**Top 75 questions**

[**https://leetcode.com/list/xi4ci4ig/**](https://leetcode.com/list/xi4ci4ig/)

[**https://designgurus.org/course/grokking-the-coding-interview**](https://designgurus.org/course/grokking-the-coding-interview)

**Pep Level code:**

[**https://github.com/rajneeshkumar146/pepcoding-Batches/tree/master/2022**](https://github.com/rajneeshkumar146/pepcoding-Batches/tree/master/2022)

**Java tips:**

**Lists**

**Creating a normal List:**

List<Integer> list=new ArrayList<Integer>(); // notice we use ArrayList

**Creating list of list**

List<List<Integer>> result=new ArrayList<List<Integer>>(); // creating list of list

List<Integer> tempResult=new ArrayList<Integer>(); // creating list

result.add(new ArrayList<>(tempResult)); //adding list to a list object

**Placing an element at a particular position in arraylist**

Say arralist is [10,20,30,40]

arrList.set(2, 3000) // answer will be [10,20,3000, 40]

Notice the length of arraylist would not change as it happens in add()

**Sorting an ArrayList**

int[][] envelopes = { { 5, 4 }, { 6, 9 }, { 6, 7 }, { 2, 3 } };

// This is used in Russian envolope problem (LIS) in DP

// Notice the arrow is -> (single dash) (NOT =>)

      Arrays.sort(envelopes, (a, b) -> a[0] - b[0]);

**Sorting a Array of class object**

// This is implemented in Non Overlapping Bridge problem

// Creating custom class for Bridge to store North, South values in pair

// Implementing Comparable to sort array of class (CityBridge[]) based on North property

class CityBridge implements Comparable<CityBridge> {

    int north, south;

    CityBridge(int nb, int sb) {

        this.north = nb;

        this.south = sb;

    }

    // Notice how we override Comparable's compareTo method becuase we have to sort

    // array of Class CityBridge[] on North and South property.

    // Sorting cna than be done as Arrays.sort(CityBridgesArray)

    public int compareTo(CityBridge o) {

        // if 'north' value is same, than sort on 'south' value

        if (this.north == o.north) {

            return this.south - o.south;

        }

        return this.north - o.north;

    }

}

// Creating array of classes

CityBridge[] bridgesList = new CityBridge[nbValues.length];

bridgesList[0] = new CityBridge(6, 3);

bridgesList[1] = new CityBridge(2, 7);

// Now Sorting can be done like :

Arrays.sort(bridgesList);

// This will bring [2,7] object at index 0 and [6,3] at index 1

* Another way of creating/returning array with values :

new int[] { a, b };

* To get minimum , use

int min = (int) 1e9;

**instead of**

int min = Integer.MAX\_VALUE;

because it gives buffer.

* **Convert String to array and Array to String**

        // String to Array

        String str = "abc";

        char[] arr = str.toCharArray(); // ['a', 'b', 'c']

        // Array to String

        String newStr = new String(arr);  // "abc" (arr is char[] array)

        // or

        String newStr2 = String.valueOf(arr); // "abc"

* **Convert int to String**

        // int to String

  String str= String.valueOf(1); // “1”

* **Making a copy of object**

**Ans: Copy Constructor as shown below**

    private static class Node {

        int val;

        ArrayList<Node> children = new ArrayList<>();

        Node(int data) {

            val = data;

        }

        // Cloning an object so that if changes are made in 1 , other should not get

        // affected

        public Node(Node another) {

            // For non primitives properties (like array, object),

            // always create new object like we did not below for arraylist children

// below

            ArrayList<Node> newChildren = new ArrayList<>();

            for (Node n : another.children) {

                newChildren.add(n);

            }

            this.children = newChildren; // you can access

        }

    }

**Usage:**

  Node root = PopulateTreeHelper();

        Node nodeCopy = new Node(root); // making a copy

* How to **use dictionary** (HashMap) in Java

HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

for(int i : nums){  
 if(map.containsKey(i)){  
 map.put(i, map.get(i) + 1);  
 }else{  
 map.put(i,1);  
 }

}

for(int i : nums){  
 if(map.get(i) == 1){  
 return i;  
 }  
 }

* **Printing a HashMap**

// Printing the filled map to show the output, see VerticalOrderTraversal

problem

TreeMap<Integer, List<Integer>> map = new TreeMap<>();

        // notice you can pull 'keys' using keySet

        for (int key : map.keySet()) {

            // and have to pull valyes using key (not index)

            String value = map.get(key).toString();

            System.out.println(key + " " + value);

        }

* **Difference between TreeMap and HashMap? When to use what?**

Both store keyValue pair, but TreeMap also maintain order of keys e.g. [-1,0,1,2,3]……irrespective of key-insertion order

* **returning a list directly:**

return new int[] {2,3};

//this will return a list of int[] with two items: 2 and 3.

* **How to declare a queue in Java:**

 Queue<Node> queue = new ArrayDeque<>();

// add item

 queue.add(root);

// removing item

Node node = queue.remove();

* **Difference between peek() and poll() in Queue**

The **peek()** method only retrieved the element at the head but the **poll()** also removes the element along with the retrieval. It returns NULL if the queue is empty.

Node n = queue.peek(); // only retrieves

Node n = queue.poll(); // retrieves and removes

**Recursion**:

1. Euler for recursion: https://www.youtube.com/watch?v=R7qja\_gZrvI

left print -> Left call -> print -> Right call

2. Use Faith and Expectation:

Faith is if run(n) will do its job, than run(n-1) will do its job automatically and than add a base case

* **Differnece between Flood fill and Maze Path**  is :

In maze path - you move in specific direction, but in flood fill you can move in **any direction** to reach from one point to other, so you have to manage the **visited cells**(mark them as true before loop and mark them as false after loop ends) in flood fill to avoid crossing the same cell.

* **Combination**- hamesha forward move karna hota

**Arrangement/Permutation**- direction backward kar sakte ho beech mai

E..g say string is **abcde …**you haveto find combination so possible combination can be :

ab, abc, abce -- these are all combinations because we are moving in same direction

**Another example:**

Permutation of two things from three given things a, b, c is ab, ba, bc, cb, ac, ca

Combination of two things from three given things a, b, c is ab, bc, ca

**If the order is important**, **then the problem is related to permutation**, and the possible number of samples will be, XY, YX, YZ, ZY, XZ, ZX. In this case, XY is distinct from the sample YX, YZ is distinct from the sample ZY and XZ is distinct from the sample ZX.

If the order is unnecessary, then the question is relevant to the combination, and the possible samples will be XY, YZ and ZX.

**Point:** Permutations are always more than Combinations

However, abec, abdc - are arrangements (not combinations) because they are not forward looking in 1 direction

* **For question –CoinChange\_GetMinCoins\_LC322 (Leetcode 322)**

**Icon

Description automatically generated with medium confidence**

Say coins are {2,3,5,7} and target sum is 10, we have to find minimum number of coins required

We start with 2 in the loop in solution code, remaining number is 8(10-2) – now our funciton will return minimum no. of coins required to get 8 (10-2).

We have to add 1 because we have used 1 coin already i.e 2.

Base condition return 0 since minimum coins required to get 0 is 0.

* You can get the row and column of a cell in a 2D matrix. By its serial number with the following logic

**int row = index / nums.length** (size of 2Darray)

**int row = index % nums.length** (size of 2Darray)

* You can also **store 2D array into 1D array:**

Now, for this r and c can be found using above forumales (r= idx / nums.length and c= idx % nums.length)  
But how would you find the idx:

using formula  **R\*N+C**  where N is number of columns.

**Note:** (Check Level up lecture of 1st Apr).

* **Working with charcters**

Below code will store occurences of each character in a string and than remove duplicate charcters and store the count.

int[] freq = new int[26];

        // Below for loop with store the occurences count of charcters in a

string in freq[] array

        // e.g. "sendmoremoney" will store [0,0,0,0,3.....] 3 because 'e'

occured 3 times

        for (int i = 0; i < superString.length(); i++) {

            freq[superString.charAt(i) - 'a']++;

        }

        superString = "";

        // Below For loop will remove the duplicate values and order them

alphabetically "demnorsy"

        for (int i = 0; i < 26; i++) {

            if (freq[i] > 0) {

                superString += (char) (i + 'a');

            }

}

       System.out.println(superString);

* **Bit manipulation**

1 byte= 8 bit

1 int = 4 byte= 32 bit

1 long = 8 byte = 64 bit

Most significant bit -> 1001 <- Least significant bit(LSB)

**^ (XOR)**

0 ^ 0 = 0

0 ^ 1 = 1

1 ^ 1 = 0

1 ^ 0 = 1

**! (Complement)**

!0 = 1

!1 = 0

**~ (1’s complement) – reverses the bits**

m = 100101

~m =011010

**-(2’s complement)**

Turns the number negative

**<< Left shift**

X=1011

X << 2 = 101100 (appends zeroes to the right and 2 digits from left will be dropped)

**>> Right shift**

X=1011

X >> 2 = 111011 (Right shift is speical here, if MSB is 1 than 1s will be appended on the left and if MSB is 0 than 0 will be appeneded and 2 bits from right will be dropped)

Y = 0011

X >> 2 = 000011

**-Getting the last set bit**

**n & (-n)**

* **Important Formula**

Shifting bits towards **left** by **n** digits -> will result in final number as **num\* (2^n)**

E.g.

5 is 101 -> shifting by 2 bits will result in 10100 - which is 5\*(2^2) = 20

Similarly , Shifting bits towards right by **n** digits -> will result in final number as **num/ (2^n)**

* **Tricks to find number of set bits(1) in a binary number**(See BitManiplationQuestions.java and video of 6th Apr )

1. Hamming weight method

Logic of hamming weght…when you AND a number ‘n’ with ‘n-1’ , number of set bits reduces by 1

So keep doing AND till all 1s becomes 0s.

Text, letter, whiteboard

Description automatically generated

Other appraoch is to AND with ‘1’ so what will happen that if you get 1 that means last most bit is 1, keep counting and keep ‘right shifting’ the number by 1.

All Palendromic partitions Euler diagram:

AllPallendromicPartitions\_LC131

Graphical user interface

Description automatically generated with low confidence

Original String is “**pepcocp”..** We take substrings which are palaledromic and start by taking first character which is also a pallndrome **e..g** We start with **pepcocp** whichbreak into “p” and “pep” (only pallendrome). When we take “p” - we pass rest of the string i.e. “epcocp” into recursive call. Again we take first bit i.e. “e” and now we don’t have any substring which is pallndrome

* **Linked list**

**Important question**: Remove nth node from end.

Uses 2-pointer approach

**Note**: Understand the importance of adding a dummy Node at the beginning to handle case where the input list is [1,2] and element to be removed is 2nd from the end i.e. 1.

We use the concept of dummyNode when we use the ‘current’ and ‘previous’ pointers and want to avoid ‘previous’ to be null becuase we have to set previous.next in the logic

This concept is also used in the implmenetation of LC 92: **ReverseSpecificNodes** in “Basics\_Day3.java”

-**Sorting a linked list and Merge 2 linked list**

See “Basics\_Day2.java”.

We use Merge sort to sort a linked list by first splitting it till the last node and then merging it iteratively in a sorted form.

**Dynamic Programming-** Dynamic programming is a technique that breaks the problems into sub-problems(overlapping sub problems), and saves the result for future purposes(memoization) so that we do not need to compute the result again.  Ex: **Fibonnaci**

**2 ways-**

1. Recursion and Memoization – Top Down (occupies more memory in the call stack)
2. Tabulation – Bottom up – Uses loop

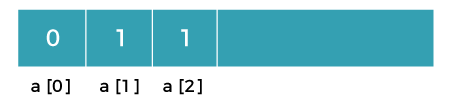
Check Fibonacci.java, for each implementations – **These implementations are the base of DP problems, understand them**

**Bottom Explanation**

The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward.

In the bottom-up approach, we start from the base case to find the answer for the end. We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.

We use for loop to iterate over the sub-problems.



We keep calcaulating next result from previous result in Bottom up as shown above.

- **Differnece between DP and Recursion** is DP uses Recursion but with caching(memoization).

**- Important on DP solutions**

1) First columns and rows are generally kept empty in DP 2-D arrays

so if you talk of dp[i][j]...you are filling for word[i-1][j-1]

See diagram **EditDistance.png** and code in this "**DP**" folder in “PeplevelUp”

2)One important thing about base conditions is if there are 2 base conditions , in one condition you are returning 1 and in other you are returning zero - always put the one returning 1 above the one returning 0.

E.g. Count\_distinct subsequences (LC 115)quesiton of DP.

* Target Sum Subset and Coin Change problem are same, only in TargetSumSubsetwe do not repeat numbers.

If given nos are 2,3,5,6

For 7,  
TargetSum subset will be {2,5}

CoinChange will be {2,2,3} and {2,5}

* **Differnece between SubArray and Subsets**:

SubArrays are continous as shown below: (Elements are not skipped)

A person standing in front of a whiteboard with red writing

Description automatically generated with medium confidence

**SubArrays above**

SubSubsets can be non-continuous

**A picture containing text, whiteboard, outdoor, people

Description automatically generated**

**SubSets** (har element ke pass choice hoti hai wo aayega ya nahi aayegs)

Total no of subsets possible are **2^n**

**Differnce between SubString and Subsequence:**

The difference between a longest common substring and a longest common subsequence is the criterion of contiguity. The symbols that make up a longest common substring must appear in both strings as a contiguous string.

“abc” has following subsequences- [abc, ab, bc, ac, a, b, c,””]

“abc” substring – [abc, ab, bc,a,b,c] - has to be continuous e.g. “ac” is not a substring.

**- SubString solutions are solved using 2D arrays (Not recursion and memoization)**

e.g.

1) Longest Common SubString

2) Longest Pallendromic SubString