout << "maxArea = " << getMaxArea(hist, n) << endl;

return 0;

}

**Output**

maxArea = 12

**Time Complexity :**O(n)

**Space Complexity :**O(n)

# 290. [Length of the Longest Valid Substring](https://practice.geeksforgeeks.org/problems/valid-substring0624/1)

Given a string **S**consisting only of opening and closing parenthesis 'ie **'('**and **')'**, find out the length of the longest valid(well-formed) parentheses substring.  
**NOTE:**Length of the smallest valid substring **( )** is **2**.

**Example 1:**

**Input: S** = "(()("

**Output:** 2

**Explanation:** The longest valid

substring is "()". Length = 2.

**Example 2:**

**Input: S** = "()(())("

**Output:** 6

**Explanation:** The longest valid

substring is "()(())". Length = 6.

**Your Task:**  
You dont need to read input or print anything. Complete the function **findMaxLen()**which takes S as input parameter and returns the maxlength.

**Expected Time Complexity:**O(n)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 <= |S| <= 105

## Solution:

A **Simple Approach** is to find all the substrings of given string. For every string, check if it is a valid string or not. If valid and length is more than maximum length so far, then update maximum length. We can check whether a substring is valid or not in linear time using a stack (See [this](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/) for details). Time complexity of this solution is O(n2.

An **Efficient Solution** can solve this problem in O(n) time. The idea is to store indexes of previous starting brackets in a stack. The first element of the stack is a special element that provides index before the beginning of valid substring (base for next valid string).

1) Create an empty stack and push -1 to it.

The first element of the stack is used

to provide a base for the next valid string.

2) Initialize result as 0.

3) If the character is '(' i.e. str[i] == '('),

push index'i' to the stack.

2) Else (if the character is ')')

a) Pop an item from the stack (Most of the

time an opening bracket)

b) If the stack is not empty, then find the

length of current valid substring by taking

the difference between the current index and

top of the stack. If current length is more

than the result, then update the result.

c) If the stack is empty, push the current index

as a base for the next valid substring.

3) Return result.

Below is the implementation of the above algorithm.

// C++ program to find length of the

// longest valid substring

#include <bits/stdc++.h>

using namespace std;

int findMaxLen(string str)

{

int n = str.length();

// Create a stack and push -1 as

// initial index to it.

stack<int> stk;

stk.push(-1);

// Initialize result

int result = 0;

// Traverse all characters of given string

for (int i = 0; i < n; i++)

{

// If opening bracket, push index of it

if (str[i] == '(')

stk.push(i);

// If closing bracket, i.e.,str[i] = ')'

else

{

// Pop the previous opening

// bracket's index

if (!stk.empty())

{

stk.pop();

}

// Check if this length formed with base of

// current valid substring is more than max

// so far

if (!stk.empty())

result = max(result, i - stk.top());

// If stack is empty. push current index as

// base for next valid substring (if any)

else

stk.push(i);

}

}

return result;

}

// Driver code

int main()

{

string str = "((()()";

// Function call

cout << findMaxLen(str) << endl;

str = "()(()))))";

// Function call

cout << findMaxLen(str) << endl;

return 0;

}

**Output**

4

6

**Explanation with example:**

Input: str = "(()()"

Initialize result as 0 and stack with one item -1.

For i = 0, str[0] = '(', we push 0 in stack

For i = 1, str[1] = '(', we push 1 in stack

For i = 2, str[2] = ')', currently stack has

[-1, 0, 1], we pop from the stack and the stack

now is [-1, 0] and length of current valid substring

becomes 2 (we get this 2 by subtracting stack top from

current index).

Since the current length is more than the current result,

we update the result.

For i = 3, str[3] = '(', we push again, stack is [-1, 0, 3].

For i = 4, str[4] = ')', we pop from the stack, stack

becomes [-1, 0] and length of current valid substring

becomes 4 (we get this 4 by subtracting stack top from

current index).

Since current length is more than current result,

we update result.

Another **Efficient Approach** can solve the problem in O(n) time. The idea is to maintain an array that stores the length of the longest valid substring ending at that index. We iterate through the array and return the maximum value.

1) Create an array longest of length n (size of the input

string) initialized to zero.

The array will store the length of the longest valid

substring ending at that index.

2) Initialize result as 0.

3) Iterate through the string from second character

a) If the character is '(' set longest[i]=0 as no

valid sub-string will end with '('.

b) Else

i) if s[i-1] = '('

set longest[i] = longest[i-2] + 2

ii) else

set longest[i] = longest[i-1] + 2 +

longest[i-longest[i-1]-2]

4) In each iteration update result as the maximum of

result and longest[i]

5) Return result.

Below is the implementations of the above algorithm.

// C++ program to find length of the longest valid

// substring

#include <bits/stdc++.h>

using namespace std;

int findMaxLen(string s)

{

if (s.length() <= 1)

return 0;

// Initialize curMax to zero

int curMax = 0;

vector<int> longest(s.size(), 0);

// Iterate over the string starting from second index

for (int i = 1; i < s.length(); i++)

{

if (s[i] == ')' && i - longest[i - 1] - 1 >= 0

&& s[i - longest[i - 1] - 1] == '(')

{

longest[i]

= longest[i - 1] + 2

+ ((i - longest[i - 1] - 2 >= 0)

? longest[i - longest[i - 1] - 2]

: 0);

curMax = max(longest[i], curMax);

}

}

return curMax;

}

// Driver code

int main()

{

string str = "((()()";

// Function call

cout << findMaxLen(str) << endl;

str = "()(()))))";

// Function call

cout << findMaxLen(str) << endl;

return 0;

}

**Output**

4

6

**Another approach in O(1) auxiliary space and O(N) Time complexity:**

1. The idea to solve this problem is to traverse the string on and keep track of the count of open parentheses and close parentheses with the help of two counters **left** and **right** respectively.
2. First, the string is traversed from the left towards the right and for every “**(**” encountered, the **left counter** is incremented by 1 and for every “**)**” the **right counter** is incremented by 1.
3. Whenever the left becomes equal to right, the length of the current valid string is calculated and if it greater than the current longest substring, then value of required longest substring is updated with current string length.
4. If the right counter becomes greater than the left counter, then the set of parentheses has become **invalid** and hence the left and right counters are **set to 0**.
5. After the above process, the string is similarly traversed from right to left and similar procedure is applied.

Below is the implementation of the above approach:

// C++ program to implement the above approach

#include <bits/stdc++.h>

using namespace std;

// Function to return the length of

// the longest valid substring

int solve(string s, int n)

{

// Variables for left and right counter.

// maxlength to store the maximum length found so far

int left = 0, right = 0, maxlength = 0;

// Iterating the string from left to right

for (int i = 0; i < n; i++)

{

// If "(" is encountered,

// then left counter is incremented

// else right counter is incremented

if (s[i] == '(')

left++;

else

right++;

// Whenever left is equal to right, it signifies

// that the subsequence is valid and

if (left == right)

maxlength = max(maxlength, 2 \* right);

// Reseting the counters when the subsequence

// becomes invalid

else if (right > left)

left = right = 0;

}

left = right = 0;

// Iterating the string from right to left

for (int i = n - 1; i >= 0; i--) {

// If "(" is encountered,

// then left counter is incremented

// else right counter is incremented

if (s[i] == '(')

left++;

else

right++;

// Whenever left is equal to right, it signifies

// that the subsequence is valid and

if (left == right)

maxlength = max(maxlength, 2 \* left);

// Reseting the counters when the subsequence

// becomes invalid

else if (left > right)

left = right = 0;

}

return maxlength;

}

//A much shorter and concise version of the above code

int solve2(string s, int n){

int left=0,right=0,maxlength=0,t=2;

while(t--){

left=0;

right=0;

for(int i=0;i<n;i++){

if(s[i]=='(')left++;

else right++;

if(left==right){

maxlength=max(maxlength,2\*left);

}

//when travelling from 0 to n-1

if(t%2==1 && right>left){

left=0;

right=0;

}

//when travelling from n-1 to 0

if(t%2==0 && left>right){

left=0;

right=0;

}

}

//now we need to do the same thing from the other side;

reverse(s.begin(),s.end());

}

return maxlength;

}

// Driver code

int main()

{

// Function call

cout << solve("((()()()()(((())", 16);

return 0;

}

**Output**

8

# 291. Expression contains redundant bracket or not

Given a string of balanced expression, find if it contains a redundant parenthesis or not. A set of parenthesis are redundant if the same sub-expression is surrounded by unnecessary or multiple brackets. Print ‘Yes’ if redundant, else ‘No’.  
**Note:** Expression may contain ‘**+**‘, ‘**\***‘, ‘**–**‘ and ‘**/**‘ operators. Given expression is **valid** and there are **no white** spaces present.  
**Example:** 

**Input:**

((a+b))

(a+(b)/c)

(a+b\*(c-d))

**Output:**

Yes

Yes

No

**Explanation:**

1. ((a+b)) can reduced to (a+b), this Redundant

2. (a+(b)/c) can reduced to (a+b/c) because **b** is

surrounded by **()** which is redundant.

3. (a+b\*(c-d)) doesn't have any redundant or multiple

brackets.

## Solution:

The idea is to use stack, which is discussed in [this](https://www.geeksforgeeks.org/find-expression-duplicate-parenthesis-not/) article. For any sub-expression of expression, if we are able to pick any sub-expression of expression surrounded by (), then we again left with () as part of string, we have redundant braces.   
We iterate through the given expression and for each character in the expression, if the character is an open parenthesis ‘(‘ or any of the operators or operands, we push it to the stack. If the character is close parenthesis ‘)’, then pop characters from the stack till matching open parenthesis ‘(‘ is found.   
Now for redundancy two condition will arise while popping-

1. If immediate pop hits an open parenthesis ‘(‘, then we have found a duplicate parenthesis. For example, **(((a+b))+c)** has duplicate brackets around **a+b**. When we reach the second “)” after a+b, we have “**((**” in the stack. Since the top of stack is an opening bracket, we conclude that there are duplicate brackets.
2. If immediate pop doesn’t hit any operand(‘\*’, ‘+’, ‘/’, ‘-‘) then it indicates the presence of unwanted brackets surrounded by expression. For instance, **(a)+b** contain unwanted **()** around **a** thus it is redundant.

/\* C++ Program to check whether valid

expression is redundant or not\*/

#include <bits/stdc++.h>

using namespace std;

// Function to check redundant brackets in a

// balanced expression

bool checkRedundancy(string& str)

{

// create a stack of characters

stack<char> st;

// Iterate through the given expression

for (auto& ch : str) {

// if current character is close parenthesis ')'

if (ch == ')') {

char top = st.top();

st.pop();

// If immediate pop have open parenthesis '('

// duplicate brackets found

bool flag = true;

while (!st.empty() and top != '(') {

// Check for operators in expression

if (top == '+' || top == '-' ||

top == '\*' || top == '/')

flag = false;

// Fetch top element of stack

top = st.top();

st.pop();

}

// If operators not found

if (flag == true)

return true;

}

else

st.push(ch); // push open parenthesis '(',

// operators and operands to stack

}

return false;

}

// Function to check redundant brackets

void findRedundant(string& str)

{

bool ans = checkRedundancy(str);

if (ans == true)

cout << "Yes\n";

else

cout << "No\n";

}

// Driver code

int main()

{

string str = "((b\*c)\*(a/b))";

findRedundant(str);

str = "(a+(b)/c)";

findRedundant(str);

str = "(a+b\*(c-d))";

findRedundant(str);

return 0;

}

**Output**

Yes

Yes

No

# 292. Implement Stack using Queue

Implement a Stack using two queues**q1** and**q2**.

**Example 1:**

**Input:**

push(2)

push(3)

pop()

push(4)

pop()

**Output:** 3 4

**Explanation:**

push(2) the stack will be {2}

push(3) the stack will be {2 3}

pop() poped element will be 3 the

  stack will be {2}

push(4) the stack will be {2 4}

pop()   poped element will be 4

**Example 2:**

**Input:**

push(2)

pop()

pop()

push(3)

**Output:** 2 -1

**Your Task:**

Since this is a function problem, you don't need to take inputs. You are required to complete the two methods **push()** which takes an integer **'x'** as input denoting the element to be pushed into the stack and **pop()** which returns the integer poped out from the stack(**-1** if the stack is empty).

**Expected Time Complexity:**O(1) for **push()**and O(N) for **pop()**(or vice-versa).  
**Expected Auxiliary Space:**O(1) for both **push()**and **pop()**.

**Constraints:**  
1 <=Number of queries <= 100  
1 <= values of the stack <= 100

## Solution:

**Method 1 (By making push operation costly)**   
This method makes sure that newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’. ‘q2’ is used to put every new element at front of ‘q1’.

1. **push(s, x)** operation’s step are described below:
   * Enqueue x to q2
   * One by one dequeue everything from q1 and enqueue to q2.
   * Swap the names of q1 and q2
2. **pop(s)** operation’s function are described below:
   * Dequeue an item from q1 and return it.

Below is the implementation of the above approach:

/\* Program to implement a stack using

two queue \*/

#include <bits/stdc++.h>

using namespace std;

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;

}

};

// Driver code

int main()

{

Stack s;

s.push(1);

s.push(2);

s.push(3);

cout << "current size: " << s.size()

<< endl;

cout << s.top() << endl;

s.pop();

cout << s.top() << endl;

s.pop();

cout << s.top() << endl;

cout << "current size: " << s.size()

<< endl;

return 0;

}

**Output :**

current size: 3

3

2

1

current size: 1

**Method 2 (By making pop operation costly)**   
In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

1. **push(s, x)**operation:
   * Enqueue x to q1 (assuming size of q1 is unlimited).
2. **pop(s)** operation:
   * One by one dequeue everything except the last element from q1 and enqueue to q2.
   * Dequeue the last item of q1, the dequeued item is result, store it.
   * Swap the names of q1 and q2
   * Return the item stored in step 2.

/\* Program to implement a stack

using two queue \*/

#include <bits/stdc++.h>

using namespace std;

class Stack {

queue<int> q1, q2;

int curr\_size;

public:

Stack()

{

curr\_size = 0;

}

void pop()

{

if (q1.empty())

return;

// Leave one element in q1 and

// push others in q2.

while (q1.size() != 1) {

q2.push(q1.front());

q1.pop();

}

// Pop the only left element

// from q1

q1.pop();

curr\_size--;

// swap the names of two queues

queue<int> q = q1;

q1 = q2;

q2 = q;

}

void push(int x)

{

q1.push(x);

curr\_size++;

}

int top()

{

if (q1.empty())

return -1;

while (q1.size() != 1) {

q2.push(q1.front());

q1.pop();

}

// last pushed element

int temp = q1.front();

// to empty the auxiliary queue after

// last operation

q1.pop();

// push last element to q2

q2.push(temp);

// swap the two queues names

queue<int> q = q1;

q1 = q2;

q2 = q;

return temp;

}

int size()

{

return curr\_size;

}

};

// Driver code

int main()

{

Stack s;

s.push(1);

s.push(2);

s.push(3);

s.push(4);

cout << "current size: " << s.size()

<< endl;

cout << s.top() << endl;

s.pop();

cout << s.top() << endl;

s.pop();

cout << s.top() << endl;

cout << "current size: " << s.size()

<< endl;

return 0;

}

**Output :**

current size: 4

4

3

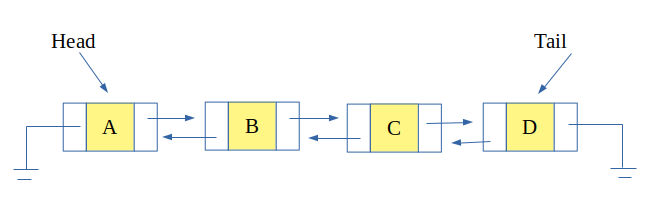
2

current size: 2

# 293. Implement Stack & Queue using Deque

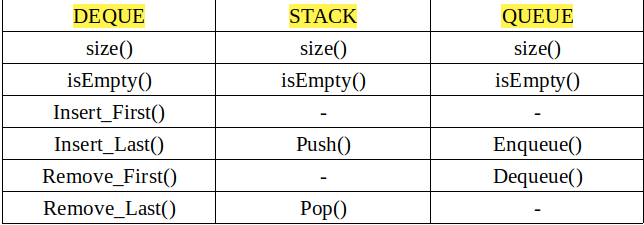
Deque also known as **double eneded queue**, as name suggests is a special kind of queue in which insertions and deletions can be done at the last as well as at the beginning.

A link-list representation of deque is such that each node points to the next node as well as the previous node. So that insertion and deletions take constant time at both the beginning and the last.



Now, deque can be used to implement a stack and queue. One simply needs to understand how deque can made to work as a stack or a queue.

The functions of deque to tweak them to work as stack and queue are list below.



**Examples: Stack**

Input : Stack : 1 2 3

Push(4)

Output : Stack : 1 2 3 4

Input : Stack : 1 2 3

Pop()

Output : Stack : 1 2

**Examples: Queue**

Input: Queue : 1 2 3

Enqueue(4)

Output: Queue : 1 2 3 4

Input: Queue : 1 2 3

Dequeue()

Output: Queue : 2 3

// C++ Program to implement stack and queue using Deque

#include <bits/stdc++.h>

using namespace std;

// structure for a node of deque

struct DQueNode {

int value;

DQueNode\* next;

DQueNode\* prev;

};

// Implementation of deque class

class Deque {

private:

// pointers to head and tail of deque

DQueNode\* head;

DQueNode\* tail;

public:

// constructor

Deque()

{

head = tail = NULL;

}

// if list is empty

bool isEmpty()

{

if (head == NULL)

return true;

return false;

}

// count the number of nodes in list

int size()

{

// if list is not empty

if (!isEmpty()) {

DQueNode\* temp = head;

int len = 0;

while (temp != NULL) {

len++;

temp = temp->next;

}

return len;

}

return 0;

}

// insert at the first position

void insert\_first(int element)

{

// allocating node of DQueNode type

DQueNode\* temp = new DQueNode[sizeof(DQueNode)];

temp->value = element;

// if the element is first element

if (head == NULL) {

head = tail = temp;

temp->next = temp->prev = NULL;

}

else {

head->prev = temp;

temp->next = head;

temp->prev = NULL;

head = temp;

}

}

// insert at last position of deque

void insert\_last(int element)

{

// allocating node of DQueNode type

DQueNode\* temp = new DQueNode[sizeof(DQueNode)];

temp->value = element;

// if element is the first element

if (head == NULL) {

head = tail = temp;

temp->next = temp->prev = NULL;

}

else {

tail->next = temp;

temp->next = NULL;

temp->prev = tail;

tail = temp;

}

}

// remove element at the first position

void remove\_first()

{

// if list is not empty

if (!isEmpty()) {

DQueNode\* temp = head;

head = head->next;

if(head) head->prev = NULL;

delete temp;

if(head == NULL) tail = NULL;

return;

}

cout << "List is Empty" << endl;

}

// remove element at the last position

void remove\_last()

{

// if list is not empty

if (!isEmpty()) {

DQueNode\* temp = tail;

tail = tail->prev;

if(tail) tail->next = NULL;

delete temp;

if(tail == NULL) head = NULL;

return;

}

cout << "List is Empty" << endl;

}

// displays the elements in deque

void display()

{

// if list is not empty

if (!isEmpty()) {

DQueNode\* temp = head;

while (temp != NULL) {

cout << temp->value << " ";

temp = temp->next;

}

cout << endl;

return;

}

cout << "List is Empty" << endl;

}

};

// Class to implement stack using Deque

class Stack : public Deque {

public:

// push to push element at top of stack

// using insert at last function of deque

void push(int element)

{

insert\_last(element);

}

// pop to remove element at top of stack

// using remove at last function of deque

void pop()

{

remove\_last();

}

};

// class to implement queue using deque

class Queue : public Deque {

public:

// enqueue to insert element at last

// using insert at last function of deque

void enqueue(int element)

{

insert\_last(element);

}

// dequeue to remove element from first

// using remove at first function of deque

void dequeue()

{

remove\_first();

}

};

// Driver Code

int main()

{

// object of Stack

Stack stk;

// push 7 and 8 at top of stack

stk.push(7);

stk.push(8);

cout << "Stack: ";

stk.display();

// pop an element

stk.pop();

cout << "Stack: ";

stk.display();

// object of Queue

Queue que;

// insert 12 and 13 in queue

que.enqueue(12);

que.enqueue(13);

cout << "Queue: ";

que.display();

// delete an element from queue

que.dequeue();

cout << "Queue: ";

que.display();

cout << "Size of Stack is " << stk.size() << endl;

cout << "Size of Queue is " << que.size() << endl;

return 0;

}

**Output:**

Stack: 7 8

Stack: 7

Queue: 12 13

Queue: 13

Size of Stack is 1

Size of Queue is 1

# 294. Stack Permutations (Check if an array is stack permutation of other)

A **stack permutation** is a permutation of objects in the given input queue which is done by transferring elements from input queue to the output queue with the help of a stack and the built-in push and pop functions.  
The well defined rules are: 

1. Only dequeue from the input queue.
2. Use inbuilt push, pop functions in the single stack.
3. Stack and input queue must be empty at the end.
4. Only enqueue to the output queue.

There are a huge number of permutations possible using a stack for a single input queue.   
Given two arrays, both of unique elements. One represents the input queue and the other represents the output queue. Our task is to check if the given output is possible through stack permutation.  
**Examples:**

Input : First array: 1, 2, 3

Second array: 2, 1, 3

Output : Yes

Procedure:

push 1 from input to stack

push 2 from input to stack

pop 2 from stack to output

pop 1 from stack to output

push 3 from input to stack

pop 3 from stack to output

Input : First array: 1, 2, 3

Second array: 3, 1, 2

Output : Not Possible

## Solution:

The idea to do this is we will try to convert the input queue to output queue using a stack, if we are able to do so then the queue is permutable otherwise not.   
**Below is the step by step algorithm to do this**: 

1. Continuously pop elements from the input queue and check if it is equal to the top of output queue or not, if it is not equal to the top of output queue then we will push the element to stack.
2. Once we find an element in input queue such the top of input queue is equal to top of output queue, we will pop a single element from both input and output queues, and compare the top of stack and top of output queue now. If top of both stack and output queue are equal then pop element from both stack and output queue. If not equal, go to step 1.
3. Repeat above two steps until the input queue becomes empty. At the end if both of the input queue and stack are empty then the input queue is permutable otherwise not.

Below is implementation of above idea:

// Given two arrays, check if one array is

// stack permutation of other.

#include<bits/stdc++.h>

using namespace std;

// function to check if input queue is

// permutable to output queue

bool checkStackPermutation(int ip[], int op[], int n)

{

// Input queue

queue<int> input;

for (int i=0;i<n;i++)

input.push(ip[i]);

// output queue

queue<int> output;

for (int i=0;i<n;i++)

output.push(op[i]);

// stack to be used for permutation

stack <int> tempStack;

while (!input.empty())

{

int ele = input.front();

input.pop();

if (ele == output.front())

{

output.pop();

while (!tempStack.empty())

{

if (tempStack.top() == output.front())

{

tempStack.pop();

output.pop();

}

else

break;

}

}

else

tempStack.push(ele);

}

// If after processing, both input queue and

// stack are empty then the input queue is

// permutable otherwise not.

return (input.empty()&&tempStack.empty());

}

// Driver program to test above function

int main()

{

// Input Queue

int input[] = {1, 2, 3};

// Output Queue

int output[] = {2, 1, 3};

int n = 3;

if (checkStackPermutation(input, output, n))

cout << "Yes";

else

cout << "Not Possible";

return 0;

}

**Output:** 

Yes

**Same Leetcode question:**

Given two integer arrays pushed and popped each with distinct values, return true*if this could have been the result of a sequence of push and pop operations on an initially empty stack, or*false*otherwise.*

**Example 1:**

**Input:** pushed = [1,2,3,4,5], popped = [4,5,3,2,1]

**Output:** true

**Explanation:** We might do the following sequence:

push(1), push(2), push(3), push(4),

pop() -> 4,

push(5),

pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

**Example 2:**

**Input:** pushed = [1,2,3,4,5], popped = [4,3,5,1,2]

**Output:** false

**Explanation:** 1 cannot be popped before 2.

**Constraints:**

* 1 <= pushed.length <= 1000
* 0 <= pushed[i] <= 1000
* All the elements of pushed are **unique**.
* popped.length == pushed.length
* popped is a permutation of pushed.

**Solution:**

class Solution {

public:

bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {

int n = pushed.size(), i=0, j=0;

stack<int> st;

while(i<n){

st.push(pushed[i++]);

while(!st.empty() && st.top()==popped[j]){

st.pop();

j++;

}

}

return st.empty();

}

};

# 295. Implement Queue using Stack

Implement a Queue using 2 stacks**s1** and**s2** .  
A Query **Q** is of 2 Types  
**(i)** 1 x (a query of this type means  pushing **'x'** into the queue)  
**(ii)** 2   (a query of this type means to pop element from queue and print the poped element)

**Example 1:**

**Input:**

5

1 2 1 3 2 1 4 2

**Output:**

2 3

**Explanation:**

In the first testcase

1 2 the queue will be {2}

1 3 the queue will be {2 3}

2   poped element will be 2 the queue

  will be {3}

1 4 the queue will be {3 4}

2   poped element will be 3.

**Example 2:**

**Input:**

4

1 2 2 2 1 4

**Output:**

2 -1

**Explanation:**

In the second testcase

1 2 the queue will be {2}

2   poped element will be 2 and

  then the queue will be empty

2   the queue is empty and hence -1

1 4 the queue will be {4}.

**Your Task:**  
You are required to complete the two methods **push** which take one argument an integer **'x'** to be pushed into the queue and **pop** which returns a integer poped out from other queue(-1 if the queue is empty). The **printing** is done **automatically**by the**driver code**.

**Expected Time Complexity** : O(1) for **push()**and O(N) for **pop()**or O(N) for **push()**and O(1) for **pop()**   
**Expected Auxilliary Space**: O(1).

**Constraints:**  
1 <=Q <= 100  
1 <= x <= 100

Note:The **Input/Ouput** format and **Example** given are used for system's internal purpose, and should be used by a user for **Expected Output** only. As it is a function problem, hence a user should not read any input from stdin/console. The task is to complete the function specified, and not to write the full code.

## Solution:

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

**Method 1 (By making enQueue operation costly)** This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

*enQueue(q, x):*

* *While stack1 is not empty, push everything from stack1 to stack2.*
* *Push x to stack1 (assuming size of stacks is unlimited).*
* *Push everything back to stack1.*

*Here time complexity will be O(n)*

*deQueue(q):*

* *If stack1 is empty then error*
* *Pop an item from stack1 and return it*

*Here time complexity will be O(1)*

Below is the implementation of the above approach:

// CPP program to implement Queue using

// two stacks with costly enQueue()

#include <bits/stdc++.h>

using namespace std;

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;

}

};

// Driver code

int main()

{

Queue q;

q.enQueue(1);

q.enQueue(2);

q.enQueue(3);

cout << q.deQueue() << '\n';

cout << q.deQueue() << '\n';

cout << q.deQueue() << '\n';

return 0;

}

**Output:**

1

2

3

**Complexity Analysis:**

* **Time Complexity:**
  + **Push operation:** O(N).   
    In the worst case we have empty whole of stack 1 into stack 2.
  + **Pop operation:** O(1).   
    Same as pop operation in stack.
* **Auxiliary Space:** O(N).   
  Use of stack for storing values.

**Method 2 (By making deQueue operation costly)**In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

Here time complexity will be O(1)

deQueue(q)

1) If both stacks are empty then error.

2) If stack2 is empty

While stack1 is not empty, push everything from stack1 to stack2.

3) Pop the element from stack2 and return it.

Here time complexity will be O(n)

Method 2 is definitely better than method 1.   
Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty. So, the amortized complexity of the dequeue operation becomes .   
Implementation of method 2:

// CPP program to implement Queue using

// two stacks with costly deQueue()

#include <bits/stdc++.h>

using namespace std;

struct Queue {

stack<int> s1, s2;

// Enqueue an item to the queue

void enQueue(int x)

{

// Push item into the first stack

s1.push(x);

}

// Dequeue an item from the queue

int deQueue()

{

// if both stacks are empty

if (s1.empty() && s2.empty()) {

cout << "Q is empty";

exit(0);

}

// if s2 is empty, move

// elements from s1

if (s2.empty()) {

while (!s1.empty()) {

s2.push(s1.top());

s1.pop();

}

}

// return the top item from s2

int x = s2.top();

s2.pop();

return x;

}

};

// Driver code

int main()

{

Queue q;

q.enQueue(1);

q.enQueue(2);

q.enQueue(3);

cout << q.deQueue() << '\n';

cout << q.deQueue() << '\n';

cout << q.deQueue() << '\n';

return 0;

}

**Output:**

1 2 3

**Complexity Analysis:**

* **Time Complexity:**
  + **Push operation:** O(1).   
    Same as pop operation in stack.
  + **Pop operation:** O(N).   
    In the worst case we have empty whole of stack 1 into stack 2
* **Auxiliary Space:** O(N).   
  Use of stack for storing values.

**Queue can also be implemented using one user stack and one Function Call Stack.**Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

*enQueue(x)*

1) Push *x* to *stack1*.

*deQueue:*

1) If *stack1* is empty then error.

2) If *stack1* has only one element then return it.

3) Recursively pop everything from the stack1, store the popped item

in a variable *res*, push the *res* back to stack1 and return *res*

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1*in deQueue() and all other items are pushed back in step

**3. Implementation of method 2 using Function Call Stack:**

// CPP program to implement Queue using

// one stack and recursive call stack.

#include <bits/stdc++.h>

using namespace std;

struct Queue {

stack<int> s;

// Enqueue an item to the queue

void enQueue(int x)

{

s.push(x);

}

// Dequeue an item from the queue

int deQueue()

{

if (s.empty()) {

cout << "Q is empty";

exit(0);

}

// pop an item from the stack

int x = s.top();

s.pop();

// if stack becomes empty, return

// the popped item

if (s.empty())

return x;

// recursive call

int item = deQueue();

// push popped item back to the stack

s.push(x);

// return the result of deQueue() call

return item;

}

};

// Driver code

int main()

{

Queue q;

q.enQueue(1);

q.enQueue(2);

q.enQueue(3);

cout << q.deQueue() << '\n';

cout << q.deQueue() << '\n';

cout << q.deQueue() << '\n';

return 0;

}

**Output:**

1 2 3

**Complexity Analysis:**

* **Time Complexity:**
  + **Push operation :**O(1).   
    Same as pop operation in stack.
  + **Pop operation :**O(N).   
    The difference from above method is that in this method element is returned and all elements are restored back in a single call.
* **Auxiliary Space:** O(N).   
  Use of stack for storing values.

# 296. Implement "n" queue in an array

*Create a data structure kQueues that represents k queues. Implementation of kQueues should use only one array, i.e., k queues should use the same array for storing elements. Following functions must be supported by kQueues.*  
*enqueue(int x, int qn) –> adds x to queue number ‘qn’ where qn is from 0 to k-1*  
*dequeue(int qn) –> deletes an element from queue number ‘qn’ where qn is from 0 to k-1*

## Solution:

**Method 1 (Divide the array in slots of size n/k)**   
A simple way to implement k queues is to divide the array in k slots of size n/k each, and fix the slots for different queues, i.e., use arr[0] to arr[n/k-1] for the first queue, and arr[n/k] to arr[2n/k-1] for queue2 where arr[] is the array to be used to implement two queues and size of array be n.  
The problem with this method is an inefficient use of array space. An enqueue operation may result in overflow even if there is space available in arr[]. For example, consider k as 2 and array size n as 6. Let we enqueue 3 elements to first and do not enqueue anything to the second queue. When we enqueue the 4th element to the first queue, there will be overflow even if we have space for 3 more elements in the array.  
**Method 2 (A space efficient implementation)**   
The idea is similar to the [stack post](https://www.geeksforgeeks.org/efficiently-implement-k-stacks-single-array/), here we need to use three extra arrays. In stack post, we needed two extra arrays, one more array is required because in queues, enqueue() and dequeue() operations are done at different ends.  
Following are the three extra arrays are used:   
1) **front[]**: This is of size k and stores indexes of front elements in all queues.   
2) **rear[]**: This is of size k and stores indexes of rear elements in all queues.   
2) **next[]**: This is of size n and stores indexes of next item for all items in array arr[].   
Here arr[] is the actual array that stores k stacks.  
Together with k queues, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable ‘free’.  
All entries in front[] are initialized as -1 to indicate that all queues are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to the next slot. Top of the free stack, ‘free’ is initialized as 0.  
Following is C++ implementation of the above idea.

// A C++ program to demonstrate implementation of k queues in a single

// array in time and space efficient way

#include<iostream>

#include<climits>

using namespace std;

// A C++ class to represent k queues in a single array of size n

class kQueues

{

// Array of size n to store actual content to be stored in queue

int \*arr;

// Array of size k to store indexes of front elements of the queue

int \*front;

// Array of size k to store indexes of rear elements of queue

int \*rear;

// Array of size n to store next entry in all queues

int \*next;

int n, k;

int free; // To store the beginning index of the free list

public:

//constructor to create k queue in an array of size n

kQueues(int k, int n);

// A utility function to check if there is space available

bool isFull() { return (free == -1); }

// To enqueue an item in queue number 'qn' where qn is from 0 to k-1

void enqueue(int item, int qn);

// To dequeue an from queue number 'qn' where qn is from 0 to k-1

int dequeue(int qn);

// To check whether queue number 'qn' is empty or not

bool isEmpty(int qn) { return (front[qn] == -1); }

};

// Constructor to create k queues in an array of size n

kQueues::kQueues(int k1, int n1)

{

// Initialize n and k, and allocate memory for all arrays

k = k1, n = n1;

arr = new int[n];

front = new int[k];

rear = new int[k];

next = new int[n];

// Initialize all queues as empty

for (int i = 0; i < k; i++)

front[i] = -1;

// Initialize all spaces as free

free = 0;

for (int i=0; i<n-1; i++)

next[i] = i+1;

next[n-1] = -1; // -1 is used to indicate end of free list

}

// To enqueue an item in queue number 'qn' where qn is from 0 to k-1

void kQueues::enqueue(int item, int qn)

{

// Overflow check

if (isFull())

{

cout << "\nQueue Overflow\n";

return;

}

int i = free; // Store index of first free slot

// Update index of free slot to index of next slot in free list

free = next[i];

if (isEmpty(qn))

front[qn] = i;

else

next[rear[qn]] = i;

next[i] = -1;

// Update next of rear and then rear for queue number 'qn'

rear[qn] = i;

// Put the item in array

arr[i] = item;

}

// To dequeue an from queue number 'qn' where qn is from 0 to k-1

int kQueues::dequeue(int qn)

{

// Underflow checkSAS

if (isEmpty(qn))

{

cout << "\nQueue Underflow\n";

return INT\_MAX;

}

// Find index of front item in queue number 'qn'

int i = front[qn];

front[qn] = next[i]; // Change top to store next of previous top

// Attach the previous front to the beginning of free list

next[i] = free;

free = i;

// Return the previous front item

return arr[i];

}

/\* Driver program to test kStacks class \*/

int main()

{

// Let us create 3 queue in an array of size 10

int k = 3, n = 10;

kQueues ks(k, n);

// Let us put some items in queue number 2

ks.enqueue(15, 2);

ks.enqueue(45, 2);

// Let us put some items in queue number 1

ks.enqueue(17, 1);

ks.enqueue(49, 1);

ks.enqueue(39, 1);

// Let us put some items in queue number 0

ks.enqueue(11, 0);

ks.enqueue(9, 0);

ks.enqueue(7, 0);

cout << "Dequeued element from queue 2 is " << ks.dequeue(2) << endl;

cout << "Dequeued element from queue 1 is " << ks.dequeue(1) << endl;

cout << "Dequeued element from queue 0 is " << ks.dequeue(0) << endl;

return 0;

}

Output:

Dequeued element from queue 2 is 15

Dequeued element from queue 1 is 17

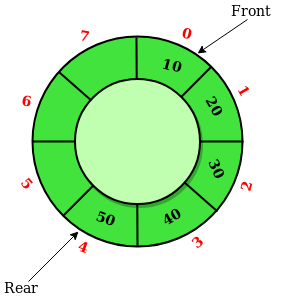
Dequeued element from queue 0 is 11

Time complexities of enqueue() and dequeue() is O(1).  
The best part of the above implementation is, if there is a slot available in the queue, then an item can be enqueued in any of the queues, i.e., no wastage of space. This method requires some extra space. Space may not be an issue because queue items are typically large, for example, queues of employees, students, etc where every item is of hundreds of bytes. For such large queues, the extra space used is comparatively very less as we use three integer arrays as extra space.

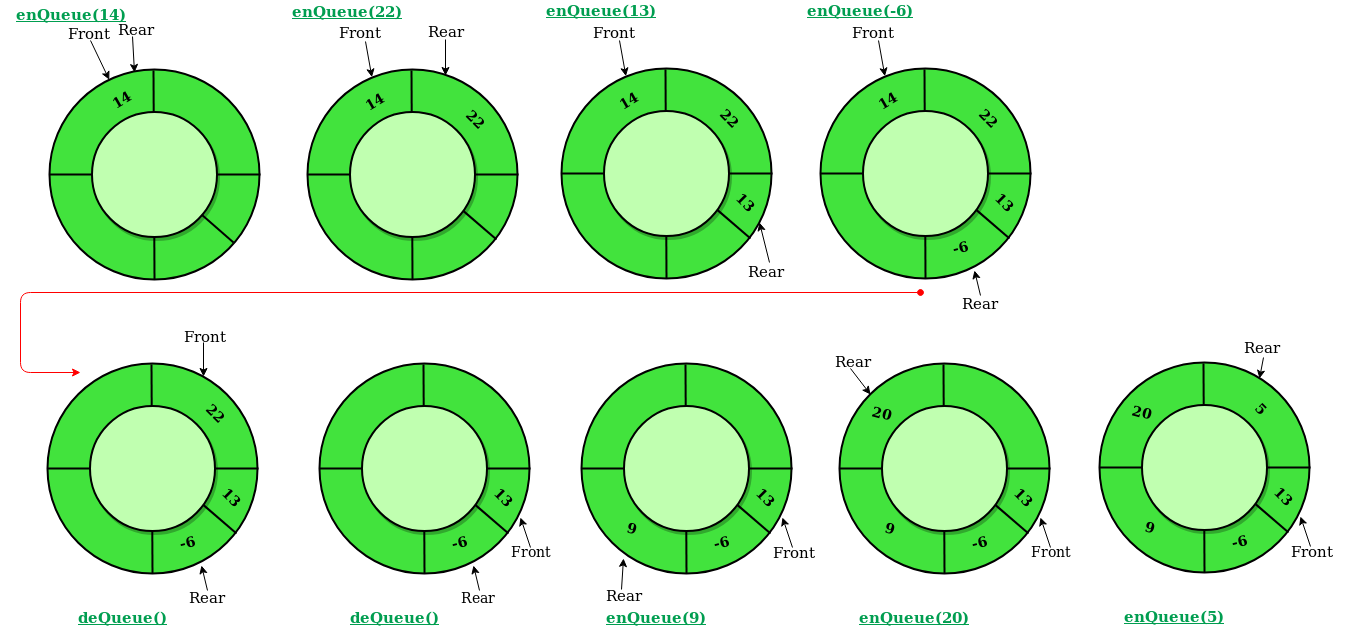
# 297. Implement a Circular queue

# Array Implementation

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **‘Ring Buffer’**. 



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Operations on Circular Queue: 

* **Front:** Get the front item from queue.
* **Rear:** Get the last item from queue.
* **enQueue(value)**This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
  1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
* **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
  1. Check whether queue is Empty means check (front==-1).
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

// C or C++ program for insertion and

// deletion in Circular Queue

#include<bits/stdc++.h>

using namespace std;

class Queue

{

// Initialize front and rear

int rear, front;

// Circular Queue

int size;

int \*arr;

Queue(int s)

{

front = rear = -1;

size = s;

arr = new int[s];

}

void enQueue(int value);

int deQueue();

void displayQueue();

};

/\* Function to create Circular queue \*/

void Queue::enQueue(int value)

{

if ((front == 0 && rear == size-1) ||

(rear == (front-1)%(size-1)))

{

printf("\nQueue is Full");

return;

}

else if (front == -1) /\* Insert First Element \*/

{

front = rear = 0;

arr[rear] = value;

}

else if (rear == size-1 && front != 0)

{

rear = 0;

arr[rear] = value;

}

else

{

rear++;

arr[rear] = value;

}

}

// Function to delete element from Circular Queue

int Queue::deQueue()

{

if (front == -1)

{

printf("\nQueue is Empty");

return INT\_MIN;

}

int data = arr[front];

arr[front] = -1;

if (front == rear)

{

front = -1;

rear = -1;

}

else if (front == size-1)

front = 0;

else

front++;

return data;

}

// Function displaying the elements

// of Circular Queue

void Queue::displayQueue()

{

if (front == -1)

{

printf("\nQueue is Empty");

return;

}

printf("\nElements in Circular Queue are: ");

if (rear >= front)

{

for (int i = front; i <= rear; i++)

printf("%d ",arr[i]);

}

else

{

for (int i = front; i < size; i++)

printf("%d ", arr[i]);

for (int i = 0; i <= rear; i++)

printf("%d ", arr[i]);

}

}

/\* Driver of the program \*/

int main()

{

Queue q(5);

// Inserting elements in Circular Queue

q.enQueue(14);

q.enQueue(22);

q.enQueue(13);

q.enQueue(-6);

// Display elements present in Circular Queue

q.displayQueue();

// Deleting elements from Circular Queue

printf("\nDeleted value = %d", q.deQueue());

printf("\nDeleted value = %d", q.deQueue());

q.displayQueue();

q.enQueue(9);

q.enQueue(20);

q.enQueue(5);

q.displayQueue();

q.enQueue(20);

return 0;

}

**Output:** 

Elements in Circular Queue are: 14 22 13 -6

Deleted value = 14

Deleted value = 22

Elements in Circular Queue are: 13 -6

Elements in Circular Queue are: 13 -6 9 20 5

Queue is Full

**Time Complexity:** Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.  
**Applications:** 

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

# Circular Linked List Implementation

In this post another method of circular queue implementation is discussed, using Circular Singly Linked List.

Operations on Circular Queue:

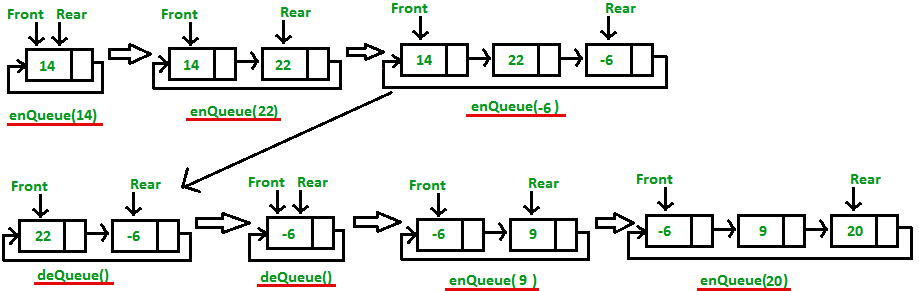
* **Front:**Get the front item from queue.
* **Rear:** Get the last item from queue.
* **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

**Steps:**

* 1. Create a new node dynamically and insert value into it.
  2. Check if front==NULL, if it is true then front = rear = (newly created node)
  3. If it is false then rear=(newly created node) and rear node always contains the address of the front node.
* **deQueue()** This function is used to delete an element from the circular queue. In a queue, the element is always deleted from front position.

**Steps:**

* 1. Check whether queue is empty or not means front == NULL.
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front = rear = NULL else move the front forward in queue, update address of front in rear node and return the element.

[](https://media.geeksforgeeks.org/wp-content/uploads/Operations-on-Circular-Queue.png)

Below is the implementation of above approach:

// C++ program for insertion and

// deletion in Circular Queue

#include <bits/stdc++.h>

using namespace std;

// Structure of a Node

struct Node {

int data;

struct Node\* link;

};

struct Queue {

struct Node \*front, \*rear;

};

// Function to create Circular queue

void enQueue(Queue\* q, int value)

{

struct Node\* temp = new Node;

temp->data = value;

if (q->front == NULL)

q->front = temp;

else

q->rear->link = temp;

q->rear = temp;

q->rear->link = q->front;

}

// Function to delete element from Circular Queue

int deQueue(Queue\* q)

{

if (q->front == NULL) {

printf("Queue is empty");

return INT\_MIN;

}

// If this is the last node to be deleted

int value; // Value to be dequeued

if (q->front == q->rear) {

value = q->front->data;

free(q->front);

q->front = NULL;

q->rear = NULL;

}

else // There are more than one nodes

{

struct Node\* temp = q->front;

value = temp->data;

q->front = q->front->link;

q->rear->link = q->front;

free(temp);

}

return value;

}

// Function displaying the elements of Circular Queue

void displayQueue(struct Queue\* q)

{

struct Node\* temp = q->front;

printf("\nElements in Circular Queue are: ");

while (temp->link != q->front) {

printf("%d ", temp->data);

temp = temp->link;

}

printf("%d", temp->data);

}

/\* Driver of the program \*/

int main()

{

// Create a queue and initialize front and rear

Queue\* q = new Queue;

q->front = q->rear = NULL;

// Inserting elements in Circular Queue

enQueue(q, 14);

enQueue(q, 22);

enQueue(q, 6);

// Display elements present in Circular Queue

displayQueue(q);

// Deleting elements from Circular Queue

printf("\nDeleted value = %d", deQueue(q));

printf("\nDeleted value = %d", deQueue(q));

// Remaining elements in Circular Queue

displayQueue(q);

enQueue(q, 9);

enQueue(q, 20);

displayQueue(q);

return 0;

}

**Output:**

Elements in Circular Queue are: 14 22 6

Deleted value = 14

Deleted value = 22

Elements in Circular Queue are: 6

Elements in Circular Queue are: 6 9 20

**Time Complexity:**Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

**Note:**In case of linked list implementation, a queue can be easily implemented without being circular. However, in the case of array implementation, we need a circular queue to save space.

# 298. LRU Cache Implementation

Design a data structure that works like a LRU Cache. Here **cap** denotes the capacity of the cache and Q denotes the number of queries. Query can be of two types:

1. **SET** **x** **y** : sets the value of the key **x** with value **y**
2. **GET** **x** : gets the key of **x** if present else returns **-1**.

The LRUCache class has two methods **get**() and **set**() which are defined as follows.

1. **get(key)**   : returns the value of the key if it already exists in the cache otherwise returns **-1.**
2. **set(key, value)** : if the key is already present, update its value. If not present, add the key-value pair to the cache. If the cache reaches its capacity it should invalidate the least recently used item before inserting the new item.
3. In the **constructor** of the class the capacity of the cache should be intitialized.

**Example 1:**

**Input:**

cap = 2

Q = 2

Queries = SET 1 2 GET 1

**Output:** 2

**Explanation:**

Cache Size = 2

SET 1 2 GET 1

SET 1 2 : 1 -> 2

GET 1 : Print the value corresponding

to Key 1, ie 2.

**Example 2:**

**Input:**

cap = 2

Q = 8

Queries = SET 1 2 SET 2 3 SET 1 5

SET 4 5 SET 6 7 GET 4 SET 1 2 GET 3

**Output:** 5 -1

**Explanation:**

Cache Size = 2

SET 1 2 : 1 -> 2

SET 2 3 : 1 -> 2, 2 -> 3 (the most recently

used one is kept at the rightmost position)

SET 1 5 : 2 -> 3, 1 -> 5

SET 4 5 : 1 -> 5, 4 -> 5 (Cache size is 2, hence

we delete the least recently used key-value pair)

SET 6 7 : 4 -> 5, 6 -> 7

GET 4 : Prints 5 (The cache now looks like

6 -> 7, 4->5)

SET 1 2 : 4 -> 5, 1 -> 2

(Cache size is 2, hence we delete the least

recently used key-value pair)

GET 3 : No key value pair having

key = 3. Hence, -1 is printed.

**Your Task:**  
You don't need to read input or print anything . Complete the constructor and get(), set() methods of the class LRUcache.

**Expected Time Complexity:**O(1) for both **get()**and **set().**  
**Expected Auxiliary Space:**O(1) for both **get()**and **set().**  
(Although, you may use extra space for cache storage and implementation purposes).

**Constraints:**  
1 <= cap <= 1000  
1 <= Q <= 100000  
1 <= x, y <= 1000

## Solution:

**1. Brute-force Approach:**  
We will keep an array of Nodes and each node will contain the following information:

struct Node

{

int key;

int value;

// It shows the time at which the key is stored.

// We will use the timeStamp to find out the

// least recently used (LRU) node.

int timeStamp;

Node(int \_key, int \_value)

{

key = \_key;

value = \_value;

// currentTimeStamp from system

timeStamp = currentTimeStamp;

}

};

The size of the array will be equal to the given capacity of cache.

**(a)** For **get(int key):** We can simply iterate over the array and compare the key of each node with the given key and return the value stored in the node for that key. If we don’t find any such node, return simply -1.  
Time Complexity: **O(n)**

**(b)** For **set(int key, int value):** If the array if full, we have to delete one node from the array. To find the LRU node, we will iterate through the array and find the node with least timeStamp value. We will simply insert the new node (with new key and value) at the place of the LRU node.  
If the array is not full, we can simply insert a new node in the array at the last current index of the array.  
Time Complexity: **O(n)**

**2. Optimized Approach:**  
The key to solve this problem is using a double linked list which enables us to quickly move nodes.   
The LRU cache is a hash map of keys and double linked nodes. The hash map makes the time of get() to be O(1). The list of double linked nodes make the nodes adding/removal operations O(1).

**Code using Doubly Linked List and HashMap:**

#include <bits/stdc++.h>

using namespace std;

class LRUCache{

public:

class node

{

public:

int key;

int value;

node \* prev;

node \* next;

node(int \_key,int \_value)

{

key = \_key;

value = \_value;

}

};

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

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

int cap;

map<int, node \*> m;

// Constructor for initializing the

// cache capacity with the given value.

LRUCache(int capacity)

{

cap = capacity;

head->next = tail;

tail->prev = head;

}

void addnode(node \* temp)

{

node \* dummy = head->next;

head->next = temp;

temp->prev = head;

temp->next = dummy;

dummy->prev = temp;

}

void deletenode(node \* temp)

{

node \* delnext = temp->next;

node \* delprev = temp->prev;

delnext->prev = delprev;

delprev->next = delnext;

}

// This method works in O(1)

int get(int key)

{

if (m.find(key) != m.end())

{

node \* res = m[key];

m.erase(key);

int ans = res->value;

deletenode(res);

addnode(res);

m[key] = head->next;

cout << "Got the value : " << ans

<< " for the key: " << key << "\n";

return ans;

}

cout << "Did not get any value for the key: "

<< key << "\n";

return -1;

}

// This method works in O(1)

void set(int key, int value)

{

cout << "Going to set the (key, value) : ("

<< key << ", " << value << ")" << "\n";

if (m.find(key) != m.end())

{

node \* exist = m[key];

m.erase(key);

deletenode(exist);

}

if (m.size() == cap)

{

m.erase(tail->prev->key);

deletenode(tail->prev);

}

addnode(new node(key, value));

m[key] = head->next;

}

};

// Driver code

int main()

{

cout << "Going to test the LRU "

<< "Cache Implementation\n";

LRUCache \* cache = new LRUCache(2);

// It will store a key (1) with value

// 10 in the cache.

cache->set(1, 10);

// It will store a key (1) with value 10 in the

// cache.

cache->set(2, 20);

cout << "Value for the key: 1 is "

<< cache->get(1) << "\n"; // returns 10

// Evicts key 2 and store a key (3) with

// value 30 in the cache.

cache->set(3, 30);

cout << "Value for the key: 2 is "

<< cache->get(2) << "\n"; // returns -1 (not found)

// Evicts key 1 and store a key (4) with

// value 40 in the cache.

cache->set(4, 40);

cout << "Value for the key: 1 is "

<< cache->get(1) << "\n"; // returns -1 (not found)

cout << "Value for the key: 3 is "

<< cache->get(3) << "\n"; // returns 30

cout << "Value for the key: 4 is "

<< cache->get(4) << "\n"; // return 40

return 0;

}

**Output:**

Going to test the LRU Cache Implementation

Going to set the (key, value) : (1, 10)

Going to set the (key, value) : (2, 20)

Got the value : 10 for the key: 1

Value for the key: 1 is 10

Going to set the (key, value) : (3, 30)

Did not get any value for the key: 2

Value for the key: 2 is -1

Going to set the (key, value) : (4, 40)

Did not get any value for the key: 1

Value for the key: 1 is -1

Got the value : 30 for the key: 3

Value for the key: 3 is 30

Got the value : 40 for the key: 4

Value for the key: 4 is 40

# 299. [Reverse a Queue using recursion](https://practice.geeksforgeeks.org/problems/queue-reversal/1)

Given a Queue **Q** containing **N** elements. The task is to reverse the Queue. Your task is to complete the function **rev(),**that reverses the **N** elements of the queue.

**Example 1:**

**Input:**

6

4 3 1 10 2 6

**Output:**

6 2 10 1 3 4

**Explanation:**

After reversing the given

elements of the queue , the resultant

queue will be 6 2 10 1 3 4.

**Example 2:**

**Input:**

4

4 3 2 1

**Output:**

1 2 3 4

**Explanation:**

After reversing the given

elements of the queue , the resultant

queue will be 1 2 3 4.

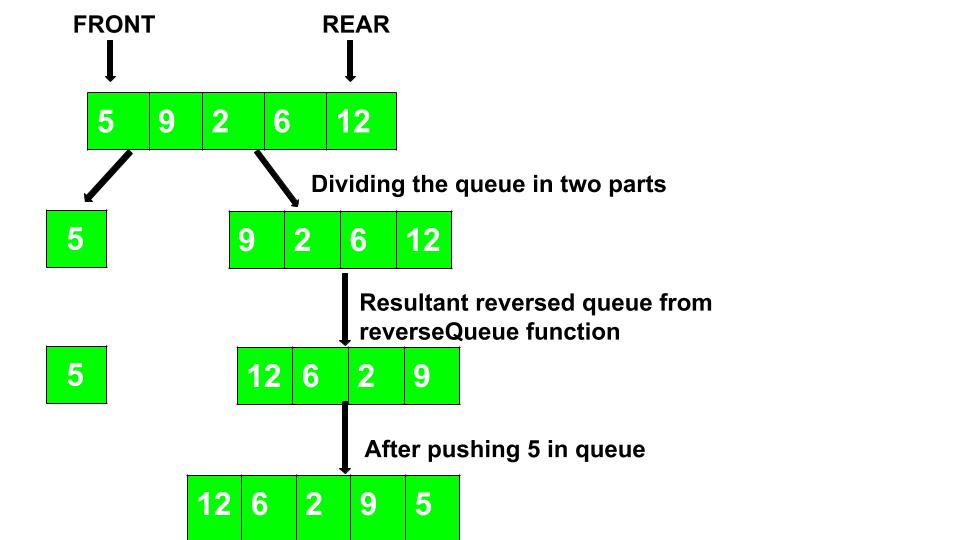
**Your Task:**  
 You only need to **complete**the **function rev**that takes a **queue**as **parameter**and **returns**the **reversed queue**. The **printing**is done **automatically**by the**driver code**.

**Expected Time Complexity** : O(n)  
**Expected Auxilliary Space** : O(n)

**Constraints:**  
1 ≤ N ≤ 105  
1 ≤ elements of Queue ≤ 105

## Solution:

1. The pop element from the queue if the queue has elements otherwise return empty queue.
2. Call reverseQueue function for the remaining queue.
3. Push the popped element in the resultant reversed queue.



**Pseudo Code :** 

queue reverseFunction(queue)

{

if (queue is empty)

return queue;

else {

data = queue.front()

queue.pop()

queue = reverseFunction(queue);

q.push(data);

return queue;

}

}

// C++ code for reversing a queue

#include <bits/stdc++.h>

using namespace std;

// Utility function to print the queue

void printQueue(queue<long long int> Queue)

{

while (!Queue.empty()) {

cout << Queue.front() << " ";

Queue.pop();

}

}

// Recursive function to reverse the queue

void reverseQueue(queue<long long int>& q)

{

// Base case

if (q.empty())

return;

// Dequeue current item (from front)

long long int data = q.front();

q.pop();

// Reverse remaining queue

reverseQueue(q);

// Enqueue current item (to rear)

q.push(data);

}

// Driver code

int main()

{

queue<long long int> Queue;

Queue.push(56);

Queue.push(27);

Queue.push(30);

Queue.push(45);

Queue.push(85);

Queue.push(92);

Queue.push(58);

Queue.push(80);

Queue.push(90);

Queue.push(100);

reverseQueue(Queue);

printQueue(Queue);

}

**Output:**

100 90 80 58 92 85 45 30 27 56

**Time Complexity :** O(n).

**Without using recursion:**

**Approach:** For reversing the queue one approach could be to store the elements of the queue in a temporary data structure in a manner such that if we re-insert the elements in the queue they would get inserted in reverse order. So now our task is to choose such data-structure which can serve the purpose. According to the approach, the data-structure should have the property of ‘LIFO’ as the last element to be inserted in the data structure should actually be the first element of the reversed queue. The stack could help in approaching this problem. This will be a two-step process: 

1. Pop the elements from the queue and insert into the stack. (Topmost element of the stack is the last element of the queue)
2. Pop the elements of the stack to insert back into the queue. (The last element is the first one to be inserted into the queue)

// CPP program to reverse a Queue

#include <bits/stdc++.h>

using namespace std;

// Utility function to print the queue

void Print(queue<int>& Queue)

{

while (!Queue.empty()) {

cout << Queue.front() << " ";

Queue.pop();

}

}

// Function to reverse the queue

void reverseQueue(queue<int>& Queue)

{

stack<int> Stack;

while (!Queue.empty()) {

Stack.push(Queue.front());

Queue.pop();

}

while (!Stack.empty()) {

Queue.push(Stack.top());

Stack.pop();

}

}

// Driver code

int main()

{

queue<int> Queue;

Queue.push(10);

Queue.push(20);

Queue.push(30);

Queue.push(40);

Queue.push(50);

Queue.push(60);

Queue.push(70);

Queue.push(80);

Queue.push(90);

Queue.push(100);

reverseQueue(Queue);

Print(Queue);

}

**Output:** 

100, 90, 80, 70, 60, 50, 40, 30, 20, 10

**Complexity Analysis:** 

* **Time Complexity:** O(n).   
  As we need to insert all the elements in the stack and later to the queue.
* **Auxiliary Space:** O(N).   
  Use of stack to store values.

# 300. Reverse the first “K” elements of a queue

Given an integer **K**and a [queue](http://www.geeksforgeeks.org/queue-data-structure/) of integers, we need to reverse the order of the first**K** elements of the queue, leaving the other elements in the same relative order.

Only following standard operations are allowed on queue.

* enqueue(x) : Add an item x to rear of queue
* dequeue() : Remove an item from front of queue
* size() : Returns number of elements in queue.
* front() : Finds front item.

**Example 1:**

**Input:**

5 3

1 2 3 4 5

**Output:**

3 2 1 4 5

**Explanation:**

After reversing the given

input from the 3rd position the resultant

output will be 3 2 1 4 5.

**Example 2:**

**Input:**

4 4

4 3 2 1

**Output:**

1 2 3 4

**Explanation:**

After reversing the given

input from the 4th position the resultant

output will be 1 2 3 4.

**Your Task:**  
**Complete**the **provided function** **modifyQueue**that takes **queue and k** as **parameters**and **returns**a **modified**queue. The **printing**is done **automatically**by the **driver code**.

**Expected TIme Complexity** : O(n)  
**Expected Auxilliary Space**: O(n)

**Constraints:**  
1 <= N <= 1000  
1 <= K <= N

**Note:**The **Input/Ouput** format and **Example** given are used for system's internal purpose, and should be used by a user for **Expected Output** only. As it is a function problem, hence a user should not read any input from stdin/console. The task is to complete the function specified, and not to write the full code.

## Solution:

The idea is to use an auxiliary [stack](https://www.geeksforgeeks.org/stack-data-structure/).

1. Create an empty stack.
2. One by one dequeue first K items from given queue and push the dequeued items to stack.
3. Enqueue the contents of stack at the back of the queue
4. Dequeue (size-k) elements from the front and enque them one by one to the same queue.

// C++ program to reverse first

// k elements of a queue.

#include <bits/stdc++.h>

using namespace std;

/\* Function to reverse the first

K elements of the Queue \*/

void reverseQueueFirstKElements(

int k, queue<int>& Queue)

{

if (Queue.empty() == true

|| k > Queue.size())

return;

if (k <= 0)

return;

stack<int> Stack;

/\* Push the first K elements

into a Stack\*/

for (int i = 0; i < k; i++) {

Stack.push(Queue.front());

Queue.pop();

}

/\* Enqueue the contents of stack

at the back of the queue\*/

while (!Stack.empty()) {

Queue.push(Stack.top());

Stack.pop();

}

/\* Remove the remaining elements and

enqueue them at the end of the Queue\*/

for (int i = 0; i < Queue.size() - k; i++) {

Queue.push(Queue.front());

Queue.pop();

}

}

/\* Utility Function to print the Queue \*/

void Print(queue<int>& Queue)

{

while (!Queue.empty()) {

cout << Queue.front() << " ";

Queue.pop();

}

}

// Driver code

int main()

{

queue<int> Queue;

Queue.push(10);

Queue.push(20);

Queue.push(30);

Queue.push(40);

Queue.push(50);

Queue.push(60);

Queue.push(70);

Queue.push(80);

Queue.push(90);

Queue.push(100);

int k = 5;

reverseQueueFirstKElements(k, Queue);

Print(Queue);

}

**Output:**

50 40 30 20 10 60 70 80 90 100

**Complexity Analysis:**

* **Time Complexity:** O(n+k).   
  Where ‘n’ is the total number of elements in the queue and ‘k’ is the number of elements to be reversed. This is because firstly the whole queue is emptied into the stack and after that first ‘k’ elements are emptied and enqueued in the same way.
* **Auxiliary Space :**Use of stack to store values for the purpose of reversing-: O(n)

# 301. Interleave the first half of the queue with second half

Given a queue of integers of even length, rearrange the elements by interleaving the first half of the queue with the second half of the queue.

Only a stack can be used as an auxiliary space.

**Examples:**

Input : 1 2 3 4

Output : 1 3 2 4

Input : 11 12 13 14 15 16 17 18 19 20

Output : 11 16 12 17 13 18 14 19 15 20

## Solution:

Following are the steps to solve the problem:  
1.Push the first half elements of queue to stack.  
2.Enqueue back the stack elements.  
3.Dequeue the first half elements of the queue and enqueue them back.  
4.Again push the first half elements into the stack.  
5.Interleave the elements of queue and stack.

// C++ program to interleave the first half of the queue

// with the second half

#include <bits/stdc++.h>

using namespace std;

// Function to interleave the queue

void interLeaveQueue(queue<int>& q)

{

// To check the even number of elements

if (q.size() % 2 != 0)

cout << "Input even number of integers." << endl;

// Initialize an empty stack of int type

stack<int> s;

int halfSize = q.size() / 2;

// Push first half elements into the stack

// queue:16 17 18 19 20, stack: 15(T) 14 13 12 11

for (int i = 0; i < halfSize; i++) {

s.push(q.front());

q.pop();

}

// enqueue back the stack elements

// queue: 16 17 18 19 20 15 14 13 12 11

while (!s.empty()) {

q.push(s.top());

s.pop();

}

// dequeue the first half elements of queue

// and enqueue them back

// queue: 15 14 13 12 11 16 17 18 19 20

for (int i = 0; i < halfSize; i++) {

q.push(q.front());

q.pop();

}

// Again push the first half elements into the stack

// queue: 16 17 18 19 20, stack: 11(T) 12 13 14 15

for (int i = 0; i < halfSize; i++) {

s.push(q.front());

q.pop();

}

// interleave the elements of queue and stack

// queue: 11 16 12 17 13 18 14 19 15 20

while (!s.empty()) {

q.push(s.top());

s.pop();

q.push(q.front());

q.pop();

}

}

// Driver program to test above function

int main()

{

queue<int> q;

q.push(11);

q.push(12);

q.push(13);

q.push(14);

q.push(15);

q.push(16);

q.push(17);

q.push(18);

q.push(19);

q.push(20);

interLeaveQueue(q);

int length = q.size();

for (int i = 0; i < length; i++) {

cout << q.front() << " ";

q.pop();

}

return 0;

}

**Output:**

11 16 12 17 13 18 14 19 15 20

**Time complexity:** O(n).  
**Auxiliary Space:** O(n).

# 302. Find the first circular tour that visits all Petrol Pumps

Suppose there is a circle. There are **N** petrol pumps on that circle. You will be given two sets of data.  
**1.** The amount of petrol that every petrol pump has.  
**2.** Distance from that petrol pump to the next petrol pump.  
Find a starting point where the truck can start to get through the complete circle without exhausting its petrol in between.  
**Note :**  Assume for 1 litre petrol, the truck can go 1 unit of distance.

**Example 1:**

**Input:**

N = 4

Petrol = 4 6 7 4

Distance = 6 5 3 5

**Output:** 1

**Explanation: T**here are 4 petrol pumps with

amount of petrol and distance to next

petrol pump value pairs as {4, 6}, {6, 5},

{7, 3} and {4, 5}. The first point from

where truck can make a circular tour is

2nd petrol pump. Output in this case is 1

(index of 2nd petrol pump).

**Your Task:**  
Your task is to complete the function **tour**() which takes the required data as inputs and returns an integer denoting a point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). If there exists multiple such starting points, then the function must return the first one out of those. (return -1 otherwise)

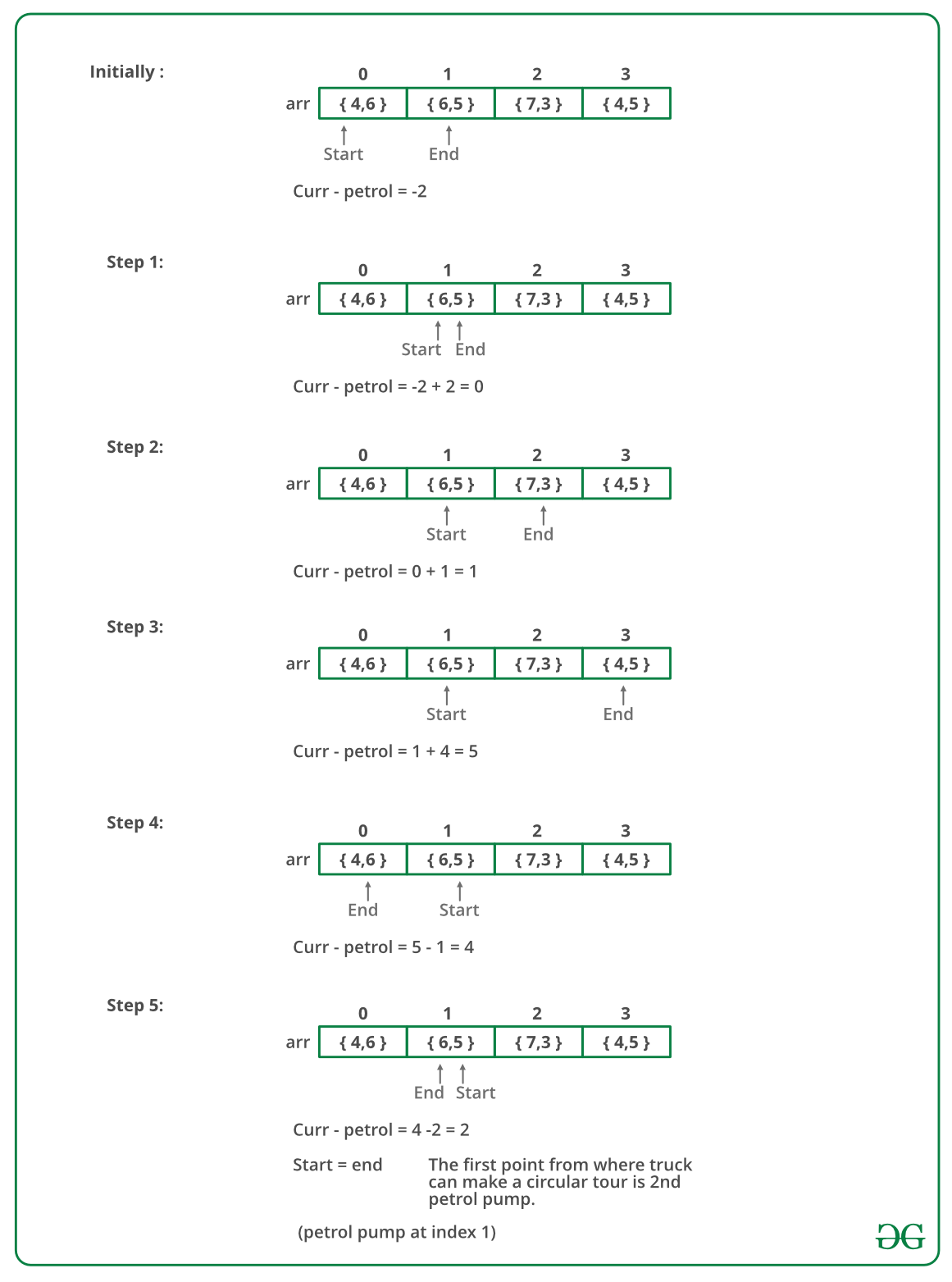
**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space**: O(1)

**Constraints:**  
2 ≤ N ≤ 10000  
1 ≤ petrol, distance ≤ 1000

## Solution:

A **Simple Solution** is to consider every petrol pumps as a starting point and see if there is a possible tour. If we find a starting point with a feasible solution, we return that starting point. The worst case time complexity of this solution is O(n^2).  
An efficient approach is to **use a Queue**to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or the current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeuing petrol pumps until the queue becomes empty.  
Instead of creating a separate queue, we use the given array itself as a queue. We maintain two index variables start and end that represent the rear and front of the queue.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

// C++ program to find circular tour for a truck

#include <bits/stdc++.h>

using namespace std;

// A petrol pump has petrol and distance to next petrol pump

class petrolPump

{

public:

int petrol;

int distance;

};

// The function returns starting point if there is a possible solution,

// otherwise returns -1

int printTour(petrolPump arr[], int n)

{

// Consider first petrol pump as a starting point

int start = 0;

int end = 1;

int curr\_petrol = arr[start].petrol - arr[start].distance;

/\* Run a loop while all petrol pumps are not visited.

And we have reached first petrol pump again with 0 or more petrol \*/

while (end != start || curr\_petrol < 0)

{

// If curremt amount of petrol in truck becomes less than 0, then

// remove the starting petrol pump from tour

while (curr\_petrol < 0 && start != end)

{

// Remove starting petrol pump. Change start

curr\_petrol -= arr[start].petrol - arr[start].distance;

start = (start + 1) % n;

// If 0 is being considered as start again, then there is no

// possible solution

if (start == 0)

return -1;

}

// Add a petrol pump to current tour

curr\_petrol += arr[end].petrol - arr[end].distance;

end = (end + 1) % n;

}

// Return starting point

return start;

}

// Driver code

int main()

{

petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};

int n = sizeof(arr)/sizeof(arr[0]);

int start = printTour(arr, n);

(start == -1)? cout<<"No solution": cout<<"Start = "<<start;

return 0;

}

**Output:**

start = 2

**Time Complexity:** We are visiting each petrol pump exactly once, therefore the time complexity is **O(n)**

**Auxiliary Space:**O(1)

Another efficient solution can be to find out the first petrol pump where the amount of petrol is greater than or equal to the distance to be covered to reach the next petrol pump. Now we mark that petrol pump as **start**and now we check whether we can finish the journey towards the end point. If in the middle, at any petrol pump, the amount of petrol is less than the distance to be covered to reach the next petrol pump, then we can say we cannot complete the circular tour from **start**. We again try to find out the next point from where we can start our journey i.e. the next petrol pump where the amount of petrol is greater than or equal to the distance to be covered and we mark it as **start**. We need not look at any petrol pump in between the initial petrol pump marked as **start**and the new **start**as we know that we cannot complete the journey if we start from any middle petrol pump because eventually we will arrive at a point where amount of petrol is less than the distance. Now we repeat the process until we reach the last petrol pump and update our **start**as and when required. After we reach our last petrol pump, we try to reach our first petrol pump from the last and let’s say we have a remaining amount of petrol as **curr\_petrol**. Now we again start traveling from the first petrol pump and take the advantage of our **curr\_petrol**and try to reach the **start**. If we can reach the **start**, then we may conclude that **start**can be our starting point.

Below is the implementation of the above approach:

// C++ program to find circular tour for a truck

#include <bits/stdc++.h>

using namespace std;

// A petrol pump has petrol and distance to next petrol pump

class petrolPump {

public:

int petrol;

int distance;

};

// The function returns starting point if there is a

// possible solution, otherwise returns -1

int printTour(petrolPump arr[], int n)

{

int start;

for (int i = 0; i < n; i++) {

// Identify the first petrol pump from where we

// might get a full circular tour

if (arr[i].petrol >= arr[i].distance) {

start = i;

break;

}

}

// To store the excess petrol

int curr\_petrol = 0;

int i;

for (i = start; i < n;) {

curr\_petrol += (arr[i].petrol - arr[i].distance);

// If at any point remaining petrol is less than 0,

// it means that we cannot start our journey from

// current start

if (curr\_petrol < 0) {

// We move to the next petrol pump

i++;

// We try to identify the next petrol pump from

// where we might get a full circular tour

for (; i < n; i++) {

if (arr[i].petrol >= arr[i].distance) {

start = i;

// Reset rem\_petrol

curr\_petrol = 0;

break;

}

}

}

else {

// Move to the next petrolpump if curr\_petrol is

// >= 0

i++;

}

}

// If remaining petrol is less than 0 while we reach the

// first petrol pump, it means no circular tour is

// possible

if (curr\_petrol < 0) {

return -1;

}

for (int j = 0; j < start; j++) {

curr\_petrol += (arr[j].petrol - arr[j].distance);

// If remaining petrol is less than 0 at any point

// before we reach initial start, it means no

// circular tour is possible

if (curr\_petrol < 0) {

return -1;

}

}

// If we have successfully reached intial\_start, it

// means can get a circular tour from final\_start, hence

// return it

return start;

}

// Driver code

int main()

{

petrolPump arr[] = { { 6, 4 }, { 3, 6 }, { 7, 3 } };

int n = sizeof(arr) / sizeof(arr[0]);

int start = printTour(arr, n);

(start == -1) ? cout << "No solution"

: cout << "Start = " << start;

return 0;

}

**Output:**

start = 2

**Time Complexity:** O(n)

**Auxiliary Space:** O(1)

**My Implementation:**

int tour(petrolPump p[],int n)

{

int extra=0, req=0, start=0;

for(int i=0;i<n;i++){

if(extra+p[i].petrol-p[i].distance >= 0)

extra = extra + p[i].petrol - p[i].distance;

else{

req = req + extra + p[i].petrol - p[i].distance;

extra = 0;

start = i+1;

}

}

if(req+extra>=0)

return start;

else

return -1;

}

**Easy to understand soln:**

class Solution {

public int canCompleteCircuit(int[] gas, int[] cost) {

int n = gas.length;

int total\_tank = 0;

int curr\_tank = 0;

int starting\_station = 0;

for (int i = 0; i < n; ++i) {

total\_tank += gas[i] - cost[i];

curr\_tank += gas[i] - cost[i];

// If one couldn't get here,

if (curr\_tank < 0) {

// Pick up the next station as the starting one.

starting\_station = i + 1;

// Start with an empty tank.

curr\_tank = 0;

}

}

return total\_tank >= 0 ? starting\_station : -1;

}

}

# 303. Minimum time required to rot all oranges

Given a grid of dimension **nxm** where each cell in the grid can have values 0, 1 or 2 which has the following meaning:  
**0**: Empty cell  
**1** : Cells have fresh oranges  
**2** : Cells have rotten oranges

We have to determine what is the minimum time required to rot all oranges. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (**up**, **down**, **left** and **right**) in unit time. 

**Example 1:**

**Input:** grid = {{0,1,2},{0,1,2},{2,1,1}}

**Output:** 1

**Explanation:** The grid is-

0 1 2

0 1 2

2 1 1

Oranges at positions (0,2), (1,2), (2,0)

will rot oranges at (0,1), (1,1), (2,2) and

(2,1) in unit time.

**Example 2:**

**Input:** grid = {{2,2,0,1}}

**Output:** -1

**Explanation:** The grid is-

2 2 0 1

Oranges at (0,0) and (0,1) can't rot orange at

(0,3).

**Your Task:**  
You don't need to read or print anything, Your task is to complete the function **orangesRotting()**which takes grid as input parameter and returns the minimum time to rot all the fresh oranges. If not possible returns -1.

**Expected Time Complexity:**O(n\*m)  
**Expected Auxiliary Space:**O(n)

**Constraints:**  
1 ≤ n, m ≤ 500

## Solution:

**Naive Solution:**

* **Approach:** The idea is very basic. Traverse through all oranges in multiple rounds. In every round, rot the oranges to the adjacent position of oranges which were rotten in the last round.
* **Algorithm:**
  1. Create a variable *no = 2* and *changed = false*
  2. Run a loop until there is no cell of the matrix which is changed in an iteration.
  3. Run a nested loop and traverse the matrix. If the element of the matrix is equal to *no* then assign the adjacent elements to no + 1 if the adjacent element’s value is equal to 1, i.e. not rotten, and update changed to true.
  4. Traverse the matrix and check if there is any cell which is 1. If 1 is present return -1
  5. Else return no – 2
* **Implementation:**

// C++ program to rot all oranges when u can move

// in all the four direction from a rotten orange

#include <bits/stdc++.h>

using namespace std;

const int R = 3;

const int C = 5;

// Check if i, j is under the array limits of row and column

bool issafe(int i, int j)

{

if (i >= 0 && i < R && j >= 0 && j < C)

return true;

return false;

}

int rotOranges(int v[R][C])

{

bool changed = false;

int no = 2;

while (true) {

for (int i = 0; i < R; i++) {

for (int j = 0; j < C; j++) {

// Rot all other oranges present at

// (i+1, j), (i, j-1), (i, j+1), (i-1, j)

if (v[i][j] == no) {

if (issafe(i + 1, j) && v[i + 1][j] == 1) {

v[i + 1][j] = v[i][j] + 1;

changed = true;

}

if (issafe(i, j + 1) && v[i][j + 1] == 1) {

v[i][j + 1] = v[i][j] + 1;

changed = true;

}

if (issafe(i - 1, j) && v[i - 1][j] == 1) {

v[i - 1][j] = v[i][j] + 1;

changed = true;

}

if (issafe(i, j - 1) && v[i][j - 1] == 1) {

v[i][j - 1] = v[i][j] + 1;

changed = true;

}

}

}

}

// if no rotten orange found it means all

// oranges rottened now

if (!changed)

break;

changed = false;

no++;

}

for (int i = 0; i < R; i++) {

for (int j = 0; j < C; j++) {

// if any orange is found to be

// not rotten then ans is not possible

if (v[i][j] == 1)

return -1;

}

}

// Because initial value for a rotten

// orange was 2

return no - 2;

}

// Driver function

int main()

{

int v[R][C] = { { 2, 1, 0, 2, 1 },

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

{ 1, 0, 0, 2, 1 } };

cout << "Max time incurred: " << rotOranges(v);

return 0;

}

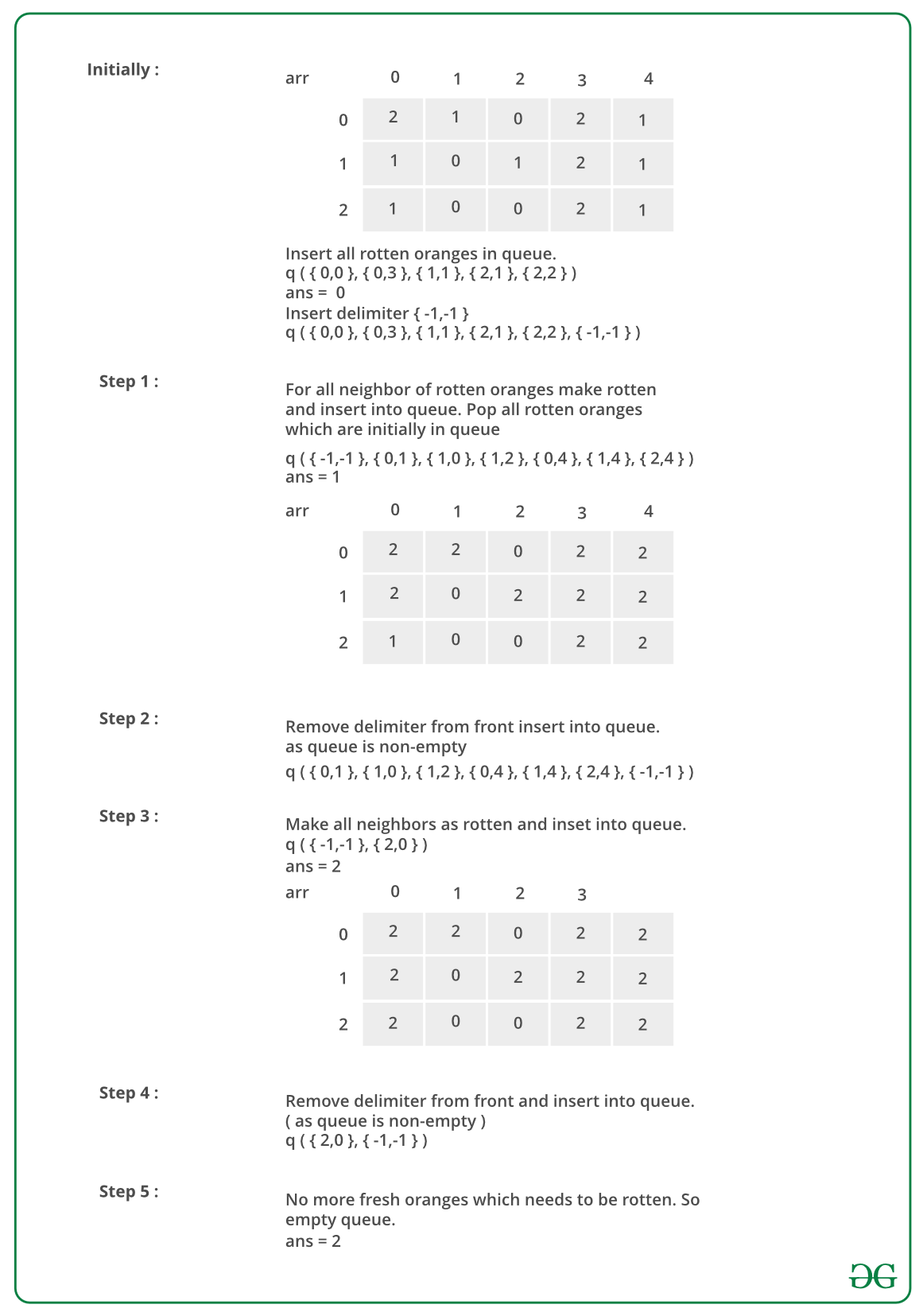
**Output:**

Max time incurred: 2

* **Complexity Analysis:**
  + **Time Complexity**: O((R\*C) \* (R \*C)).   
    The matrix needs to be traversed again and again until there is no change in the matrix, that can happen max(R \*C)/2 times. So time complexity is O((R \* C) \* (R \*C)).
  + **Space Complexity:**O(1).   
    No extra space is required.

**Efficient Solution**

* **Approach:** The idea is to use [Breadth First Search](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/). The condition of oranges getting rotten is when they come in contact with other rotten oranges. This is similar to breadth-first search where the graph is divided into layers or circles and the search is done from lower or closer layers to deeper or higher layers. In the previous approach, the idea was based on BFS but the implementation was poor and inefficient. To find the elements whose values are *no* the whole matrix had to be traversed. So that time can be reduced by using this efficient approach of BFS.
* **Algorithm:**
  1. Create an empty queue *Q*.
     1. Find all rotten oranges and enqueue them to Q. Also, enqueue a delimiter to indicate the beginning of the next time frame.
     2. Run a loop While Q is not empty
     3. Do following while delimiter in Q is not reached
  2. Dequeue an orange from the queue, rot all adjacent oranges. While rotting the adjacent, make sure that the time frame is incremented only once. And the time frame is not incremented if there are no adjacent oranges.
  3. Dequeue the old delimiter and enqueue a new delimiter. The oranges rotten in the previous time frame lie between the two delimiters.
* **Dry run of the above approach:**



* **Implementation:**

// C++ program to find minimum time required to make all

// oranges rotten

#include<bits/stdc++.h>

#define R 3

#define C 5

using namespace std;

// function to check whether a cell is valid / invalid

bool isvalid(int i, int j)

{

return (i >= 0 && j >= 0 && i < R && j < C);

}

// structure for storing coordinates of the cell

struct ele {

int x, y;

};

// Function to check whether the cell is delimiter

// which is (-1, -1)

bool isdelim(ele temp)

{

return (temp.x == -1 && temp.y == -1);

}

// Function to check whether there is still a fresh

// orange remaining

bool checkall(int arr[][C])

{

for (int i=0; i<R; i++)

for (int j=0; j<C; j++)

if (arr[i][j] == 1)

return true;

return false;

}

// This function finds if it is possible to rot all oranges or not.

// If possible, then it returns minimum time required to rot all,

// otherwise returns -1

int rotOranges(int arr[][C])

{

// Create a queue of cells

queue<ele> Q;

ele temp;

int ans = 0;

// Store all the cells having rotten orange in first time frame

for (int i=0; i<R; i++)

{

for (int j=0; j<C; j++)

{

if (arr[i][j] == 2)

{

temp.x = i;

temp.y = j;

Q.push(temp);

}

}

}

// Separate these rotten oranges from the oranges which will rotten

// due the oranges in first time frame using delimiter which is (-1, -1)

temp.x = -1;

temp.y = -1;

Q.push(temp);

// Process the grid while there are rotten oranges in the Queue

while (!Q.empty())

{

// This flag is used to determine whether even a single fresh

// orange gets rotten due to rotten oranges in current time

// frame so we can increase the count of the required time.

bool flag = false;

// Process all the rotten oranges in current time frame.

while (!isdelim(Q.front()))

{

temp = Q.front();

// Check right adjacent cell that if it can be rotten

if (isvalid(temp.x+1, temp.y) && arr[temp.x+1][temp.y] == 1)

{

// if this is the first orange to get rotten, increase

// count and set the flag.

if (!flag) ans++, flag = true;

// Make the orange rotten

arr[temp.x+1][temp.y] = 2;

// push the adjacent orange to Queue

temp.x++;

Q.push(temp);

temp.x--; // Move back to current cell

}

// Check left adjacent cell that if it can be rotten

if (isvalid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1) {

if (!flag) ans++, flag = true;

arr[temp.x-1][temp.y] = 2;

temp.x--;

Q.push(temp); // push this cell to Queue

temp.x++;

}

// Check top adjacent cell that if it can be rotten

if (isvalid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {

if (!flag) ans++, flag = true;

arr[temp.x][temp.y+1] = 2;

temp.y++;

Q.push(temp); // Push this cell to Queue

temp.y--;

}

// Check bottom adjacent cell if it can be rotten

if (isvalid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1) {

if (!flag) ans++, flag = true;

arr[temp.x][temp.y-1] = 2;

temp.y--;

Q.push(temp); // push this cell to Queue

}

Q.pop();

}

// Pop the delimiter

Q.pop();

// If oranges were rotten in current frame than separate the

// rotten oranges using delimiter for the next frame for processing.

if (!Q.empty()) {

temp.x = -1;

temp.y = -1;

Q.push(temp);

}

// If Queue was empty than no rotten oranges left to process so exit

}

// Return -1 if all arranges could not rot, otherwise return ans.

return (checkall(arr))? -1: ans;

}

// Driver program

int main()

{

int arr[][C] = { {2, 1, 0, 2, 1},

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

{1, 0, 0, 2, 1}};

int ans = rotOranges(arr);

if (ans == -1)

cout << "All oranges cannot rotn";

else

cout << "Time required for all oranges to rot => " << ans << endl;

return 0;

}

**Output**

Time required for all oranges to rot => 2

* **Complexity Analysis:**
  + **Time Complexity:** O( R \*C).   
    Each element of the matrix can be inserted into the queue only once so the upper bound of iteration is O(R\*C), i.e. the number of elements. So time complexity is O(R \*C).
  + **Space Complexity:** O(R\*C).   
    To store the elements in a queue O(R\*C) space is needed.

**My code:**

int orangesRotting(vector<vector<int>>& grid) {

int oranges=0, \_time=0, m=grid.size(), n=grid[0].size();

queue<pair<int, int>> q;

for(int i=0;i<m;i++){

for(int j=0;j<n;j++){

if(grid[i][j]==1)

oranges++;

else if(grid[i][j]==2){

oranges++;

q.push(make\_pair(i,j));

}

}

}

int count = q.size();

while(count<oranges && !q.empty()){

int size = q.size();

while(size--){

pair<int, int> p = q.front();

int i = p.first, j = p.second;

q.pop();

if(j-1>=0 && grid[i][j-1]==1){

grid[i][j-1] = 2;

q.push(make\_pair(i,j-1));

}

if(j+1<n && grid[i][j+1]==1){

grid[i][j+1] = 2;

q.push(make\_pair(i,j+1));

}

if(i-1>=0 && grid[i-1][j]==1){

grid[i-1][j] = 2;

q.push(make\_pair(i-1,j));

}

if(i+1<m && grid[i+1][j]==1){

grid[i+1][j] = 2;

q.push(make\_pair(i+1,j));

}

}

count += q.size();

\_time++;

}

if(count==oranges)

return \_time;

return -1;

}

# 304. [Distance of nearest cell having 1 in a binary matrix](https://practice.geeksforgeeks.org/problems/distance-of-nearest-cell-having-1/0)

Given a binary grid. Find the distance of nearest 1 in the grid for each cell.  
The distance is calculated as **|i1 – i2| + |j1 – j2|**, where i1, j1 are the row number and column number of the current cell and i2, j2 are the row number and column number of the nearest cell having value 1.

**Example 1:**

**Input:** grid = {{0,1,1,0},{1,1,0,0},{0,0,1,1}}

**Output:** {{1,0,0,1},{0,0,1,1},{1,1,0,0}}

**Explanation:** The grid is-

0 1 1 0

1 1 0 0

0 0 1 1

0's at (0,0), (0,3), (1,2), (1,3), (2,0) and

(2,1) are at a distance of 1 from 1's at (0,1),

(0,2), (0,2), (2,3), (1,0) and (1,1)

respectively.

**Example 2:**

**Input:** grid = {{1,0,1},{1,1,0},{1,0,0}}

**Output:** {{0,1,0},{0,0,1},{0,1,2}}

**Explanation:** The grid is-

1 0 1

1 1 0

1 0 0

0's at (0,1), (1,2), (2,1) and (2,2) are at a

distance of 1, 1, 1 and 2 from 1's at (0,0),

(0,2), (2,0) and (1,1) respectively.

**Yout Task:**  
You don't need to read or print anything, Your task is to complete the function **nearest()**which takes grid as input parameter and returns a matrix of same dimensions where the value at index (i, j) in the resultant matrix signifies the minimum distance of 1 in the matrix from grid[i][j].

**Expected Time Complexity:**O(n\*m)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 ≤ n, m ≤ 500

## Solution:

**Method 1:** This method uses a simple brute force approach to arrive at the solution.

* **Approach:** The idea is to traverse the matrix for each cell and find the minimum distance, To find the minimum distance traverse the matrix and find the cell which contains 1 and calculates the distance between two cells and store the minimum distance.
* **Algorithm :**
  1. Traverse the matrix from start to end (using two nested loops)
  2. For every element find the closest element which contains 1. To find the closest element traverse the matrix and find the minimum distance.
  3. Fill the minimum distance in the matrix.

**Implementation:**

// C++ program to find distance of nearest

// cell having 1 in a binary matrix.

#include<bits/stdc++.h>

#define N 3

#define M 4

using namespace std;

// Print the distance of nearest cell

// having 1 for each cell.

void printDistance(int mat[N][M])

{

int ans[N][M];

// Initialize the answer matrix with INT\_MAX.

for (int i = 0; i < N; i++)

for (int j = 0; j < M; j++)

ans[i][j] = INT\_MAX;

// For each cell

for (int i = 0; i < N; i++)

for (int j = 0; j < M; j++)

{

// Traversing the whole matrix

// to find the minimum distance.

for (int k = 0; k < N; k++)

for (int l = 0; l < M; l++)

{

// If cell contain 1, check

// for minimum distance.

if (mat[k][l] == 1)

ans[i][j] = min(ans[i][j],

abs(i-k) + abs(j-l));

}

}

// Printing the answer.

for (int i = 0; i < N; i++)

{

for (int j = 0; j < M; j++)

cout << ans[i][j] << " ";

cout << endl;

}

}

// Driven Program

int main()

{

int mat[N][M] =

{

0, 0, 0, 1,

0, 0, 1, 1,

0, 1, 1, 0

};

printDistance(mat);

return 0;

}

**Output:**

3 2 1 0

2 1 0 0

1 0 0 1

**Complexity Analysis:**

* **Time Complexity:** O(N2\*M2).   
  For every element in the matrix, the matrix is traversed and there are N\*M elements So the time complexity is O(N2\*M2).
* **Space Complexity:**O(1).   
  No extra space is required.

**Method 2:** This method uses the [BFS or breadth-first search](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) technique to arrive at the solution.

vector<vector<int>>nearest(vector<vector<int>>grid)

{

queue<pair<int, int>> q;

int m = grid.size(), n = grid[0].size();

vector<vector<int>> res(m, vector<int> (n, INT\_MAX-1));

for(int i=0;i<m;i++){

for(int j=0;j<n;j++){

if(grid[i][j]==1){

res[i][j] = 0;

q.push(make\_pair(i,j));

}

}

}

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

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

while(!q.empty()){

pair<int, int> p = q.front();

q.pop();

for(int i=0;i<4;i++){

int x = p.first + r[i], y = p.second + c[i];

if(x>=0 && x<m && y>=0 && y<n){

if(res[p.first][p.second]+1 < res[x][y]){

res[x][y] = res[p.first][p.second]+1;

q.push(make\_pair(x,y));

}

}

}

}

return res;

}

**Complexity Analysis:**

* **Time Complexity:** O(N\*M).   
  In BFS traversal every element is traversed only once so time Complexity is O(M\*N).
* **Space Complexity:**O(M\*N).   
  To store every element in the matrix O(M\*N) space is required.

# 305. First negative integer in every window of size “k”

Given an array **A[]**of size **N** and a positive integer **K**, find the first negative integer for each and every window(contiguous subarray) of size **K**.

**Example 1:**

**Input :**

N = 5

A[] = {-8, 2, 3, -6, 10}

K = 2

**Output :**

-8 0 -6 -6

**Explanation :**

First negative integer for each window of size k

**{-8, 2}** = -8

**{2, 3}** = 0 (does not contain a negative integer)

**{3, -6}** = -6

**{-6, 10}** = -6

**Example 2:**

**Input :**

N = 8

A[] = {12, -1, -7, 8, -15, 30, 16, 28}

K = 3

**Output :**

-1 -1 -7 -15 -15 0

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **printFirstNegativeInteger()** which takes the array **A[]**, its size **N**and an integer **K**as inputs and returns the first negative number in every window of size K starting from the first till the end. If a window does not contain a negative integer , then return 0 for that window.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(K)  
  
**Constraints:**  
1 <= N <= 105  
1 <= A[i] <= 105  
1 <= K <= N

## Solution:

**Naive Approach:** Run two loops. In the outer loop, take all subarrays(windows) of size k. In the inner loop, get the first negative integer of the current subarray(window).

// C++ implementation to find the first negative

// integer in every window of size k

#include <bits/stdc++.h>

using namespace std;

// function to find the first negative

// integer in every window of size k

void printFirstNegativeInteger(int arr[], int n, int k)

{

// flag to check whether window contains

// a negative integer or not

bool flag;

// Loop for each subarray(window) of size k

for (int i = 0; i<(n-k+1); i++)

{

flag = false;

// traverse through the current window

for (int j = 0; j<k; j++)

{

// if a negative integer is found, then

// it is the first negative integer for

// current window. Print it, set the flag

// and break

if (arr[i+j] < 0)

{

cout << arr[i+j] << " ";

flag = true;

break;

}

}

// if the current window does not

// contain a negative integer

if (!flag)

cout << "0" << " ";

}

}

// Driver program to test above functions

int main()

{

int arr[] = {12, -1, -7, 8, -15, 30, 16, 28};

int n = sizeof(arr)/sizeof(arr[0]);

int k = 3;

printFirstNegativeInteger(arr, n, k);

return 0;

}

**Output :**

-1 -1 -7 -15 -15 0

**Time Complexity :** The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)\*k) which can also be written as O(nk) when k is comparitively much smaller than n, otherwise when k tends to reach n, complexity becomes O(k).

**Efficient Approach:** It is a variation of the problem of [Sliding Window Maximum](https://www.geeksforgeeks.org/sliding-window-maximum-maximum-of-all-subarrays-of-size-k/).   
We create a Dequeue, Di of capacity k, that stores only useful elements of the current window of k elements. An element is useful if it is in the current window and it is a negative integer. We process all array elements one by one and maintain Di to contain useful elements of current window and these useful elements are all negative integers. For a particular window, if Di is not empty then the element at front of the Di is the first negative integer for that window, else that window does not contain a negative integer.

// C++ implementation to find the first negative

// integer in every window of size k

#include <bits/stdc++.h>

using namespace std;

// function to find the first negative

// integer in every window of size k

void printFirstNegativeInteger(int arr[], int n, int k)

{

// A Double Ended Queue, Di that will store indexes of

// useful array elements for the current window of size k.

// The useful elements are all negative integers.

deque<int> Di;

/\* Process first k (or first window) elements of array \*/

int i;

for (i = 0; i < k; i++)

// Add current element at the rear of Di

// if it is a negative integer

if (arr[i] < 0)

Di.push\_back(i);

// Process rest of the elements, i.e., from arr[k] to arr[n-1]

for ( ; i < n; i++)

{

// if Di is not empty then the element at the

// front of the queue is the first negative integer

// of the previous window

if (!Di.empty())

cout << arr[Di.front()] << " ";

// else the window does not have a

// negative integer

else

cout << "0" << " ";

// Remove the elements which are out of this window

while ( (!Di.empty()) && Di.front() < (i - k + 1))

Di.pop\_front(); // Remove from front of queue

// Add current element at the rear of Di

// if it is a negative integer

if (arr[i] < 0)

Di.push\_back(i);

}

// Print the first negative

// integer of last window

if (!Di.empty())

cout << arr[Di.front()] << " ";

else

cout << "0" << " ";

}

// Driver program to test above functions

int main()

{

int arr[] = {12, -1, -7, 8, -15, 30, 16, 28};

int n = sizeof(arr)/sizeof(arr[0]);

int k = 3;

printFirstNegativeInteger(arr, n, k);

return 0;

}

**Output:**

-1 -1 -7 -15 -15 0

**Time Complexity:** O(n)   
**Auxiliary Space:** O(k)

**Optimized Approach:**: It is also possible to accomplish this with constant space. The idea is to have a variable firstNegativeIndex to keep track of the first negative element in the k sized window. At every iteration, we skip the elements which no longer fall under the current k size window (firstNegativeIndex <= i – k) as well as the positive elements.

Below is the solution based upon this approach.

// C++ code for First negative integer

// in every window of size k

#include <iostream>

using namespace std;

void printFirstNegativeInteger(int arr[], int k, int n)

{

int firstNegativeIndex = 0;

int firstNegativeElement;

for(int i = k - 1; i < n; i++)

{

// skip out of window and positive elements

while((firstNegativeIndex < i ) &&

(firstNegativeIndex <= i - k ||

arr[firstNegativeIndex] > 0))

{

firstNegativeIndex ++;

}

// check if a negative element is found, otherwise use 0

if(arr[firstNegativeIndex] < 0)

{

firstNegativeElement = arr[firstNegativeIndex];

}

else

{

firstNegativeElement = 0;

}

cout<<firstNegativeElement << " ";

}

}

// Driver code

int main()

{

int arr[] = { 12, -1, -7, 8, -15, 30, 16, 28};

int n = sizeof(arr)/sizeof(arr[0]);

int k = 3;

printFirstNegativeInteger(arr, k, n);

}

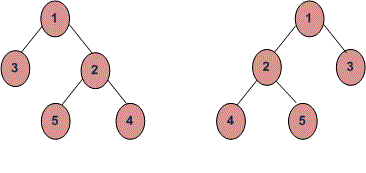
**Output:**

-1 -1 -7 -15 -15 0

**Time Complexity:** O(n)   
**Auxiliary Space:** O(1)

# 306. [Check if all levels of two trees are anagrams or not.](https://www.geeksforgeeks.org/check-if-all-levels-of-two-trees-are-anagrams-or-not/)

Given two binary trees, we have to check if each of their levels are anagrams of each other or not.   
**Example:** 



**Tree 1:**

Level 0 : 1

Level 1 : 3, 2

Level 2 : 5, 4

**Tree 2:**

Level 0 : 1

Level 1 : 2, 3

Level 2 : 4, 5

As we can clearly see all the levels of above two binary trees are anagrams of each other, hence return true.

## Solution:

**Naive Approach:** Below is the step by step explanation of the naive approach to do this: 

1. Write a recursive program for level order traversal of a tree.
2. Traverse each level of both the trees one by one and store the result of traversals in 2 different vectors, one for each tree.
3. Sort both the vectors and compare them iteratively for each level, if they are same for each level then return true else return false.

**Time Complexity:** O(n^2), where n is the number of nodes.

**Efficient Approach:**   
The idea is based on below article.   
[Print level order traversal line by line | Set 1](https://www.geeksforgeeks.org/print-level-order-traversal-line-line/)   
We traverse both trees simultaneously level by level. We store each level both trees in vectors (or array). To check if two vectors are anagram or not, we sort both and then compare.  
**Time Complexity:** **O(nlogn)**, where n is the number of nodes.

/\* Iterative program to check if two trees are level

by level anagram. \*/

#include <bits/stdc++.h>

using namespace std;

// A Binary Tree Node

struct Node

{

struct Node \*left, \*right;

int data;

};

// Returns true if trees with root1 and root2

// are level by level anagram, else returns false.

bool areAnagrams(Node \*root1, Node \*root2)

{

// Base Cases

if (root1 == NULL && root2 == NULL)

return true;

if (root1 == NULL || root2 == NULL)

return false;

// start level order traversal of two trees

// using two queues.

queue<Node \*> q1, q2;

q1.push(root1);

q2.push(root2);

while (1)

{

// n1 (queue size) indicates number of Nodes

// at current level in first tree and n2 indicates

// number of nodes in current level of second tree.

int n1 = q1.size(), n2 = q2.size();

// If n1 and n2 are different

if (n1 != n2)

return false;

// If level order traversal is over

if (n1 == 0)

break;

// Dequeue all Nodes of current level and

// Enqueue all Nodes of next level

vector<int> curr\_level1, curr\_level2;

while (n1 > 0)

{

Node \*node1 = q1.front();

q1.pop();

if (node1->left != NULL)

q1.push(node1->left);

if (node1->right != NULL)

q1.push(node1->right);

n1--;

Node \*node2 = q2.front();

q2.pop();

if (node2->left != NULL)

q2.push(node2->left);

if (node2->right != NULL)

q2.push(node2->right);

curr\_level1.push\_back(node1->data);

curr\_level2.push\_back(node2->data);

}

// Check if nodes of current levels are

// anagrams or not.

sort(curr\_level1.begin(), curr\_level1.end());

sort(curr\_level2.begin(), curr\_level2.end());

if (curr\_level1 != curr\_level2)

return false;

}

return true;

}

// Utility function to create a new tree Node

Node\* newNode(int data)

{

Node \*temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// Driver program to test above functions

int main()

{

// Constructing both the trees.

struct Node\* root1 = newNode(1);

root1->left = newNode(3);

root1->right = newNode(2);

root1->right->left = newNode(5);

root1->right->right = newNode(4);

struct Node\* root2 = newNode(1);

root2->left = newNode(2);

root2->right = newNode(3);

root2->left->left = newNode(4);

root2->left->right = newNode(5);

areAnagrams(root1, root2)? cout << "Yes" : cout << "No";

return 0;

}

Yes

**Note:** In the above program we are comparing the vectors storing each level of a tree directly using not equal to function ‘ != ‘ which compares the vectors first on the basis of their size and then on the basis of their content, hence saving our work of iteratively comparing the vectors.

# 307. Sum of minimum and maximum elements of all subarrays of size “k”.

Given an array of both positive and negative integers, the task is to compute sum of minimum and maximum elements of all sub-array of size k.  
Examples: 

Input : arr[] = {2, 5, -1, 7, -3, -1, -2}

K = 4

Output : 18

Explanation : Subarrays of size 4 are :

{2, 5, -1, 7}, min + max = -1 + 7 = 6

{5, -1, 7, -3}, min + max = -3 + 7 = 4

{-1, 7, -3, -1}, min + max = -3 + 7 = 4

{7, -3, -1, -2}, min + max = -3 + 7 = 4

Sum of all min & max = 6 + 4 + 4 + 4

= 18

## Solution:

**Method 1 (Simple)**   
Run two loops to generate all subarrays of size k and find maximum and minimum values. Finally, return sum of all maximum and minimum elements.   
Time taken by this solution is O(nk).

**Method 2 (Efficient using Dequeue)**   
The idea is to use Dequeue data structure and sliding window concept. We create two empty double-ended queues of size k (‘S’ , ‘G’) that only store indexes of elements of current window that are not useless. An element is useless if it can not be maximum or minimum of next subarrays.

a) In deque 'G', we maintain decreasing order of

values from front to rear

b) In deque 'S', we maintain increasing order of

values from front to rear

1) First window size K

1.1) For deque 'G', if current element is greater

than rear end element, we remove rear while

current is greater.

1.2) For deque 'S', if current element is smaller

than rear end element, we just pop it while

current is smaller.

1.3) insert current element in both deque 'G' 'S'

2) After step 1, front of 'G' contains maximum element

of first window and front of 'S' contains minimum

element of first window. Remaining elements of G

and S may store maximum/minimum for subsequent

windows.

3) After that we do traversal for rest array elements.

3.1) Front element of deque 'G' is greatest and 'S'

is smallest element of previous window

3.2) Remove all elements which are out of this

window [remove element at front of queue ]

3.3) Repeat steps 1.1 , 1.2 ,1.3

4) Return sum of minimum and maximum element of all

sub-array size k.

Below is implementation of above idea

// C++ program to find sum of all minimum and maximum

// elements Of Sub-array Size k.

#include<bits/stdc++.h>

using namespace std;

// Returns sum of min and max element of all subarrays

// of size k

int SumOfKsubArray(int arr[] , int n , int k)

{

int sum = 0; // Initialize result

// The queue will store indexes of useful elements

// in every window

// In deque 'G' we maintain decreasing order of

// values from front to rear

// In deque 'S' we maintain increasing order of

// values from front to rear

deque< int > S(k), G(k);

// Process first window of size K

int i = 0;

for (i = 0; i < k; i++)

{

// Remove all previous greater elements

// that are useless.

while ( (!S.empty()) && arr[S.back()] >= arr[i])

S.pop\_back(); // Remove from rear

// Remove all previous smaller that are elements

// are useless.

while ( (!G.empty()) && arr[G.back()] <= arr[i])

G.pop\_back(); // Remove from rear

// Add current element at rear of both deque

G.push\_back(i);

S.push\_back(i);

}

// Process rest of the Array elements

for ( ; i < n; i++ )

{

// Element at the front of the deque 'G' & 'S'

// is the largest and smallest

// element of previous window respectively

sum += arr[S.front()] + arr[G.front()];

// Remove all elements which are out of this

// window

while ( !S.empty() && S.front() <= i - k)

S.pop\_front();

while ( !G.empty() && G.front() <= i - k)

G.pop\_front();

// remove all previous greater element that are

// useless

while ( (!S.empty()) && arr[S.back()] >= arr[i])

S.pop\_back(); // Remove from rear

// remove all previous smaller that are elements

// are useless

while ( (!G.empty()) && arr[G.back()] <= arr[i])

G.pop\_back(); // Remove from rear

// Add current element at rear of both deque

G.push\_back(i);

S.push\_back(i);

}

// Sum of minimum and maximum element of last window

sum += arr[S.front()] + arr[G.front()];

return sum;

}

// Driver program to test above functions

int main()

{

int arr[] = {2, 5, -1, 7, -3, -1, -2} ;

int n = sizeof(arr)/sizeof(arr[0]);

int k = 3;

cout << SumOfKsubArray(arr, n, k) ;

return 0;

}

**Output:**

14

**Time Complexity:** O(n)

# 308. [Minimum sum of squares of character counts in a given string after removing “k” characters.](https://practice.geeksforgeeks.org/problems/game-with-string/0)

Given a string **s** of lowercase alphabets and a number k, the task is to print the minimum value of the string after removal of **‘k’** characters. The value of a string is defined as the sum of squares of the count of each distinct character.

**Example 1:**

**Input:** s = abccc, k = 1

**Output:** 6

**Explaination:**

We remove c to get the value as 12 + 12 + 22

**Example 2:**

**Input:** s = aabcbcbcabcc, k = 3

**Output:** 27

**Explaination:** We remove two 'c' and one 'b'.

Now we get the value as 32 + 32 + 32.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **minValue()** which takes s and k as input parameters and returns the minimum possible required value.

**Expected Time Complexity:** O(N\*logN)  where N is the length of string  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 ≤ k ≤ |string length| ≤ 100

## Solution:

One clear observation is that we need to remove character with highest frequency. One trick is the character ma  
A **Simple solution** is to use sorting technique through all current highest frequency reduce up to k times. For After every reduce again sort frequency array.   
A **Better Solution** used to Priority Queue which has to the highest element on top. 

1. Initialize empty priority queue.
2. Count frequency of each character and Store into temp array.
3. Remove K characters which have highest frequency from queue.
4. Finally Count Sum of square of each element and return it.

Below is the implementation of the above idea.

// C++ program to find min sum of squares

// of characters after k removals

#include <bits/stdc++.h>

using namespace std;

const int MAX\_CHAR = 26;

// Main Function to calculate min sum of

// squares of characters after k removals

int minStringValue(string str, int k)

{

int l = str.length(); // find length of string

// if K is greater than length of string

// so reduced string will become 0

if (k >= l)

return 0;

// Else find Frequency of each character and

// store in an array

int frequency[MAX\_CHAR] = { 0 };

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

frequency[str[i] - 'a']++;

// Push each char frequency into a priority\_queue

priority\_queue<int> q;

for (int i = 0; i < MAX\_CHAR; i++)

q.push(frequency[i]);

// Removal of K characters

while (k--) {

// Get top element in priority\_queue,

// remove it. Decrement by 1 and again

// push into priority\_queue

int temp = q.top();

q.pop();

temp = temp - 1;

q.push(temp);

}

// After removal of K characters find sum

// of squares of string Value

int result = 0; // Initialize result

while (!q.empty()) {

int temp = q.top();

result += temp \* temp;

q.pop();

}

return result;

}

// Driver Code

int main()

{

string str = "abbccc"; // Input 1

int k = 2;

cout << minStringValue(str, k) << endl;

str = "aaab"; // Input 2

k = 2;

cout << minStringValue(str, k);

return 0;

}

**Output:**

6

2

**Time Complexity: O(n\*logn)**

**Space Complexity: O(26)**

# 309. Queue based approach or first non-repeating character in a stream.

## Same as ques 161.(Linked List)

# 310. Next Smaller Element

Given an array, print the Next Smaller Element (NSE) for every element. The NSE for an element x is the first smaller element on the right side of x in array. Elements for which no smaller element exist (on right side), consider NSE as -1.   
Examples:   
**a)**For any array, rightmost element always has NSE as -1.   
**b)**For an array which is sorted in increasing order, all elements have NSE as -1.   
**c)**For the input array **[4, 8, 5, 2, 25}**, the NSE for each element are as follows.

Element NSE

4 --> 2

8 --> 5

5 --> 2

2 --> -1

25 --> -1

**d)** For the input array**[13, 7, 6, 12},** the next smaller elements for each element are as follows.

## Solution:

**Method 1 (Simple)**   
Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first smaller element for the element picked by outer loop. If a smaller element is found then that element is printed as next, otherwise, -1 is printed.

**Time Complexity**: O(n^2) The worst case occurs when all elements are sorted in decreasing order.

// Simple C++ program to print

// next smaller elements in a given array

#include "bits/stdc++.h"

using namespace std;

/\* prints element and NSE pair

for all elements of arr[] of size n \*/

void printNSE(int arr[], int n)

{

int next, i, j;

for (i = 0; i < n; i++)

{

next = -1;

for (j = i + 1; j < n; j++)

{

if (arr[i] > arr[j])

{

next = arr[j];

break;

}

}

cout << arr[i] << " -- "

<< next << endl;

}

}

// Driver Code

int main()

{

int arr[]= {11, 13, 21, 3};

int n = sizeof(arr) / sizeof(arr[0]);

printNSE(arr, n);

return 0;

}

**Output**

11 -- 3

13 -- 3

21 -- 3

3 -- -1

**Method 2 (Using Stack)**   
This problem is similar to [next greater element](https://www.geeksforgeeks.org/next-greater-element/). Here we maintain items in increasing order in the stack (instead of decreasing in next greater element problem).

1. Push the first element to stack.
2. Pick rest of the elements one by one and follow following steps in loop.
   * Mark the current element as *next*.
   * If stack is not empty, then compare *next* with stack top. If *next*is smaller than top then *next*is the NSE for the top. Keep popping from the stack while top is greater than *next*. *next* becomes the NSE for all such popped elements
   * Push *next* into the stack
3. After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

**Note:**To achieve same order, we use a stack of pairs, where first element is the value and second element is index of array element.

**Time Complexity: O(n)**

// A Stack based C++ program to find next

// smaller element for all array elements

#include <bits/stdc++.h>

using namespace std;

// prints NSE for elements of array arr[] of size n

void printNSE(int arr[], int n)

{

stack<pair<int, int> > s;

vector<int> ans(n);

// iterate for rest of the elements

for (int i = 0; i < n; i++) {

int next = arr[i];

// if stack is empty then this element can't be NSE

// for any other element, so just push it to stack

// so that we can find NSE for it, and continue

if (s.empty()) {

s.push({ next, i });

continue;

}

// while stack is not empty and the top element is

// greater than next

// a) NSE for top is next, use top's index to

// maintain original order

// b) pop the top element from stack

while (!s.empty() && s.top().first > next) {

ans[s.top().second] = next;

s.pop();

}

// push next to stack so that we can find NSE for it

s.push({ next, i });

}

// After iterating over the loop, the remaining elements

// in stack do not have any NSE, so set -1 for them

while (!s.empty()) {

ans[s.top().second] = -1;

s.pop();

}

for (int i = 0; i < n; i++) {

cout << arr[i] << " ---> " << ans[i] << endl;

}

}

// Driver program to test above functions

int main()

{

int arr[] = { 11, 13, 21, 3 };

int n = sizeof(arr) / sizeof(arr[0]);

printNSE(arr, n);

return 0;

}

**Output**

11 ---> 3

13 ---> 3

21 ---> 3

3 ---> -1