Data Structures & Algorithms for Interviews

Vignesh Narayanan

Agenda

- Time vs Space Complexity (The never ending war)
- Arrays
- Stacks
- Queues

if time permits

Linked Lists;

else

Will be covered in tutoring session 2 next week;

9 egg puzzle ?

A friend gives you 9 eggs -- 8 are 10 oz., but 1 is not.

The friend also gives you a very accurate balance and tells you to determine the odd egg out.

How would you do that ?

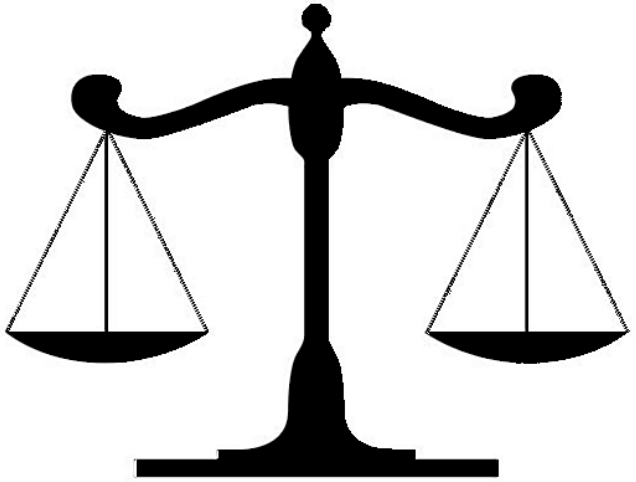
Analogies to

Time & Space

Complexities ?

Completeness vs.

Optimality ?



ABC of programming interview

Ask

Before

Coding


ARRAYS
ARRAYS EVERYWHERE
quickmeme.com

Find the repeating and the missing

Given an array where one number in it is missing and one number occurs twice. Find these two numbers.

Questions:

Size ?

Range ?

Sorted ?

Multiple duplicate elements ?

More refined problem statement

Given an **unsorted array** of **size n**. Array elements are in **range from 1 to n**. One number from set {1, 2, …n} is missing and one number occurs twice in array. Find these two numbers.

Now solutions ?

Method 1 (Use Sorting)

1) Sort the input array.
2) Traverse the array and check for missing and repeating.

Time Complexity: O(nLogn)

Method 2 (Use count array)

1) Create a temp array temp[] of size n with all initial values as 0.
2) Traverse the input array arr[], and do following for each arr[i]
……a) if(temp[arr[i]] == 0) temp[arr[i]] = 1;
……b) if(temp[arr[i]] == 1) output “arr[i]” //repeating
3) Traverse temp[] and output the array element having value as 0 (This is the missing element)

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

Can you do better ?
(PS: Interviewers favorite sentence!)

While traversing the array, use absolute value of every element as index and make the value at this index as negative to mark it visited.

If something is already marked negative then this is the repeating element.

To find missing, traverse the array again and look for a positive value.

```
void printTwoElements(int arr[], int size)
{
    int i;
    printf("\n The repeating element is");

    for(i = 0; i < size; i++)
    {
        if(arr[abs(arr[i])-1] > 0)
            arr[abs(arr[i])-1] = -arr[abs(arr[i])-1];
        else
            printf(" %d ", abs(arr[i]));
    }

    printf("\nand the missing element is ");
    for(i=0; i<size; i++)
    {
        if(arr[i]>0)
            printf("%d",i+1);
    }
}
```

Find the maximum repeating number

Given an array of size *n*, the array contains numbers in range from 0 to *k-1* where *k* is a positive integer and *k* <= *n.* Find the maximum repeating number in this array.

The **naive approach:**

Run two loops, the outer loop picks an element one by one, the inner loop counts number of occurrences of the picked element.

Finally return the element with maximum count.

Time complexity of this approach is *O(n^2)*.

A **better approach**:

Create a count array of size k and initialize all elements of *count[]* as 0.

Iterate through all elements of input array, and for every element *arr[i]*, increment *count[arr[i]]*.

Finally, iterate through *count[]* and return the index with maximum value.

This approach takes O(n) time, but requires O(k) space.

Can we do better ?

Iterate though input array *arr[]*, for every element *arr[i]*, increment *arr[arr[i]%k]* by *k*

Find the maximum value in the modified array.

Index of the maximum value is the maximum repeating element.

If we want to get the original array back, we can iterate through the array one more time and do *arr[i] = arr[i] % k* where *i* varies from 0 to *n-1*.

```
// Returns maximum repeating element in arr[0..n-1].
// The array elements are in range from 0 to k-1
int maxRepeating(int* arr, int n, int k)
{
    // Iterate though input array, for every element
    // arr[i], increment arr[arr[i]%k] by k
    for (int i = 0; i< n; i++)
        arr[arr[i]%k] += k;

    // Find index of the maximum repeating element
    int max = arr[0], result = 0;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
            result = i;
        }
    }

    /* Uncomment this code to get the original array back
       for (int i = 0; i< n; i++)
          arr[i] = arr[i]%k; */

    // Return index of the maximum element
    return result;
}
```

Similar problems

- **Find the Missing Number:** You are given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.
- **Find the Number Occurring Odd Number of Times:** Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in O(n) time & constant space.

XOR !

Find the smallest missing number

Given a **sorted** array of n integers where each integer is in the range from 0 to m-1 and m > n. Find the smallest number that is missing from the array.

Example:

Input: {0, 1, 2, 6, 9}

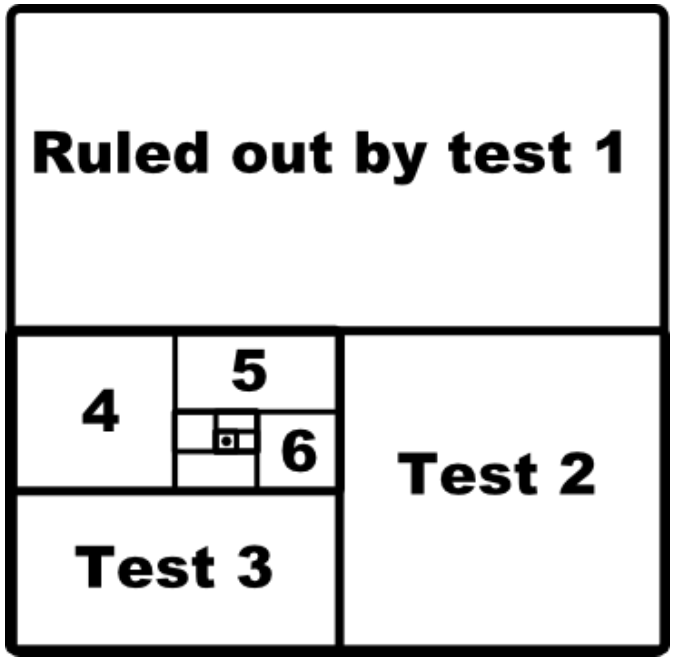
n = 5

m = 10

Output: 3

Newton's law of programming interview !

If you can find a linear time O(n) algorithm by yourself during an interview, then there exists (mostly always) a logarithmic time O(log n) algorithm that the interviewer expects from you !



Binary Search !

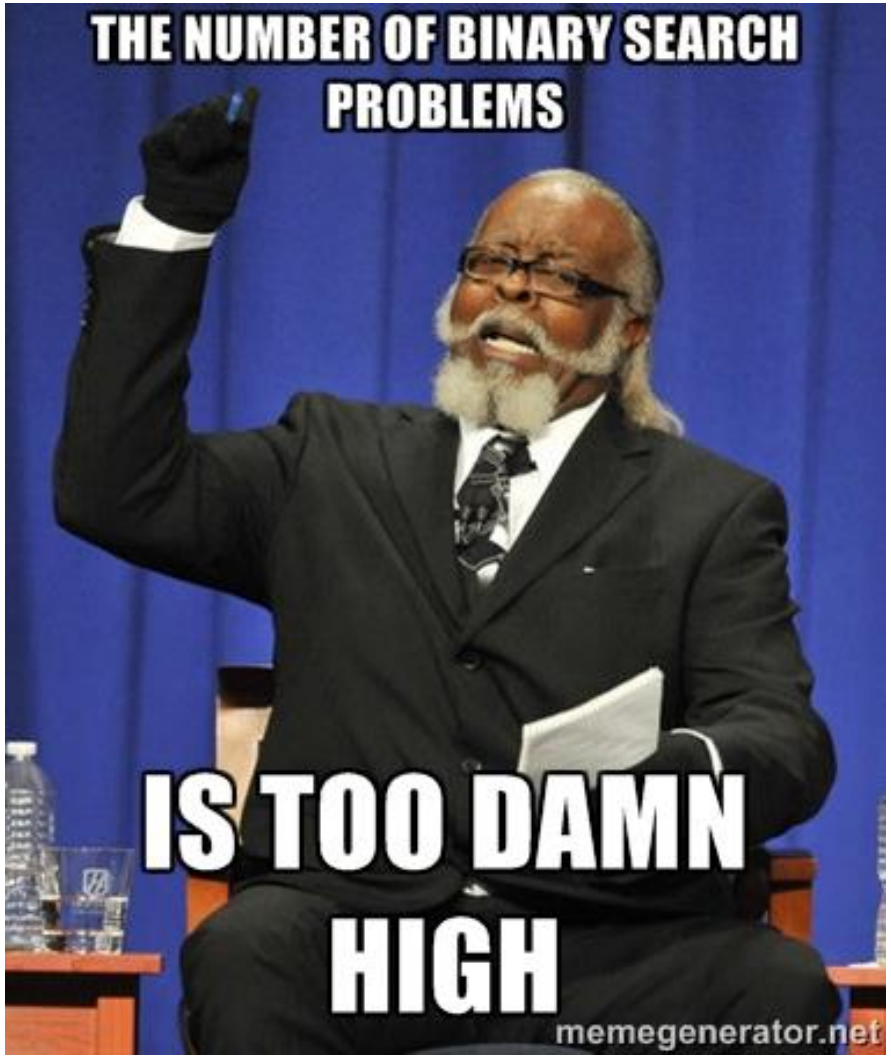
1) If the first element is not same as its index then return first index
…2) Else get the middle index say mid
……a) If arr[mid] greater than mid then the required element lies in left half.
……b) Else the required element lies in right half.

```
int findFirstMissing(int array[], int start, int end) {

    if(start  > end)
      return end + 1;

    if (start != array[start])
      return start;

    int mid = (start + end) / 2;

    if (array[mid] > mid)
      return findFirstMissing(array, start, mid);
    else
      return findFirstMissing(array, mid + 1, end);
}
```

Similar Problems

- **Count the number of occurrences in a sorted array:** Given a sorted array arr[] and a number x, write a function that counts the occurrences of x in arr[].
- **Find the maximum element in an array which is first increasing and then decreasing:** Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.
- **Search an element in a sorted and rotated array:** An element in a sorted array can be found in O(log n) time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in O(log n) time.

What we learnt so far ?



Find pair that sums up to a given number

Given an array A[] of n numbers and another number x, determine whether or not there exist two elements in S whose sum is exactly x.

Naive? Run 2 loops

```
hasArrayTwoCandidates (A[], ar_size, sum)
1) Sort the array in non-decreasing order.
2) Initialize two index variables to find the candidate
   elements in the sorted array.
      (a) Initialize first to the leftmost index: l = 0
      (b) Initialize second  the rightmost index:  r = ar_size-1
3) Loop while l < r.
      (a) If (A[l] + A[r] == sum)  then return 1
      (b) Else if( A[l] + A[r] <  sum )  then l++
      (c) Else r--
4) No candidates in whole array - return 0
```

Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then (nlogn) in worst case. If we use Quick Sort then O(n^2) in worst case.

Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is O(n) for merge sort and O(1) for Heap Sort.

```
1) Initialize Binary Hash Map M[] = {0, 0, …}
2) Do following for each element A[i] in A[]
   (a)If M[x  - A[i]] is set then print the pair (A[i], x - A[i])
   (b)Set M[A[i]]
```

Similar Problems

- **Find a pair with the given difference:** Given an unsorted array and a number n, find if there exists a pair of elements in the array whose difference is n.
- **Find a triplet that sum to a given value:** Given an array and a value, find if there is a triplet in array whose sum is equal to the given value. If there is such a triplet present in array, then print the triplet and return true. Else return false. For example, if the given array is {12, 3, 4, 1, 6, 9} and given sum is 24, then there is a triplet (12, 3 and 9) present in array whose sum is 24.
- **Pythagorean Triplet in an array:** Given an array of integers, write a function that returns true if there is a triplet (a, b, c) that satisfies $a^2 + b^2 = c^2$.

Shuffle a given array

Fisher–Yates shuffle Algorithm works in O(n) time complexity. The assumption here is, we are given a function rand() that generates random number in O(1) time.

The idea is to start from the last element, swap it with a randomly selected element from the whole array (including last). Now consider the array from 0 to n-2 (size reduced by 1), and repeat the process till we hit the first element.

```
To shuffle an array a of n elements (indices 0..n-1):
  for i from n - 1 downto 1 do
       j = random integer with 0 <= j <= i
       exchange a[j] and a[i]
```

Rearrange positive and negative numbers

- An array contains both positive and negative numbers in random order. Rearrange the array elements so that positive and negative numbers are placed alternatively. Number of positive and negative numbers need not be equal. If there are more positive numbers they appear at the end of the array. If there are more negative numbers, they too appear in the end of the array.
- For example, if the input array is [-1, 2, -3, 4, 5, 6, -7, 8, 9], then the output should be [9, -7, 8, -3, 5, -1, 2, 4, 6]

- The solution is to first separate positive and negative numbers using partition process of QuickSort.
- In the partition process, consider 0 as value of pivot element so that all negative numbers are placed before positive numbers.
- Once negative and positive numbers are separated, we start from the first negative number and first positive number, and swap every alternate negative number with next positive number.

```
// The main function that rearranges elements of given array.  It puts
// positive elements at even indexes (0, 2, ..) and negative numbers at
// odd indexes (1, 3, ..).
void rearrange(int arr[], int n)
{
    // The following few lines are similar to partition process
    // of QuickSort.  The idea is to consider 0 as pivot and
    // divide the array around it.
    int i = -1;
    for (int j = 0; j < n; j++)
    {
        if (arr[j] < 0)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    // Now all positive numbers are at end and negative numbers at
    // the beginning of array. Initialize indexes for starting point
    // of positive and negative numbers to be swapped
    int pos = i+1, neg = 0;

    // Increment the negative index by 2 and positive index by 1, i.e.,
    // swap every alternate negative number with next positive number
    while (pos < n && neg < pos && arr[neg] < 0)
    {
        swap(&arr[neg], &arr[pos]);
        pos++;
        neg += 2;
    }
}
```

Rearrange positive and negative numbers (Variation)

- Given an array of positive and negative numbers, arrange them in an alternate fashion such that every positive number is followed by negative and vice-versa **maintaining (preserving) the input order of appearance**.
- Example:

  Input: arr[] = {1, 2, 3, -4, -1, 4}

  Output: arr[] = {-4, 1, -1, 2, 3, 4}

- The idea is to process array from left to right.
- While processing, find the first out of place element in the remaining unprocessed array.
- An element is out of place if it is negative and at odd index, or it is positive and at even index.
- Once we find an out of place element, we find the first element after it with opposite sign.
- We right rotate the sub array between these two elements (including these two).

Similar Problems

- **Segregate Even and Odd numbers:** Given an array A[], write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers.

  Example:
  Input = {12, 34, 45, 9, 8, 90, 3}
  Output = {12, 34, 8, 90, 45, 9, 3}

- **Segregate 0s and 1s in an array:** You are given an array of 0s and 1s in random order. Segregate 0s on left side and 1s on right side of the array. Traverse array only once.

  Example:
  Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]
  Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

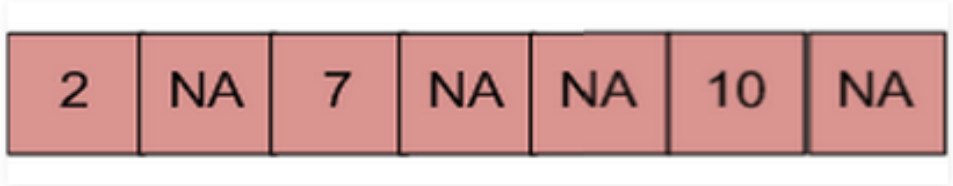
Problems that uses 2 input arrays

- Input array’s are mostly sorted
- See if you can use the merge phase of the merge sort

Merge an array of size n into another array of size m+n

There are two sorted arrays. First one is of size m+n containing only m elements. Another one is of size n and contains n elements. Merge these two arrays into the first array of size m+n such that the output is sorted.

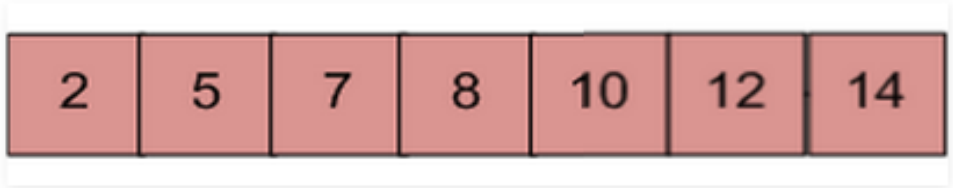
Input: array with m+n elements (mPlusN[]).



NA => Value is not filled/available in array mPlusN[]. There should be n such array blocks.

Input: array with n elements (N[]).

Output: N[] merged into mPlusN[] (Modified mPlusN[])



Algorithm:

```
Let first array be mPlusN[] and other array be N[]
1) Move m elements of mPlusN[] to end.
2) Start from nth element of mPlusN[] and 0th element of N[] and merge them
    into mPlusN[].
```

Median of two sorted arrays

There are 2 sorted arrays A and B of size n each. Write an algorithm to find the median of the array obtained after merging the above 2 arrays(i.e. array of length 2n).

- **Method 1 (Simply count while Merging)**
  Use merge procedure of merge sort. Keep track of count while comparing elements of two arrays. If count becomes n(For 2n elements), we have reached the median. Take the average of the elements at indexes n-1 and n in the merged array.

```
int getMedian(int ar1[], int ar2[], int n)
{
    int i = 0;  /* Current index of i/p array ar1[] */
    int j = 0; /* Current index of i/p array ar2[] */
    int count;
    int m1 = -1, m2 = -1;

    /* Since there are 2n elements, median will be average
     of elements at index n-1 and n in the array obtained after
     merging ar1 and ar2 */
    for (count = 0; count <= n; count++)
    {
        /*Below is to handle case where all elements of ar1[] are
          smaller than smallest(or first) element of ar2[]*/
        if (i == n)
        {
            m1 = m2;
            m2 = ar2[0];
            break;
        }

        /*Below is to handle case where all elements of ar2[] are
          smaller than smallest(or first) element of ar1[]*/
        else if (j == n)
        {
            m1 = m2;
            m2 = ar1[0];
            break;
        }

        if (ar1[i] < ar2[j])
        {
            m1 = m2;  /* Store the prev median */
            m2 = ar1[i];
            i++;
        }
        else
        {
            m1 = m2;  /* Store the prev median */
            m2 = ar2[j];
            j++;
        }
    }

    return (m1 + m2)/2;
}
```

Method 2 (By comparing the medians of two arrays)

Algorithm:

```
1) Calculate the medians m1 and m2 of the input arrays ar1[]
   and ar2[] respectively.
2) If m1 and m2 both are equal then we are done.
     return m1 (or m2)
3) If m1 is greater than m2, then median is present in one
   of the below two subarrays.
    a)  From first element of ar1 to m1 (ar1[0...|_n/2_|])
    b)  From m2 to last element of ar2  (ar2[|_n/2_|...n-1])
4) If m2 is greater than m1, then median is present in one
   of the below two subarrays.
   a)  From m1 to last element of ar1  (ar1[|_n/2_|...n-1])
   b)  From first element of ar2 to m2 (ar2[0...|_n/2_|])
5) Repeat the above process until size of both the subarrays
   becomes 2.
6) If size of the two arrays is 2 then use below formula to get
  the median.
    Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2
```

Example:

```
ar1[] = {1, 12, 15, 26, 38}
ar2[] = {2, 13, 17, 30, 45}
```

For above two arrays m1 = 15 and m2 = 17

For the above ar1[] and ar2[], m1 is smaller than m2. So median is present in one of the following two subarrays.

```
[15, 26, 38] and [2, 13, 17]
```

Let us repeat the process for above two subarrays:

```
 m1 = 26 m2 = 13.
```

m1 is greater than m2. So the subarrays become

```
  [15, 26] and [13, 17]
Now size is 2, so median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2
                         = (max(15, 13) + min(26, 17))/2
                         = (15 + 17)/2
                         = 16
```

Method 3 (By doing binary search for the median):

```
1) Get the middle element of ar1[] using array indexes left and right.
   Let index of the middle element be i.
2) Calculate the corresponding index j of ar2[]
     j = n - i - 1
3) If ar1[i] >= ar2[j] and ar1[i] <= ar2[j+1] then ar1[i] and ar2[j]
   are the middle elements.
     return average of ar2[j] and ar1[i]
4) If ar1[i] is greater than both ar2[j] and ar2[j+1] then
     do binary search in left half  (i.e., arr[left ... i-1])
5) If ar1[i] is smaller than both ar2[j] and ar2[j+1] then
     do binary search in right half (i.e., arr[i+1....right])
6) If you reach at any corner of ar1[] then do binary search in ar2[]
```

- Example:

  ar1[] = {1, 5, 7, 10, 13}

  ar2[] = {11, 15, 23, 30, 45}
- Middle element of ar1[] is 7.
- Let us compare 7 with 23 and 30, since 7 smaller than both 23 and 30, move to right in ar1[].
- Do binary search in {10, 13}, this step will pick 10.
- Now compare 10 with 15 and 23.
- Since 10 is smaller than both 15 and 23, again move to right.
- Only 13 is there in right side now.
- Since 13 is greater than 11 and smaller than 15, terminate here.
- We have got the median as 12.

Union and Intersection of 2 sorted arrays

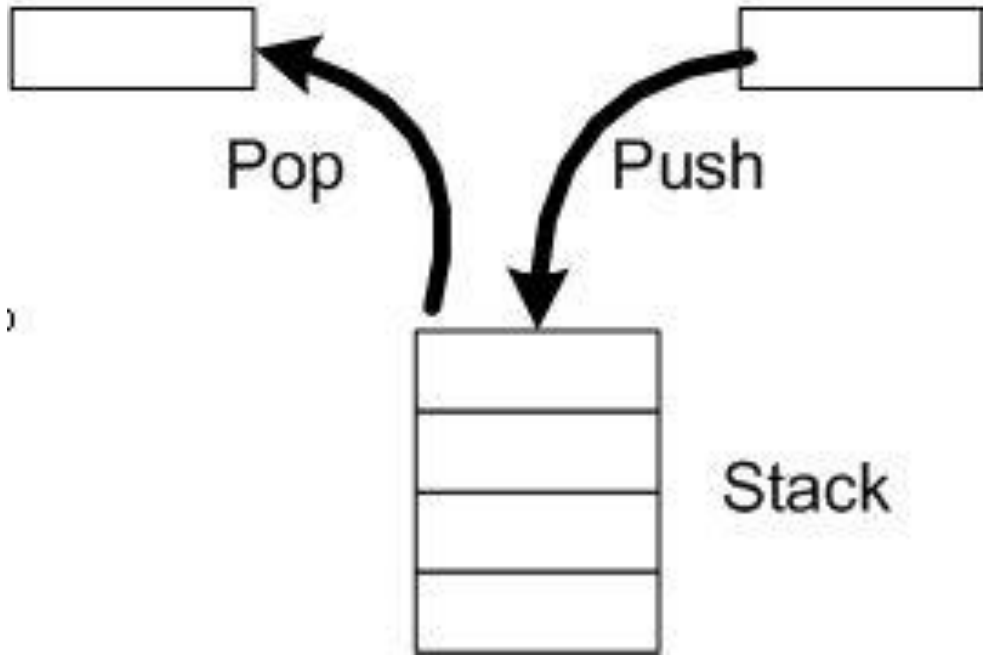
A = {1,2,3,4,5}

B = {2,5,6,7}

Union : 1,2,3,4,5,6,7

Intersection: 2,5

Stacks



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.

1) Push the first element to stack.

2) Pick rest of the elements one by one and follow following steps in loop.

....a) Mark the current element as *next*.

....b) If stack is not empty, then pop an element from stack and compare it with *next*.

....c) If next is greater than the popped element, then *next* is the next greater element for the popped element.

....d) Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements

....g) If *next* is smaller than the popped element, then push the popped element back.

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

Implement Stack using Queues

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.

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′.

```
push(s, x) // x is the element to be pushed and s is stack
  1) Enqueue x to q2
  2) One by one dequeue everything from q1 and enqueue to q2.
  3) Swap the names of q1 and q2
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

pop(s)
  1) Dequeue an item from q1 and return it.
```

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.

```
push(s,  x)
  1) Enqueue x to q1 (assuming size of q1 is unlimited).

pop(s)
  1) One by one dequeue everything except the last element from q1 and enqueue to q2.
  2) Dequeue the last item of q1, the dequeued item is result, store it.
  3) Swap the names of q1 and q2
  4) Return the item stored in step 2.
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.
```

Design and Implement Special Stack Data Structure

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).

Solution: Use two stacks: one to store actual stack elements and 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 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.

The 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

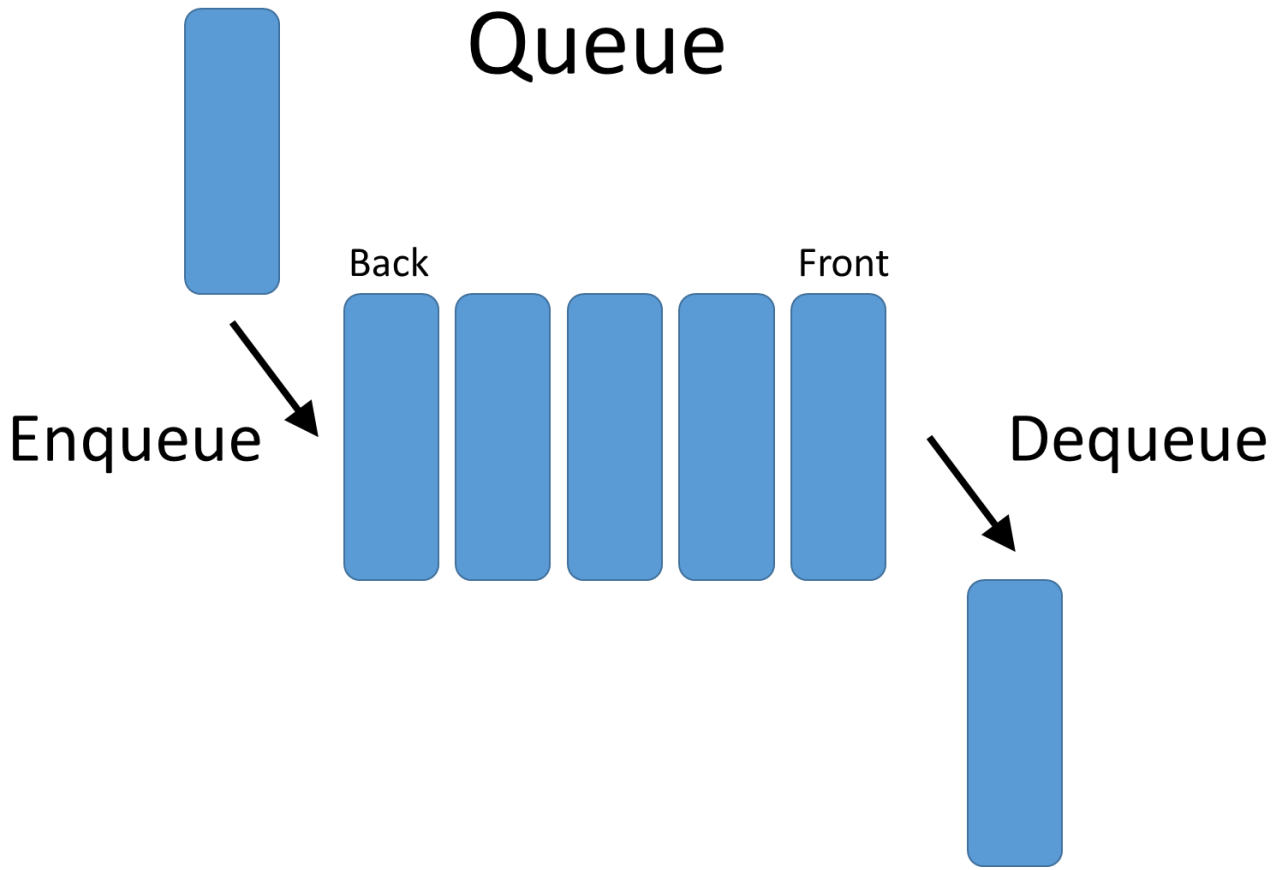
1) Return the top element of auxiliary stack.

We can see that **all above operations are O(1)**.

Similar Problem

- **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.

Queues



Implement Queue using Stacks

We are given a Stack data structure that supports standard operations like push() and pop(). We need to implement a Queue data structure using only instances of Queue.

Method 1 (By making enQueue operation costly)

This method makes sure that newly 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)
  1) While stack1 is not empty, push everything from satck1 to stack2.
  2) Push x to stack1 (assuming size of stacks is unlimited).
  3) Push everything back to stack1.

dnQueue(q)
  1) If stack1 is empty then error
  2) Pop an item from stack1 and return it
```

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).

deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
       While stack1 is not empty, push everything from satck1 to stack2.
  3) Pop the element from stack2 and return it.
```

Maximum of all sub arrays of size k (Added a O(n) method)

- Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.
- Example:
- Input :
  arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}
  k = 3
  Output :
  3 3 4 5 5 5 6

```cpp
// A Dequeue (Double ended queue) based method for printing maixmum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int>  Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i)
    {
        // For very element, the previous smaller elements are useless so
        // remove them from Qi
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();  // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while ( (!Qi.empty()) && Qi.front() <= i - k)
            Qi.pop_front();  // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

         // Add current element at the rear of Qi
        Qi.push_back(i);
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}
```