

Leet code, making dynamic programming simple:

<https://leetcode.com/discuss/study-guide/1490172/dynamic-programming-is-simple>

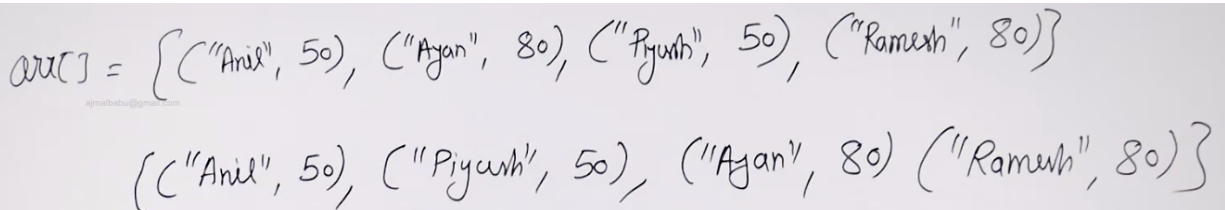
<https://www.educative.io/courses/grokking-the-coding-interview/xl0ElGxR6Bq>

Log in with ajmalbabu@gmail.com

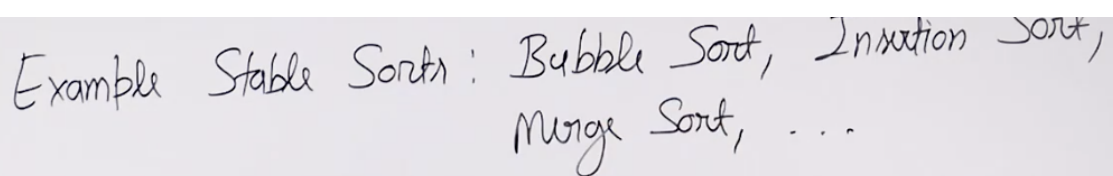
Course name: <https://practice.geeksforgeeks.org/batch/cip-1#>

Sorting:

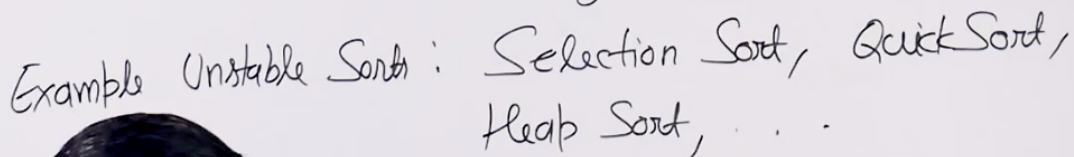
Stable sorting: Ensure that people who have the same marks appear in the same order in the original array as below. Let us say we are sorting by marks. Stability is important when there are multiple fields in the original array, but not applicable if all the elements are integers in the original array. Unstable can produce different output from the original



Handwritten example of stable sorting. The initial array is $arr[] = \{ ("Anil", 50), ("Ayan", 80), ("Piyush", 50), ("Ramesh", 80) \}$. The sorted array is $\{ ("Anil", 50), ("Piyush", 50), ("Ayan", 80), ("Ramesh", 80) \}$. The elements with the same mark (50) maintain their relative order from the original array.



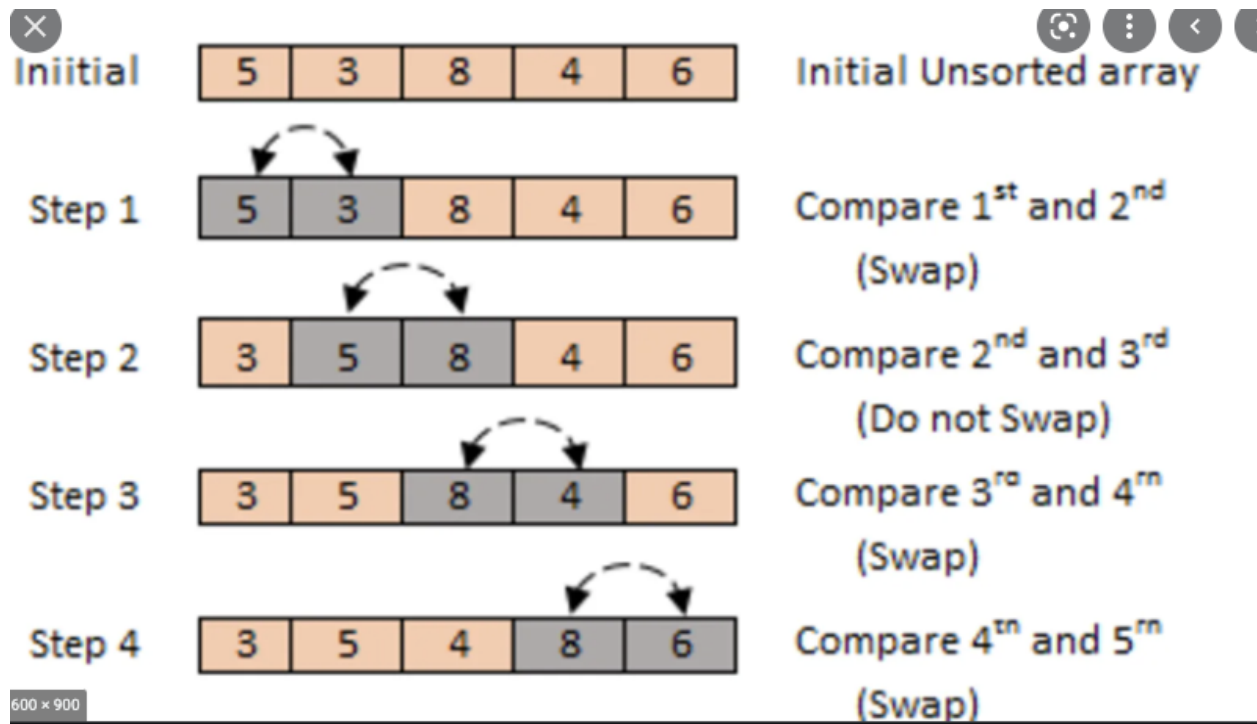
Example Stable Sorts: Bubble Sort, Insertion Sort, Merge Sort, ...



Example Unstable Sorts: Selection Sort, Quick Sort, Heap Sort, ...

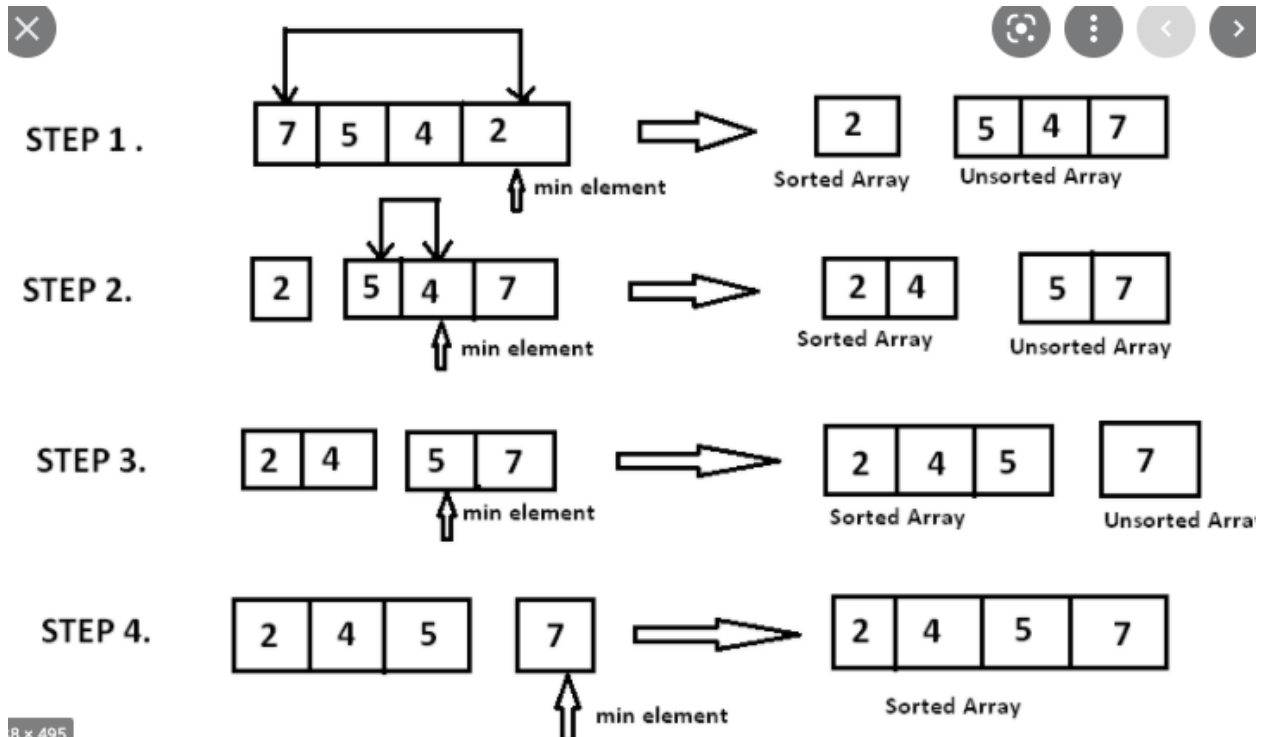
1. Bubble Sort

$O(N^2)$



```
for (int i = 0; i < elements.length; i++) {  
    for (int j = 0; j < elements.length - 1 - i; j++) {  
        if (elements[j] > elements[j + 1]) {  
            int temp = elements[j];  
            elements[j] = elements[j + 1];  
            elements[j + 1] = temp;  
        }  
    }  
}
```

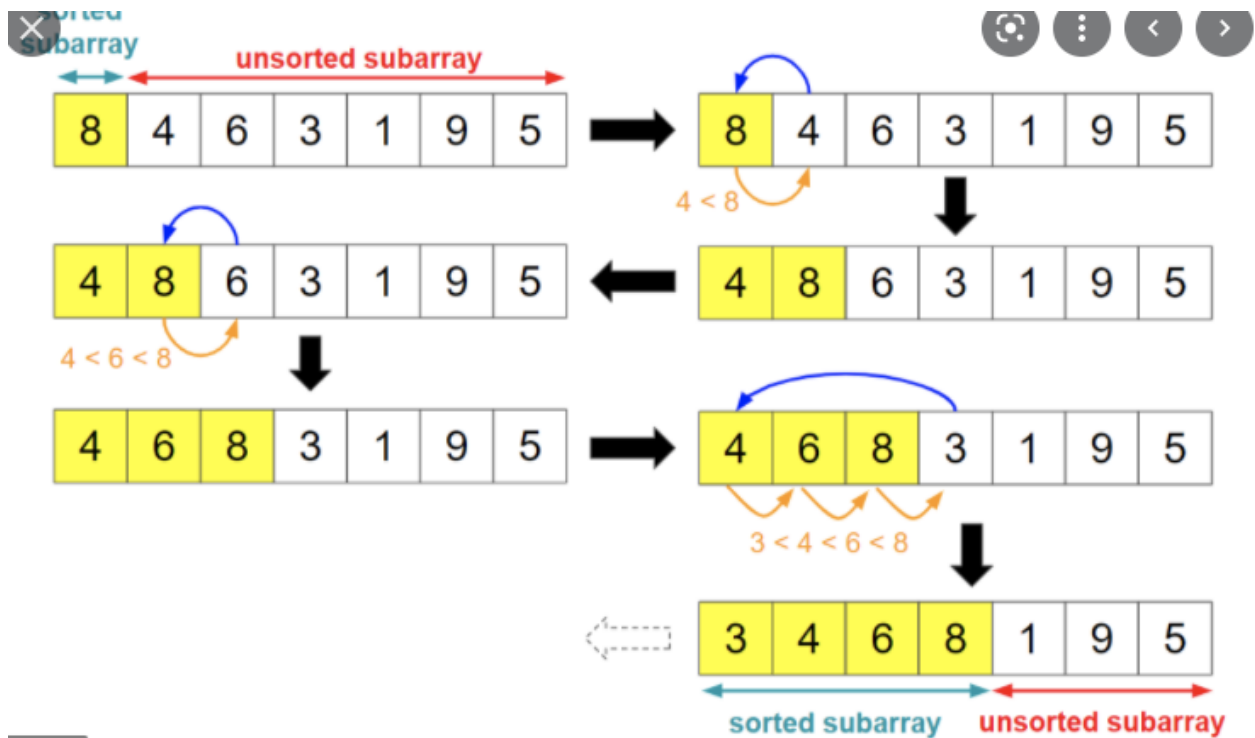
2. Selection Sort $O(N^2)$



8 x 495

```
for (int i = 0; i < elements.length - 1; i++) {
    int lowestPos = i;
    for (int j = i + 1; j < elements.length; j++) {
        if (elements[j] < elements[lowestPos]) {
            lowestPos = j;
        }
    }
    int temp = elements[i];
    elements[i] = elements[lowestPos];
    elements[lowestPos] = temp;
}
```

3. Insertion sort $O(N^2)$



```
for (int i = 1; i < arr.length; i++) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}
```

Better version from me below

```

    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = i + 1; j > 0; j--) {
            if (arr[j] < arr[j - 1]) {
                int temp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = temp;
            } else break;
        }
    }

```

4. Merge Sort $O(N \log N)$ for worst-case and space complexity = $O(N)$

```

// arr = 5 = {0,1,2,3,4}
private static void mergeSort(int[] arr, int l, int h) {
    if (l < h) {
        int m = (h - l) / 2;
        mergeSort(arr, l, h: l + m);
        mergeSort(arr, l: l + m + 1, h);
        merge(arr, l, h);
    }
}

```

```

private static void merge(int[] arr, int l, int h) {
    int m = (h - l) / 2;
    int[] a1 = new int[m + 1];
    int[] a2 = new int[h - l - m]; // or (h - l + 1 - (m + 1))

    int i = 0, a1Index = 0, a2Index = 0;

    for (i = l; i <= l + m; i++)
        a1[a1Index++] = arr[i];
    for (i = l + m + 1; i <= h; i++)
        a2[a2Index++] = arr[i];

    i = l;
    a1Index = 0;
    a2Index = 0;

    while (a1Index < a1.length && a2Index < a2.length) {
        if (a1[a1Index] == a2[a2Index]) {
            arr[i++] = a1[a1Index++];
            arr[i++] = a2[a2Index++];
        } else if (a1[a1Index] < a2[a2Index]) {
            arr[i++] = a1[a1Index++];
        } else {
            arr[i++] = a2[a2Index++];
        }
    }

    for (; a1Index < a1.length; a1Index++)
        arr[i++] = a1[a1Index];
    for (; a2Index < a2.length; a2Index++)
        arr[i++] = a2[a2Index];
}

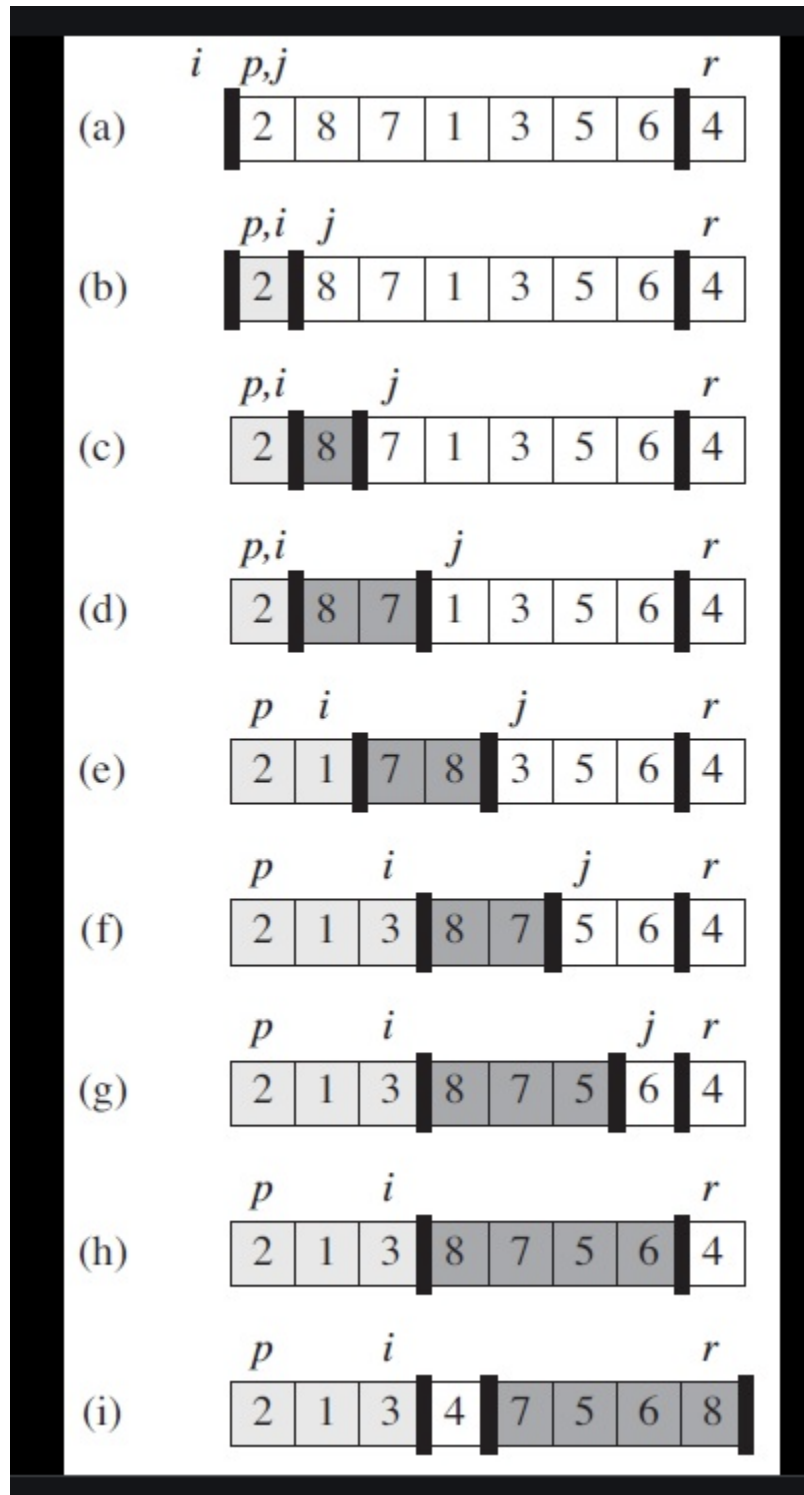
```

1. Naive partition $O(N)$ space is $O(1)$

```
int[] arr = {2, 7, 8, 3, 7}; // 2,3,7,7,8, p = 2
int p = 1;
int low = 0, high = 4;

int pivotCount = 0;
int[] arr1 = new int[high - low + 1];
int targetIndex = 0;
for (int i = low; i <= high; i++) {
    if (arr[i] <= arr[p]) {
        if (arr[i] < arr[p])
            arr1[targetIndex++] = arr[i];
        else
            pivotCount++;
    }
}
for (int i = 0; i < pivotCount; i++) {
    arr1[targetIndex++] = arr[p];
}
int lastPivot = low + targetIndex - 1;
for (int i = low; i <= high; i++) {
    if (arr[i] > arr[p])
        arr1[targetIndex++] = arr[i];
}
```

2. Lomuto partition $O(N * \log N)$ and space $O(1)$, time complexity worst-case $O(N^2)$, and the average case is $O(n \log n)$. See a more detailed explanation in Hoare below




```
private static void qSort(int[] arr) { qSort(arr, l: 0, h: arr.length - 1); }
```

```
private static void qSort(int[] arr, int l, int h) {
    if (l < h) {
        int p = lomutoPartition(arr, l, h);
        qSort(arr, l, h: p - 1);
        qSort(arr, l: p + 1, h);
    }
}
```

```
private static int lomutoPartition(int[] arr, int l, int h) {

    int pivot = arr[h]; // randomly pick pivot & swap to end, so folks know
                        // this algorithm can't manipulate how?

    int slow = l - 1; // "l-" is key
    for (int fast = l; fast < h; fast++) {
        if (arr[fast] < pivot) { // < Addl swap for last element, so lesser
                                // movement of elements
            swap(arr, ++slow, fast);
        }
    }
    swap(arr, ++slow, h);
    return slow;
}
```

fast < h is a key idea if it is **fast <= h** there will be too many movements, the only drawback for **fast < h** is one additional code for the last element (pivot) swap

HOARE

3. Hoare partition average case $O(N \log N)$, space complexity $O(1)$
4. worst case (for fully sorted array happens for every element) and time complexity will be N^2 ($n + n-1 + n-2 + n-3 \dots 1$), space complexity $O(1)$
5. average complexity when 10% tree happens is $= N + 9/10N + 9/(9/10)N + \dots 1 = N * \log(n)$ base $9/10 = N \log(n)$, space complexity $O(1)$

```

private static void qSort(int[] arr) {
    int r = new Random().nextInt(arr.length);
    swap(arr, from: 0, r);
    qSort(arr, l: 0, h: arr.length - 1);
}

private static void qSort(int[] arr, int l, int h) {
    if (l < h) {
        int p = hoarePartition(arr, l, h);
        qSort(arr, l, p); // This is important (p - 1) is bad option, that will exclude elements.
        qSort(arr, l: p + 1, h);
    }
}

private static int hoarePartition(int[] arr, int l, int h) {
    int pivot = arr[l];
    int i = l - 1; // since do while go one step back
    int j = h + 1; // since do while go one step ahead
    while (true) {
        do { // do while important, else infinite loop happens after
            // first swap, need to increment and check condition
            i++;
        } while (arr[i] < pivot); // <= is a bad option,
        // as the first element will get stuck in its position
        // and sorting will be wrong
        do {
            j--;
        } while (arr[j] > pivot);
        if (i >= j) return j; // no need to do for == as it is same value/index
        // return "j" is key, returning "i" will include higher value to left side
        // and right side of pivot is not all sorted yet, so final result won't be sorted
        // safer bet is to return "j"
        swap(arr, i, j);
    }
}

```

HeapSort ($N * \log(N)$) for all cases

```

public void sort(int[] arr) {
    int n = arr.length;
    buildheap(arr, n);
    swap(arr, from: 0, to: n - 1);

    for (int i = n - 1; i > 0; i--) {
        heapify(arr, i, 0);
        swap(arr, from: 0, to: i - 1);
    }
}

```

```

public void buildheap(int[] arr, int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
}

```

```

void heapify(int[] arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest]) largest = l;

    if (r < n && arr[r] > arr[largest]) largest = r;

    if (largest != i) {
        swap(arr, i, largest);
        heapify(arr, n, largest);
    }
}

```

```
private void swap(int[] arr, int from, int to) {
    int temp = arr[from];
    arr[from] = arr[to];
    arr[to] = temp;
}
```

Counting Sort $O(N+K)$ space $o(n+k)$

```
private static void sort(int[] arr, int k) {
    int[] count = new int[k];
    for (int i = 0; i < arr.length; i++) {
        count[arr[i]]++;
    }
    for (int i = 1; i < count.length; i++) {
        count[i] = count[i - 1] + count[i];
    }
    int[] sorted = new int[arr.length];
    for (int i = arr.length - 1; i >= 0; i--) { // starting from n - 1 to get stable sort
        sorted[count[arr[i]] - 1] = arr[i]; // -1 since arr[0] = 1, if there is a 0 value
                                           // in the input array arr[0] - 1 = 1 - 1 = 0
        count[arr[i]]--;
    }
    for (int i = 0; i < arr.length; i++) {
        arr[i] = sorted[i];
    }
}
```

Radix Sort time complexity = $O(d * (n+10))$, where d = number of digits in largest number and space complexity is $O(n)$

```

public static void sort(int[] arr) {
    int max = Integer.MIN_VALUE;
    for (int i : arr) {
        if (i > max) max = i;
    }
    for (int exp = 1; max / exp > 0; exp = exp * 10) {
        sort(arr, exp);
    }
}

```

```

private static void sort(int[] arr, int exp) {
    int[] count = new int[10];
    for (int val : arr) {
        count[(val / exp) % 10]++;
    }
    for (int i = 1; i < count.length; i++) {
        count[i] += count[i - 1];
    }
    int[] sorted = new int[arr.length];

    for (int i = arr.length - 1; i >= 0; i--) {
        sorted[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < arr.length; i++) {
        arr[i] = sorted[i];
    }
}

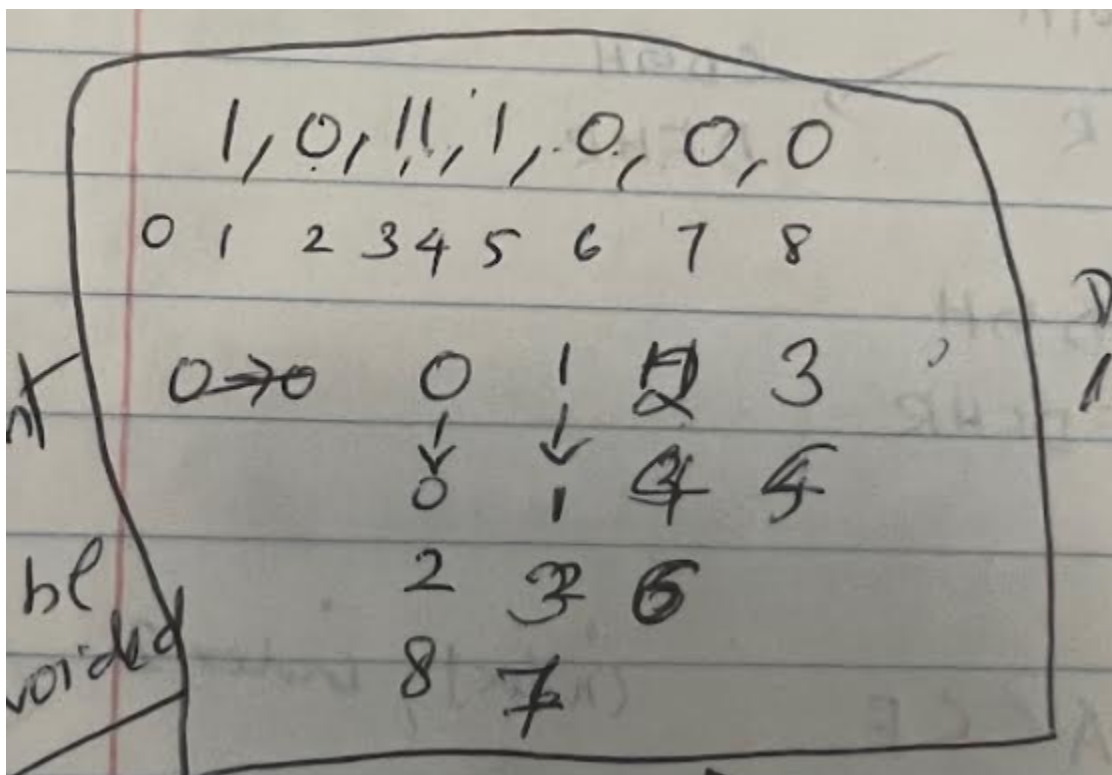
```

Bucket sort - When data is uniformly distributed from ranges e.g. 1-100, 200-300 ranges, etc. separate by bucket and sort each bucket. Space = n , time = $n \log n$

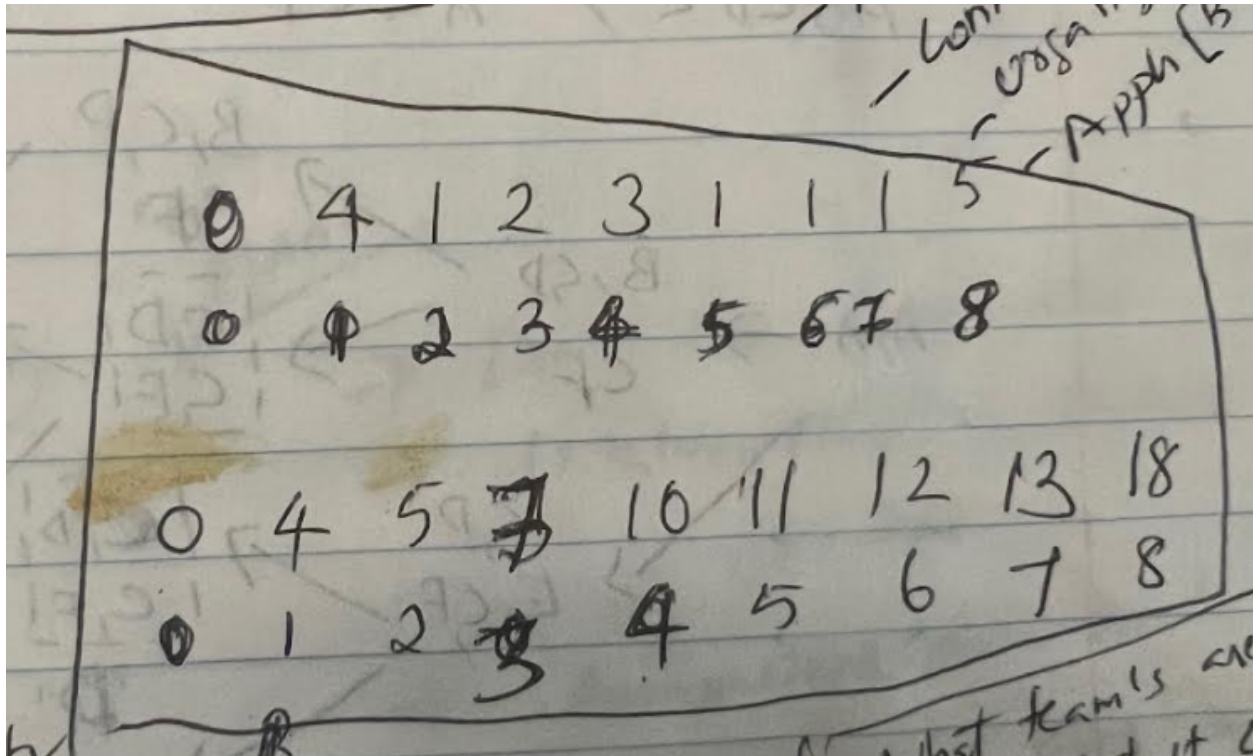
When memory writes are costly like EPROM, the best algorithm is selection sort, though it has n^2 CPU memory operations or swaps are N only

For a small array of 20 elements are small, insertion sort works better even though it is N^2

Hashing - 0s and 1s



Hashing - sum



Leet code documentation:

1. ReverseLinkedList

To reverse a linked list with two pointers

2. ComparatorTest

All sort of functional comparator

3. Container with most water [Link](#)

Two pointers from both side and move left and right

4. SplitArrayLargestSum - Hard problem and binary search did well here

5. SwappingNodeInLinkedList [Link](#)

The solution was to swap the value not to adjust the pointers

7. Reach a given score

Easy Accuracy: 76.47% Submissions: 2898 Points: 2

Consider a game where a player can score **3** or **5** or **10** points in a move. Given a total score **n**, find the number of distinct combinations to reach the given score.

Example 1:

Input:

`n = 8`

Output: 1

Explanation: when `n = 8`, `{3,5}` and `{5,3}` are the two possible permutations but these represent the same combination. Hence output is 1.

Example 2:

Input:

`n = 20`

Output: 4

Explanation: When `n = 20`, `{10,10}`, `{5,5,5,5}`, `{10,5,5}` and `{3,3,3,3,3,5}` are different possible permutations. Hence output will be 4.

Your Task:

Complete **count()** function which takes **N** as an argument and returns the **number of ways/combinations** to reach the given score.

Expected Time Complexity: $O(N)$.

Expected Auxiliary Space: $O(N)$.

Constraints:

$1 \leq n \leq 1000$


```
public static int count(int n)
{
    //Your code here
}
```

10. Count ways to reach the n'th stair

Medium Accuracy: 42.67% Submissions: 58239 Points: 4

There are **n** stairs, a person standing at the bottom wants to reach the top. The person can climb either **1 stair or 2 stairs at a time**. Count the number of ways, the person can reach the top (**order does matter**).

Example 1:

Input:

n = 4

Output: 5

Explanation:

You can reach 4th stair in 5 ways.

Way 1: Climb 2 stairs at a time.

Way 2: Climb 1 stair at a time.

Way 3: Climb 2 stairs, then 1 stair and then 1 stair.

Way 4: Climb 1 stair, then 2 stairs then 1 stair.

Way 5: Climb 1 stair, then 1 stair and then 2 stairs.

Example 2:

Input:

n = 10

Output: 89

Explanation:

There are 89 ways to reach the 10th stair.

Your Task:

Complete the function **countWays()** which takes the top stair number m as input parameters and returns the answer $\% 10^9+7$.

Expected Time Complexity : $O(n)$

Expected Auxiliary Space: $O(1)$

Constraints:

$$1 \leq n \leq 10^4$$

```
class Solution
{
    //Function to count number of ways to reach the nth stair.
    int countWays(int n)
    {
        // your code here
    }
}
```

11. Count ways to N'th Stair(Order does not matter)

Medium Accuracy: 51.45% Submissions: 29132 Points: 4

There are **N** stairs, and a person standing at the bottom wants to reach the top. The person can climb either **1 stair or 2 stairs at a time**. Count the number of ways, the person can reach the top (**order does not matter**).

Note: Order does not matter means for $n=4$ {1 2 1},{2 1 1},{1 1 2} are considered same.

Example 1:

Input:

$N = 4$

Output: 3

Explanation: You can reach 4th stair in 3 ways.

3 possible ways are:

1, 1, 1, 1

1, 1, 2

2, 2

Example 2:

Input:

N = 5

Output: 3

Explanation:

You may reach the 5th stair in 3 ways.

The 3 possible ways are:

1, 1, 1, 1, 1

1, 1, 1, 2

1, 2, 2

Your Task:

Your task is to complete the function **countWays()** which takes single argument(N) and returns the answer.

Expected Time Complexity: O(N)

Expected Auxiliary Space: O(N)

Constraints:

1 <= N <= 10⁶

```
class Solution
{
    //Function to count number of ways to reach the nth stair
    //when order does not matter.
    Long countWays(int m)
    {
        // your code here
    }
}
```

16. Unique BST's

Medium Accuracy: 44.17% Submissions: 41964 Points: 4

Given an integer. Find how many **structurally unique binary search trees** are there that stores the values from 1 to that integer (inclusive).

Example 1:

Input:

N = 2

Output: 2

Explanation: for N = 2, there are 2 unique BSTs



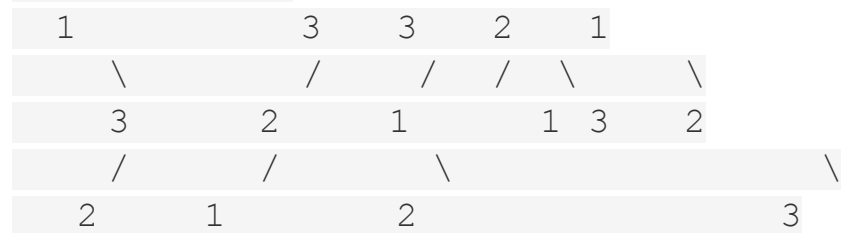
Example 2:

Input:

N = 3

Output: 5

Explanation: for N = 3, there are 5 possible BSTs



Your Task:

You don't need to read input or print anything. Your task is to complete the function **numTrees()** which takes the integer N as input and returns the total number of Binary Search Trees possible with keys [1.....N] inclusive. Since the answer can be very large, return the **answer modulo $1e9 + 7$** .

Expected Time Complexity: $O(N^2)$.

Expected Auxiliary Space: $O(N)$.

Constraints:

$1 \leq N \leq 1000$

```
class Solution
{
    //Function to return the total number of possible unique BST.
    static int numTrees(int N)
    {
        // Your code goes here

    }
}
```

18. Max sum subarray by removing at most one element

Medium Accuracy: 46.3% Submissions: 14110 Points: 4

You are given array **A** of size **n**. You need to find the maximum-sum sub-array with the condition that you are allowed to skip at most one element.

Example 1:

Input:

`n = 5`

`A[] = {1, 2, 3, -4, 5}`

Output: 11

Explanation: We can get maximum sum subarray by skipping -4.

Example 2:

Input:

`n = 8`

`A[] = {-2, -3, 4, -1, -2, 1, 5, -3}`

Output: 9

Explanation: We can get maximum sum subarray by skipping -2 as `[4, -1, 1, 5]`

sums to 9, which is the maximum achievable sum.

Your Task:

Your task is to complete the function **maxSumSubarray** that take array and size as parameters and returns the maximum sum.

Expected Time Complexity: $O(N)$.

Expected Auxiliary Space: $O(N)$.

Constraints:

$1 \leq n \leq 100$

$-10^3 \leq A_i \leq 10^3$

```
class Solution
{
    //Function to return maximum sum subarray by removing at most one element.
    public static int maxSumSubarray(int A[], int n)
    {
        //add code here.
    }
}
```

19. Longest Increasing Subsequence

Medium Accuracy: 46.69% Submissions: 73912 Points: 4

This problem is part of GFG SDE Sheet. [Click here to view more.](#)

Given an array of integers, find the **length** of the **longest (strictly) increasing subsequence** from the given array.

Example 1:

Input:

N = 16

A[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5

```
13, 3, 11, 7, 15}
```

Output: 6

Explanation: Longest increasing subsequence

0 2 6 9 13 15, which has length 6

Example 2:

Input:

N = 6

A[] = {5, 8, 3, 7, 9, 1}

Output: 3

Explanation: Longest increasing subsequence

5 7 9, with length 3

Your Task:

Complete the function **longestSubsequence()** which takes the input array and its size as input parameters and returns the **length** of the **longest increasing subsequence**.

Expected Time Complexity : $O(N \log(N))$

Expected Auxiliary Space: $O(N)$

Constraints:

$1 \leq N \leq 10^5$

$0 \leq A[i] \leq 10^6$

```
class Solution
{
    //Function to find length of longest increasing subsequence.
    static int longestSubsequence(int size, int a[])
    {
        // code here
    }
}
```

20. Longest Common Subsequence

Medium Accuracy: 49.98% Submissions: 81238 Points: 4

This problem is part of GFG SDE Sheet. [Click here](#) to view more.

Given two sequences, find the length of longest subsequence present in both of them. Both the strings are of uppercase.

Example 1:

Input:

```
A = 6, B = 6  
str1 = ABCDGH  
str2 = AEDFHR
```

Output: 3

Explanation: LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

Example 2:

Input:

```
A = 3, B = 2  
str1 = ABC  
str2 = AC
```

Output: 2

Explanation: LCS of "ABC" and "AC" is "AC" of length 2.

Your Task:

Complete the function **lcs()** which takes the length of two strings respectively and two strings as input parameters and returns the length of the longest subsequence present in both of them.

Expected Time Complexity : $O(|str1|*|str2|)$

Expected Auxiliary Space: $O(|str1|*|str2|)$

Constraints:

1<=size(str1),size(str2)<=103

class Solution

```
{
    //Function to find the length of longest common subsequence in two strings.
    static int lcs(int x, int y, String s1, String s2)
    {
        // your code here
    }
}
```