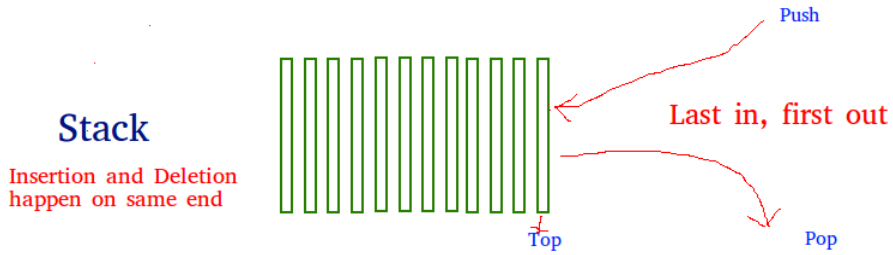
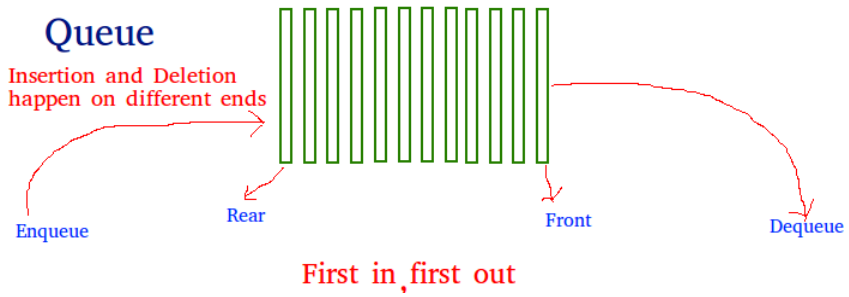
1. Implement Stack using Queues

The problem is opposite of this post. We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.





Recommended: Please solve it on *"PRACTICE"* first, before moving on to the solution.

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

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;
}
// This code is contributed by Chhavi
```

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;
```

```
}
// This code is contributed by Chhavi
```

Output :

```
current size: 4
4
3
2
current size: 2
```

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

Question: 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. All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

Example:

```
Consider the following SpecialStack
16  --> TOP
15
29
19
18

When getMin() is called it should return 15,
which is the minimum element in the current stack.

If we do pop two times on stack, the stack becomes
29  --> TOP
19
18

When getMin() is called, it should return 18
which is the minimum in the current stack.
```

An approach that uses O(1) time and O(n) extra space is discussed here.

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 Inserted	Present Stack	minELe
3	3	3
5	5 3	3
2	1 5 3	2
1	0 1 5 3	1
1	1 0 1 5 3	1
-1	-3 1 0 1 5 3	-1

- 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()

Number Removed	Orignal Number	Present Stack	minEle
-	-	-3 1 0 1 5 3	-1
-3	-1	1 0 1 5 3	1
1	1	0 1 5 3	1
0	1	1 5 3	2
1	2	5 3	3
5	5	3	3

- 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
        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
```

Exercise:

Similar approach can be used to find the maximum element as well. Implement a stack that supports getMax() in O(1) time and constant extra space

3. Reverse a stack without using extra space in O(n)

Reverse a Stack without using recursion and extra space. Even the functional Stack is not allowed.

Examples:

```
Input : 1->2->3->4
Output : 4->3->2->1

Input :  6->5->4
Output : 4->5->6
```

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

We have discussed a way of reversing a string in below post.

Reverse a Stack using Recursion

The above solution requires O(n) extra space. We can reverse a string in O(1) time if we internally represent the stack as a linked list. Reverse a stack would require a reversing a linked list which can be done with O(n) time and O(1) extra space.

Note that push() and pop() operations still take O(1) time.

```
// CPP program to implement Stack
// using linked list so that reverse
// can be done with O(1) extra space.
#include<bits/stdc++.h>
using namespace std;

class StackNode {
    public:
    int data;
    StackNode *next;

    StackNode(int data)
    {
        this->data = data;
        this->next = NULL;
    }
};

class Stack {
```

```
    StackNode *top;

    public:

    // Push and pop operations
    void push(int data)
    {
        if (top == NULL) {
            top = new StackNode(data);
            return;
        }
        StackNode *s = new StackNode(data);
        s->next = top;
        top = s;
    }

    StackNode* pop()
    {
        StackNode *s = top;
        top = top->next;
        return s;
    }

    // prints contents of stack
    void display()
    {
        StackNode *s = top;
        while (s != NULL) {
            cout << s->data << " ";
            s = s->next;
        }
        cout << endl;
    }

    // Reverses the stack using simple
    // linked list reversal logic.
    void reverse()
    {
        StackNode *prev, *cur, *succ;
        cur = prev = top;
        cur = cur->next;
        prev->next = NULL;
        while (cur != NULL) {

            succ = cur->next;
            cur->next = prev;
            prev = cur;
            cur = succ;
        }
        top = prev;
    }
};
```

```
// driver code
int main()
{
    Stack *s = new Stack();
    s->push(1);
    s->push(2);
    s->push(3);
    s->push(4);
    cout << "Original Stack" << endl;;
    s->display();
    cout << endl;

    // reverse
    s->reverse();

    cout << "Reversed Stack" << endl;
    s->display();

    return 0;
}
// This code is contribute by Chhavi.
```

Output:

```
Original Stack
4 3 2 1
Reversed Stack
1 2 3
```

Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

1. For any array, rightmost element always has next greater element as -1.
2. For an array which is sorted in decreasing order, all elements have next greater element as -1.
3. For the input array [4, 5, 2, 25}, the next greater elements for each element are as follows.

```
Element       NGE
   4      -->   5
   5      -->   25
   2      -->   25
   25     -->   -1
```

d) For the input array [13, 7, 6, 12}, the next greater elements for each element are as follows.

```
  Element        NGE
   13      -->    -1
   7       -->     12
   6       -->     12
   12      -->     -1
```

Recommended: Please solve it on "*PRACTICE* " first, before moving on to the 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.

filter_none

edit

play_arrow

brightness_4

```
// 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;
}

// This code is contributed
// by Akanksha Rai(Abby_akku)
```

Output:

```
11 -- 13
13 -- 21
21 -- -1
3 -- -1
```

Time Complexity: O(n^2). The worst case occurs when all elements are sorted in decreasing order.

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

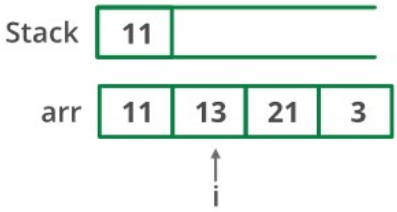
- Finally, push the next in the stack.
- After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

Below image is a dry run of the above approach:

Below is the implementation of the above approach:

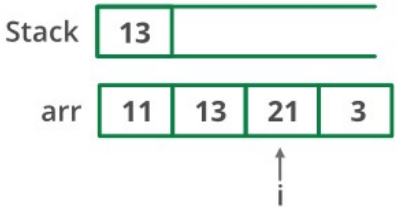
```
// A Stack based C++ program to find next
// greater element for all array elements.
#include <bits/stdc++.h>
using namespace std;

/* 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]);
```

Initially :



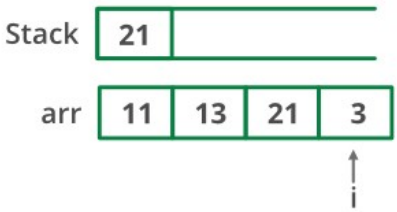
Step 1:

arr[i] > S.top
S.pop()
Next greater element of 11 is 13
S.push(arr[i])



Step 2:

arr[i] > S.top()
S.pop()
Next greater element of 13 is 21
S.push(arr[i])



Step 3:

arr[i] < S.top()
S.push(arr[i])

i reached to end of the array

Step 4:

For all elements in stack next greater element is
Next greater element of 3 is -1
S.pop()

Step 5:

Next greater element of 21 is -1
S.pop()

```
  }

  /* 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 program to test above functions */
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).
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 algorithm

5. Smallest Greater Element on Right Side

Given an array of distinct elements, print the closest greater element for every element. The closest greater element for an element x is the smallest element on the right side of x in array which is greater than x. Elements for which no greater element exist, consider next greater element as -1.

Examples:

```
Input: arr[] = {4, 5, 2, 25}
Output:
Element       NGE
   4      -->   5
   5      -->   25
   2      -->   25
   25     -->   -1

Input: arr[] = {4, 10, 7}
Output:
Element       NGE
   4       -->  7
   10      -->   -1
   7       -->   -1
```

Approach: In this post, we will be discussing how to find the Next Greater Element using C++ STL(set).
Finding the smallest greater element on the right side will be like finding the first greater element of the current element in a list that is sorted.
Consider example 1, The sorted list would look like 2, 4, 5, 25.
Here for element 4, the greater element is 5 as it is next to it, so we print 5 and remove 4 because it would not be greater to the other elements since it is no longer on anyone's right.
Similarly, for 5 it is 25 and we remove 5 from the list, as 5 will not be on the right side of 2 or 25, so it can be deleted.

Given below are the steps to find the Next Greater Element of every index element.

- Insert all the elements in a Set, it will store all the elements in an increasing order.
- Iterate on the array of elements, and for each index, find the upper_bound of the current index element. The upper_bound() returns an iterator which can point to the following position.
    1. If the iterator is pointing to a position past the last element, then there exists no NGE to the current index element.
    2. If the iterator points to a position referring to an element, then that element is the NGE to the current index element.
- Find the position of current index element at every traversal and remove it from the set using >lower_bound() and erase() functions of set.

Below is the implementation of the above approach.

```
// C++ program to print the
// NGE's of array elements using
// C++ STL
#include <bits/stdc++.h>
using namespace std;

// Function to print the NGE
void printNGE(int a[], int n)
{

    set<int> ms;

    // insert in the multiset container
    for (int i = 0; i < n; i++)
        ms.insert(a[i]);

    cout << "Element   "
         << "NGE";

    // traverse for all array elements
    for (int i = 0; i < n; i++) {

        // find the upper_bound in set
        auto it = ms.upper_bound(a[i]);

        // if points to the end, then
        // no NGE of that element
        if (it == ms.end()) {
            cout << "\n   " << a[i]
                 << " ----> " << -1;
        }

        // print the element at that position
        else {
            cout << "\n   " << a[i]
                 << " ----> " << *it;
        }

        // find the first occurrence of
        // the index element and delete it
        it = ms.lower_bound(a[i]);

        // delete one occurrence
        // from the container
        ms.erase(it);
    }
}

// Driver Code
int main()
{
```

```
    int a[] = { 4, 5, 2, 25 };
    int n = sizeof(a) / sizeof(a[0]);

    // Function call to print the NGE
    printNGE(a, n);
    return 0;
}
```

Output:

```
Element   NGE
   4 ----> 5
   5 ----> 25
   2 ----> 25
   25 ----> -1
```

Time Complexity: O(N log N)

6. 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?

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

```
/* 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;
}

// 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 << "Middle Element is " << findMiddle(ms) << endl;
    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Item popped is 77
Item popped is 66
Middle Element is 33
```

7. 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 which produces the set of merged intervals for the given set of intervals.

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 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 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.

```
1. 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 a 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 stack. We can avoid use of extra space by doing merge operations in-place. Below are detailed steps.

```
1) Sort all intervals in decreasing 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 start time of previous interval with end time of current interval.

Below is implementation of 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 {
            arr[index] = arr[i];
            index++;
        }
    }

    // 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]
```

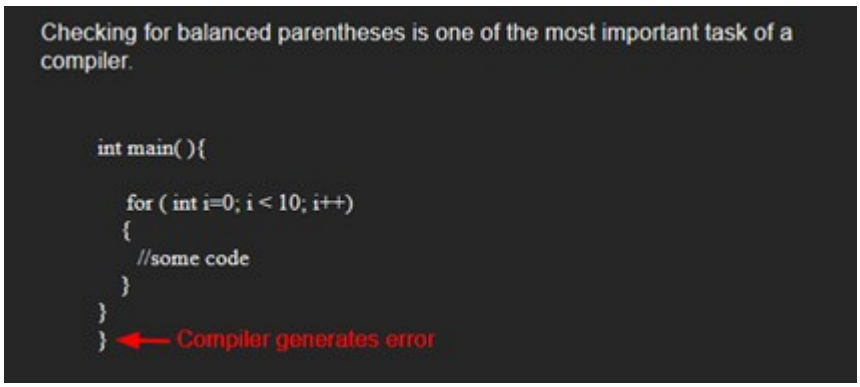
8. Check for balanced parentheses in an expression

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

Example:

Input: exp = “[()]{}{[()()]()}”
Output: Balanced

Input: exp = “[(])”
Output: Not Balanced



Algorithm:

- Declare a character stack 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 parenthesis 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:

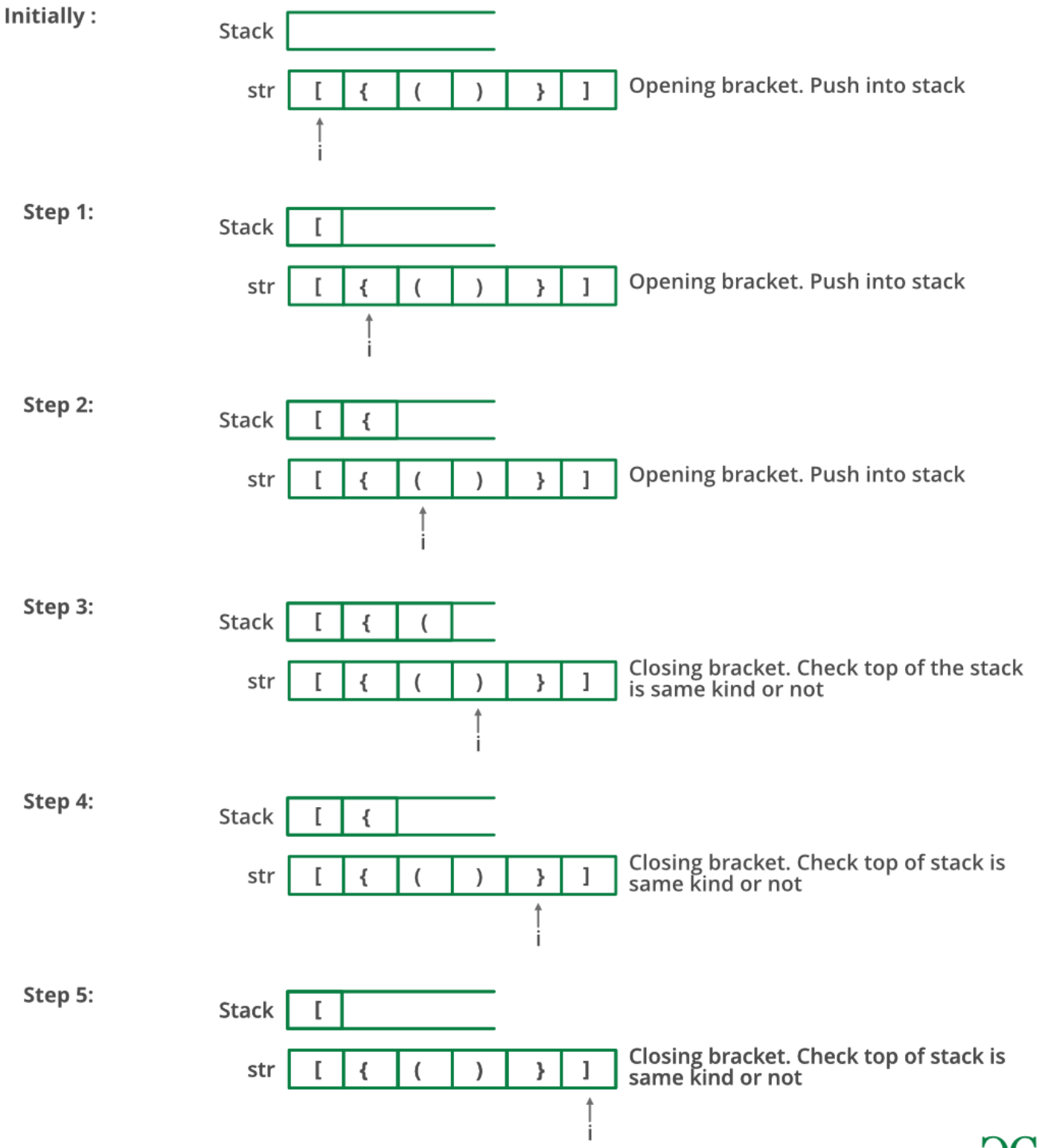
filter_none

edit

play_arrow

brightness_4

```
// CPP program to check for balanced parenthesis.
```




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

// function to check if paranthesis are balanced
bool areParanthesisBalanced(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 program to test above function
```

```
int main()
{
    string expr = "{()}[]";

    if (areParanthesisBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
}
```

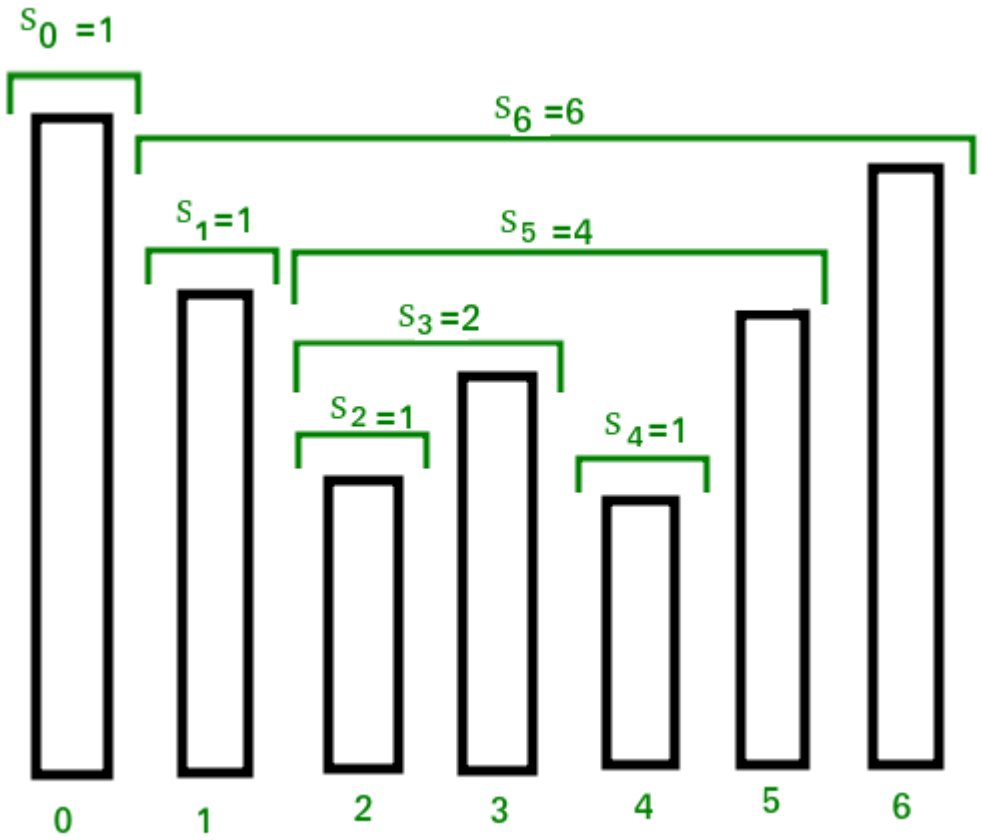
Output:

```
Balanced
```

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

9. The Stock Span Problem

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.
The span Si of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.
For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}



A Simple but inefficient method

Traverse the input price array. For every element being visited, traverse elements on left of it and increment the span value of it while elements on the left side are smaller.

Following is implementation of this method.

```
// C++ program for brute force method
// to calculate stock span values
#include <bits/stdc++.h>
using namespace std;

// Fills array S[] with span values
void calculateSpan(int price[], int n, int S[])
{
    // Span value of first day is always 1
    S[0] = 1;

    // Calculate span value of remaining days
    // by linearly checking previous days
    for (int i = 1; i < n; i++)
    {
        S[i] = 1; // Initialize span value

        // Traverse left while the next element
        // on left is smaller than price[i]
        for (int j = i - 1; (j >= 0) &&
                (price[i] >= price[j]); j--)
            S[i]++;
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver code
int main()
{
    int price[] = { 10, 4, 5, 90, 120, 80 };
    int n = sizeof(price) / sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);
```

```
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output :

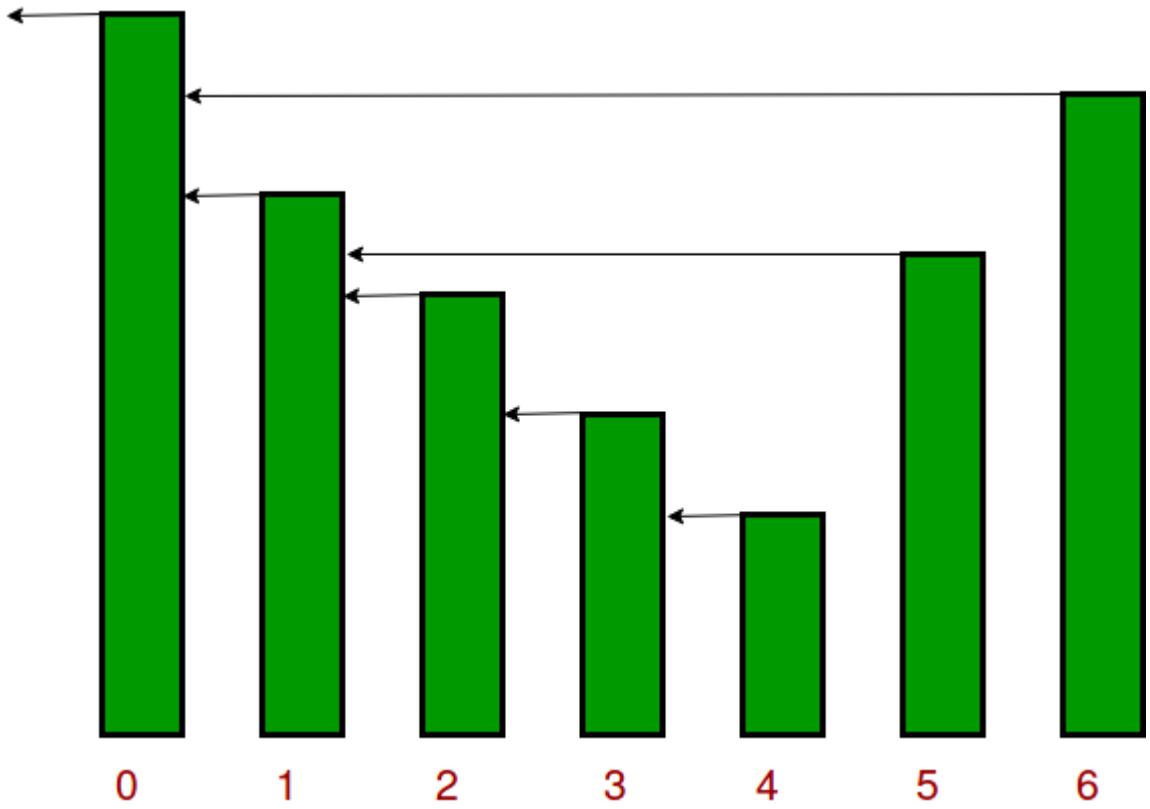
```
1 1 2 4 5 1
```

Time Complexity of the above method is O(n^2). We can calculate stock span values in O(n) time.

A Linear Time Complexity Method

We see that S[i] on day i can be easily computed if we know the closest day preceding i, such that the price is greater than on that day than the price on day i. If such a day exists, let's call it h(i), otherwise, we define h(i) = -1.

The span is now computed as S[i] = i – h(i). See the following diagram.



To implement this logic, we use a stack as an abstract data type to store the days i, h(i), h(h(i)) and so on. When we go from day i-1 to i, we pop the days when the price of the stock was less than or equal to price[i] and then push the value of day i back into the stack.

Following is the implementation of this method.

```
// C++ linear time solution for stock span problem
#include <iostream>
#include <stack>
using namespace std;

// A stack based efficient method to calculate
```

```
// stock span values
void calculateSpan(int price[], int n, int S[])
{
    // Create a stack and push index of first
    // element to it
    stack<int> st;
    st.push(0);

    // Span value of first element is always 1
    S[0] = 1;

    // Calculate span values for rest of the elements
    for (int i = 1; i < n; i++) {
        // Pop elements from stack while stack is not
        // empty and top of stack is smaller than
        // price[i]
        while (!st.empty() && price[st.top()] <= price[i])
            st.pop();

        // If stack becomes empty, then price[i] is
        // greater than all elements on left of it,
        // i.e., price[0], price[1], ..price[i-1].  Else
        // price[i] is greater than elements after
        // top of stack
        S[i] = (st.empty()) ? (i + 1) : (i - st.top());

        // Push this element to stack
        st.push(i);
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
    int price[] = { 10, 4, 5, 90, 120, 80 };
    int n = sizeof(price) / sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);
```

```
    return 0;
}
```

Output:

```
1 1 2 4 5 1
```

Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of array is added and removed from stack at most once. So there are total 2n operations at most. Assuming that a stack operation takes O(1) time, we can say that the time complexity is O(n).

Auxiliary Space: O(n) in worst case when all elements are sorted in decreasing order.

Another approach: (without using stack)

```
// C++ program for a linear time solution for stock
// span problem without using stack
#include <iostream>
#include <stack>
using namespace std;

// An efficient method to calculate stock span values
// implementing the same idea without using stack
void calculateSpan(int A[], int n, int ans[])
{
    // Span value of first element is always 1
    ans[0] = 1;

    // Calculate span values for rest of the elements
    for (int i = 1; i < n; i++) {
        int counter = 1;
        while ((i - counter) >= 0 && A[i] > A[i - counter]) {
            counter += ans[i - counter];
        }
        ans[i] = counter;
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
    int price[] = { 10, 4, 5, 90, 120, 80 };
    int n = sizeof(price) / sizeof(price[0]);
    int S[n];
```

```
    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}
```

Output:

```
1 1 2 4 5 1
```

10. Minimum number of bracket reversals needed to make an expression balanced

Given an expression with only ‘}’ and ‘{‘. The expression may not be balanced. Find minimum number of bracket reversals to make the expression balanced.

Examples:

```
Input:  exp = "}{"
Output: 2
We need to change '}' to '{' and '{' to
'}' so that the expression becomes balanced,
the balanced expression is '{}'

Input:  exp = "{{{"
Output: Can't be made balanced using reversals

Input:  exp = "{{{{"
Output: 2

Input:  exp = "{{{{}}"
Output: 1

Input:  exp = "}{{}}{{{"
Output: 3
```

One simple observation is, the string can be balanced only if total number of brackets is even (there must be equal no of ‘{‘ and ‘}’)

A **Naive Solution** is to consider every bracket and recursively count number of reversals by taking two cases (i) keeping the bracket as it is (ii) reversing the bracket. If we get a balanced expression, we update result if number of steps followed for reaching here is smaller than the minimum so far. Time complexity of this solution is $O(2^n)$.

An **Efficient Solution** can solve this problem in O(n) time. The idea is to first remove all balanced part of expression. For example, convert “**}{{}}**{{{” to “}{{{” by removing highlighted part. If we take a closer look, we can notice that, after removing balanced part, we always end up with an expression of the form }}…}{{…{, an expression that contains 0 or more number of closing brackets followed by 0 or more numbers of opening brackets.

How many minimum reversals are required for an expression of the form “}}..}{{..{” ?. Let m be the total number of closing brackets and n be the number of opening brackets. We need ⌈m/2⌉ + ⌈n/2⌉ reversals. For example }}}}{{ requires 2+1 reversals.

Below is implementation of above idea.

```
// C++ program to find minimum number of
// reversals required to balance an expression
#include<bits/stdc++.h>
```

```
using namespace std;


// Returns count of minimum reversals for making
// expr balanced. Returns -1 if expr cannot be
// balanced.
int countMinReversals(string expr)
{
    int len = expr.length();

    // length of expression must be even to make
    // it balanced by using reversals.
    if (len%2)
       return -1;

    // After this loop, stack contains unbalanced
    // part of expression, i.e., expression of the
    // form "}}..}{{..{"
    stack<char> s;
    for (int i=0; i<len; i++)
    {
        if (expr[i]=='}' && !s.empty())
        {
            if (s.top()=='{')
                s.pop();
            else
                s.push(expr[i]);
        }
        else
            s.push(expr[i]);
    }

    // Length of the reduced expression
    // red_len = (m+n)
    int red_len = s.size();

    // count opening brackets at the end of
    // stack
    int n = 0;
    while (!s.empty() && s.top() == '{')
    {
        s.pop();
        n++;
    }

    // return ceil(m/2) + ceil(n/2) which is
    // actually equal to (m+n)/2 + n%2 when
    // m+n is even.
    return (red_len/2 + n%2);
}
```

```
// Driver program to test above function
int main()
{
   string expr = "}}{{";
   cout << countMinReversals(expr);
   return 0;
}
```

Output:

```
2
```

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

11. Find if an expression has duplicate parenthesis or not

Given a balanced expression, find if it contains duplicate parenthesis or not. A set of parenthesis are duplicate if the same subexpression is surrounded by multiple parenthesis.

Examples:

```
Below expressions have duplicate parenthesis -
((a+b)+((c+d)))
The subexpression "c+d" is surrounded by two
pairs of brackets.

(((a+(b)))+(c+d))
The subexpression "a+(b)" is surrounded by two
pairs of brackets.

(((a+(b))+c+d))
The whole expression is surrounded by two
pairs of brackets.

Below expressions don't have any duplicate parenthesis -
((a+b)+(c+d))
No subsexpression is surrounded by duplicate
brackets.

((a+(b))+(c+d))
No subsexpression is surrounded by duplicate
brackets.
```

It may be assumed that the given expression is valid and there are not any white spaces present.

The idea is to use stack. Iterate through the given expression and for each character in the expression, if the character is a open parenthesis ‘(‘ or any of the operators or operands, push it to the top of the stack. If the character is close parenthesis ‘)’, then pop characters from the stack till matching open parenthesis ‘(‘ is found and a counter is used, whose value is incremented for every character encountered till the opening parenthesis ‘(‘ is found. If the number of characters encountered between the opening and closing parenthesis pair, which is equal to the value of the counter, is less than 1, then a pair of duplicate parenthesis is found else there is no occurrence of redundant parenthesis pairs. For example, (((a+b))+c) has duplicate brackets around “a+b”. When the second “)” after a+b is encountered, the stack contains “((“. Since the top of stack is a opening bracket, it can be concluded that there are duplicate brackets.

Below is the implementation of above idea :

```
// C++ program to find duplicate parenthesis in a
// balanced expression
#include <bits/stdc++.h>
using namespace std;

// Function to find duplicate parenthesis in a
// balanced expression
bool findDuplicateparenthesis(string str)
{
```

```
    // create a stack of characters
    stack<char> Stack;

    // Iterate through the given expression
    for (char ch : str)
    {
        // if current character is close parenthesis ')'
        if (ch == ')')
        {
            // pop character from the stack
            char top = Stack.top();
            Stack.pop();

            // stores the number of characters between a
            // closing and opening parenthesis
            // if this count is less than or equal to 1
            // then the brackets are redundant else not
            int elementsInside = 0;
            while (top != '(')
            {
                elementsInside++;
                top = Stack.top();
                Stack.pop();
            }
            if(elementsInside < 1) {
                return 1;
            }
        }

        // push open parenthesis '(', operators and
        // operands to stack
        else
            Stack.push(ch);
    }

    // No duplicates found
    return false;
}


// Driver code
int main()
{
    // input balanced expression
    string str = "(((a+(b))+(c+d)))";

    if (findDuplicateparenthesis(str))
        cout << "Duplicate Found ";
    else
        cout << "No Duplicates Found ";
```

```
    return 0;
}
```

Output:

```
Duplicate Found
```

Time complexity of above solution is O(n).
Auxiliary space used by the program is O(n)

12. The Celebrity Problem

*In a party of N people, only one person is known to everyone. Such a person **may be present** in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like "**does A know B?** ". Find the stranger (celebrity) in minimum number of questions.*

We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function *HaveAcquaintance(A, B)* which returns *true* if A knows B, *false* otherwise. How can we solve the problem.

We measure the complexity in terms of calls made to *HaveAcquaintance().*

Method 1 (Graph)

We can 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]. We have N_{C2} pairs. If celebrity is present in the party, we will have one sink node in the graph with outdegree of zero, and indegree of N-1. We can find the sink node in (N) time, but the overall complexity is $O(N^2)$ as we need to construct the graph first.

Method 2 (Recursion)

We can decompose the problem into combination of smaller instances. Say, if we know celebrity of N-1 persons, can we extend the solution to N? We have two possibilities, Celebrity(N-1) may know N, or N already knew Celebrity(N-1). In the former case, N will be celebrity if N doesn't know anyone else. In the later case we need to check that Celebrity(N-1) doesn't know N.

Solve the problem of smaller instance during divide step. On the way back, we find the celebrity (if present) from the smaller instance. During combine stage, check whether the returned celebrity is known to everyone and he doesn't know anyone. The recurrence of the recursive decomposition is,

$T(N) = T(N-1) + O(N)$

$T(N) = O(N^2)$. You may try writing pseudo code to check your recursion skills.

Method 3 (Using Stack)

The graph construction takes $O(N^2)$ time, it is similar to brute force search. In case of recursion, we reduce the problem instance by not more than one, and also combine step may examine M-1 persons (M – instance size).

We have following observation based on elimination technique (Refer *Polya's How to Solve It* book).

- If A knows B, then A can't be celebrity. Discard A, and *B may be celebrity*.
- If A doesn't know B, then B can't be celebrity. Discard B, and *A may be celebrity*.

- Repeat above two steps till we left with only one person.
- Ensure the remained person is celebrity. (Why do we need this step?)

We can use stack to verity celebrity.

1. Push all the celebrities into a stack.
2. Pop off top two persons from the stack, discard one person based on return status of *HaveAcquaintance(A, B)*.
3. Push the remained person onto stack.
4. Repeat step 2 and 3 until only one person remains in the stack.
5. Check the remained person in stack doesn't have acquaintance with anyone else.

We will discard N elements utmost (Why?). If the celebrity is present in the party, we will call *HaveAcquaintance()* 3(N-1) times. Here is code using stack.

```
// 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;

    int C; // Celebrity

    // Push everybody to stack
```

```
for (int i = 0; i < n; i++)
    s.push(i);

// Extract top 2
int A = s.top();
s.pop();
int B = s.top();
s.pop();

// Find a potential celevrity
while (s.size() > 1)
{
    if (knows(A, B))
    {
        A = s.top();
        s.pop();
    }
    else
    {
        B = s.top();
        s.pop();
    }
}

// Potential candidate?
C = s.top();
s.pop();

// Last candidate was not
// examined, it leads one
// excess comparison (optimize)
if (knows(C, B))
    C = B;

if (knows(C, A))
    C = A;

// Check if C is actually
// a celebrity or not
for (int i = 0; i < n; i++)
{
    // If any person doesn't
    // know 'a' or 'a' 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;
}
```

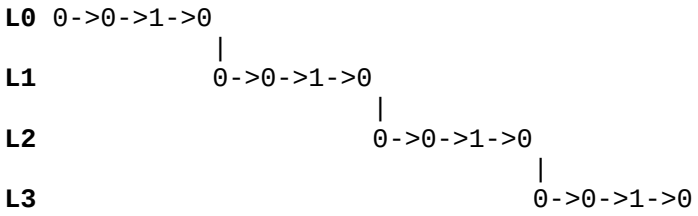
Output :

```
Celebrity ID 2
```

Complexity O(N). Total comparisons 3(N-1). Try the above code for successful MATRIX {{0, 0, 0, 1}, {0, 0, 0, 1}, {0, 0, 0, 1}, {0, 0, 0, 1}}.

Note: You may think that why do we need a new graph as we already have access to input matrix. Note that the matrix MATRIX used to help the hypothetical function *HaveAcquaintance(A, B)*, but never accessed via usual notation MATRIX[i, j]. We have access to the input only through the function *HaveAcquiantance(A, B)*. Matrix is just a way to code the solution. We can assume the cost of hypothetical function as O(1).

If still not clear, assume that the function *HaveAcquiantance* accessing information stored in a set of linked lists arranged in levels. List node will have *next* and *nextLevel* pointers. Every level will have N nodes i.e. an N element list, *next* points to next node in the current level list and the *nextLevel* pointer in last node of every list will point to head of next level list. For example the linked list representation of above matrix looks like,



The function *HaveAcquanintance(i, j)* will search in the list for *j-th* node in the *i-th* level. Out goal is to minimize calls to *HaveAcquanintance* function.

Method 4 (Using two Pointers)

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. We will find a celebrity candidate at the end of the loop. Go through each person again and check whether this is the celebrity. Below is C++ implementation.

```
// C++ program to find
// celebrity in O(n) time
// and O(1) extra space
```

```
#include <bits/stdc++.h>
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 id of celebrity
int findCelebrity(int n)
{
    // Initialize two pointers
    // as two corners
    int a = 0;
    int b = n - 1;

    // Keep moving while
    // the two pointers
    // don't become same.
    while (a < b)
    {
        if (knows(a, b))
            a++;
        else
            b--;
    }

    // Check if a is actually
    // a celebrity or not
    for (int i = 0; i < n; i++)
    {
        // If any person doesn't
        // know 'a' or 'a' doesn't
        // know any person, return -1
        if ( (i != a) &&
                (knows(a, i) ||
                !knows(i, a)) )
            return -1;
    }

    return a;
```

```
}

// 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
```

13. Find the first circular tour that visits all petrol pumps

Suppose there is a circle. There are n petrol pumps on that circle. You are 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.

Calculate the first 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). Expected time complexity is O(n). Assume for 1-litre petrol, the truck can go 1 unit of distance.

For example, let there be 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 the truck can make a circular tour is 2nd petrol pump. Output should be “start = 1” (index of 2nd petrol pump).

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.

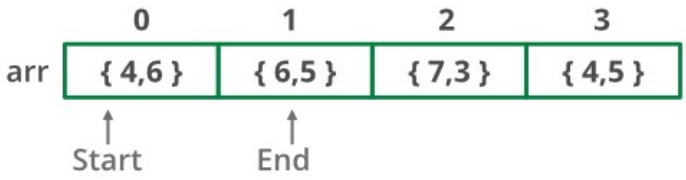
Instead of creating a separate queue, we use the given array itself as a queue. We maintain two index variables start and end that represent the rear and front of the queue.

Below image is a dry run of the above approach:

Below is the implementation of the above approach:

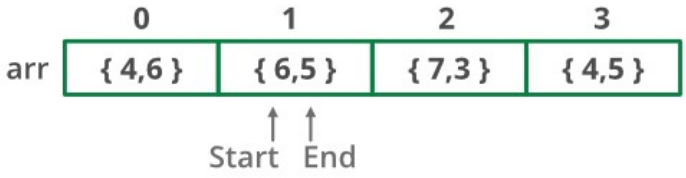
```
// C++ program to find circular tour for a truck
#include <bits/stdc++.h>
using namespace std;

// 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)
```

Initially :



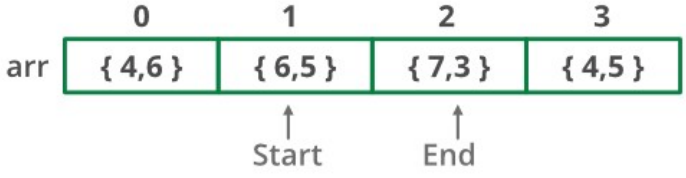
Curr - petrol = -2

Step 1:



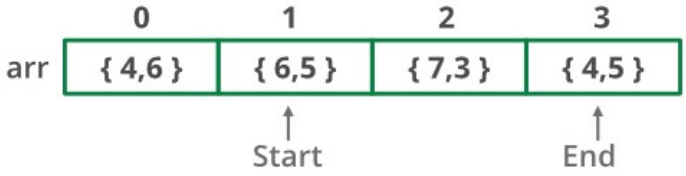
Curr - petrol = -2 + 2 = 0

Step 2:



Curr - petrol = 0 + 1 = 1

Step 3:

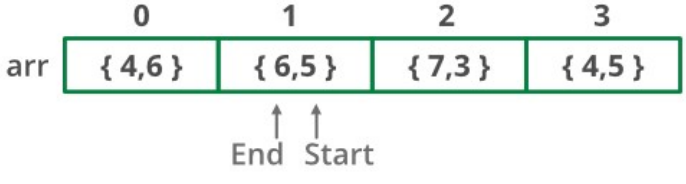


Curr - petrol = 1 + 4 = 5

Step 4:

Curr - petrol = 5 - 1 = 4

Step 5:



Curr - petrol = 4 -2 = 2

Start = end The first point from where truck can make a circular tour is 2nd petrol pump.

(petrol pump at index 1)

```
{
    // 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;
}
```

```
// This code is contributed by rathbhupendra
```
Output:

```
start = 2
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to the total number of enqueue operations. Therefore the time complexity is **O(n)**.

Auxiliary Space: O(1)