# Tips and Tricks

## Searching

### For searching, in case of array, if you are implementing binary search, then the function should accept a lo and hi so that it knows which part of the array it is searching.

### To find floor and search, always check the next and previous of middle value to see if they are greater and less than the number you are searching for

### Find 3rd largest element in 1 pass

* Make 3 variables. First, second and third.
* When you see an element bigger than first
  + third = second
  + second = first
  + first = new element
* When you see an element bigger than second
  + third = second
  + second = new element
* When you see an element bigger than third
  + third = new element

### Find 2nd largest element using minimum comparisons

* Use merge sort technique.
* Compare every two element and keep track of first and second
* Now compare the result of those and keep track of first and second

### Find missing number in an array from 1 to n

* Take the highest number and take it as 'n'.
* Use (n\*(n+1))/2 to calculate the sum.
* Now subtract each element from this sum.
* The answer left in the end will be your answer.

## **Two numbers in array with +ve and -ve** **numbers whose sum is closest to zero**

* 1st method - Use index from left and right
  + Sort the array
  + Find the sum of first and last.
  + If sum > 0 then take the sum of first and second last
  + If sum < 0 then take the sum of second and last
  + If sum = 0 then you have your answer
  + Keep doing this recursively
* 2nd method - Hashing
  + For every value, save the complementary value in a HashMap. The complementary value is saved as key.
  + Check if any number is present in the HashMap.
  + If it is present then the key and the value are the two values that were needed.

## **Two numbers in array whose sum is closest to a specific number**

* 1st method - Use index from left and right
  + Sort the array
  + Keep a minimum index which will save the index for which it was closest
  + Find the sum of first and last.
  + If sum > number then take the sum of first and second last
  + If sum < number then take the sum of second and last
  + If the sum is closer than the closest you have seen till now then you have found your numbers.
  + Keep doing this recursively
* 2nd method - Hashing
  + For every value, save the complementary value in a HashMap. The complementary value is saved as key.
  + Check if any number is present in the HashMap.
  + If it is present then the key and the value are the two values that were needed.

### Find a pair with given difference

* Make a HashMap
* Go through every element and see if it is there in the HashMap
* If it is then print the key and value and return
* If not then add the following to HashMap
  + hm.put(arr[i] - difference, arr[i])
  + hm.put(arr[i] + difference, arr[i])

## **For frequency questions, use a HashMap to count the** **frequency**

## **If an array is sorted and the question is asking for frequency of certain number then use binary search**

* Use binary search to find the first occurrence.
* Use another binary search to find the last occurrence.
* Then use this two information to find the frequency.

## **Find union and intersection of two arrays**

* Make two indices for both the arrays.
* See which one of the index is smaller and add that to the new arraylist. Increment that index.
* If it was same then in union add any one of them to the list and increment both the indices.
* In case of intersection add only in the case when both of them are similar.

## **Only one element is occurring odd number of times**

* The idea is that there a certain number of elements so that the half of them is odd then the middle number and the number before that should be same. If not then there has been an element before that has been odd.
* If it is same then an element after that is odd because it is necessary that an element is odd.
* If the half of it is even then the previous number should not be the same. If it is then there is an odd element on the left.

## Sorting

## **Sort 0s, 1s and 2s (Dutch National Flag Algorithm)**

* You are supposed to make the 3 regions in the original array. These will be marked using the indices.
* LoIndex = 0, MidIndex = 0 and HiIndex = arr.length - 1.
* Now arr[MidIndex] will be observed.
  + If arr[MidIndex] = 0 then swap arr[LoIndex] with arr[MidIndex] and increment LoIndex and MidIndex
  + If arr[MidIndex] = 1 then increment MidIndex
  + If arr[MidIndex] = 2 then swap arr[MidIndex] with arr[HiIndex] and decrement HiIndex.
    - Dont increment MidIndex in this case.
* Example
  + 0, 1, 2, 0, 1, 2
  + 0, 1, 2, 0, 1, 2 - lo = 0 mid = 0 hi = 5
  + 0, 1, 2, 0, 1, 2 - lo = 1 mid = 1 hi = 5
  + 0, 1, 2, 0, 1, 2 - lo = 1 mid = 2 hi = 5
  + 0, 1, 2, 0, 1, 2 - lo = 1 mid = 2 hi = 4
  + 0, 1, 1, 0, 2, 2 - lo = 1 mid = 2 hi = 3
  + 0, 0, 1, 1, 2, 2 - lo = 2 mid = 3 hi = 3

## **In a non stored array, how many inversions are there. Inversions mean a[i] > a[j] when i < j**

* In 2,4,1,3,5 has 3 inversions (2,1), (4,1) and (2,3).
* Basically, do merge sort.
* Whenever a term on the right-side array is less than left side array means there is an inversion so increment the inversion counter.

## **Sorting huge numbers**

* For string huge numbers, convert them to string.

# Binary Tree

## Find inorder successor

* Keep track of a global boolean = false and Node.
* When you see the boolean is positive, save the current value of node as successor and change the value of boolean to false.
* Whenever you see the node for which you have to find the inorder successor, change the boolean to true.

## Using parent array to contruct binary tree

* Use a hashmap to track integer to their node.
* Now use the hashmap to get the chil and node and form a relation between them.

# Binary Search Tree

* Use global variables a lot in these. It has helped me specially in recursion.
* Use queue all the time when it comes to levelorder.
* It helps a lot of you keep track of the elements by tracking minimum and maximum. This has been shown below in array to BST.

## From array to Binary Search Tree

* Keep track of min and max.
* At start this will be Integer.MIN\_VALUE and Integer.MAX\_VALUE.
* Now when you put the root, it still Integer.MIN\_VALUE and Integer.MAX\_VALUE.
* If the next element is smaller than root then all the elements below it will be in the range Integer.MIN\_VALUE and root.value.
* If the next element is greater than root then all the elements below it will be in the range root.value and Integer.MAX\_VALUE.

## Check if Binrary Tree is Binary Search Tree

* Make a recursive function that accepts value min and max
* First time it will take Integer.MIN\_VALUE and Integer.MAX\_VALUE.
* Now when we call node.left, the minimum will be Integer.MIN\_VALUE and maximum will be node.value
* Now when we call node.right, the minimum will be node.value and maximum will be Integer.MAX\_VALUE.
* This helps in keeping track of the range of the values.

## Reverse Inorder

* Inorder(root.right)
* Inorder(root.left)

## Inorder Successor

* Make a global variable of successor node and Boolean found.
* Initialize the node as false and found as false
* Do inorder transversal.
* Check when (found == true) save node as the current root and change found to false.
* When you actually find the node for which you are trying to find the successor, change the found = true.
* Remember that last step has to be implemented after second last step.
* Global variable node will have your answer.

## Inorder Predecessor

* Make a global variable of successor node and Boolean found.
* Initialize the node as false and found as false
* Do reverse inorder transversal.
* Check when (found == true) save node as the current root and change found to false.
* When you actually find the node for which you are trying to find the successor, change the found = true.
* Remember that last step has to be implemented after second last step.
* Global variable node will have your answer.

## Levelorder to Binary Search Tree

* Make two Queues one that contains nodes and nodesDetailed.
* The nodesDetailed is an additional class which contains the value of the node, the minimum value of the node and the maximum value of the node.
* Now make a node of root and detailedNode of root and add it to the queues. The detailed node min will be Integer.MIN\_VALUE and maximum value will be Integer.MAX\_VALUE.
* Now keep increasing the index in array and poll the queue.
* If the value in the levelorder is greater than minimum value and less than value then create a new node, add this node to the left of the previous node, then add the detailedVersion of this node and node to the queues.
* If it is greater than value and less than maximum then add it to the right.
* First check the left, if you found left then keep using the same node that you polled and check if the next term in level is right.
* If you found the right then it will also be assigned and if not then in the next iteratino of while loop it will be assigned to the left of the next element polled from the queue.
* If the queue got empty then a tree from that levelorder could not be found and return null.
* If the index reached the end of the array then a binary search tree was made and now return the root of the tree.

## Binary Search Tree to Min Heap

* The property of min heap is that parent should always be smaller than its children.
* Take inorder of BST
* Then do preorder.
* While preorder, keep assigning the values.

## Find kth Smallest Element

* Maintain a global variable of k and make it equal to kth.
* Do inorder traversal.
* In the middle of recursive statement keep decrementing k.
* If k == 0 then we have found our node.
* Save this node in another global variable and at the end return this node.

## Lowest Common Ancestor

* First node is smaller than second node
* If node value is less than first node then go right.
* If node is greater than second node then go left.
* If node is greater than first node and less than second node then you have found your node.

## Distance between nodes

* Calculate height of first node.
* Calculate height of second node.
* Calculate height of ancestor.
* Distance is (node1 - ancestor) + (node2 - ancestor)

## Remove nodes outside a range

* Traverse pre order
* If node is less than minimum then node becomes node.right.
* If node is more than maximum then node becomes node.left.

# Trie

## Find the frequency of all strings in an array of strings

* Create a trie.
* The node will be special because it will not be a boolean but an integer containing the number of words occurred.
* Add all the words in the trie and increment the number of words in that array.
* Now do a preorder traversal of arrays and find the string and the count of each array and it to an arraylist.

## Find the highest frequency string in an array of strings

* Create a trie.
* The node will be special because it will not be a boolean but an integer containing the number of words occurred.
* Add all the words in the trie and increment the number of words in that array.
* Now do a preorder of arrays and find the node with the highest count. That string will be of the highest frequency.

## Autocomplete part of a word to a word from a list of strings

* Create a trie with all the words in the list of strings.
* Now find the path of the tree of the query and get the end of the path.
* Now from this node, get all the words possible by doing an inorder traversal.
* Once you get the words, add the query in front of it and all these words are your answer.

## Find every subarray of a string

* Create a trie.
* Add word in the following way:
  + Lets say the word is bawika.
  + So add "bawika".
  + So add "awika".
  + So add "wika".
  + So add "ika".
  + So add "ka".
  + So add "a".
* Now basically do inorder traversal and everytime that function is called recursively add the string to the ArrayList.
* The ArrayList will have the answer at the end.