Data Structures and Algorithms

Contents

[1. Design Strategies and Sorting 5](#_Toc137141789)

[1.1. Brute Force 5](#_Toc137141790)

[1.1.1. Selection Sort 5](#_Toc137141791)

[1.1.2. Bubble Sort 6](#_Toc137141792)

[1.2. Decrease and Conquer 7](#_Toc137141793)

[1.2.1. Insertion Sort 7](#_Toc137141794)

[1.3. Divide and Conquer 7](#_Toc137141795)

[1.3.1. Merge Sort 8](#_Toc137141796)

[1.3.2. Quick Sort 9](#_Toc137141797)

[1.4. Transform and Conquer 11](#_Toc137141798)

[1.4.1. Heap Sort 11](#_Toc137141799)

[1.4.2. Counting Sort 13](#_Toc137141800)

[1.4.3. Radix Sort 13](#_Toc137141801)

[1.5. Cheat Sheet on sorting algorithms 13](#_Toc137141802)

[1.6. Problem Solving – Pre Sorting 13](#_Toc137141803)

[1.6.1. Sort an Array – Leet Code 912 13](#_Toc137141804)

[1.6.2. Two Sum – Leet Code 1 13](#_Toc137141805)

[1.6.3. Three Sum – Leet Code 15 15](#_Toc137141806)

[1.6.4. 3Sum Closest – Leet Code 16 17](#_Toc137141807)

[1.6.5. Four Sum – Leet Code 18 18](#_Toc137141808)

[1.6.6. Attend Meetings – Leet Code 252 20](#_Toc137141809)

[1.6.7. Merge Intervals – Leet Code 56 21](#_Toc137141810)

[1.7. Problem Solving – Extension of Merge Sort 22](#_Toc137141811)

[1.7.1. Inter Section of two arrays - Leet code 349 22](#_Toc137141812)

[1.7.2. Inter Section of two arrays - Leet code 350 23](#_Toc137141813)

[1.7.3. Inter Section of three arrays – Leet Code 1213 24](#_Toc137141814)

[1.7.4. Count of Smaller Numbers After Self – Leet Code 315 25](#_Toc137141815)

[1.7.5. Inversion Pairs 27](#_Toc137141816)

[1.7.6. Reverse Pairs 27](#_Toc137141817)

[1.7.7. Merge Sorted Array – Leet Code 88 27](#_Toc137141818)

[1.8. Problem Solving – Extension of Quick Sort (Quick Select) 28](#_Toc137141819)

[1.8.1. Find Kth smallest / largest element – Leet code 215 28](#_Toc137141820)

[1.8.2. K Closest Points to Origin – Leet code 973 29](#_Toc137141821)

[1.8.3. Top K Frequent Elements – Leet Code 347 29](#_Toc137141822)

[1.9. Problem Solving – Extension of Lomuto and Hoare’s partition 30](#_Toc137141823)

[1.9.1. Sort based on 2 flag colors 30](#_Toc137141824)

[1.9.2. Sort based on 3 flag colors – Dutch Flag – Leet Code 75 30](#_Toc137141825)

[1.9.3. Sort based on even odd – Leet Code 905 32](#_Toc137141826)

[1.9.4. Sort based on parity 2 – Leet Code 922 32](#_Toc137141827)

[1.10. Problem Solving – Extension of Heap 33](#_Toc137141828)

[1.10.1. Merge k sorted arrays 33](#_Toc137141829)

[1.10.2. Merge k sorted linked lists – Leet Code 23 35](#_Toc137141830)

[1.10.3. Kth largest element in stream 37](#_Toc137141831)

[1.10.4. Kth smallest element in stream 39](#_Toc137141832)

[1.10.5. Take a sequence of strings in streaming fashion and find kth longest string 40](#_Toc137141833)

[1.10.6. Sliding Window Median – Leet Code 480 40](#_Toc137141834)

[1.10.7. Find Median from Data Stream – Leet Code295 41](#_Toc137141835)

[1.10.8. Find median of two sorted array/ stream – Leet Code 4 42](#_Toc137141836)

[1.11. Coding Patterns 42](#_Toc137141837)

[1.11.1. Two Pointer 42](#_Toc137141838)

[1.11.2. K the largest or Smallest using Heap 43](#_Toc137141839)

[2. Recursion 43](#_Toc137141840)

[2.1. Recursion In General 43](#_Toc137141841)

[2.1.1. Power of two 43](#_Toc137141842)

[2.1.2. X Power N – Leet Code 50 44](#_Toc137141843)

[2.1.3. Fibanocci – Leet Code 509 44](#_Toc137141844)

[2.1.4. Maximum number of Nodes in BST – Catalan Numbers - TBD 45](#_Toc137141845)

[2.1.5. Tower of Hanoi – Geeks for Geeks 45](#_Toc137141846)

[2.2. Combinatorial Problems - Subset/Permutation of a set 46](#_Toc137141847)

[2.2.1. Binary String 46](#_Toc137141848)

[2.2.2. Permutations – Leet Code 46 47](#_Toc137141849)

[2.2.3. Permutations II - Leet Code 47 48](#_Toc137141850)

[2.2.4. Subsets – Lee Code 78 49](#_Toc137141851)

[2.2.5. Subsets II – Leet Code 90 50](#_Toc137141852)

[2.2.6. Combination Sum II – Leet Code 40 50](#_Toc137141853)

[2.2.7. Palindromic Decomposition Of A String 51](#_Toc137141854)

[3. Trees 53](#_Toc137141855)

[3.1. Binary Trees 53](#_Toc137141856)

[3.1.1. Search in a Binary Search Tree – Leet Code 700 53](#_Toc137141857)

[3.1.2. Insert into a Binary Search Tree – Leet Code 701 53](#_Toc137141858)

[3.1.3. Min/Max in BST 54](#_Toc137141859)

[3.1.4. Delete Node in a BST – Leet Code 450 54](#_Toc137141860)

[3.1.5. Iterative Postorder Traversal - Geeks for Geeks 56](#_Toc137141861)

[3.1.6. Binary Search Tree Iterator – Leet Code 173 57](#_Toc137141862)

[3.2. BFS 57](#_Toc137141863)

[3.2.1. Binary Tree Level Order Traversal – Leet Code 102 57](#_Toc137141864)

[3.2.2. N-ary Tree Level Order Traversal – Leet Code 429 58](#_Toc137141865)

[3.2.3. Binary Tree Level Order Traversal II – Leet Code 107 59](#_Toc137141866)

[3.2.4. Binary Tree Right Side View – Leet Code 199 59](#_Toc137141867)

[3.2.5. Binary Tree Zigzag Level Order Traversal – Leet Code 103 60](#_Toc137141868)

[3.2.6. Convert a Binary Tree into its Mirror Tree – Geeks for Geeks 61](#_Toc137141869)

[3.2.7. Clone Binary Tree – Leet Code 1379 61](#_Toc137141870)

[3.2.8. Populating Next Right Pointers in Each Node – Leet Code 116 63](#_Toc137141871)

[3.3. DFS – Depth first search 63](#_Toc137141872)

[3.3.1. Path Sum – Leet Code 112 64](#_Toc137141873)

[3.3.2. Path Sum II – Leet Code 113 64](#_Toc137141874)

[3.3.3. Diameter of Binary Tree – Leet Code 543 65](#_Toc137141875)

[3.3.4. Convert Sorted Array to Binary Search Tree – Leet Code 108 66](#_Toc137141876)

[3.3.5. Construct Binary Tree from Inorder and Postorder Traversal – Leet Code 106 67](#_Toc137141877)

[4. Graphs 68](#_Toc137141878)

[5. Dynamic Programming 68](#_Toc137141879)

[5.1. Top Down – Memoization 68](#_Toc137141880)

[5.1.1. Climbing Stairs – Leet Code 70 68](#_Toc137141881)

[5.1.2. N Choose K – IK 68](#_Toc137141882)

[5.1.3. Unique Paths – Leet Code 62 68](#_Toc137141883)

[5.1.4. Minimum Path Sum– Leet Code 64 69](#_Toc137141884)

[5.1.5. Min Cost Climbing Stairs – Leet Code 746 70](#_Toc137141885)

[5.1.6. Fdlsa’ 70](#_Toc137141886)

[5.1.7. Lfds’a 70](#_Toc137141887)

[5.2. Bottom Up - Tabulation 70](#_Toc137141888)

[5.2.1. Climbing Stairs – Leet Code 70 70](#_Toc137141889)

[5.2.2. N Choose K – IK 71](#_Toc137141890)

[5.2.3. Unique Paths – Leet Code 62 71](#_Toc137141891)

[5.2.4. Minimum Path Sum – Leet Code 64 72](#_Toc137141892)

[5.2.5. Min Cost Climbing Stairs – Leet Code 746 72](#_Toc137141893)

[**Pushing back 0 in the end of cost vector helps, as we can treat it as destination** 73](#_Toc137141894)

[6. Notes on library functions 73](#_Toc137141895)

[6.1. Vectors 73](#_Toc137141896)

[6.2. Queues - FIFO 73](#_Toc137141897)

[6.3. Stacks - LIFO 73](#_Toc137141898)

[6.4. Priority Queue 74](#_Toc137141899)

[6.5. Maps 74](#_Toc137141900)

[6.6. Sets 74](#_Toc137141901)

[6.7. Pairs 74](#_Toc137141902)

[6.8. Strings 74](#_Toc137141903)

# Design Strategies and Sorting

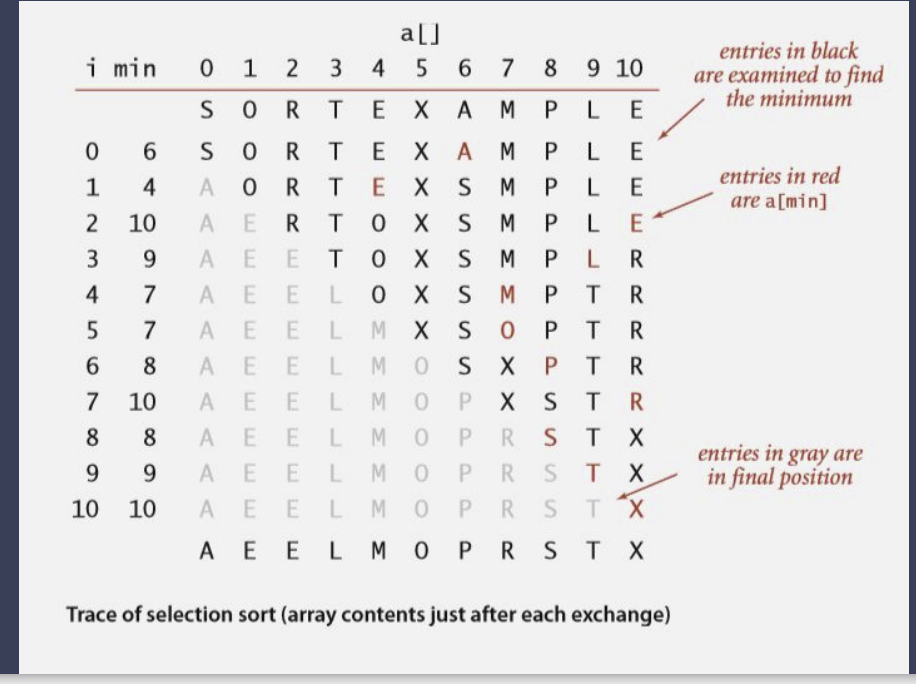
## Brute Force

In brute force approach the problem is solved by going through exhaustive operations.

For ex: in sorting we might do an exhaustive comparison and swapping in a brute force solution.

### Selection Sort

Selecting one smallest/largest element in each pass and doing it for “n” iterations. The selected element will be kept in appropriate position. The selection happens by choosing the first non-sorted element and checking if there is smaller/larger element in rest of the input.



vector<int> selectionSort(vector<int> &nums) {

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

        int minIdx = i;

        for(int j=i+1; j < nums.size();j++){

            if(nums[minIdx] > nums[j])

                minIdx = j;

        }

        swap(nums[i], nums[minIdx]);

    }

    return nums;

}

**TC**: O(n2)

**SC**: O(1)

Inline Sort

**Not Stable Sort**

### Bubble Sort

In each pass bubble up smallest/biggest element. Scan the array from right to left, compare two elements and swap them if needed to make sure the smallest/biggest element keeps moving to left. Do the operation for n times.

vector<int> bubbleSort(vector<int> &nums) {

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

        for(int j=nums.size()-1; j > i;j--){

            if(nums[j-1] > nums[j])

                swap(nums[j], nums[j-1]);

        }

    }

    return nums;

}

**TC**: O(n2)

**SC**: O(1)

Inline Sort

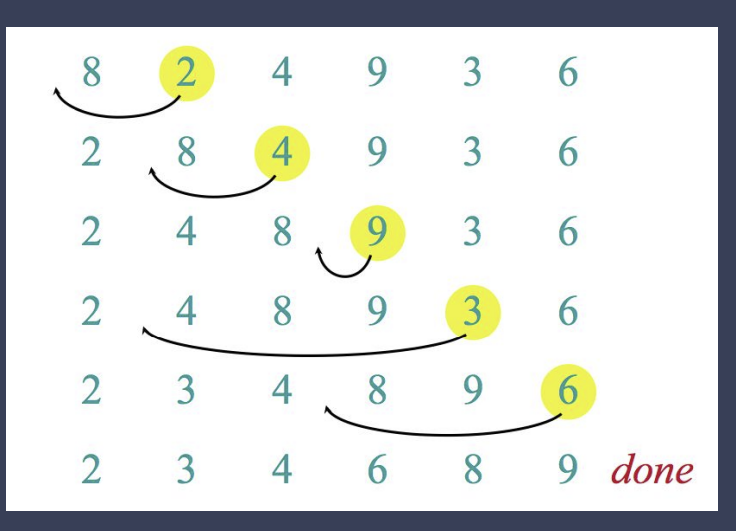
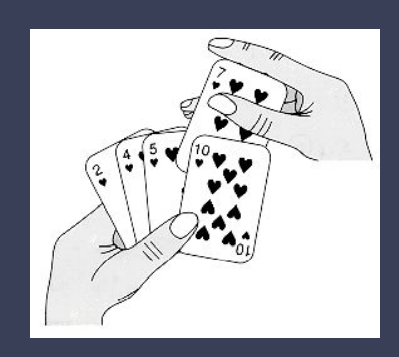
**Not Stable Sort**

## Decrease and Conquer

Decrease the problem size each iteration. Can be done in 2 ways.

* Decrease the size of problem from n to n-1. Ex: Selection and Bubble sort.
* Assume the problem of size n-1 solved, and solve nth item. Ex: Insertion sort

### Insertion Sort



vector<int> insertionSort(vector<int> &nums) {

    for(int i = 0; i < nums.size()-1; i++){

        int j = i + 1;

        int temp = nums[j];

        while (j > 0 && temp < nums[j-1]) {

            nums[j] = nums[j-1];

            j--;

        }

        nums[j] = temp;

    }

    return nums;

}

**TC**: Best Case: O(n), Avg Case: O(n2), Worst Case: O(n2)

**SC**: O(1)

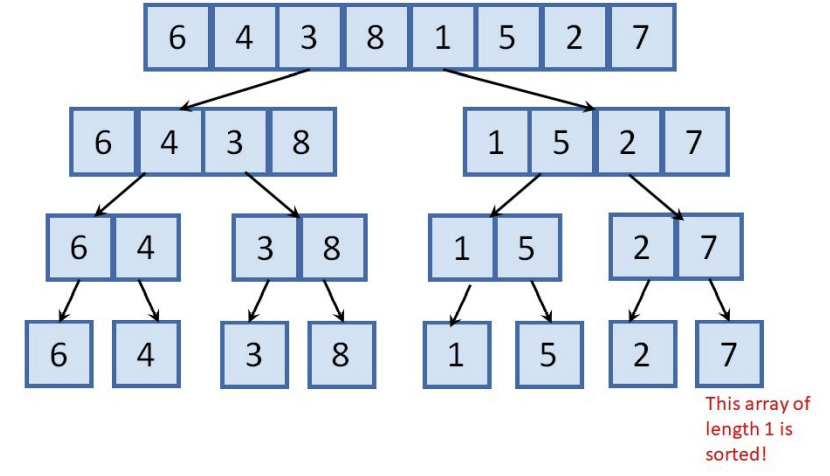
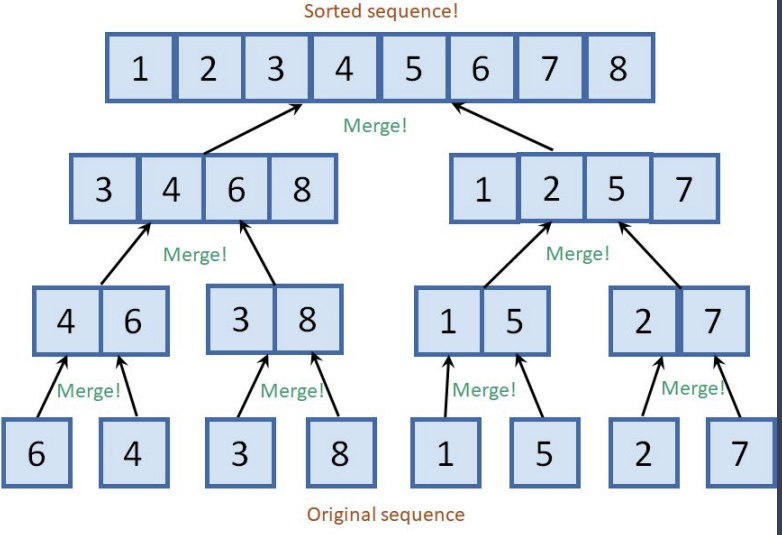
Inline Sort

Stable Sort

## Divide and Conquer

Divide the problem into 2 or more sub problems until it becomes too easy to solve, example of size 0 or 1.

### Merge Sort

In merge sort keep dividing the array until its size is 1, and start conquer/merge/solve each sub set.

For merging operation, a two-pointer solution, pointing to each subset and copying the small element to axillary array and moving the pointer helps.

void mergeHelper(vector<int> &a, int s, int e) {

    // Leaf Node, nothing to do

    if (s == e)

        return;

    // Divide

    int m = (s+e)/2;

    mergeHelper(a, s, m);

    mergeHelper(a, m+1, e);

    // Combine/Merge

    int i = s, j = m + 1;

    vector<int> aux;

    while (i <= m && j <= e) {

        if (a[i] <= a[j])

            aux.push\_back(a[i++]);

        else

            aux.push\_back(a[j++]);

    }

    // Combine left over items

    while (i <= m ) aux.push\_back(a[i++]);

    while (j <= e ) aux.push\_back(a[j++]);

    // Copy axillary array into original array

    for(i=0; i<aux.size();i++)

        a[s+i] = aux[i];

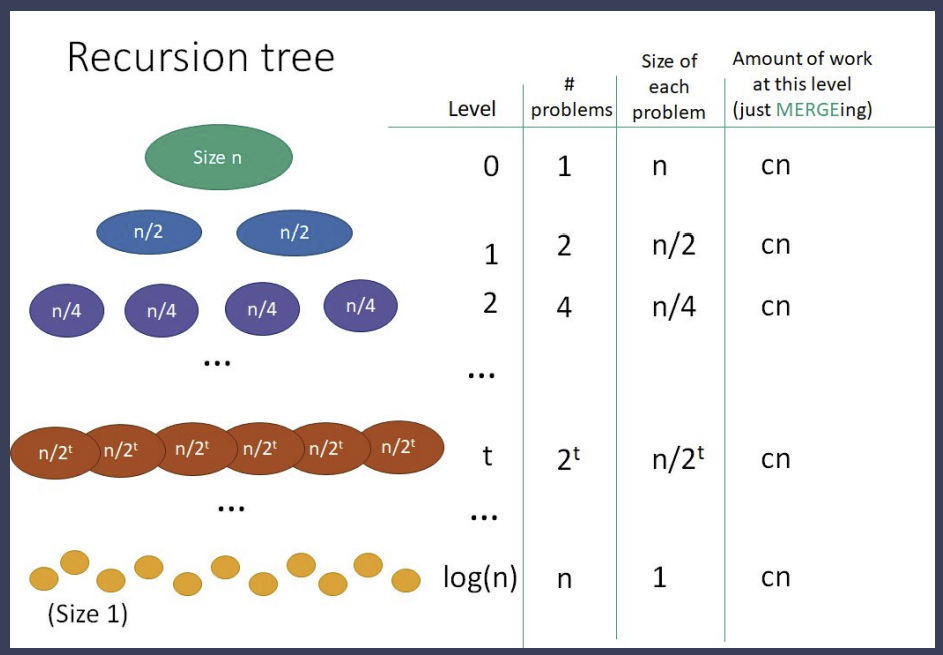
}

vector<int> mergeSort(vector<int> &nums) {

    mergeHelper(nums, 0, nums.size()-1);

    return nums;

}

****

**TC**: O(n logn)

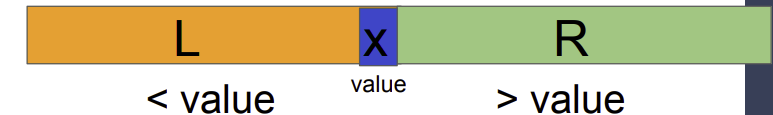
**SC**: O(n) for axillary array

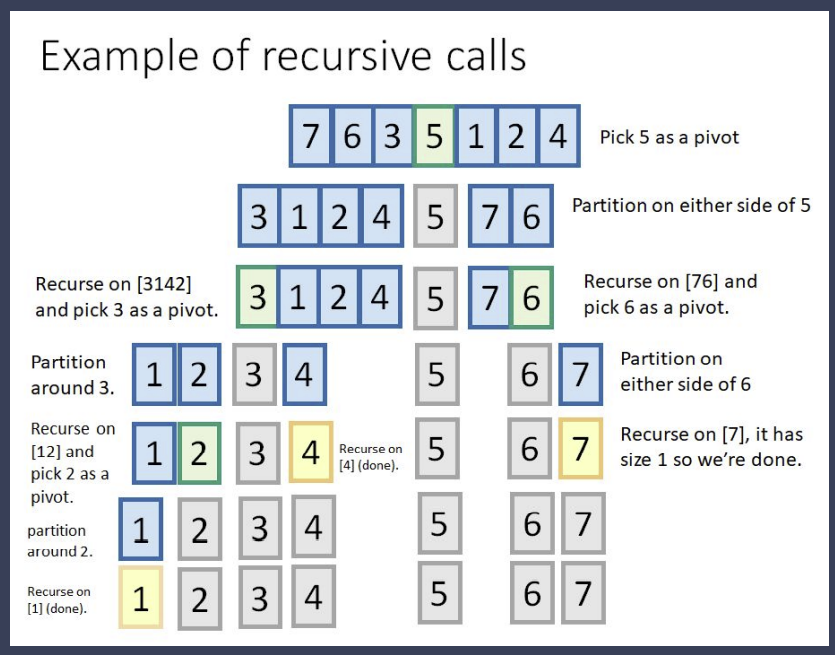
**Not Inline Sort**

Stable Sort

### Quick Sort

In quick sort choose a pivot at random index, and move all smaller elements to left and bigger elements to right of the pivot. And divide the set until size 1. The array gets sorted.





The partitioning can be done **Inline** in two ways – Lomuto, Hoare.

void quickHelper(vector<int> &a, int s, int e) {

    // Leaf Node, nothing to do

    if (s >= e)

        return;

    // Choose a random index as pivot by moving to to front.

    // This avoids hitting worst case frequently.

    int r = s + rand() % (e-s+1);

    swap(a[r],a[s]);

    // Lomuto’s partition

    // int pivot = a[s];

    // int es = s; // end of small partition

    // int j = s + 1;

    // while (j <= e) {

    //     if (pivot >= a[j]) {

    //         es++;

    //         swap(a[j], a[es]);

    //     }

    //     j++;

    // }

    // Hoare's partition

    int pivot = a[s];

    int i = s + 1;

    int j = e;

    while (i <= j) {

        // if (a[i] <= pivot)

        //     i++;

        // else if (a[j] > pivot)

        //     j--;

        // else

        //     swap(a[i++],a[j--]);

        while(i <= j && pivot >= a[i] ) i++;

        while(i <= j && pivot < a[j] ) j--;

        if (i < j)

            swap(a[i],a[j]);

    }

    // Move the pivot to end of small partition.

    // This moves pivot to its position in sorted array.

    swap(a[s], a[j]);

    // leave the element at end of small partition and process remaining

    quickHelper(a, s, j-1);

    quickHelper(a, j+1, e);

}

vector<int> quickSort(vector<int> &nums) {

    quickHelper(nums, 0, nums.size()-1);

    return nums;

}

**TC**: Best Case: O(n logn), Avg Case: O(n logn), Worst Case: O(n2)

**SC**: O(1)

Inline Sort

**Not Stable Sort**

## Transform and Conquer

Transform the input into another data structure and conquer.

### Heap Sort

In heap sort the entire input can be transformed into complete binary heap (priority queue) and be sorted.

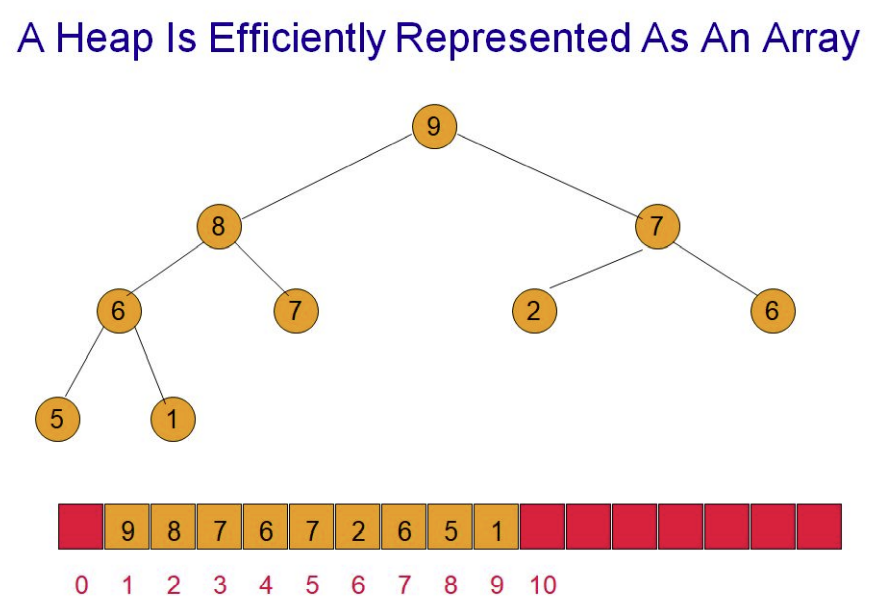
Priority queue can be Max priority queue or min priority queue.

The extract operation of max priority queue gives maximum number.

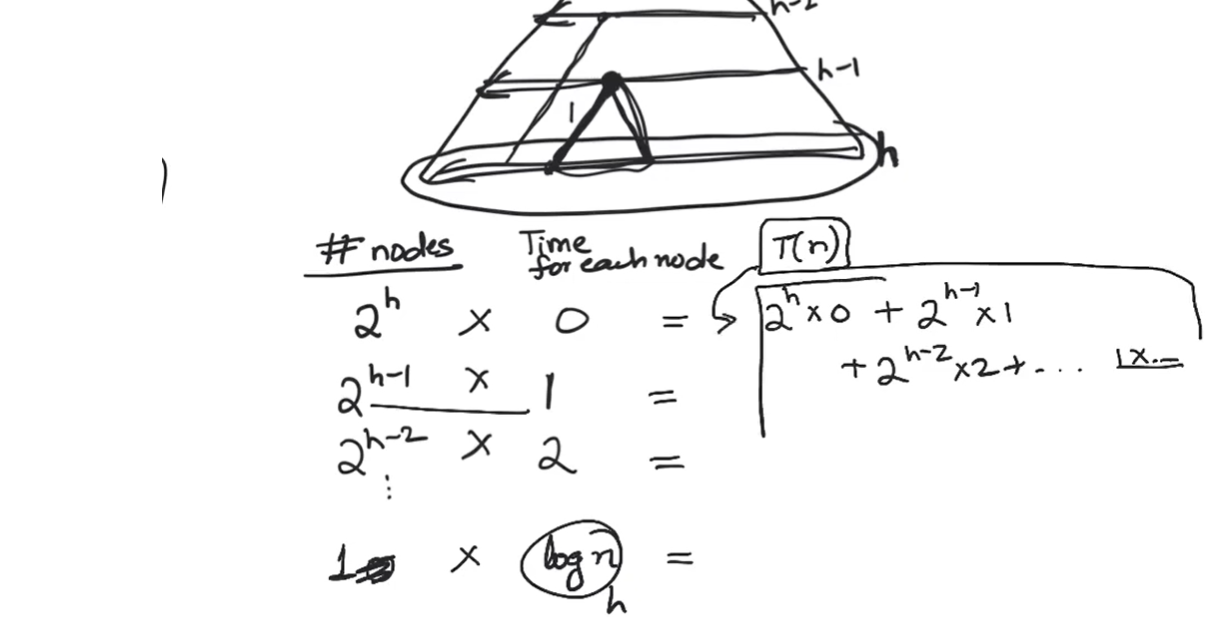
Properties of max priority queue

* Priority of any node is bigger than child
* Always a complete binary tree

Insertion can be done in end and heapify up to maintain properties of heap. Similarly extraction.



Build Heap Inline Time Complexity O(n)



int parent(int child) {

    return (child-1) / 2;

}

int leftChild(int parent) {

    return (2 \* parent) + 1;

}

int rightChild(int parent) {

    return (2 \* parent) + 2;

}

void heapifydown(vector<int> &a, int parent, int size){

    int left = leftChild(parent);

    int right = rightChild(parent);

    int largest = parent;

    if(left < size && a[parent] < a[left])

        largest = left;

    if (right < size && a[largest] < a[right])

        largest = right;

    if (largest != parent) {

        swap(a[parent], a[largest]);

        heapifydown(a, largest, size);

    }

}

void heapifyup(vector<int> &a, int child){

    if(child && a[parent(child)] < a[child]) {

        swap(a[parent(child)],a[child]);

        heapifyup(a,parent(child));

    }

}

void buildHeap(vector<int> &a) {

    for(int i=a.size()/2; i >= 0; i--)

        heapifydown(a,i, a.size());

}

void extractAll(vector<int> &a) {

    int size = a.size() - 1;

    for(int i = 0; i < size; i++) {

        swap(a[0], a[size-i]);

        heapifydown(a, 0, size-i);

    }

}

vector<int> heapSort(vector<int> &a) {

    buildHeap(a);

    printVector(a);

    extractAll(a);

    return a;

}

**TC**: Best Case: O(n logn

**SC**: O(1)

Inline Sort

**Not Stable Sort**

### Counting Sort

### Radix Sort

## Cheat Sheet on sorting algorithms

## Problem Solving – Pre Sorting

### Sort an Array – Leet Code 912

<https://leetcode.com/problems/sort-an-array/>

Both heap sort and merge sort are accepted.

### Two Sum – Leet Code 1

<https://leetcode.com/problems/two-sum/>

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

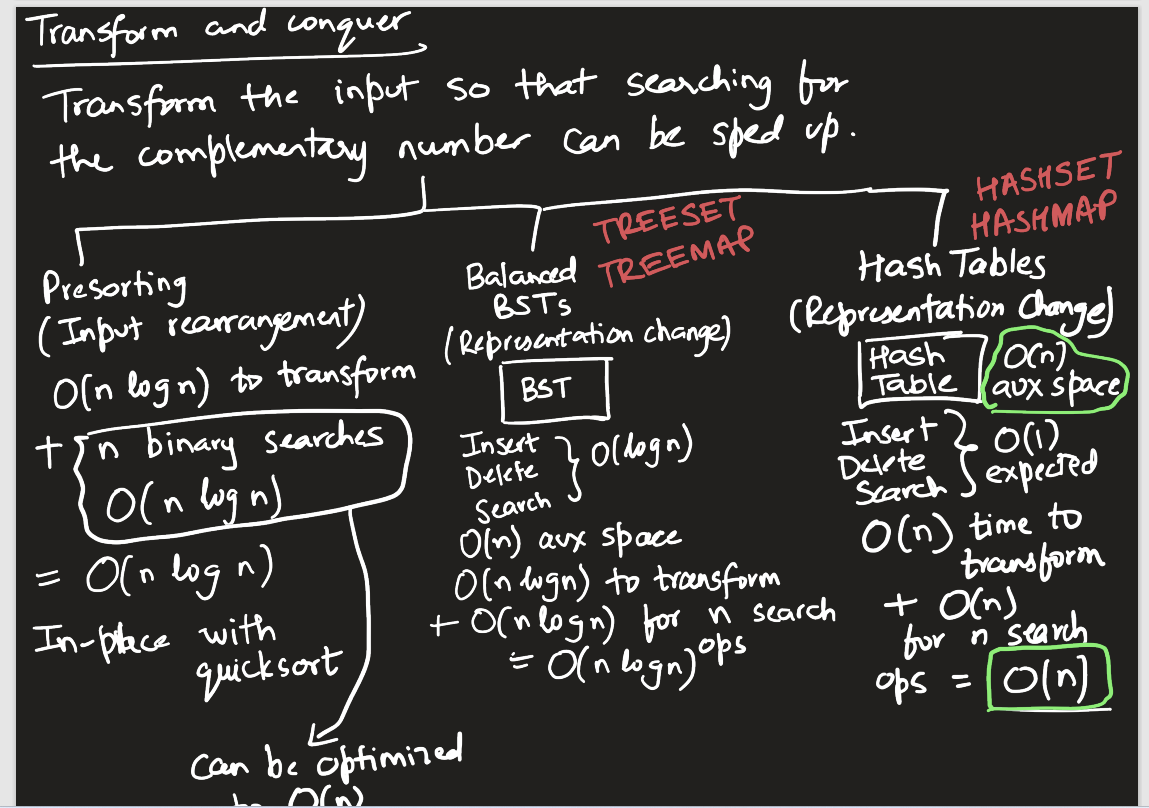
**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]



#if 1 // Solution 1

// The below solution can be used if its sorted.

// Use two pointers to get two sum.

vector<int> twoSum(vector<int>& a, int target) {

    int l = 0, r = a.size() - 1;

    while(l < r) {

        if (a[l] + a[r] == target)

            return {l,r};

        else if (a[l]+a[r] > target)

            r--;

        else

            l++;

    }

    return {};

}

#else // Solution 2

// Use heap to get two sum

vector<int> twoSum(vector<int>& a, int target) {

    map<int,int> hmap;

    for (int i=0; i < a.size();i++) {

        int rem = target - a[i];

        auto it = hmap.find(rem);

        if ( it != hmap.end()) {

            return {it->second, i};

        }

        hmap[a[i]] = i;

    }

    return {};

}

#endif

Two Pointer

TC: O(n) + Sorting if done

SC: O(1)

Hash Map

TC: O(n)

SC: O(n)



### Three Sum – Leet Code 15

<https://leetcode.com/problems/3sum/>

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

**Input:** nums = [-1,0,1,2,-1,-4]

**Output:** [[-1,-1,2],[-1,0,1]]

**Explanation:**

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

**Example 2:**

**Input:** nums = [0,1,1]

**Output:** []

**Explanation:** The only possible triplet does not sum up to 0.

**Example 3:**

**Input:** nums = [0,0,0]

**Output:** [[0,0,0]]

**Explanation:** The only possible triplet sums up to 0.

vector<vector<int>> threeSum(vector<int>& a, int t = 0) {

    vector<vector<int>> res;

    int i = 0;

    sort(a.begin(),a.end());

    while ( i < a.size()-2 ) {

        int rem = t - a[i];

        int l = i+1;

        int r = a.size()-1;

        while (l < r) {

            if (a[l] + a[r] == rem) {

                res.push\_back({a[i],a[l],a[r]});

                l++;r--;

                // Skip duplicates

                while( (l < r) && (a[l] == a[l-1])) l++;

                while( (l < r) && (a[r] == a[r+1])) r--;

            } else if (a[l] + a[r] > rem)

                r--;

            else

                l++;

        }

        i++;

        // Skip duplicates

        while( (i < a.size()-2) && (a[i] == a[i-1])) i++;

    }

    return res;

}

vector<vector<int>> getTwoSum(vector<int>& a, int startpos, int target) {

    vector<vector<int>> res;

    int l = startpos;

    int r = a.size() - 1;

    while(l < r) {

        int sum = a[l] + a[r];

        if (sum == target) {

            res.push\_back({a[l],a[r]});

            l++; r--;

            while(l<r && a[l] == a[l-1]) l++;

            while(l<r && a[r] == a[r+1]) r--;

        } else if (sum > target)

            r--;

        else

            l++;

    }

    return res;

}

vector<vector<int>> threeSum(vector<int>& a, int t = 0) {

    sort(a.begin(), a.end());

    vector<vector<int>> threeSum;

    for(int i = 0; i < a.size()-2; ) {

        vector<vector<int>> twoSums = getTwoSum(a, i+1, t-a[i]);

        for(auto twoSum: twoSums) {

            threeSum.push\_back(twoSum);

            threeSum.back().push\_back(a[i]);

        }

        i++;

        while((i < a.size()-2 ) && (a[i] == a[i-1])) i++;

    }

    return threeSum;

}

**Skipping duplicates is one important thing.**

**Maintaining separate index for result array helped.**

**TBD: Add a method with heap as well.**

**TC: O(n2)** - O (n logn) for Sorting + O (n2) for three sum loops.

**SC**: **O(1)**

### 3Sum Closest – Leet Code 16

https://leetcode.com/problems/3sum-closest/description/

Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to target.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

**Example 1:**

**Input:** nums = [-1,2,1,-4], target = 1

**Output:** 2

**Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

**Example 2:**

**Input:** nums = [0,0,0], target = 1

**Output:** 0

**Explanation:** The sum that is closest to the target is 0. (0 + 0 + 0 = 0).

int threeSumClosest(vector<int>& nums, int target) {

    int i = 0;

    int closest = nums[0] + nums[1] + nums[2];

    sort(nums.begin(), nums.end());

    while (i < nums.size()-2){

        int l = i + 1;

        int r = nums.size()-1;

        while(l < r) {

            int sum = nums[i] + nums[l] + nums[r];

            closest = abs( sum - target) < abs(closest - target) ? sum : closest;

            if (sum == target)

                return sum;

            else if (sum > target )

                r--;

            else

                l++;

        }

        i++;

    }

    return closest;

}

**TC: O(n2)** - O (n logn) for Sorting + O (n2) for three sum loops.

**SC**: **O(1)**

### Four Sum – Leet Code 18

<https://leetcode.com/problems/4sum/>

Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that:

0 <= a, b, c, d < n

a, b, c, and d are distinct.

nums[a] + nums[b] + nums[c] + nums[d] == target

You may return the answer in any order.

Example 1:

Input: nums = [1,0,-1,0,-2,2], target = 0

Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

Example 2:

Input: nums = [2,2,2,2,2], target = 8

Output: [[2,2,2,2]]

vector<vector<int>> twoSum(vector<int>& a, int l, long long int t) {

    vector<vector<int>> res;

    int r = a.size()-1;

    while(l < r) {

        if (a[l]+a[r] == t) {

            res.push\_back({a[l],a[r]});

            l++; r--;

            // Skip duplicates.

            while ((l < r) && (a[l] == a[l-1])) l++;

            while (( l < r) && (a[r] == a[r+1])) r--;

        }

        else if (a[l]+a[r] > t)

            r--;

        else

            l++;

    }

    return res;

}

vector<vector<int>> threeSum(vector<int>& a, int s , long long int t) {

    vector<vector<int>> res;

    int i = s;

    int k = 0;

    // Need 2 elements for two sum.

    while (i < a.size()-2) {

        vector<vector<int>> temp = twoSum(a, i+1, t-a[i]);

        int j = 0;

        while (j < temp.size()) {

            res.push\_back(temp[j]);

            res[k].push\_back(a[i]);

            j++; k++;

        }

        i++;

        // Skip duplicates.

        while ((i < a.size()-2) && (a[i] == a[i-1])) i++;

    }

    return res;

}

vector<vector<int>> fourSum(vector<int> a, int target) {

    vector<vector<int>> res;

    int i = 0;

    int j = 0;

    sort(a.begin(),a.end());

    while ( i < a.size() - 3) {

        vector<vector<int>> temp = threeSum(a, i+1, target - a[i]);

        int k = 0;

        while (k < temp.size()) {

            res.push\_back(temp[k]);

            res[j].push\_back(a[i]);

            j++; k++;

        }

        i++;

        // Skip duplicates

        while (( i < a.size() - 3) && (a[i] == a[i-1])) i++;

    }

    return res;

}

**Maintaining separate index for result array helped.**

**TC: O(n3)** - O (n logn) for Sorting + O (n3) for nested loops.

**SC**: **O(1)**

### Attend Meetings – Leet Code 252

https://leetcode.com/problems/meeting-rooms/

<https://uplevel.interviewkickstart.com/resource/rc-codingproblem-404890-794635-1014-6106>

int can\_attend\_all\_meetings(vector<vector<int>> &intervals) {

    sort(intervals.begin(), intervals.end());

    int i = 1;

    while (i < intervals.size()) {

        // Overlapping in below cases

        // If they start at same time or

        // first one ends later than second one starts

        if ((intervals[i-1][0] == intervals[i][0]) ||

            (intervals[i-1][1] > intervals[i][0])) {

            return 0;

        }

        i++;

    }

    // Write your code here.

    return 1;

}

**TC: O(n logn)** - O (n logn) for Sorting + O (n) for one pass.

**SC**: **O(1)**

### Merge Intervals – Leet Code 56

<https://leetcode.com/problems/merge-intervals/description/>

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

**Example 1:**

**Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]

**Output:** [[1,6],[8,10],[15,18]]

**Explanation:** Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

**Example 2:**

**Input:** intervals = [[1,4],[4,5]]

**Output:** [[1,5]]

**Explanation:** Intervals [1,4] and [4,5] are considered overlapping.

vector<int> getOverlappingInterval(vector<int> i1, vector<int> i2) {

    // If both intervals start at same time they are overlapping.

    if (i1[0] == i2[0]) {

        if (i1[1] > i2[1])

            return i1;

        else

            return i2;

    } else if (i1[1] >= i2[0]) {

        return {i1[0], max(i1[1], i2[1])};

    }

    return {};

}

vector<vector<int>> merge(vector<vector<int>>& intervals) {

    vector<vector<int>> res;

    int i = 1;

    sort(intervals.begin(), intervals.end());

    res.push\_back(intervals[0]);

    while ( i < intervals.size()) {

        vector<int> temp = getOverlappingInterval(res.back(), intervals[i]);

        if (temp.size() != 0) {

            res.pop\_back();

            res.push\_back(temp);

        } else {

            res.push\_back(intervals[i]);

        }

        i++;

    }

    return res;

}

**TC: O(n logn)** - O (n logn) for Sorting + O (n) for one pass.

**SC**: **O(1)**

## Problem Solving – Extension of Merge Sort

### Inter Section of two arrays - Leet code 349

Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must be unique and you may return the result in any order.

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2]

Example 2:

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [9,4]

Explanation: [4,9] is also accepted.

vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {

    int i = 0; // Pointing to nums1

    int j = 0; // Pointing to nums2

    vector<int> res;

    sort(nums1.begin(), nums1.end());

    sort(nums2.begin(), nums2.end());

    while (i < nums1.size() && j < nums2.size()) {

        if (nums1[i] == nums2[j]) {

            res.push\_back(nums1[i]);

            i++; j++;

            // Skip the duplicates

            while (i < nums1.size() && (nums1[i] == nums1[i-1])) i++;

            while (j < nums2.size() && (nums2[j] == nums2[j-1])) j++;

        } else if (nums1[i] < nums2[j])

            i++;

        else

            j++;

    }

    return res;

}

**TC: O(n + m)**

**SC**: **O(1)**

### Inter Section of two arrays - Leet code 350

https://leetcode.com/problems/intersection-of-two-arrays-ii/description/

Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

**Example 1:**

**Input:** nums1 = [1,2,2,1], nums2 = [2,2]

**Output:** [2,2]

**Example 2:**

**Input:** nums1 = [4,9,5], nums2 = [9,4,9,8,4]

**Output:** [4,9]

**Explanation:** [9,4] is also accepted.

vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {

    int i = 0; // Pointing to nums1

    int j = 0; // Pointing to nums2

    vector<int> res;

    sort(nums1.begin(), nums1.end());

    sort(nums2.begin(), nums2.end());

    while (i < nums1.size() && j < nums2.size()) {

        if (nums1[i] == nums2[j]) {

            res.push\_back(nums1[i]);

            i++; j++;

        } else if (nums1[i] < nums2[j])

            i++;

        else

            j++;

    }

    return res;

}

**TC: O(n + m)**

**SC**: **O(1)**

### Inter Section of three arrays – Leet Code 1213

<https://leetcode.com/problems/intersection-of-three-sorted-arrays/>

<https://uplevel.interviewkickstart.com/resource/rc-codingproblem-404892-799432-1015-6111>

Given three arrays sorted in the ascending order, return their intersection sorted array in the ascending order.

{

"arr1": [2, 5, 10],

"arr2": [2, 3, 4, 10],

"arr3": [2, 4, 10]

}

[2, 10]

{

"arr1": [1, 2, 3],

"arr2": [],

"arr3": [2, 2]

}

[-1]

vector<int> find\_intersection(vector<int> &arr1, vector<int> &arr2, vector<int> &arr3) {

    vector<int> res;

    int i = 0; // Pointing arr1

    int j = 0; // Pointing arr2

    int k = 0; // Pointing arr3

    while (i < arr1.size() && j < arr2.size() && k < arr3.size()) {

        if (arr1[i] == arr2[j] && arr1[i] == arr3[k]) {

            res.push\_back(arr1[i]);

            i++; j++; k++;

            // Skip if duplicates not allowed.

            // while (i < arr1.size() && arr1[i] == arr1[i-1]) i++;

            // while (j < arr2.size() && arr2[j] == arr2[j-1]) j++;

            // while (k < arr3.size() && arr3[k] == arr3[k-1]) k++;

        } else if (arr1[i] < arr2[j] && arr1[i] < arr3[k])

            i++;

        else if (arr2[j] < arr3[k])

            j++;

        else

            k++;

    }

    if (res.size() == 0)

        res.push\_back(-1);

    return res;

}

**TC: O(n + m + o)**

**SC**: **O(1)**

### Count of Smaller Numbers After Self – Leet Code 315

https://leetcode.com/problems/count-of-smaller-numbers-after-self/

Given an integer array nums, return an integer array counts where counts[i] is the number of smaller elements to the right of nums[i].

Example 1:

Input: nums = [5,2,6,1]

Output: [2,1,1,0]

Explanation:

To the right of 5 there are 2 smaller elements (2 and 1).

To the right of 2 there is only 1 smaller element (1).

To the right of 6 there is 1 smaller element (1).

To the right of 1 there is 0 smaller element.

Example 2:

Input: nums = [-1]

Output: [0]

Example 3:

Input: nums = [-1,-1]

Output: [0,0]

class Element {

    public:

        int value;

        int origIdx;

        int small;

        Element(int val, int idx) : value(val), origIdx(idx),small(0) {}

};

void helper(vector<class Element \*> &a, int s, int e) {

    if (s == e) return;

    int m = (s+e) / 2;

    helper(a, s, m);

    helper(a, m+1, e);

    int i = s, j = m + 1;

    int acc = 0;

    vector<class Element \*> aux;

    while(i <= m && j <= e) {

        if (a[i]->value > a[j]->value) {

            aux.push\_back(a[j++]);

            acc++;

        } else {

            a[i]->small += acc;

            aux.push\_back(a[i++]);

        }

    }

    while(i <= m) {

        a[i]->small += acc;

        aux.push\_back(a[i++]);

    }

    while(j <= e) aux.push\_back(a[j++]);

    for(i=0; i < aux.size(); i++)

        a[s+i] = aux[i];

}

vector<int> countSmaller(vector<int>& nums) {

    vector<class Element \*> a;

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

        a.push\_back(new Element(nums[i],i));

    }

    helper(a, 0, a.size()-1);

    vector<int> res (nums.size());

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

        res[a[i]->origIdx] = a[i]->small;

    }

    return res;

}

**TC: O(n logn)**

**SC**: **O(n)**

**The tricky part is using acc wisely. The explanation is if a number in left half is bigger, all elements of right array need to be added into smaller elements count.**

### Inversion Pairs

### Reverse Pairs

### Merge Sorted Array – Leet Code 88

<https://leetcode.com/problems/merge-sorted-array/>

You are given two integer arrays nums1 and nums2, sorted in **non-decreasing order**, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

**Merge** nums1 and nums2 into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

**Example 1:**

**Input:** nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

**Output:** [1,2,2,3,5,6]

**Explanation:** The arrays we are merging are [1,2,3] and [2,5,6].

The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

**Example 2:**

**Input:** nums1 = [1], m = 1, nums2 = [], n = 0

**Output:** [1]

**Explanation:** The arrays we are merging are [1] and [].

The result of the merge is [1].

**Example 3:**

**Input:** nums1 = [0], m = 0, nums2 = [1], n = 1

**Output:** [1]

**Explanation:** The arrays we are merging are [] and [1].

The result of the merge is [1].

Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

    int k = nums1.size() - 1;

    m--; n--;

    while (n >=0 && m >= 0) {

        if (nums1[m] > nums2[n])

            nums1[k--] = nums1[m--];

        else

            nums1[k--] = nums2[n--];

    }

    while (n >= 0 )

        nums1[k--] = nums2[n--];

}

**TC: O(n + m)**

**SC**: **O(1)**

## Problem Solving – Extension of Quick Sort (Quick Select)

### Find Kth smallest / largest element – Leet code 215

<https://leetcode.com/problems/kth-largest-element-in-an-array/>

void helper(vector<int> a, int s, int e, int t){

    // Base Case

    if (s >= e) return;

    // Swap with random index.

    int r = s + rand() % (e-s+1);

    swap(a[s], a[r]);

    // Perform Lomuto's partition.

    int i = s;

    int j = s + 1;

    int pivot = a[s];

    while (j <= e) {

        if (a[j] <= pivot) {

            ++i;

            swap(a[j], a[i]);

        }

        j++;

    }

    swap(a[i], a[s]);

    if (i == t)

        return ;

    else if (t > i)

        return helper(a, i+1, e, t);

    else

        return helper(a, s, i-1, t);

    return;

}

int findKthLargest(vector<int>& nums, int k) {

    helper(nums, 0, nums.size()-1, nums.size() - k);

    return  nums[nums.size() - k];

}

**TC: O(n) – Need to analyze**

**SC**: **O(1)**

### K Closest Points to Origin – Leet code 973

https://leetcode.com/problems/k-closest-points-to-origin/

### Top K Frequent Elements – Leet Code 347

<https://leetcode.com/problems/top-k-frequent-elements/>

Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

Example 1:

Input: nums = [1,1,1,2,2,3], k = 2

Output: [1,2]

Example 2:

Input: nums = [1], k = 1

Output: [1]

void helper(vector<int> &a, int s, int e, int t) {

    // Base Case

    if (s >= e) return ;

    // Swap with random index.

    int r = s + rand() % (e-s+1);

    swap(a[s], a[r]);

    // Perform Lomuto's partition.

    int i = s;

    int j = s + 1;

    int pivot = a[s];

    while (j <= e) {

        if (a[j] <= pivot) {

            ++i;

            swap(a[j], a[i]);

        }

        j++;

    }

    swap(a[i], a[s]);

    if (i == t)

        return ;

    else if (t > i)

        return helper(a, i+1, e, t);

    else

        return helper(a, s, i-1, t);

    return ;

}

vector<int> topKFrequent(vector<int>& nums, int k) {

    // Build a frequency map.

    map<int, int> hmap;

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

        hmap[nums[i]]++;

    // Invert Key->Value in frequency map. Frequency shall be key.

    map<int, vector<int>> ihmap;

    vector<int> frequencies;

    for (auto it=hmap.begin(); it != hmap.end(); it++)

    {

        ihmap[it->second].push\_back(it->first);

        frequencies.push\_back(it->second);

    }

    // Perform quick select n-k position.

    helper(frequencies, 0, frequencies.size()-1, frequencies.size()-k);

    // Pop up top k elements

    vector<int> res;

    for (int i=frequencies.size()-1; i >= 0 && k > 0; i--)

    {

        res.push\_back(ihmap[frequencies[i]].back());

        ihmap[frequencies[i]].pop\_back();

        k--;

    }

    return res;

}

**TC: O(n logn)**

**SC**: **O(n)**

## Problem Solving – Extension of Lomuto and Hoare’s partition

### Sort based on 2 flag colors

### Sort based on 3 flag colors – Dutch Flag – Leet Code 75

<https://leetcode.com/problems/sort-colors/description/>

Given an array nums with n objects colored red, white, or blue, sort them [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm)so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

**Example 1:**

**Input:** nums = [2,0,2,1,1,0]

**Output:** [0,0,1,1,2,2]

**Example 2:**

**Input:** nums = [2,0,1]

**Output:** [0,1,2]

// Lomuto's partition

void sortColors(vector<int>& nums) {

    int ts = 0, os = 0, c = 0;

    while (c < nums.size()) {

        if (nums[c] == 2)

            c++;

        else if (nums[c] == 1)

            swap(nums[c++],nums[ts++]);

        else {

            swap(nums[c],nums[ts]);

            swap(nums[os],nums[ts]);

            os++; ts++; c++;

        }

    }

    return ;

}

// Hoare's partition

void sortColors(vector<int>& nums) {

    int l = 0, r = nums.size()-1, c = 0;

    while (c <= r) {

        if (nums[c] == 1)

            c++;

        else if (nums[c] == 0)

            swap(nums[c++],nums[l++]);

        else {

            swap(nums[c],nums[r--]);

        }

    }

    return ;

}

**TC: O(n)**

**SC**: **O(1)**

### Sort based on even odd – Leet Code 905

<https://leetcode.com/problems/sort-array-by-parity/>

Given an integer array nums, move all the even integers at the beginning of the array followed by all the odd integers.

Return ***any array****that satisfies this condition*.

**Example 1:**

**Input:** nums = [3,1,2,4]

**Output:** [2,4,3,1]

**Explanation:** The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3] would also be accepted.

**Example 2:**

**Input:** nums = [0]

**Output:** [0]

// Lomuto's partition

vector<int> sortArrayByParity(vector<int>& nums) {

    int bo =0, c= 0;

    while (c < nums.size()) {

        if (nums[c] %2 == 0)

            swap(nums[c],nums[bo++]);

        c++;

    }

    return nums;

}

// Hoare's partition

vector<int> sortArrayByParity(vector<int>& nums) {

    int l = 0, r = nums.size()-1;

    while (l < r) {

        while ((l < r) && (nums[l] %2 == 0)) l++;

        while ((l < r) && (nums[r] %2 != 0)) r--;

        if (l < r)

            swap(nums[l], nums[r]);

    }

    return nums;

}

**TC: O(n)**

**SC**: **O(1)**

### Sort based on parity 2 – Leet Code 922

<https://leetcode.com/problems/sort-array-by-parity-ii/description/>

Given an array of integers nums, half of the integers in nums are **odd**, and the other half are **even**.

Sort the array so that whenever nums[i] is odd, i is **odd**, and whenever nums[i] is even, i is **even**.

Return *any answer array that satisfies this condition*.

**Example 1:**

**Input:** nums = [4,2,5,7]

**Output:** [4,5,2,7]

**Explanation:** [4,7,2,5], [2,5,4,7], [2,7,4,5] would also have been accepted.

**Example 2:**

**Input:** nums = [2,3]

**Output:** [2,3]

// Hoare's partition

vector<int> sortArrayByParityII(vector<int>& nums) {

    int e = 0, o = 1;

    while (e < nums.size() && o < nums.size()) {

        while (e < nums.size() && (nums[e] %2 == 0)) e += 2;

        while (o < nums.size() && (nums[o] %2 == 1)) o += 2;

        if (e < nums.size() && o < nums.size())

            swap(nums[e], nums[o]);

    }

    return nums;

}

**TC: O(n)**

**SC**: **O(1)**

## Problem Solving – Extension of Heap

### Merge k sorted arrays

class HeapElement {

public:

    int ele; // Element value

    int pos; // Array position, 0-based

    HeapElement(int ele\_, int pos\_) : ele(ele\_), pos(pos\_) {}

};

void populateMinHeap(vector<HeapElement \*> minHeap, int last){

    for(int i = 1; i < last;i++) {

        cout << minHeap[i]->ele << " : " << minHeap[i]->pos << endl;

    }

}

void insertMinHeap(vector<HeapElement \*> &a, int pos, HeapElement \* nele) {

    int c = pos;

    int p = c / 2;

    a[c] = nele;

    cout <<  "Insterting at pos "<< pos << " Element " << nele->ele << " : " << nele->pos << endl;

    while (p >= 1) {

        if (a[p]->ele > a[c]->ele) {

            swap(a[p],a[c]);

            c = p;

            p = c/2;

            continue;

        }

        break;

    }

}

HeapElement \*extractMinHeap(vector<HeapElement \*> &minHeap, int last) {

    HeapElement \*ret = minHeap[1];

    swap(minHeap[1], minHeap[last]);

    last--;

    int p = 1, c = 2 \*p;

    while (c <= last) {

        int s = minHeap[c+1] < minHeap[c] ? c+1 : c;

        s = min (s, last);

        if(minHeap[p] > minHeap[s]) {

            swap(minHeap[p], minHeap[s]);

            p = s;

            c = 2 \* p;

            continue;

        }

        break;

    }

    return ret;

}

vector<int> mergeKSortedArrays(vector<vector<int>> &nums) {

    vector<int> res;

    vector<HeapElement \*> minHeap(nums.size()+1);

    // Build min heap for first element of K sorted arrays.

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

    {

        insertMinHeap(minHeap, i+1, new HeapElement(nums[i].front(),i));

        nums[i].erase(nums[i].begin());

    }

    populateMinHeap(minHeap, minHeap.size());

    int minHeapElements = minHeap.size()-1;

    while (minHeapElements) {

        HeapElement \*temp = extractMinHeap(minHeap, minHeapElements);

        cout <<  "Extracted " << temp->ele << " : " << temp->pos << endl;

        if (nums[temp->pos].size() != 0) {

            insertMinHeap(minHeap, minHeapElements, new HeapElement(nums[temp->pos].front(),temp->pos));

            nums[temp->pos].erase(nums[temp->pos].begin());

        } else {

            minHeapElements--;

        }

        res.push\_back(temp->ele);

        delete temp;

    }

    return res;

}

### Merge k sorted linked lists – Leet Code 23

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

Input: lists = [[1,4,5],[1,3,4],[2,6]]

Output: [1,1,2,3,4,4,5,6]

Explanation: The linked-lists are:

[

1->4->5,

1->3->4,

2->6

]

merging them into one sorted list:

1->1->2->3->4->4->5->6

Example 2:

Input: lists = []

Output: []

Example 3:

Input: lists = [[]]

Output: []

https://leetcode.com/problems/merge-k-sorted-lists/description/

void populateMinHeap(vector<ListNode \*> minHeap, int last){

    for(int i = 1; i <= last;i++) {

        cout << minHeap[i]->val << endl;

    }

}

void insertMinHeap(vector<ListNode\*> &a, int pos, ListNode \*ele){

    int c = pos;

    int p = c / 2;

    a[c] = ele;

    // cout <<  "Insterting at pos "<< pos << " Element " << ele->node->val << endl;

    while (p >= 1) {

        if (a[p]->val > a[c]->val) {

            swap(a[p], a[c]);

            c = p;

            p = c/2;

            continue;

        }

        break;

    }

}

ListNode \*extractMinHeap(vector<ListNode\*> &a, int last) {

    ListNode \*first = a[1];

    int p = 1, c = 2 \* p;

    swap(a[p], a[last]);

    last--;

    while(c <= last) {

        int s = a[c+1]->val < a[c]->val ? c+1 : c;

        s = min(s, last);

        if (a[p]->val > a[s]->val) {

            swap(a[p], a[s]);

            p = s;

            c = 2 \* p;

            continue;

        }

        break;

    }

    return first;

}

ListNode\* mergeKLists(vector<ListNode\*> lists) {

    ListNode\* dummy = new ListNode(0);

    ListNode \*prev = dummy;

    if (lists.size() == 0 )

        return nullptr;

    vector<ListNode \*> minHeap(lists.size()+1);

    int heapElements = minHeap.size() - 1;

    for(int i=0, j = 0; i < lists.size(); i++) {

        if (lists[i] != nullptr)

            insertMinHeap(minHeap, ++j, lists[i]);

        else

            heapElements--;

    }

    // populateMinHeap(minHeap, heapElements);

    while(heapElements != 0) {

        ListNode \*temp = extractMinHeap(minHeap,heapElements);

        if (temp->next != nullptr) {

            insertMinHeap(minHeap, heapElements, temp->next);

        } else

            heapElements--;

        prev->next = temp;

        prev = prev->next;

        // cout <<  "Extracted " << temp->val << " " << temp->next << " " << heapElements << endl;

    }

    // cout << "Returning" << endl;

    return dummy->next;

}

### Kth largest element in stream

Kth largest – Make a MIN HEAP of k elements, from then push only the elements bigger than root. Kth largest will be at root.

<https://leetcode.com/problems/kth-largest-element-in-an-array/>

Given an integer array nums and an integer k, return *the* kth *largest element in the array*.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

You must solve it in O(n) time complexity.

**Example 1:**

**Input:** nums = [3,2,1,5,6,4], k = 2

**Output:** 5

**Example 2:**

**Input:** nums = [3,2,3,1,2,4,5,5,6], k = 4

**Output:** 4

void insertMinHeap(vector<int> &a, int pos, int element){

    int c = pos;

    int p = c / 2;

    a[c] = element;

    while (p >= 1) {

        if (a[p] > a[c]) {

            swap(a[p], a[c]);

            c = p;

            p = c/2;

            continue;

        }

        break;

    }

}

int extractMinHeap(vector<int> &a, int last) {

    int first = a[1];

    int p = 1, c = 2 \* p;

    swap(a[p], a[last]);

    last--;

    while (c <= last) {

        int s = a[c+1] < a[c] ? c+1 : c;

        s = min (s, last);

        if (a[p] > a[s]) {

            swap(a[p], a[s]);

            p = s;

            c = 2 \*p;

            continue;

        }

        break;

    }

    return first;

}

int findKthLargest(vector<int>& nums, int k) {

    vector<int> minHeap(k+1);

    // Make a min heap of K elements

    for(int i=0; i<k; i++)

    {

        insertMinHeap(minHeap, i+1, nums[i]);

    }

    // Insert remaining elements

    for(int i=k; i < nums.size(); i++)

    {

        if (nums[i] >= minHeap[1]) {

            extractMinHeap(minHeap, k);

            insertMinHeap(minHeap, k, nums[i]);

        }

    }

    // Return the root element

    return minHeap[1];

}

**TC: O(n logk)**

**SC**: **O(k)**

### Kth smallest element in stream

void insertMaxHeap(vector<int> &a, int pos, int element){

    int c = pos;

    int p = c / 2;

    a[c] = element;

    while (p >= 1) {

        if (a[p] < a[c]) {

            swap(a[p], a[c]);

            c = p;

            p = c/2;

            continue;

        }

        break;

    }

}

int extractMaxHeap(vector<int> &a, int last) {

    int first = a[1];

    int p = 1, c = 2 \* p;

    swap(a[p], a[last]);

    last--;

    while (c <= last) {

        int s = a[c+1] > a[c] ? c+1 : c;

        s = min (s, last);

        if (a[p]  < a[s]) {

            swap(a[p], a[s]);

            p = s;

            c = 2 \*p;

            continue;

        }

        break;

    }

    return first;

}

int findKthSmallest(vector<int>& nums, int k) {

    vector<int> maxHeap(k+1);

    // Make a min heap of K elements

    for(int i=0; i<k; i++)

    {

        insertMaxHeap(maxHeap, i+1, nums[i]);

    }

    // Insert remaining elements

    for(int i=k; i < nums.size(); i++)

    {

        if (nums[i] <= maxHeap[1]) {

            extractMaxHeap(maxHeap, k);

            insertMaxHeap(maxHeap, k, nums[i]);

        }

    }

    // Return the root element

    return maxHeap[1];

}

**TC: O(n logk)**

**SC**: **O(k)**

### Take a sequence of strings in streaming fashion and find kth longest string

### Sliding Window Median – Leet Code 480

<https://leetcode.com/problems/sliding-window-median/>

The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle values.

For examples, if arr = [2,3,4], the median is 3.

For examples, if arr = [1,2,3,4], the median is (2 + 3) / 2 = 2.5.

You are given an integer array nums and an integer k. There is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the median array for each window in the original array. Answers within 10-5 of the actual value will be accepted.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [1.00000,-1.00000,-1.00000,3.00000,5.00000,6.00000]

Explanation:

Window position Median

--------------- -----

[1 3 -1] -3 5 3 6 7 1

1 [3 -1 -3] 5 3 6 7 -1

1 3 [-1 -3 5] 3 6 7 -1

1 3 -1 [-3 5 3] 6 7 3

1 3 -1 -3 [5 3 6] 7 5

1 3 -1 -3 5 [3 6 7] 6

Example 2:

Input: nums = [1,2,3,4,2,3,1,4,2], k = 3

Output: [2.00000,3.00000,3.00000,3.00000,2.00000,3.00000,2.00000]

vector<double> medianSlidingWindow(vector<int>& nums, int k) {

    multiset<int> currentK;

    vector<double> medianK;

    bool even = (k % 2 == 0);

    int i = 0;

    for(; i < nums.size(); i++) {

        currentK.insert(nums[i]);

        if(i+1 >= k) {

            if(even) {

                auto it = currentK.begin();

                advance(it, k/2-1);

                int x = \*it;

                advance(it, 1);

                medianK.push\_back ((double)(\*it)/2 + (double)(x)/2);

            }

            else {

                auto it = currentK.begin();

                advance(it, k/2);

                medianK.push\_back ((double)(\*it));

            }

            currentK.erase(currentK.lower\_bound(nums[i+1-k]));

        }

    }

    return medianK;

}

### 

### 

### Find Median from Data Stream – Leet Code295

<https://leetcode.com/problems/find-median-from-data-stream/>

    priority\_queue<int> maxHeap;

    priority\_queue<int, vector<int>, greater<int>> minHeap;

void addNum(int num) {

    maxHeap.push(num);

    minHeap.push(maxHeap.top());

    maxHeap.pop();

    if(minHeap.size() > maxHeap.size()) {

        maxHeap.push(minHeap.top());

        minHeap.pop();

    }

}

double findMedian() {

    if(maxHeap.size() == minHeap.size())

        return (double) (maxHeap.top() + minHeap.top())/2;

    else if(maxHeap.size() > minHeap.size())

        return maxHeap.top();

    else

        return minHeap.top();

}

### Find median of two sorted array/ stream – Leet Code 4

<https://leetcode.com/problems/median-of-two-sorted-arrays/>

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

**Example 1:**

**Input:** nums1 = [1,3], nums2 = [2]

**Output:** 2.00000

**Explanation:** merged array = [1,2,3] and median is 2.

**Example 2:**

**Input:** nums1 = [1,2], nums2 = [3,4]

**Output:** 2.50000

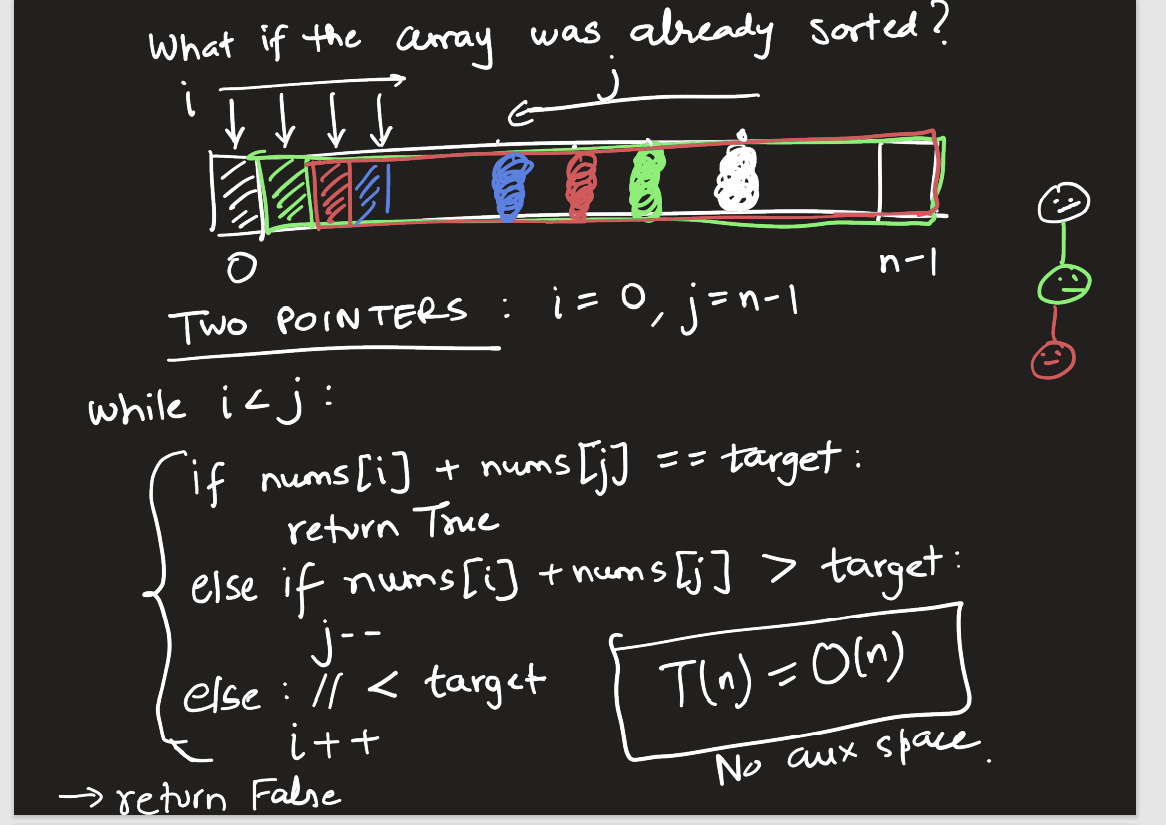
**Explanation:** merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

## Coding Patterns

### Two Pointer

On a sorted input two pointers can be used from extreme ends for various application involving sum calculation etc.,

For 2 sum it can be used as follows



### K the largest or Smallest using Heap

Kth largest – Make a MIN HEAP of k elements, from then push only the elements bigger than root. Kth largest will be at root.

Kth smallest – Make a MAX HEAP of k elements, push only smaller element.

# Recursion

## Recursion In General

Recursion usually handy in implementing divide and conquer strategy, where as iterative approach is for decrease and conquer (which also can be done by recursive).

Recursive can be done in at least two ways, 1- decrease size by one, 2- decrease size by half.

### Power of two

// Decrease and Conquer

int poweroftwo(int n) {

    // Base Case

    if (n == 0)

        return 1;

    else

        return 2 \* poweroftwo(n-1);

    return 0;

}

TC: O(n)

SC: O(1)

// Divide and Conquer

int poweroftwo(int n) {

    // 2 ^ n = 2^n/2 \* 2 ^n/2   : Even

    // 2 ^ n = 2^n/2 \* 2 ^n/2 \*x  : Odd

    // Base case

    if (n == 0)

        return 1;

    else if (n%2 == 0) {

        int temp = poweroftwo(n/2);

        return temp \* temp;

    } else {

        int temp = poweroftwo(n/2);

        return 2 \* temp \* temp;

    }

}

TC: O(log n)

SC: O(1)

### X Power N – Leet Code 50

https://leetcode.com/problems/powx-n/

int xpowern(int x, int n) {

    // x ^ n = x^n/2 \* x^n/2   : Even

    // x ^ n = x^n/2 \* x^n/2 \* x  : +ve Odd

    // x ^ n = x^n/2 \* x^n/2 \* (1/x)  : -ve Odd

    if ( n == 0)

        return 1;

    else if (n == 1)

        return x;

    else if (n == -1)

        return (1/x);

    else if (x == 0)

        return 0;

    else if (n%2 == 0 ) {

        int temp = xpowern(x, n/2);

        return temp \* temp;

    }  else if(n > 0) {

        int temp = xpowern(x, n/2);

        return x \* temp \* temp;

    }  else{

        int temp = xpowern(x, n/2);

        return (1/x) \* temp \* temp;

    }

}

TC: O(log n)

SC: O(1)

### Fibanocci – Leet Code 509

https://leetcode.com/problems/fibonacci-number/description/

// Iterative

int fibanocci(int n) {

    int prev = 1, prepprev = 0;

    int res = 0;

    for (int i = 2; i <= n; i++) {

        res = prev + prepprev;

        prepprev = prev;

        prev = res;

    }

    return res;

}

TC: O(n)

SC: O(1)

// Recursive

int fibanocci(int n) {

    // Base case

    if (n == 0)

        return 0;

    else if ( n == 1)

        return 1;

    else

        return fibanocci(n-1) + fibanocci(n-2);

}

TC: O(2^n)

SC: O(1)

// Additive Recursive

int fibanocci(int n, int b1 = 0, int b2 = 1) {

    // Base case

    if (n == 0)

        return b1;

    else

        return fibanocci(n-1, b2, b1+b2) ;

}

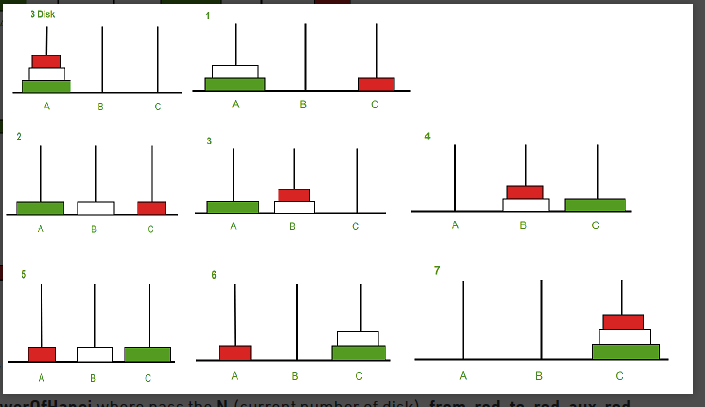
TC: O(n)

SC: O(1)

### Maximum number of Nodes in BST – Catalan Numbers - TBD

### Tower of Hanoi – Geeks for Geeks

<https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/>



vector<vector<int>> res;

void towerOfHanoi(int n, int s, int d, int a) {

    if (n == 1) {

        res.push\_back({s,d});

    } else {

        towerOfHanoi(n-1, s, a, d);

        res.push\_back({s,d});

        towerOfHanoi(n-1, a, d, s);

    }

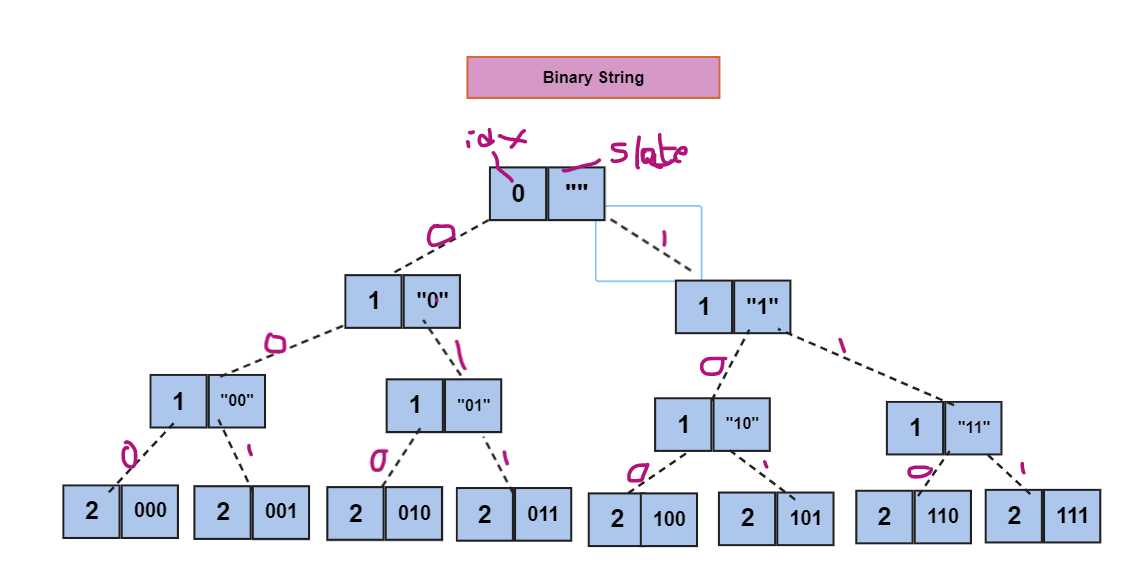
}

TC: O(2^n)

SC: O(2^n) for result

## Combinatorial Problems - Subset/Permutation of a set

### Binary String



vector<string> res;

void binaryString(int n, string &slate) {

    // cout << n << " : " << slate << endl;

    if (n == 0) {

        res.push\_back(slate);

        // return res;

    }

    else {

        slate.push\_back('0');

        binaryString(n-1, slate);

        slate.pop\_back();

        slate.push\_back('1');

        binaryString(n-1, slate);

        slate.pop\_back();

    }

}

TC: O(n \* 2^n)

SC: O(n \* 2^n) for result

### Permutations – Leet Code 46

<https://leetcode.com/problems/permutations/description/>

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

void helper(vector<int>& nums, int idx, vector<vector<int>> &res){

    if (idx == nums.size())

        res.push\_back(nums);

    else {

        for(int i=idx; i < nums.size(); i++) {

            swap(nums[idx], nums[i]);

            helper(nums, idx+1,res);

            swap(nums[idx], nums[i]);

        }

    }

}

TC: O(n! \* n)

SC: Call Stack O(n) + Result O(n! \* n)

### Permutations II - Leet Code 47

<https://leetcode.com/problems/permutations-ii/description/>

Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.

Example 1:

Input: nums = [1,1,2]

Output:

[[1,1,2],

[1,2,1],

[2,1,1]]

void helper(map<int,int> hmap, int size, vector<int> &slate, vector<vector<int>> &res){

    // cout << "Map: " << endl;

    // for (auto it=hmap.begin(); it != hmap.end();it++)

    //     cout << it->first << " : " << it->second << endl;

    // cout << "Slate: " << endl;

    // printVector(slate);

    if (size == slate.size())

        res.push\_back(slate);

    else {

        for (auto it=hmap.begin(); it != hmap.end();it++) {

            if (it->second != 0)

            {

                it->second--;

                slate.push\_back(it->first);

                helper(hmap, size, slate, res);

                slate.pop\_back();

                it->second++;

            }

        }

    }

}

vector<vector<int>> driver(vector<int> &arr){

    map<int,int> fmap;

    vector<int> slate;

    vector<vector<int>> res;

    for(int i=0; i < arr.size();i++)

        fmap[arr[i]]++;

    helper(fmap,arr.size(),slate,res);

    return res;

}

### Subsets – Lee Code 78

<https://leetcode.com/problems/subsets/description/>

Given an integer array nums of unique elements, return all possible

subsets

(the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

void helper(vector<int>& input, int idx, vector<int>& prev, vector<vector<int>>& res) {

    if (idx == input.size()) {

        res.push\_back(prev);

    } else {

        helper(input, idx+1, prev, res);

        prev.push\_back(input[idx]);

        helper(input, idx+1, prev, res);

        prev.pop\_back();

    }

}

**TC: O(2^n \* n)**

**SC: O(2^n \* n)**

### Subsets II – Leet Code 90

<https://leetcode.com/problems/subsets-ii/description/>

Given an integer array nums that may contain duplicates, return all possible

subsets

(the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,2]

Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]

void helper(vector<int> s, int start, vector<int> &slate, vector<vector<int>> &res) {

    if (start == s.size())

        res.push\_back(slate);

    else {

        int c=1, i=start;

        while((i+1 < s.size()) && (s[i] == s[++i])) c++;

        for( i=0; i < c; i++) {

            slate.push\_back(s[start]);

            helper(s, start+c, slate, res);

        }

        for( i=0; i < c; i++)

            slate.pop\_back();

        helper(s, start+c, slate, res);

    }

}

vector<vector<int>> subsetsWithDup(vector<int>& nums) {

    vector<int> prev;

    vector<vector<int>> res;

    sort(nums.begin(), nums.end());

    helper(nums, 0, prev, res);

    return res;

}

**TC: O(2^n \* n)**

**SC: O(2^n \* n)**

### Combination Sum II – Leet Code 40

<https://leetcode.com/problems/combination-sum-ii/description/>

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used once in the combination.

Note: The solution set must not contain duplicate combinations.

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8

Output:

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

]

void helper(vector<int> &input, int start, vector<int> &slate, int current, int target, vector<vector<int>> &res)

{

    // Back Tracking case

    if (current == target)

        res.push\_back(slate);

    // Base Case - Leaf Node

    else if (start == input.size() || current > target)

        return;

    else {

        int c=1, i=start;

        while( (i < input.size()-1) && (input[i] == input[++i])) c++;

        for(i=0; i<= c;i++) {

            helper(input, start+c, slate, current + i \* input[start], target, res);

            slate.push\_back(input[start]);

        }

        for(i=0; i<=c;i++) {

            slate.pop\_back();

        }

    }

}

vector<vector<int>> generate\_all\_combinations(vector<int> &arr, int target) {

    vector<vector<int>> res;

    vector<int> slate;

    sort(arr.begin(), arr.end());

    helper(arr, 0, slate, 0, target, res);

    return res;

}

### Palindromic Decomposition Of A String

Find all palindromic decompositions of a given string s.

A palindromic decomposition of string is a decomposition of the string into substrings, such that all those substrings are valid palindromes.

Example

{

"s": "abracadabra"

}

Output:

["a|b|r|a|c|ada|b|r|a", "a|b|r|aca|d|a|b|r|a", "a|b|r|a|c|a|d|a|b|r|a"]

bool isPalindrome(string input) {

    int l = 0,  r = input.length()-1;

    while( l < r) {

        if(input[l++] != input[r--]){

            return false;

        }

    }

    return true;

}

void helper(string input, int idx, string &slate, string &substring, vector<string> &res){

    cout << idx << "  " << input.length() << " " << slate << " " << substring << endl;

    if (idx == input.length()) {

        // cout << "return" << endl;

        if (isPalindrome(substring))

            res.push\_back(slate);

        return;

    }

    else {

        slate.push\_back(input[idx]);

        substring.push\_back(input[idx]);

        helper(input, idx+1, slate, substring,  res);

        slate.pop\_back();

        substring.pop\_back();

        cout << "Checking substr " << substring << endl;

        if (isPalindrome(substring)) {

            string prev(substring);

            slate.push\_back('|');

            slate.push\_back(input[idx]);

            substring = input[idx];

            helper(input, idx+1, slate, substring, res);

            slate.pop\_back();

            slate.pop\_back();// for '/'

            substring.pop\_back();

            substring = prev;

        } else {

            cout << "Rejecting " << substring << endl;

        }

    }

}

vector<string> driver(string s) {

    vector<string> res;

    string slate;

    string substr;

    slate.push\_back((char)s[0]);

    substr.push\_back((char)s[0]);

    helper(s, 1, slate, substr, res);

    return res;

}

# Trees

## Binary Trees

### Search in a Binary Search Tree – Leet Code 700

<https://leetcode.com/problems/search-in-a-binary-search-tree/description/>

You are given the root of a binary search tree (BST) and an integer val.

Find the node in the BST that the node's value equals val and return the subtree rooted with that node. If such a node does not exist, return null.

TreeNode\* searchBST(TreeNode\* root, int val) {

    TreeNode\* cur = root;

    while(cur) {

        if (val == cur->val)

            return cur;

        else if (val > cur->val)

            cur = cur->right;

        else

            cur = cur->left;

    }

    return nullptr;

}

TC: O(logn)

SC: O(1)

### Insert into a Binary Search Tree – Leet Code 701

https://leetcode.com/problems/insert-into-a-binary-search-tree/description/

You are given the root node of a binary search tree (BST) and a value to insert into the tree. Return the root node of the BST after the insertion. It is guaranteed that the new value does not exist in the original BST.

Notice that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return any of them.

TreeNode\* insertIntoBST(TreeNode\* root, int val) {

    if (root == nullptr) {

        return new TreeNode(val);

    } else {

        TreeNode \*prev;

        TreeNode \*cur = root;

        while(cur) {

            prev = cur;

            if (val > cur->val)

                cur = cur->right;

            else

                cur = cur->left;

        }

        if(val > prev->val)

            prev->right = new TreeNode(val);

        else

            prev->left = new TreeNode(val);

        return root;

    }

    return root;

}

### Min/Max in BST

int min(TreeNode \*root) {

    TreeNode \*cur = root;

    while (cur->left)

        cur = cur->left;

    return cur->val;

}

int max(TreeNode \*root) {

    TreeNode \*cur = root;

    while (cur->right)

        cur = cur->right;

    return cur->val;

}

### Delete Node in a BST – Leet Code 450

<https://leetcode.com/problems/delete-node-in-a-bst/description/>

TreeNode\* deleteNode(TreeNode\* root, int key) {

    TreeNode \*cur = root;

    TreeNode \*prev = nullptr;

    // Search for node

    while (cur) {

        if (key == cur->val)

            break;

        if (key > cur->val) {

            prev = cur;

            cur = cur->right;

        }

        else {

            prev = cur;

            cur = cur->left;

        }

    }

    // If not found or empty tree

    if (cur == nullptr)

        return root;

    // Leaf node to be deleted

    if (!cur->left && !cur->right) {

        cout << "Leaf" << endl;

        // Only root node exist and needs to be deleted

        if (prev == nullptr)

            return nullptr;

        if (key < prev->val)

            prev->left = nullptr;

        else

            prev->right = nullptr;

        return root;

    }

    // Node with one child need to be deleted

    if ((cur->left && !cur->right) || (cur->right && !cur->left)) {

        TreeNode \*child = nullptr;

        if(cur->left)

            child = cur->left;

        else

            child = cur->right;

        // Need to delete root with one child

        if (prev == nullptr)

            return child;

        if (key < prev->val)

            prev->left = child;

        else

            prev->right = child;

        return root;

    }

    // Delete node with both the childs

    if (cur->left && cur->right) {

        TreeNode \*temp = cur->right;

        TreeNode \*last = cur;

        while(temp->left) {

            last = temp;

            temp = temp->left;

        }

        cur->val = temp->val;

        if (temp == last->right)

            last->right = temp->right;

        if (temp == last->left)

            last->left = temp->right;

        return root;

    }

    return root;

}

### Iterative Postorder Traversal - Geeks for Geeks

https://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/

vector<int> postorder\_traversal(BinaryTreeNode \*root) {

    vector<int> res;

    stack<BinaryTreeNode\*> st;

    st.push(root);

    while(!st.empty()) {

        BinaryTreeNode \*temp = st.top();

        if (temp->left) {

            st.push(temp->left);

            temp->left = nullptr;

        } else if (temp->right) {

            st.push(temp->right);

            temp->right = nullptr;

        } else {

            res.push\_back(temp->val);

            st.pop();

        }

    }

    return res;

}

### Binary Search Tree Iterator – Leet Code 173

<https://leetcode.com/problems/binary-search-tree-iterator/description/>

class BSTIterator {

public:

    queue<int> q;

    void inorder(TreeNode \*root) {

        if (root == nullptr) {

            return;

        }

        else {

            inorder(root->left);

            q.push(root->val);

            inorder(root->right);

        }

    }

    BSTIterator(TreeNode\* root) {

        inorder(root);

    }

    int next() {

        if (q.empty())

            return 0;

        else {

            int ret = q.front();

            q.pop();

            return ret;

        }

    }

    bool hasNext() {

        return !q.empty();

    }

};

## BFS

### Binary Tree Level Order Traversal – Leet Code 102

<https://leetcode.com/problems/binary-tree-level-order-traversal/description/>

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

vector<vector<int>> levelOrder(TreeNode\* root) {

    vector<vector<int>> res;

    if (root == nullptr)

        return res;

    queue<TreeNode\*> q;

    q.push(root);

    while(!q.empty()) {

        int c = q.size();

        vector<int> curLevel;

        while(c-- > 0) {

            TreeNode\* temp = q.front();

            q.pop();

            curLevel.push\_back(temp->val);

            if(temp->left)

                q.push(temp->left);

            if(temp->right)

                q.push(temp->right);

        }

        res.push\_back(curLevel);

    }

    return res;

}

### N-ary Tree Level Order Traversal – Leet Code 429

<https://leetcode.com/problems/n-ary-tree-level-order-traversal/description/>

Given an n-ary tree, return the level order traversal of its nodes' values.

Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

vector<vector<int>> levelOrder(Node\* root) {

    vector<vector<int>> res;

    queue<Node\*> q;

    if (root == nullptr)

        return res;

    q.push(root);

    while(!q.empty()) {

        int c = q.size();

        vector<int> curLevelNodes;

        while(c-- > 0) {

            Node \*temp = q.front();

            q.pop();

            curLevelNodes.push\_back(temp->val);

            for (auto it:temp->children)

                q.push(it);

        }

        res.push\_back(curLevelNodes);

    }

    return res;

}

### Binary Tree Level Order Traversal II – Leet Code 107

<https://leetcode.com/problems/binary-tree-level-order-traversal-ii/>

Given the root of a binary tree, return the bottom-up level order traversal of its nodes' values. (i.e., from left to right, level by level from leaf to root).

vector<vector<int>> levelOrderBottom(TreeNode\* root) {

    queue<TreeNode\*> q;

    vector<vector<int>> res;

    if (root == nullptr)

        return res;

    q.push(root);

    while(!q.empty()) {

        int c = q.size();

        vector<int> nodes;

        while(c-- > 0 ) {

            TreeNode \*temp = q.front();

            q.pop();

            nodes.push\_back(temp->val);

            if (temp->left)

                q.push(temp->left);

            if (temp->right)

                q.push(temp->right);

        }

        res.push\_back(nodes);

    }

    reverse(res.begin(), res.end());

    return res;

}

## Binary Tree Right Side View – Leet Code 199

<https://leetcode.com/problems/binary-tree-right-side-view/>

Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

vector<int> rightSideView(TreeNode\* root) {

    if (root == nullptr)

        return {};

    vector<int> res;

    queue<TreeNode\*> q;

    q.push(root);

    while(!q.empty()) {

        int c = q.size();

        while(c){

            TreeNode \*temp = q.front();

            q.pop();

            if (c == 1)

                res.push\_back(temp->val);

            if (temp->left != nullptr)

                q.push(temp->left);

            if (temp->right != nullptr)

                q.push(temp->right);

            c--;

        }

    }

    return res;

}

### Binary Tree Zigzag Level Order Traversal – Leet Code 103

<https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/>

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

    queue<TreeNode\*> q;

    vector<vector<int>> res;

    if (root == nullptr)

        return res;

    q.push(root);

    bool flag = false;

    while(!q.empty()) {

        int c = q.size();

        vector<int> nodes;

        while(c--) {

            TreeNode\* temp = q.front();

            q.pop();

            nodes.push\_back(temp->val);

            if (temp->left)

                q.push(temp->left);

            if (temp->right)

                q.push(temp->right);

        }

        if (flag)

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

        flag = !flag;

        res.push\_back(nodes);

    }

    return res;

}

Reverse operation can be avoided by creating nodes with size and filling it from front or back

### Convert a Binary Tree into its Mirror Tree – Geeks for Geeks

<https://www.geeksforgeeks.org/write-an-efficient-c-function-to-convert-a-tree-into-its-mirror-tree/>

void mirror\_image(BinaryTreeNode \*root) {

    queue<BinaryTreeNode \*> q;

    if (root == nullptr)

        return;

    q.push(root);

    while (!q.empty()) {

        int c = q.size();

        while(c--) {

            BinaryTreeNode \*temp = q.front();

            q.pop();

            swap(temp->left, temp->right);

            if(temp->left)

                q.push(temp->left);

            if(temp->right)

                q.push(temp->right);

        }

    }

}

### Clone Binary Tree – Leet Code 1379

<https://leetcode.com/problems/find-a-corresponding-node-of-a-binary-tree-in-a-clone-of-that-tree/description/>

TreeNode\* getTargetCopy(TreeNode\* original, TreeNode\* cloned, TreeNode\* target) {

    if (!original || !cloned)

        return nullptr;

    queue<TreeNode \*> q;

    queue<TreeNode \*> cq;

    q.push(original);

    cq.push(cloned);

    while(!q.empty()) {

        int c = q.size();

        while(c--) {

            TreeNode \*temp = q.front();

            q.pop();

            TreeNode \*ctemp = cq.front();

            cq.pop();

            if (target == temp)

                return ctemp;

            if(temp->left) {

                q.push(temp->left);

                cq.push(ctemp->left);

            }

            if(temp->right) {

                q.push(temp->right);

                cq.push(ctemp->right);

            }

        }

    }

    return nullptr;

}

BinaryTreeNode \*clone\_tree(BinaryTreeNode \*root) {

    if (root == nullptr)

        return nullptr;

    queue<BinaryTreeNode \*> q;

    queue<BinaryTreeNode \*> cq;

    BinaryTreeNode \*clone = new BinaryTreeNode(root->value);

    q.push(root);

    cq.push(clone);

    while(!q.empty()) {

        int c = q.size();

        while(c--) {

            BinaryTreeNode \*temp = q.front();

            q.pop();

            BinaryTreeNode \*ctemp = cq.front();

            cq.pop();

            if(temp->left) {

                ctemp->left = new BinaryTreeNode(temp->left->value);

                q.push(temp->left);

                cq.push(ctemp->left);

            }

            if(temp->right) {

                ctemp->right = new BinaryTreeNode(temp->right->value);

                q.push(temp->right);

                cq.push(ctemp->right);

            }

        }

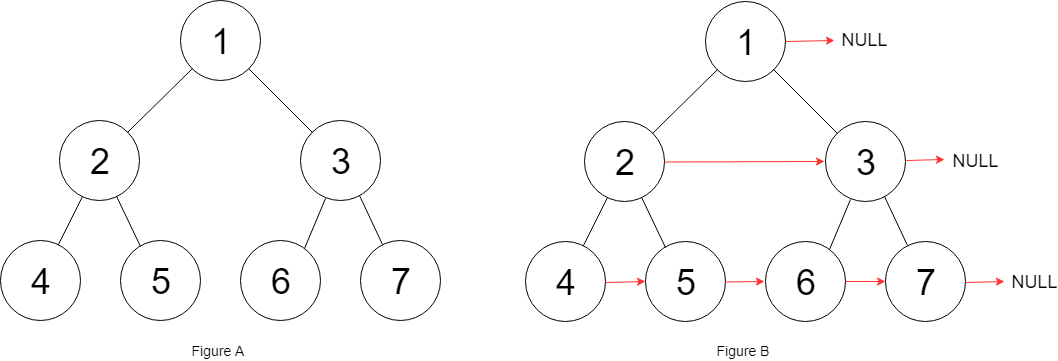
    }

    return clone;

}

### Populating Next Right Pointers in Each Node – Leet Code 116

<https://leetcode.com/problems/populating-next-right-pointers-in-each-node/description/>



Node\* connect(Node\* root) {

    if (root == nullptr)

        return nullptr;

    queue<Node \*> q;

    q.push(root);

    while(!q.empty()) {

        int c = q.size();

        Node \*prev = nullptr;

        while(c--) {

            Node \*temp = q.front();

            q.pop();

            if (prev)

                prev->next = temp;

            prev = temp;

            if(temp->left)

                q.push(temp->left);

            if(temp->right)

                q.push(temp->right);

        }

    }

    return root;

}

## DFS – Depth first search

**Template for next set of problems**

Function dfs(Node) :

#Base Case

If (node->left is null and node->right is null)

// Process Leaf Node

Return

#Recursive

If (node->left)

dfs(node->left);

if (node->right)

dfs(node->right);

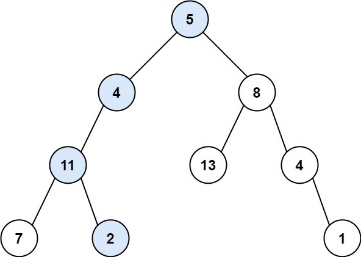
### Path Sum – Leet Code 112

<https://leetcode.com/problems/path-sum/description/>

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

Output: true

Explanation: The root-to-leaf path with the target sum is shown.

void helper(TreeNode\* root, int targetSum, bool &res) {

    if (!res && !root->left && !root->right) {

        if (targetSum == root->val)

            res =  true;

    } else {

        if (root->left)

            helper(root->left, targetSum - root->val, res);

        if (root->right)

            helper(root->right, targetSum - root->val, res);

    }

}

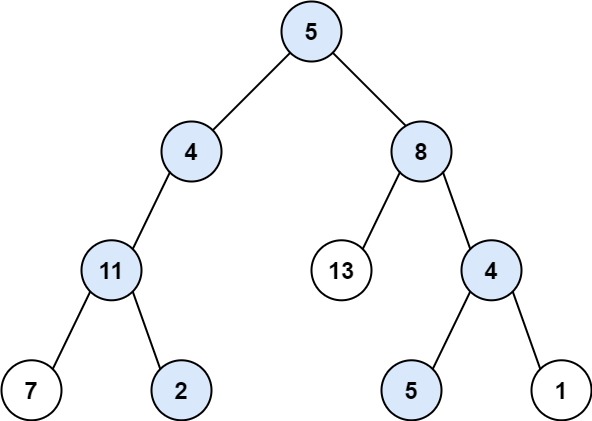
### Path Sum II – Leet Code 113

<https://leetcode.com/problems/path-sum-ii/description/>

Given the root of a binary tree and an integer targetSum, return all root-to-leaf paths where the sum of the node values in the path equals targetSum. Each path should be returned as a list of the node values, not node references.

A root-to-leaf path is a path starting from the root and ending at any leaf node. A leaf is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

Output: [[5,4,11,2],[5,8,4,5]]

Explanation: There are two paths whose sum equals targetSum:

5 + 4 + 11 + 2 = 22

5 + 8 + 4 + 5 = 22

void helper(TreeNode\* root, int targetSum, vector<int> &path, vector<vector<int>> &res) {

    if (!root->left && !root->right) {

        path.push\_back(root->val);

        if (root->val == targetSum) {

            res.push\_back(path);

        }

        path.pop\_back();

    } else {

        if (root->left) {

            path.push\_back(root->val);

            helper(root->left,targetSum - root->val,path,res);

            path.pop\_back();

        }

        if (root->right) {

            path.push\_back(root->val);

            helper(root->right,targetSum - root->val,path,res);

            path.pop\_back();

        }

    }

}

### Diameter of Binary Tree – Leet Code 543

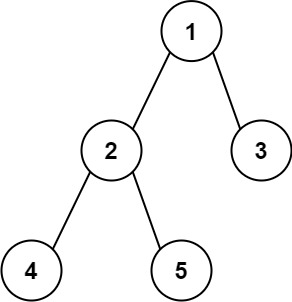
<https://leetcode.com/problems/diameter-of-binary-tree/>

Given the root of a binary tree, return the length of the diameter of the tree.

The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The length of a path between two nodes is represented by the number of edges between them.

Example 1:



Input: root = [1,2,3,4,5]

Output: 3

Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

int helper(TreeNode \*node, int &res) {

    if (!node->left && !node->right)

        return 0;

    else {

        int left = 0,right = 0 , temp = 0;

        if (node->left) {

            left = helper(node->left,res);

            temp += (left +1);

        }

        if (node->right) {

            right = helper(node->right,res);

            temp += (right +1);

        }

        res = max(temp, res);

        return (max(left, right) + 1);

    }

}

Each node in a way returns 2 values, max diameter and longest length of one side.

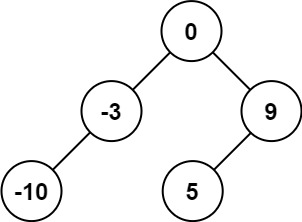
### Convert Sorted Array to Binary Search Tree – Leet Code 108

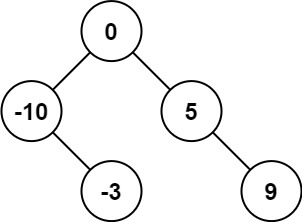
<https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/>

Given an integer array nums where the elements are sorted in ascending order, convert it to a

height-balanced binary search tree.

Example 1:





Input: nums = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: [0,-10,5,null,-3,null,9] is also accepted:

TreeNode\* helper(vector<int>& nums, int s, int e) {

    if (s > e)

        return nullptr;

    else if (s == e)

        return new TreeNode(nums[s]);

    else {

        int mid = (s + e)/2;

        TreeNode\* root = new TreeNode(nums[mid]);

        root->left = helper(nums, s, mid-1);

        root->right = helper(nums, mid+1, e);

        return root;

    }

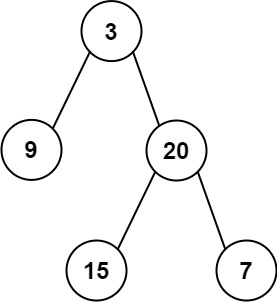
}

### Construct Binary Tree from Inorder and Postorder Traversal – Leet Code 106

<https://leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/>

Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return the binary tree.

Example 1:

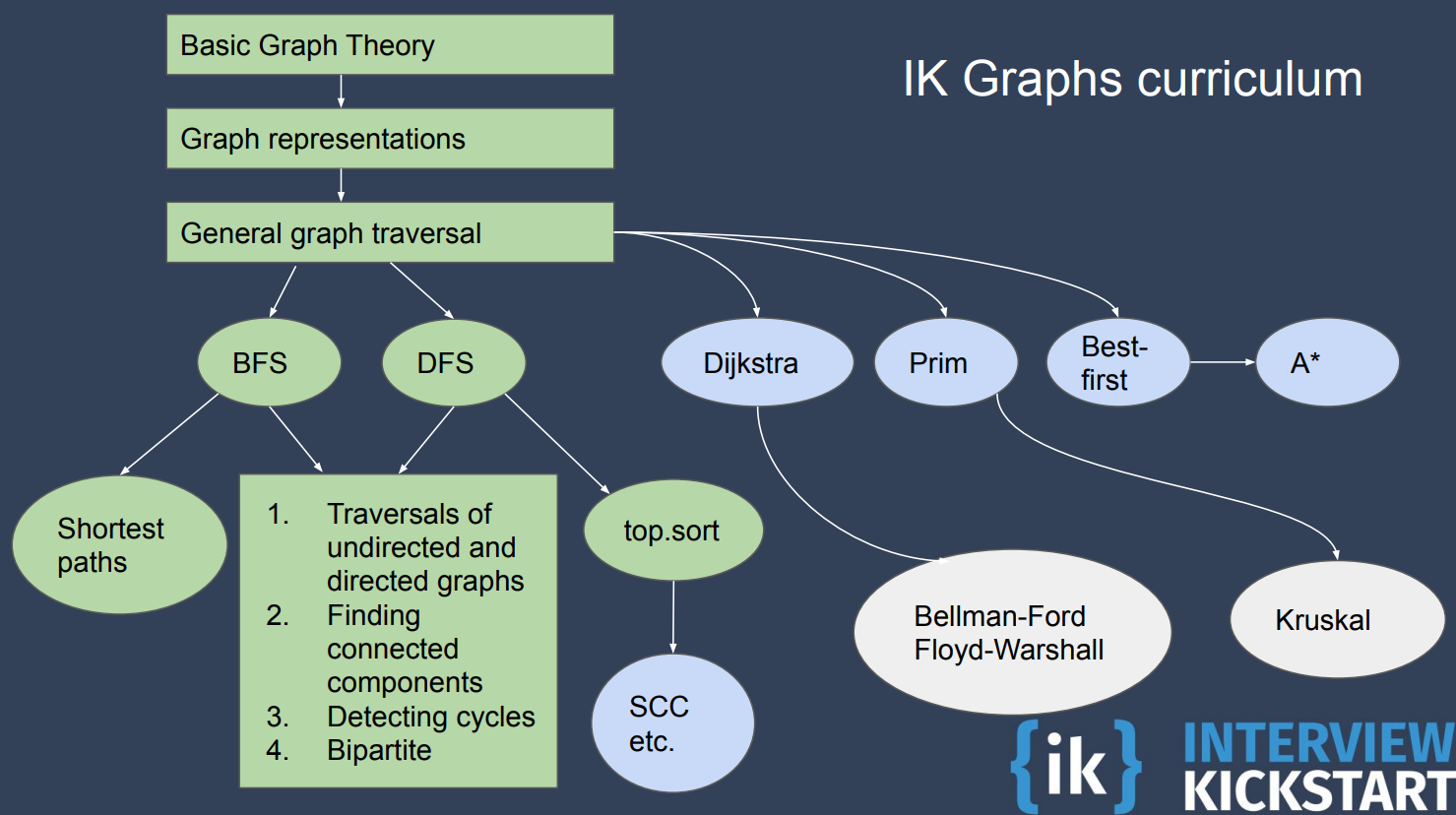


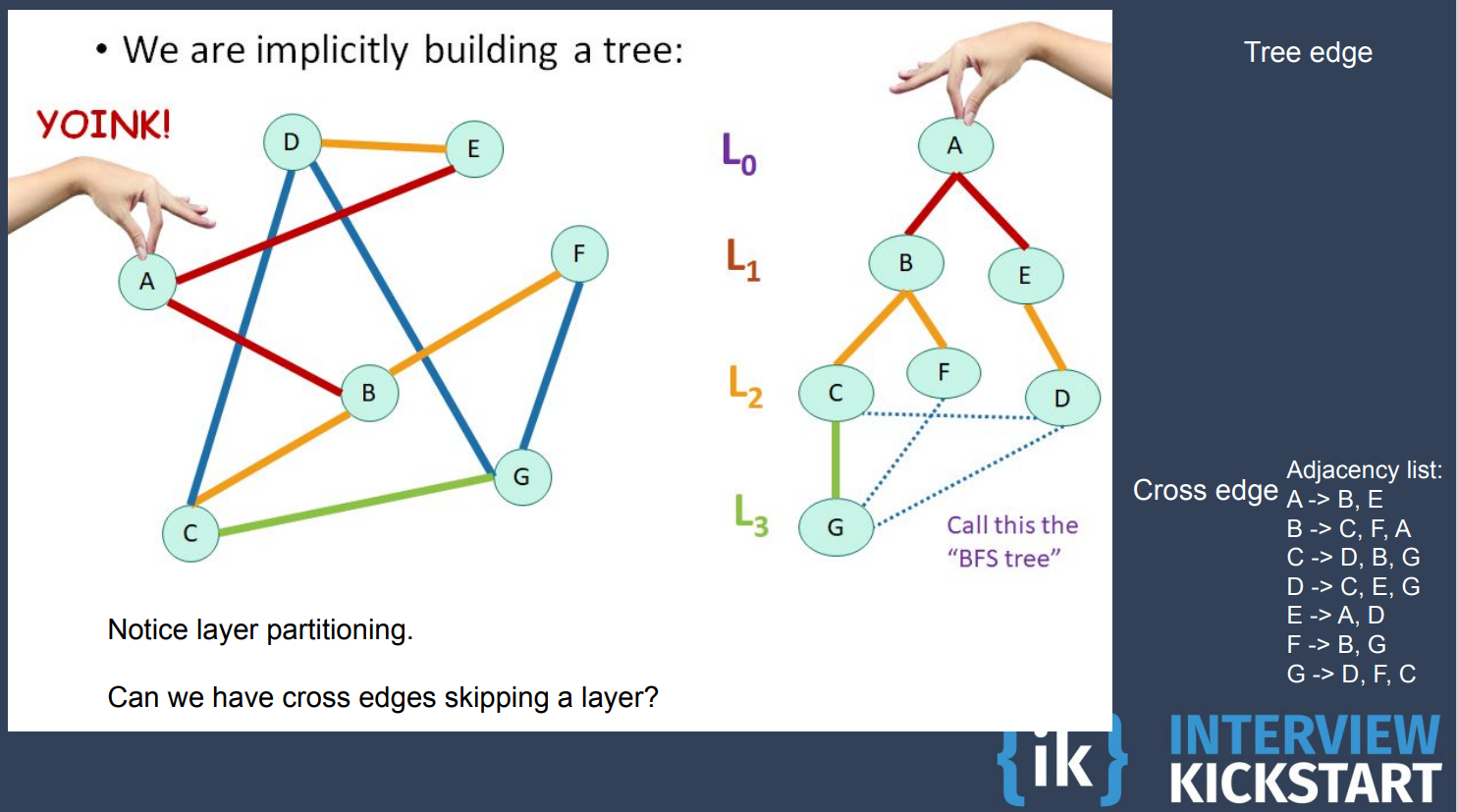
Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]

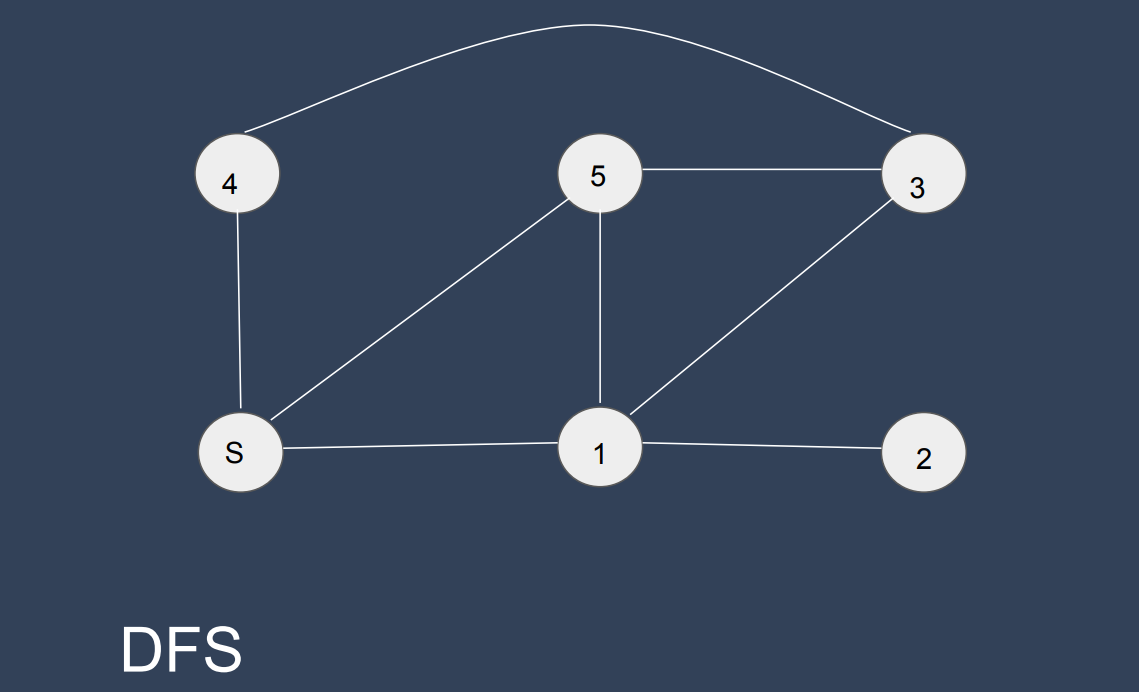
Output: [3,9,20,null,null,15,7]

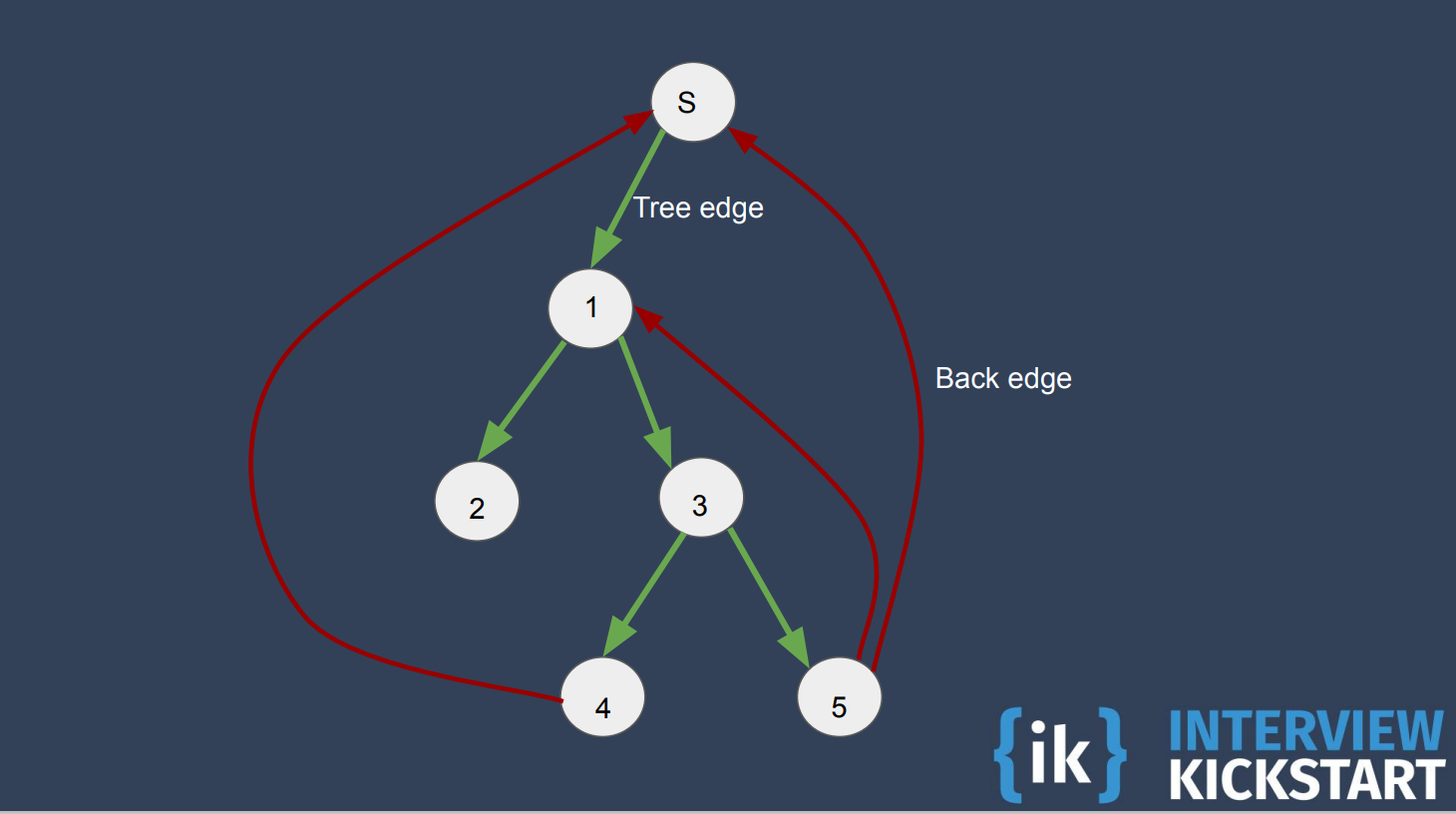
# Graphs

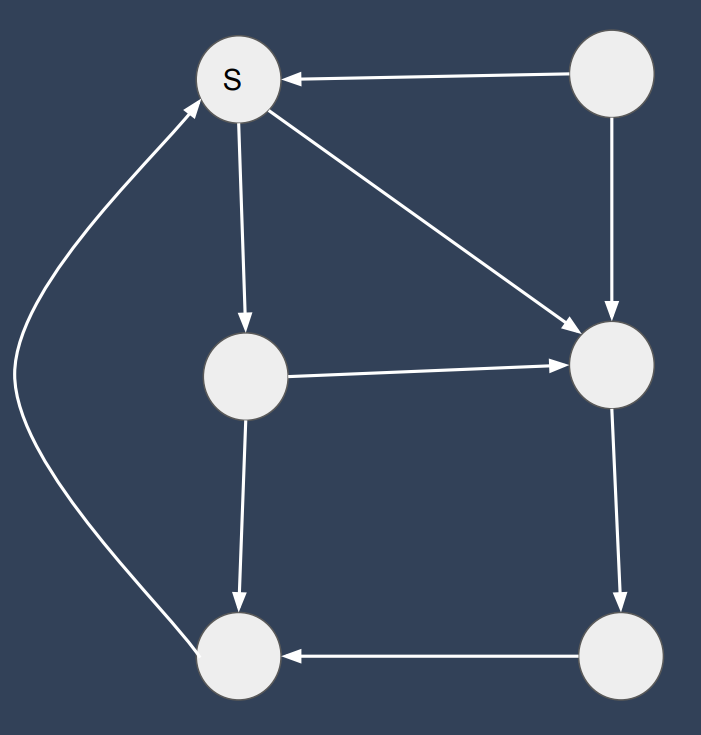
## Theory

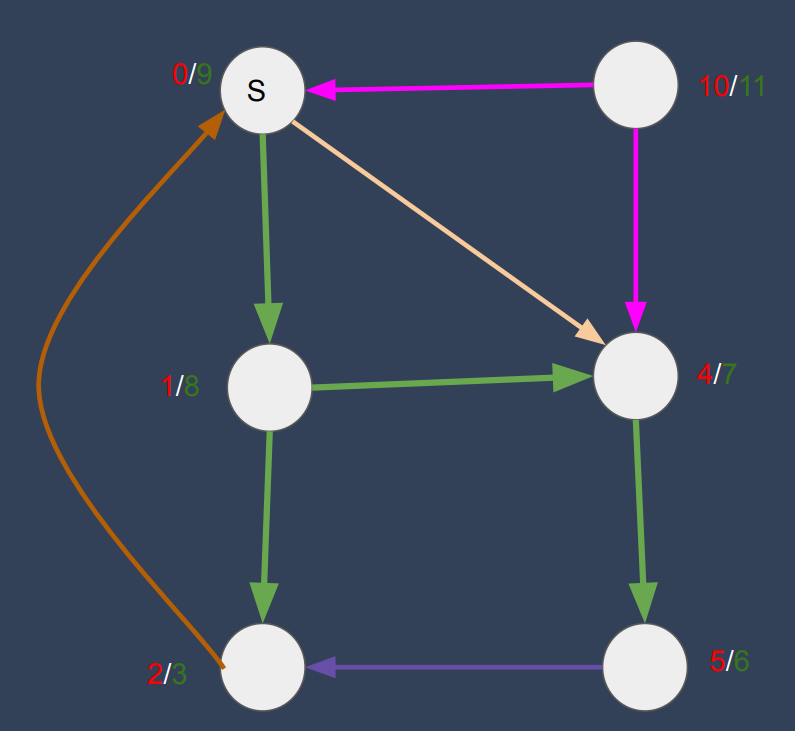


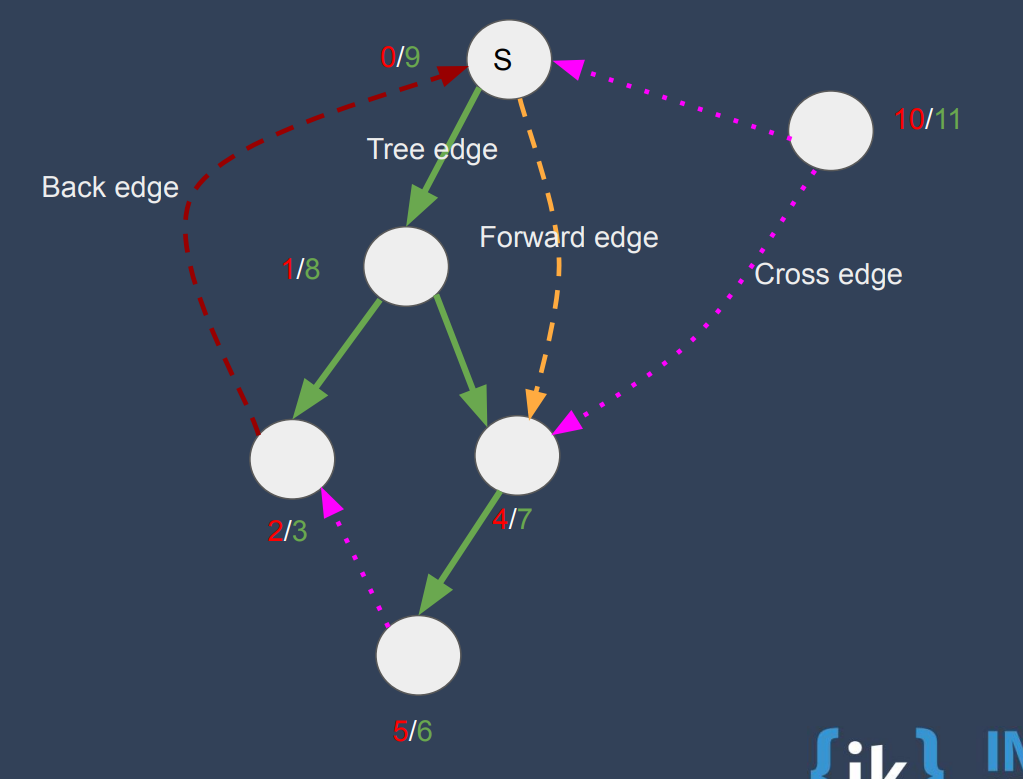


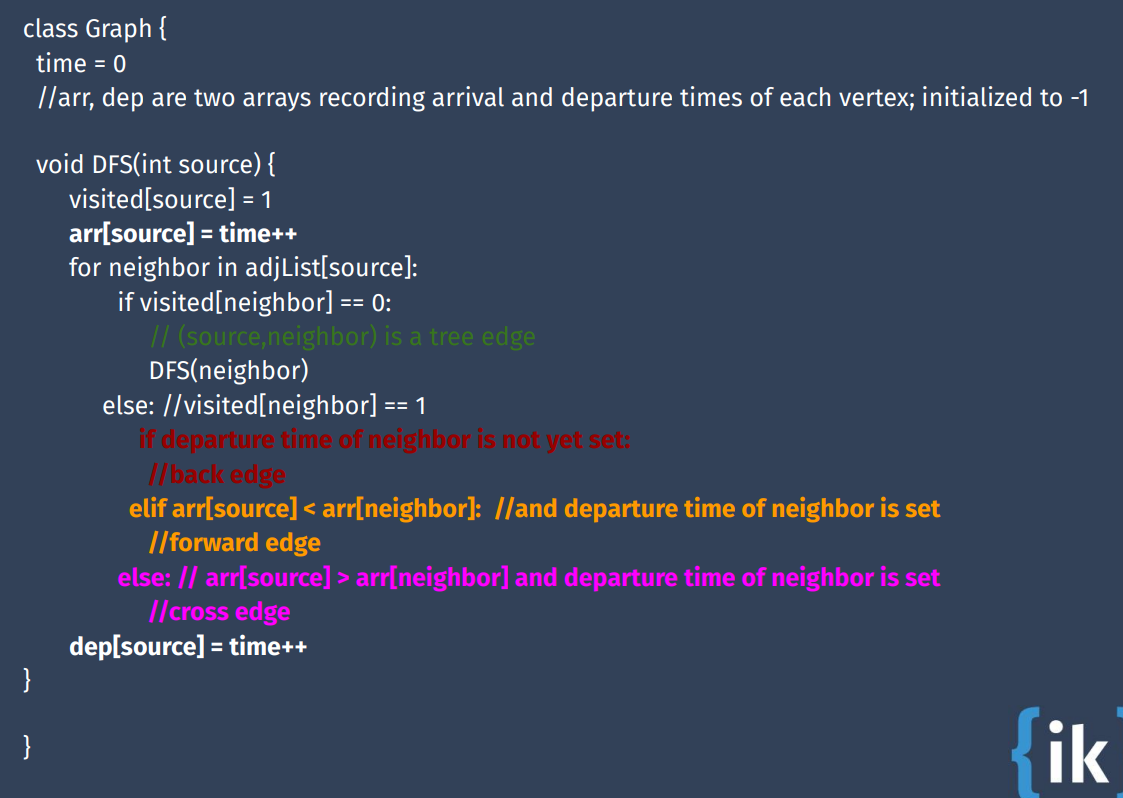












# Dynamic Programming

## Top Down – Memoization

### Climbing Stairs – Leet Code 70

<https://leetcode.com/problems/climbing-stairs/description/>

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

int helper(int n, map<int,int> &ways) {

    if (ways.find(n) == ways.end())

        ways[n] = helper(n-1, ways) + helper(n-2, ways);

    return ways[n];

}

int climbStairs(int n) {

    map<int,int> ways;

    ways[1] = 1;

    ways[2] = 2;

    return helper(n, ways);

}

### N Choose K – IK

<https://uplevel.interviewkickstart.com/resource/submissions/rc-codingproblem-404914-841138-1109-6738>

int mod = 1000000007;

int helper(int n, int r, map<pair<int,int>, int> &ncr) {

    if (n == r || r == 0)

        return 1;

    if (ncr.find(make\_pair(n,r)) == ncr.end())

        ncr[make\_pair(n,r)] = (helper(n-1,r, ncr) + helper(n-1, r-1, ncr))% mod;

    return ncr[make\_pair(n,r)];

}

int ncr(int n, int k) {

    if (k > n)

        return 0;

    map<pair<int,int>, int> nck;

    // Write your code here.

    return helper(n, k, nck);

}

### Unique Paths – Leet Code 62

https://leetcode.com/problems/unique-paths/description/

There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The test cases are generated so that the answer will be less than or equal to 2 \* 109.

int helper(int m, int n, map<pair<int, int>, int> &ways) {

    if (m <= 0 || n <= 0)

        return 1;

    if (ways.find(make\_pair(m,n)) == ways.end())

        ways[make\_pair(m,n)] = (helper(m-1, n, ways) + helper(m, n-1, ways)) ;

    return ways[make\_pair(m,n)];

}

int uniquePaths(int m, int n) {

    map<pair<int, int>, int> ways;

    return helper(m-1, n-1, ways);

}

### Minimum Path Sum– Leet Code 64

<https://leetcode.com/problems/minimum-path-sum/description/>

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

int helper(int m, int n, vector<vector<int>>& grid, map<pair<int,int>, int> &minSum ) {

    if (m == 0 && n == 0)

        return grid[m][n];

    if (m < 0 || n < 0)

        return INT\_MAX;

    if (minSum.find(make\_pair(m,n)) == minSum.end())

        minSum[make\_pair(m,n)] =  min(helper(m-1, n, grid, minSum), helper(m, n-1, grid, minSum)) + grid[m][n];

    return minSum[make\_pair(m,n)];

}

int minPathSum(vector<vector<int>>& grid) {

    map<pair<int,int>, int> minSum;

    return helper(grid.size()-1, grid[0].size()-1, grid, minSum);

}

### Min Cost Climbing Stairs – Leet Code 746

https://leetcode.com/problems/min-cost-climbing-stairs/description/

You are given an integer array cost where cost[i] is the cost of ith step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index 0, or the step with index 1.

Return the minimum cost to reach the top of the floor.

int helper(int d, vector<int> &cost, map<int, int> &minCost) {

    if (d == 0 || d == 1)

        return cost[d];

    if (minCost.find(d) == minCost.end())

        minCost[d] = cost[d] + min (helper(d-1, cost, minCost), helper(d-2, cost, minCost));

    return minCost[d];

}

int minCostClimbingStairs(vector<int>& cost) {

    cost.push\_back(0);

    map<int, int> minCost;

    return helper(cost.size()-1, cost, minCost);

}

**Pushing back 0 in the end of cost vector helps, as we can treat it as destination**

### Fdlsa’

### Lfds’a

## Bottom Up - Tabulation

### Climbing Stairs – Leet Code 70

<https://leetcode.com/problems/climbing-stairs/description/>

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

int climbStairs(int n) {

    int ways[3];

    ways[1] = 1;

    ways[2] = 2;

    for (int i = 3; i <= n; i++)

        ways[i%3] = ways[(i-1)%3] + ways[(i-2)%3];

    return ways[n%3];

}

### N Choose K – IK



int mod = 1000000007;

int ncr(int n, int k) {

    if (k > n)

        return 0;

    int tab[n+1][k+1] = {0};

    for (int r = 0; r <=n ; r++)

        tab[r][0] = 1;

    for(int c = 0; c <= k; c++)

        tab[c][c] = 1;

    for (int r = 2; r <=n ; r++) {

        for(int c = 1; c <= min(r,k); c++)

            tab[r][c] = (tab[r-1][c] + tab[r-1][c-1]) % mod;

    }

    return tab[n][k];

}

### Unique Paths – Leet Code 62

<https://leetcode.com/problems/unique-paths/description/>

https://leetcode.com/problems/unique-paths/description/

There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The test cases are generated so that the answer will be less than or equal to 2 \* 109.

int uniquePaths(int m, int n) {

    vector<vector<int>> ways(m, vector<int> (n));

    for (int row = 0; row < m; row++) {

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

            if(row == 0 || col == 0) {

                ways[row][col] = 1;

                continue;

            }

            ways[row][col] = ways[row-1][col] + ways[row][col-1];

        }

    }

    return ways[m-1][n-1];

}

### Minimum Path Sum – Leet Code 64

<https://leetcode.com/problems/minimum-path-sum/description/>

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

int minPathSum(vector<vector<int>>& grid) {

    int m = grid.size();

    int n = grid[0].size();

    vector<vector<int>> minSum(m, vector<int>(n,0));

    minSum[0][0] = grid[0][0];

    for(int row = 1; row < m; row++)

        minSum[row][0] = grid[row][0] + minSum[row-1][0];

    for(int col = 1; col < n; col++)

        minSum[0][col] = grid[0][col] + minSum[0][col-1];

    for(int row = 1; row < m; row++) {

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

            minSum[row][col] =  grid[row][col] + min (minSum[row][col-1] , minSum[row-1][col]);

        }

    }

    return minSum[m-1][n-1];

}

### Min Cost Climbing Stairs – Leet Code 746

https://leetcode.com/problems/min-cost-climbing-stairs/description/

You are given an integer array cost where cost[i] is the cost of ith step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index 0, or the step with index 1.

Return the minimum cost to reach the top of the floor.

int minCostClimbingStairs(vector<int>& cost) {

    vector<int> minCost (cost.size()+1);

    minCost[0] = cost[0];

    minCost[1] = cost[1];

    cost.push\_back(0);

    for(int i = 2; i < minCost.size(); i++)

        minCost[i] = cost[i] + min(minCost[i-1],minCost[i-2]);

    return minCost[minCost.size()-1];

}

### **Pushing back 0 in the end of cost vector helps, as we can treat it as destination**

# Notes on library functions

### Vectors

push\_back – Adds in the end

front() – Elements from front side,

back() – Elements from back

pop\_back() – remove elements from back

erase() – Can erase any element

2d Vector Init

Vector<vector<int>> x(m, vector<int>(n,0));

Vector<vector<string>> x(m, string(n,’Q’));

Min and Max

std::vector<int> v = {2, 1, 3, 6, 7, 9, 8};

int max = \*max\_element(v.begin(), v.end());

int min = \*min\_element(v.begin(), v.end());

### Queues - FIFO

Push() – add elements

Front() – First Item

Pop() – Remove first item

### Stacks - LIFO

Push() – add elements

Top() – Top Item

Pop() – Remove top item

### Priority Queue

priority\_queue<Node> maxPriorityQueue;

priority\_queue<Node, vector<Node>, greater<Node>> minHeap;

minHeap.push()

minHeap.top()

minHeap.pop()

### Maps

map<char, int> mymap;

    mymap['a'] = 1;

    for (auto it = mymap.begin();it != mymap.end(); ++it)

        cout << it->first << " = "<< it->second << '\n';

### Sets

Multiset – allows duplicates

Set- unique values

Insert(),

Find()

Erase()

Iterator arithmatics:

auto it = currentK.begin();

advance(it, k/2-1);

int x = \*it;

advance(it, 1);

set. lower\_bound(ele) 🡺 returns iterator of first occurrence of element

### Pairs

### Strings

Use length() instead of size()

Creation

String temp(size, char) – Initializes with char of Size

Additions

str + “0”

str.append(“”, number of chars)

str.push\_back(**char**)

Deletions

Str.erase() : pos, str.begin(), **str.end() – 1 (-1 since null being last character)**

Str.pop\_back()

isdigit(str[i])

(char)toupper(str[i]), (char)tolower(str[i])