Basic Java

- Bit Manipulation

  - ```
    my_byte = my_byte | (1 << pos);
    ```

- What is the difference between ArrayList and LinkedList
  - `LinkedList` and `ArrayList` are two different implementations of the List interface.
  - `LinkedList` implements it with a doubly-linked list. `ArrayList` implements it with a dynamically re-sizing array.
- What is interface
  - Interface is a group of methods that classes must implement. When implementing interface, all methods should be developed to successfully compile.
- What is final keyword
  - final means cannot be changed.
    - final class: cannot be subclassed, all methods in a final class are final
    - final method: cannot overridden by subclasses
    - final variable: can only be initialized once, its value cannot be changed after initialize
- What are implement and extends keywords
  - `extends` is for *extending* a class
  - `implements` is for *implementing* an interface
- What is `static` keyword
  - means only one copy of the method/variable shared between all instances.

  - ```
    public class MyClass {
        public static int myVariable = 0;
    }

    //Now in some other code creating two instances of MyClass
    //and altering the variable will affect all instances

    MyClass instance1 = new MyClass();
    MyClass instance2 = new MyClass();

    MyClass.myVariable = 5;  //This change is reflected in both instances
    ```

- What is heap tree?
  - Heap is a tree structure, it's a complete tree: which means totally filled other than the

rightmost leaf node can be empty. Heap can be divided as min-heap, and max-heap, for max-heap, the parent node is bigger than any of its child nodes.

- What is heap can do but stack cannot do?
    - *All* Java objects are dynamically allocated.
    - `ClassA obj = new ClassA();` the object is allocated on the heap and a reference to it is stored on the stack
- Compare between stack and heap
    - Stack is the memory set apart as spaces for a thread of execution. Stack frames are created in stack for functions. It obeys LIFO: last in first out.
    - Heap is memory that dynamically located. No enforced pattern for allocating / deallocating from the heap.
    - Usually each thread get a stack but only one heap overall. Scope:
        - Stack created and ended when thread create / end
        - Heap created and ended when appication create / end
    - Heap is slower than stack because stack's size is fixed when created and LIFO is simple. Heap is slow because dynamic allocation and synchronized to ensure multi-thread safe as it's a global accessble.
- How to implement Hashmap
    - HashMap maintains an array of buckets. Each bucket is a linkedlist of key value pairs encapsulated as Entry objects

      This array of buckets is called table. Each node of the linked list is an instance of a private class called Entry. An entry is a private static class inside HashMap which implements Map.Entry
- How to solve collision in Hashmap
    - One of the solution is: **Chaining collision resolution**. Collision happens when two keys are mapped to the same bucket in HashMap. In an `HashMap` the key is an object, that contains `hashCode()` and `equals(Object)` methods. Two process: one is use hashCode() method to get the bucket number, then if the bucket already exists keys, it will use equals method to judge if the new key is the same with already existed ones, if yes, replace, if no, add a new entry to the linked list
- Difference between Hashmap & Hashset & Hashtable
    - **HashSet**: does **not allow duplicate values**, has add method / contains method
    - **HashMap** & **HashTable**: are for saving **key-value pairs**.
        - HashTable does not allow null for key / value - HashMap allows null
        - HashTable is **synchronized**, only one thread can access in one time, so it's thread safe, but slow - HashMap is NOT sychronized, fast, but not thread safe
- HashMap resize

    - Size: 15, load factor: 0.75, when the buckets number that used is larger than size*loadFactor, the hashMap will expand to double size and recalculate all hashCode for every key already exists in the hashMap.

- GC (Garbage Collection):

- What does java uses for sort?

    - For primitive type: quick sort
    - For objects: use merge sort
    - Why different? For quick sort, even the value is the same, there might be exchange among these items, for primitive, it's OK but for reference types, it causes problem; Why not use merge sort for primitive? because merge sort requires making a clone of the array, for objects, the references takes only little memory, but for primitive types, the memory cost doubles.

- Arrays.sort 降序： `Arrays.sort(companies, Collections.reverseOrder());`

- How to use array & queue & linked list to implement stack

    - Stack: LIFO(last in first out)

    - Methods: pop(), push(), peek(), isEmpty()

```
//USE ARRAY TO IMPLEMENT STACK
public class MyStack<T> {
  // constructor
  public MyStack<T>() {
    private static final int capacity = 100;
    T[] arr = new T[capacity];
    int top = -1;
  }

  public T pop() {
    if(top == -1) throw new EmptyStackException();
    T item = arr[top];
    top--;
    return item;
  }

  public void push(T item) {
    if(top == capacity - 1) {
      throw new IllegalArgumentException("Stack overflow");
    } else {
      top++;
      arr[top] = item;
    }
  }

  public T peek() {
    if(top == -1) throw new EmptyStackException();
    T item = arr[top];
    return item;
  }

  public boolean isEmpty() {
    if(top == -1) return true;
    return false;
  }
}
```

- ```
  //USE QUEUE TO IMPLEMENT STACK, o(n) insert and o(1) remove
  Queue<Integer> qu = new LinkedList<Integer>();

  public void push(int x) {
    qu.add(x);
    int sz = qu.size();
    while (sz > 1) { // put all front to back
      qu.add(qu.remove());
      sz--;
    }
  }

  public int pop() {
    return qu.remove();
  }

  public int peek() {
    return qu.peek();
  }

  public int isEmpty() {
    return qu.isEmpty();
  }
  ```
  ```

- ```
  //USE LINKED LIST TO IMPLEMENT STACK
  public class MyStack<T> {
    private static class StackNode<T> {
      private T data;
      private StackNode next;
      public StackNode(T data) {
        this.data = data;
      }
    }

    private StackNode<T> top;

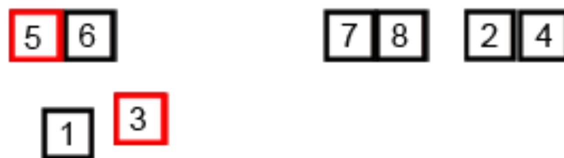
    public void push(T item) {
      StackNode<T> node = new StackNode<T>(item);
      node.next = top;
      top = node;
    }

    public T pop() {
      if(top == null) throw new EmptyStackException();
      T item = top.data;
      top = top.next;
      return item;
    }

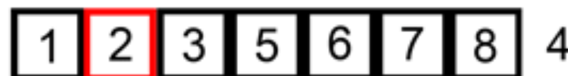
    public T peek() {
      if(top == null) throw new EmptyStackException();
      T item = top.data;
      return item;
    }

    public boolean isEmpty() {
      return top == null;
    }
  }
  ```

- What is generic types & type variable?
  - Generic type: when implementing some classes, the variable we passed in cannot be certain, they can be Integer, Character... In order to write once and fit for all types, use type variable `<T>` to represent all types
  - type variable: `<T>`
- 设计一个 data structure 包括 unique element 以及 insert order。写 insert 和 delete 两个 function
- Sort algorithms (Java's sort method?)
  ```

- Beginning with version 7, Oracle's Java implementation is using [Timsort] for object arrays bigger than 10 elements, and [Insertion sort] for arrays with less than that number of elements. The same considerations apply for both `Arrays.sort()` and `Collections.sort()`.
- QuickSort: average o(n*logn), worst o(n^2)
  - First, Pick an element, called a *pivot*, from the array.
  - Second, *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
  - Third, [Recursively] apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- TimSort: improve of merge sort, An example of merge sort. First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.
  - average and worst case: o(n*logn)



- Insertion sort: For unsorted number, scan the sorted array from back to front and find the proper position to insert.
  - average: o(n^2) best: o(n)



- Bubble sort: repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.
  - average: o(n^2) best: o(n)

5 3 1 6 7 8 2 4

- Why still useing insertion sort under 10?
    - Because big-O is only how computation changes over time, it's not the exact number. insertion sort is really O(1/2 n^2).

Leetcodes

LC101 Symmetric Tree

https://leetcode.com/problems/symmetric-tree/

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

```
   1
  / \
 2   2
/ \ / \
3  4 4  3
```

But the following [1,2,2,null,3,null,3] is not:

```
   1
  / \
 2   2
  \   \
  3    3
```

Note:
Bonus points if you could solve it both recursively and iteratively.

```
// RECURSIVELY
public class Solution {
  public boolean isSymmetric(TreeNode root) {
    if(root == null) {return true;}
    return isSymmetricCheck(root.left, root.right);
  }
```

```
  private boolean ismetricCheck(TreeNode left, TreeNode right) {
    if(left == null && right == null) {return true;}
    if((left == null && right != null) || (left != null && right == null))
{return false;}
    if(left.val != right.val) {return false;}
    return (isSymmetricCheck(left.left, right.right) &&
isSymmetricCheck(left.right, right.left));
  }
}

// NON-RECURSIVELY
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null) return true;
        if(root.left == null && root.right == null) return true;
        if(root.left == null || root.right == null) return false;

        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root.left);
        stack.push(root.right);

        TreeNode left, right;

        while(!stack.isEmpty()) {
            right = stack.pop();
            left = stack.pop();
            if(left.val != right.val) return false;

            if(left.left == null && right.right != null) return false;
            if(left.left != null && right.right == null) return false;
            if(left.left != null && right.right != null) {
                stack.push(left.left);
                stack.push(right.right);
            }

            if(left.right == null && right.left != null) return false;
            if(left.right != null && right.left == null) return false;
            if(left.right != null && right.left != null) {
                stack.push(left.right);
                stack.push(right.left);
            }
        }

        return true;
    }
}
```

LC387 First Unique Character in a String

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

```
s = "leetcode",
return 0.

s = "loveleetcode",
return 2.
```

Note: You may assume the string contain only lowercase letters.

```
// ARRAY
public class Solution {
    public int firstUniqChar(String s) {
        char[] charArray = s.toCharArray();
        int[] freq = new int[26];

        for(int i = 0; i < charArray.length; i++) {
            freq[charArray[i]-'a']++;
        }
        for(int i = 0; i < charArray.length; i++) {
            if(freq[charArray[i]-'a'] == 1) {
                return i;
            }
        }
        return -1;
    }
}
```

Convert Sorted Array to Balanced Binary Search Tree (BST)

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

```java
// FIND MID ELEMENT
public class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        if(nums.length == 0) {return null;}
        TreeNode root = pickMid(nums, 0, nums.length-1);
        return root;
    }
    public TreeNode pickMid(int[] nums, int first, int last) {
        if(first > last) return null;
        int mid = (first + last) / 2;
        TreeNode thisNode = new TreeNode(nums[mid]);
        thisNode.left = pickMid(nums, first, mid - 1);
        thisNode.right = pickMid(nums, mid + 1, last);
        return thisNode;
    }
}
```

Two Sum

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

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

Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

```java
// Hashmap, two pass
int len = nums.length;
if(len < 2) throw new IllegalArgumentException("No two sum solution");
Map<Integer, Integer> hashmap = new HashMap<Integer, Integer>();
for(int i = 0; i < len; i++) {
  hashmap.put(nums[i], i);
}
for(int i = 0; i < len; i++) {
  int diff = target - nums[i];
  if(hashmap.containsKey(diff) && hashmap.get(diff)!=i) {
    return new int[]{i, hashmap.get(diff)};
  }
}
throw new IllegalArgumentException("No two sum solution");
```

Two Sum Modified

given 2 arrays, find the values of elements that sum to be target

```java
// HASHSET
private static int[] getTwo(int[] arr1, int[] arr2, int target) {
  Set<Integer> hs = new HashSet<Integer>();
  for(int ele : arr1) {
    hs.add(ele);
  }

  // find another
  for(int ele : arr2) {
    if(hs.contains(target-ele)) {
      return new int[]{ele, target - ele};
    }
  }
  throw new IllegalArgumentException("....");
}
```

```
// SORT AND TWO POINTERS
private static int[] getTwo(int[] arr1, int[] arr2, int target) {
  Arrays.sort(arr1);
  Arrays.sort(arr2);
  int i = 0;
  int j = arr2.length-1;
  while(i<arr1.length && j>=0) {
    if(arr1[i]+arr2[j] < target) i++;
    else if(arr1[i] + arr2[j] > target) j--;
    else return new int[]{arr1[i], arr2[j]};
  }
}
```

3Sum

Given an array *S* of *n* integers, are there elements *a*, *b*, *c* in *S* such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Note: The solution set must not contain duplicate triplets.

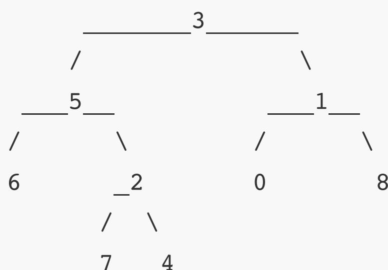
```
For example, given array S = [-1, 0, 1, 2, -1, -4],

A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

```
// SET TARGET AND TWO POINTER
public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        int len = nums.length;
        Arrays.sort(nums);
        for(int i = 0; i < len - 2; i++) {
            if(i != 0 && nums[i] == nums[i-1]) continue;
            int left = i + 1;
            int right = len - 1;
            int target = -nums[i];
            while(left < right) {
                if(nums[left] + nums[right] == target) {
                    res.add(Arrays.asList(nums[i], nums[left], nums[right]));
                    left++;
                    right--;
                    while(left < right && nums[left] == nums[left-1]) left++;
                    while(left < right && nums[right] == nums[right+1]) right--
;
                } else if (nums[left] + nums[right] > target) {
                    right--;
                } else {
                    left++;
                }
            }
        }
        return res;
    }
}
```

LC236 Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”



```
        _______3______
       /              \
    ___5__          ___1__
   /      \        /      \
   6      _2       0       8
         /  \
         7   4
```

For example, the lowest common ancestor (LCA) of nodes `5` and `1` is `3`. Another example is LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

```
// TOP TO BOTTOM, FIND PATH AND COLLIDE NODE
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        if(root == null || root == p || root == q) return root;
        Stack<TreeNode> pStack = new Stack<TreeNode>();
        Stack<TreeNode> qStack = new Stack<TreeNode>();

        TreeNode res = null;

        if(findPath(root, p, pStack) && findPath(root, q, qStack)) {

            while(!pStack.isEmpty()) {
                TreeNode pNode = pStack.pop();
                if(qStack.contains(pNode)) {
                    res = pNode;
                }
            }
        }

        return res;
    }
    private boolean findPath(TreeNode root, TreeNode target, Stack<TreeNode>
stack) {
        if(root == null) return false;
        if(root == target) {
            stack.push(root);
            return true;
        }

        if(findPath(root.left, target, stack) || findPath(root.right, target,
stack)) {
            stack.push(root);
            return true;
        }

        return false;
    }
}
```

```
// BOTTOM TO TOP, FIND NODE THAT LEFT/RIGHT BOTH HAS TARGETS
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        if(root == null) return null;
        if(root == p || root == q) return root;

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);

        if(left != null && right != null) return root;
        if(left == null && right != null) return right;
        if(left != null && right == null) return left;
        return null;
    }
}
```

Find Single Number

given an array of integers, return the only integer which only occurs once, others are in pairs. Return invalid if there are no such integers.

```
// HASHSET
public class Solution {
  public int findSingle(int[] arr) {
    Set<Integer> hashset = new HashSet<Integer>();

    for(int i = 0; i < arr.length; i++) {
      if(hashset.isEmpty() || !hashset.contains(arr[i])) {
        hashset.add(arr[i]);
      } else {
        hashset.remove(arr[i]);
      }
    }

    if(hashset.size() == 1) {
      return hashset.iterator().next();
      // Iterator iterator = hashset.iterator();
      // while(iterator.hasNext()) {iterator.next();}
    } else {
      throw new IllegalArgumentException("No such ...");
    }
  }
}
```

```
// XOR BIT MANIPULATION
public class Solution {
  public int findSingle(int[] arr) {
    int res = 0;
    //int ctr = 0;
    for(int num : arr) {
      //if(num == 0) ctr++;
      res = res ^ num;
    }
    //if(ctr > 1) throw new IllegalArgumentException("No such...");
    return res;
  }
}
```

```
// SORT AND CHECK PAIR
Arrays.sort(arr); // o(nlogn) worst: o(n^2)
```

Average Value

given an array of Integers, return the average value in double float.

```
// int: 32, long: 64, double: 64, string:...
public class Solution {
  public double getAverage(int[] arr) {
    long sum = 0;
    int num = 0;
    for(int cur : arr) {
      sum = sum + (long)cur;
      num++;
    }
    return (double)sum / (double)num;
  }
}
```

Use string to calculate when the number is extremely big.

```java
import java.math.BigInteger;
public static void main(String args[]){

    String max_long = "9223372036854775807";
    String min_long = "-9223372036854775808";

    BigInteger b1 = new BigInteger(max_long);
    BigInteger b2 = new BigInteger(min_long);

    BigInteger sum = b1.add(b1);
    BigInteger difference = b2.subtract(b1);
    BigInteger product = b1.multiply(b2);
    BigInteger quotient = b1.divide(b1);

    System.out.println("The sum is: " + sum);
    System.out.println("The difference is: " + difference);
    System.out.println("The product is: " + product);
    System.out.println("The quotient is: " + quotient);

}

// Integer.toString(number)
// String.valueOf(number)
```

Roman to Integer

```java
public class Solution {
    public int romanToInt(String s) {
        int sLen = s.length();
        if(sLen < 1) return 0;

        Map<Character, Integer> hashmap = new HashMap<>();
        hashmap.put('I', 1);
        hashmap.put('V', 5);
        hashmap.put('X', 10);
        hashmap.put('L', 50);
        hashmap.put('C', 100);
        hashmap.put('D', 500);
        hashmap.put('M', 1000);

        int res = hashmap.get(s.charAt(sLen-1));
        for(int i = sLen-2; i >= 0; i--) {
            int curr = hashmap.get(s.charAt(i));
            int next = hashmap.get(s.charAt(i+1));
            if(curr < next) {
                res -= curr;
            } else {
                res += curr;
            }
        }
        return res;
    }
}
```

FizzBuzz

Write a program that outputs the string representation of numbers from 1 to *n*.

But for multiples of three it should output “Fizz” instead of the number and for the multiples of five output “Buzz”. For numbers which are multiples of both three and five output “FizzBuzz”.

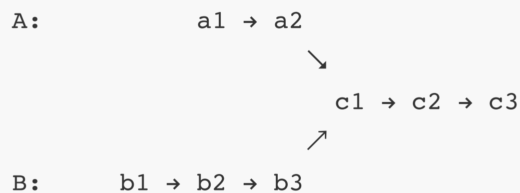
```java
// USE %
public class Solution {
    public List<String> fizzBuzz(int n) {
        List<String> lt = new ArrayList<String>();
        for(int i = 0; i < n; i++) {
            int idx = i + 1;
            if(idx % 3 == 0 && idx % 5 == 0) {
                lt.add("FizzBuzz");
            } else if(idx % 3 == 0) {
                lt.add("Fizz");
            } else if (idx % 5 == 0) {
                lt.add("Buzz");
            } else {
                lt.add(Integer.toString(idx));
            }
        }

        return lt;
    }
}
```

```java
// NO %: FIZZ BUZZ COUNTER
public class Solution {
    public List<String> fizzBuzz(int n) {
        List<String> lt = new ArrayList<String>();
        for(int i = 1, fizz = 0, buzz = 0; i <= n; i++) {
            fizz++;
            buzz++;
            if(fizz == 3 && buzz == 5) {
                fizz = 0;
                buzz = 0;
                lt.add("FizzBuzz");
            } else if (fizz == 3) {
                fizz = 0;
                lt.add("Fizz");
            } else if (buzz == 5) {
                buzz = 0;
                lt.add("Buzz");
            } else {
                lt.add(Integer.toString(i));
            }
        }

        return lt;
    }
}
```

Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:



begin to intersect at node c1.

Notes:

- If the two linked lists have no intersection at all, return `null`.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in O(n) time and use only O(1) memory.

```java
//FIND LENGTH AND MOVE TOGETHER
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA == null || headB == null) return null;
        int ALen = 0, BLen = 0;
        ListNode iterA = headA, iterB = headB;
        while (iterA != null) {
            ALen++;
            iterA = iterA.next;
        }
        while (iterB != null) {
            BLen++;
            iterB = iterB.next;
        }

        while(ALen < BLen) {
            headB = headB.next;
            BLen--;
        }
        while(ALen > BLen) {
            headA = headA.next;
            ALen--;
        }

        while(headA != headB) {
            headA = headA.next;
            headB = headB.next;
        }
        return headA;
    }
}
```

```java
// 8
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA == null || headB == null) return null;

        ListNode iterA = headA, iterB = headB;

        while(iterA != iterB) {
            if(iterA == null) iterA = headB;
            else iterA = iterA.next;
            if(iterB == null) iterB = headA;
            else iterB = iterB.next;
        }

        return iterA;
    }
}
```

OOD: Package

Two types of packages, implement: getPackageA(), getPackageB(), getAnyPackage(), addPackageA(), addPackageB()

FIFO

```
public class PackageNode {
  int val;
  int timeStamp;
  PackageNode(int x) {
    val = x;
    timeStamp = ...;
  }
}

Queue<PackageNode> qA = new LinkedList<PackageNode>();
Queue<PackageNode> qB = new LinkedList<PackageNode>();

public void addPackageA(int x) {
  PackageNode newNode = new PackageNode(x);
  qA.add(newNode);
}

public int getPackageA() {
  if(qA.isEmpty()) return null;
  PackageNode currNode = qA.remove();
  return currNode.val;
}

public int getAny() {
  if(qA.isEmpty() && qB.isEmpty()) return null;
  if(qA.isEmpty() && qB.isEmpty())...
  PackageNode aNode = qA.peek();
  PackageNode bNode = qB.peek();
  return (aNode.timeStamp > bNode.timeStamp ? bNode..);
}
```

Product of Array Except Self

Given an array of *n* integers where *n* > 1, `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Solve it **without division** and in O(*n*).

For example, given `[1,2,3,4]`, return `[24,12,8,6]`.

```
// PREFIX AND SUFFIX, FORWARD AND BACKWARD
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        int preProduct = 1;
        int sufProduct = 1;
        int[] res = new int[nums.length];

        for(int i = 0; i < nums.length; i++) {
            res[i] = preProduct;
            preProduct *= nums[i];
        }

        for(int i = nums.length - 1; i >= 0; i--) {
            res[i] *= sufProduct;
            sufProduct *= nums[i];
        }

        return res;
    }
}
```

Find the nearest value in BST

```
// BIGGER? RIGHT, SMALLER? LEFT
public int findNearestInBST(int target, Node root) {
  int res = root.val;
  Node curr = root;
  while(curr != null) {
    if(Math.abs(target-curr.val) < Math.abs(res-target)) {
      res = curr.val;
    }
    // decide left or right
    if(curr.val > target) {
      root = root.left;
    } else if (curr.val < target) {
      root = root.right;
    } else {
      return target;
    }
  }
  return res;
}
```

Group Anagrams

Given an array of strings, group anagrams together.

For example, given: `["eat", "tea", "tan", "ate", "nat", "bat"]`,
Return:

```
[
  ["ate", "eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

```
// HASHMAP SAVING STRING AND LIST, SORT TO JUDGE ANAGRAM
public class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> hm = new HashMap<String, List<String>>();

        for(String str : strs) {
            char[] charArr = str.toCharArray();
            Arrays.sort(charArr);
            String keyStr = new String(charArr);

            if(!hm.containsKey(keyStr)) {
                hm.put(keyStr, new ArrayList<String>());
                hm.get(keyStr).add(str);
            } else {
                hm.get(keyStr).add(str);
            }
        }

        return new ArrayList<List<String>>(hm.values());
    }
}
```

BST level order traversal

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

For example:
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

```
// QUEUE TO SAVE EACH LEVEL
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        List<List<Integer>> wrapList = new LinkedList<List<Integer>>();

        if(root == null) return wrapList;

        queue.add(root);
        while(!queue.isEmpty()){
            int levelNum = queue.size();
            List<Integer> subList = new LinkedList<Integer>();
            for(int i=0; i<levelNum; i++) {
                if(queue.peek().left != null) queue.add(queue.peek().left);
                if(queue.peek().right != null) queue.add(queue.peek().right);
                subList.add(queue.remove().val);
            }
            wrapList.add(subList);
        }
        return wrapList;
    }
}
```

Kth Largest Element in an Array

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example,
Given `[3,2,1,5,6,4]` and k = 2, return 5.

Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.

```
// SORT AND FIND
public int findKthLargest(int[] nums, int k) {
  Arrays.sort(nums);
  return nums[nums.length - k];
}
```

```
//  QUICK SELECT
public int findKthLargest(int[] nums, int k) {
  if (nums == null || nums.length == 0) return Integer.MAX_VALUE;
  return findKthLargest(nums, 0, nums.length - 1, nums.length - k);
}

public int findKthLargest(int[] nums, int start, int end, int k) {// quick
select: kth smallest
  if (start > end) return Integer.MAX_VALUE;

  int pivot = nums[end];// Take A[end] as the pivot,
  int left = start;
  for (int i = start; i < end; i++) {
    if (nums[i] <= pivot) {// Put numbers < pivot to pivot's left
      swap(nums, left, i);
      left++;
    }
  }
  swap(nums, left, end);// Finally, swap A[end] with A[left]

  if (left == k)// Found kth smallest number
    return nums[left];
  else if (left < k)// Check right part
    return findKthLargest(nums, left + 1, end, k);
  else // Check left part
    return findKthLargest(nums, start, left - 1, k);
}

void swap(int[] A, int i, int j) {
  int tmp = A[i];
  A[i] = A[j];
  A[j] = tmp;
}
```

Number of Islands

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
11110
11010
11000
00000
```

Answer: 1

Example 2:

```
11000
11000
00100
00011
```

Answer: 3

```
// DFS
public class Solution {
    // BFS method
    public void DFS(char[][] grid, int m, int n) {
        if(m<0 || m>=grid.length || n<0 || n>=grid[0].length) return;

        if(grid[m][n] == '1') {
            grid[m][n] = '0';
            DFS(grid, m-1, n);
            DFS(grid, m+1, n);
            DFS(grid, m, n-1);
            DFS(grid, m, n+1);
        }
    }

    public int numIslands(char[][] grid) {

        int islandNum = 0;

        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(grid[i][j]=='0') {
                    continue;
                } else {
                    islandNum++;
                    DFS(grid, i, j);
                }
            }
        }
        return islandNum;
    }
}
```

Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

```
nums = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
```

Return `4`
The longest increasing path is `[1, 2, 6, 9]`.

Example 2:

```
nums = [
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
```

Return `4`

```
// DFS
public class Solution {
    public int longestIncreasingPath(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }
        int[][] cache = new int[matrix.length][matrix[0].length];
        int max = 0;
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[0].length; j++) {
                int length = findAround(i, j, matrix, cache,
Integer.MIN_VALUE);
                max = Math.max(length, max);
            }
        }
        return max;
    }
    private int findAround(int i, int j, int[][] matrix, int[][] cache, int
pre) {
        // if out of bond OR current cell value smaller than previous cell
value.
        if (i < 0 || i >= matrix.length || j < 0 || j >= matrix[0].length ||
matrix[i][j] <= pre) {
            return 0;
        }
        // if calculated before, no need to do it again
        if (cache[i][j] > 0) {
            return cache[i][j];
        } else {
            int cur = matrix[i][j];
            int tempMax = 0;
            tempMax = Math.max(findAround(i - 1, j, matrix, cache, cur),
tempMax);
            tempMax = Math.max(findAround(i + 1, j, matrix, cache, cur),
tempMax);
            tempMax = Math.max(findAround(i, j - 1, matrix, cache, cur),
tempMax);
            tempMax = Math.max(findAround(i, j + 1, matrix, cache, cur),
tempMax);
            cache[i][j] = ++tempMax;
            return tempMax;
        }
    }
}
```

Find K Nearest Point

```
int[] distances = new int[nums.length];
// compute distance for every number
// Find Kth Smallest Element in an Array
// loop, add to Point[] and return
```

```
import java.util.PriorityQueue;
import java.util.Comparator;

public class kNearestPoint {
    public Point[] Solution(Point[] array, Point origin, int k) {
        Point[] rvalue = new Point[k];
        int index = 0;
        PriorityQueue<Point> pq = new PriorityQueue<Point> (k, new
Comparator<Point> () {
            @Override
            public int compare(Point a, Point b) {
                return (int) (getDistance(a, origin) - getDistance(b, origin));
            }
        });

        for (int i = 0; i < array.length; i++) {
            pq.offer(array[i]);
            if (pq.size() > k)
                pq.poll();
        }
        while (!pq.isEmpty())
            rvalue[index++] = pq.poll();
        return rvalue;
    }
    private double getDistance(Point a, Point b) {
        return Math.sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y -
b.y));
    }
}
```

Merge k Sorted Lists

```java
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if(lists == null || lists.length == 0){
            return null;
        }
        return mergeHelper(lists, 0, lists.length-1);
    }

    public ListNode mergeHelper(ListNode[] lists, int start, int end){
        if(start == end){
            return lists[start];
        }
        int mid = start + (end - start)/2;
        ListNode list1 = mergeHelper(lists, start, mid);
        ListNode list2 = mergeHelper(lists, mid+1, end);
        return mergeList(list1, list2);
    }

    public ListNode mergeList(ListNode list1, ListNode list2){
        ListNode dummy = new ListNode(-1), tail = dummy;
        while(list1 != null && list2 != null){
            if(list1.val < list2.val){
                tail.next = list1;
                list1 = list1.next;
            }
            else{
                tail.next = list2;
                list2 = list2.next;
            }
            tail = tail.next;
        }
        if(list1 != null){
            tail.next = list1;
        }
        else{
            tail.next = list2;
        }
        return dummy.next;
    }
}
```