### Problem Solving Template:

1. First clearly understand the problem statement and write it in your own words.
2. Come up with test cases, try to cover as many edge cases.
3. In the first solution, aim for correctness of solution. Here you can use, brute force method.
4. Clearly explain the method in simple english. Write code and execute test cases
5. Analyze the time and space complexity of brute force method.
6. Now, aim for improving the performance of algorithms by coming up with different approaches to reduce inefficiency. Write pseudo code.
7. Implement code, run and debug all test cases, review complexity.

### Linear Search:

1. This is a brute force method to find a query in a list.
2. Starting at index 0 we compare each element with the query, while incrementing the index until the query is found.
3. No precondition of ordering of elements in list.
4. Time complexity is O(N)
5. Space complexity is O(1)

| def locate\_card(cards, query):  position = 0  while position < len(cards):  if cards[position] == query:  return position  position += 1  return -1 |
| --- |

### Binary search:

1. Precondition is list has to be sorted.
2. We reduce the list scope in half in each iteration to find the query element.
3. Time complexity is O(logN). Ex- query element will be found(if present) in at most 3 iterations for a list of size 10.
4. Space complexity is O(1) also called constant in space.
5. This general algorithm can be extended to solve a number of problems, where list are sorted.

| def binary\_search(lo, hi, condition):   while lo <= hi:  mid = (lo + hi) // 2  result = condition(mid)  if result == 'found':  return mid  elif result == 'left':  hi = mid - 1  else:  lo = mid + 1  return -1 |
| --- |

Example Problems:

1. Given a list, find a query number in a list.
2. Given a rotated list, find the min. number of times a sorted list was rotated to obtain a rotated list.

What is the difference between a list and a tree? - [Refer this](https://www.geeksforgeeks.org/difference-between-array-and-tree/)

1. An array is a collection of homogeneous(same type) data items stored in [contiguous memory locations](https://www.geeksforgeeks.org/difference-between-contiguous-and-noncontiguous-memory-allocation/).
2. Tree is a data structure with nodes and edges, tree also means the root node. Binary tree means the root node has 2 child nodes (left and right).
3. Binary Tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operations are as fast as in a linked list.
4. For linked lists, the list elements need not be stored in contiguous memory locations, since each node has reference to the next node.

## 

## Binary Tree:

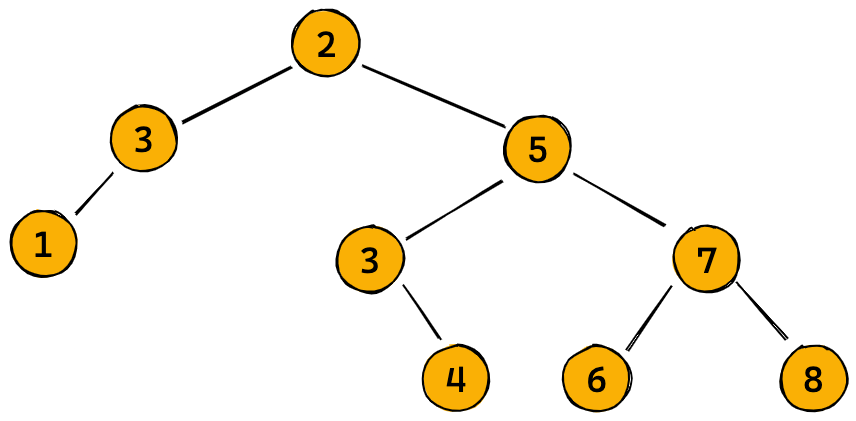
#### Example Problem Statement:

We have to create a database of 10 Million people, use a data structure that will optimize insert, update, find and list users in this database.

#### Solution:

1. With a linear approach; the insert, update and find methods will have O(N) time complexity. List method has time complexity O(1)
2. We can improve the complexity upto log(N) with BT.

### Tuple representation:



| tree\_tuple = ((1,3,None), 2, ((None, 3, 4), 5, (6, 7, 8))) |
| --- |

### Parsing Tree:

| def parse\_tuple(data):  # print(data)  if isinstance(data, tuple) and len(data) == 3:  node = TreeNode(data[1])  node.left = parse\_tuple(data[0])  node.right = parse\_tuple(data[2])  elif data is None:  node = None  else:  node = TreeNode(data)  return node |
| --- |

### Tree to Tuple:

| def to\_tuple(self):  if self is None:  return None  if self.left is None and self.right is None:  return self.key  return TreeNode.to\_tuple(self.left), self.key, TreeNode.to\_tuple(self.right) |
| --- |

### Displaying Tree:

| def display\_keys(node, space='\t', level=0):  # print(node.key if node else None, level)    # If the node is empty  if node is None:  print(space\*level + '∅')  return     # If the node is a leaf   if node.left is None and node.right is None:  print(space\*level + str(node.key))  return    # If the node has children  display\_keys(node.right, space, level+1)  print(space\*level + str(node.key))  display\_keys(node.left,space, level+1) |
| --- |

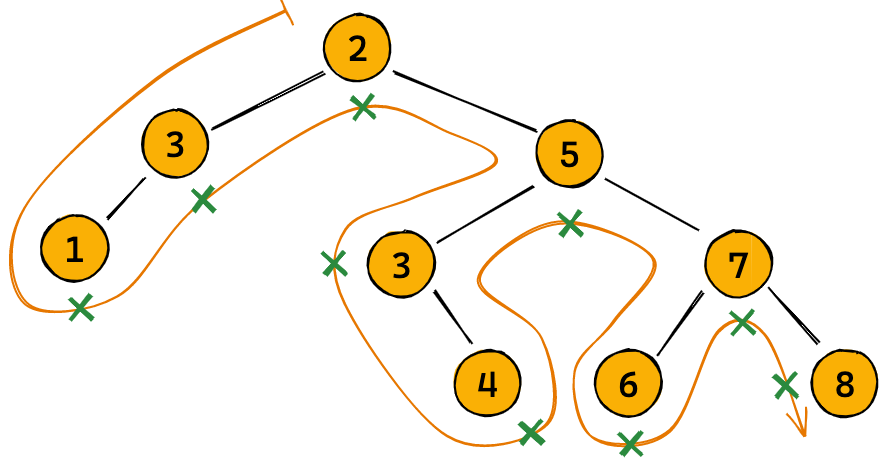
### 

### 

### 

### Tree Traversal:

#### In-order Traversal: (Left - Root - Right)



| def traverse\_in\_order(node):  if node is None:   return []  return(traverse\_in\_order(node.left) +   [node.key] +   traverse\_in\_order(node.right)) |
| --- |

#### Pre-order Traversal: (Root - Left - Right)

#### Post-order Traversal: (Left - Right - Root)

### BT Height:

| def tree\_height(node):  if node is None:  return 0  return 1 + max(tree\_height(node.left), tree\_height(node.right)) |
| --- |

### BT Size:

| def tree\_size(node):  if node is None:  return 0  return 1 + tree\_size(node.left) + tree\_size(node.right) |
| --- |

**Note-** Binary Tree will become a binary search tree, if the left subtree is less than root and right subtree is more than root.

Complete for BST

binary tree - height, size, traversal, display, parse, tree\_to\_tuple, is\_bst, For bst - insert, find, update and list, time and space complexity of this methods. is\_balanced, make\_balanced, self balancing bst (AVL method)

Hashtable in python for dictionary-

Created a hash function by using ord method on chars of key. get\_index, get\_valid\_index, linear probing idea, storing (key,value) at hash output index

Time complexity - average case is O(1) or worst case is O(n)

when is worst case? when all the keys give same hash output index, then complexity will be due to linear probing which is O(n)

Create comparison tables of time and space complexity

### Sorting methods:

#### Bubble Sort

| ## approach # start at index 0 # compare its value with value at index 1, if more swap else go to index 1 # repeat until there are no swaps in traversal of index from 0 to n-1, where n is length of list |
| --- |

| def sort(nums):  nums = list(nums)  while True:  num\_swaps = 0  for i in range(len(nums)-1):  if nums[i]>nums[i+1]:    tmp = nums[i]  nums[i]=nums[i+1]  nums[i+1]=tmp    num\_swaps +=1    if num\_swaps == 0: # means list is sorted  break   return nums |
| --- |
|  |

**Time Complexity**: O(k\*n) - max value of k is (n-1) in that case time complexity is O(n^2)

**Space Complexity**: O(1)

#### Insertion sort

| ## approach # insertion sort - keep the list sorted from left and put consecutive element from right # at the correct position, by comparing and moving in the left direction one position at a time... |
| --- |

| def sort\_insertion(nums):  nums = list(nums)  for i in range(1,len(nums)):  cur = nums.pop(i)  j=i-1  while j>=0 and cur < nums[j]:  j-=1  nums.insert(j+1,cur)    return nums |
| --- |

**Time Complexity**: O(n^2)

**Space Complexity**: O(1)

### Divide and Conquer

To performing sorting more efficiently, we'll apply a strategy called **Divide and Conquer**, which has the following general steps:

1. Divide the inputs into two roughly equal parts.
2. Recursively solve the problem individually for each of the two parts.
3. Combine the results to solve the problem for the original inputs.
4. Include terminating conditions for small or indivisible inputs.

#### Merge Sort

| ## approach # If the input list is empty or contains just one element, it is already sorted. Return it. # If not, divide the list of numbers into two roughly equal parts. # Sort each part recursively using the merge sort algorithm. You'll get back two sorted lists. # Merge the two sorted lists to get a single sorted list |
| --- |

| def merge\_sort(nums):  # Terminating condition (list of 0 or 1 elements)  if len(nums) <= 1:  return nums    # Get the midpoint  mid = len(nums) // 2    # Split the list into two halves  left = nums[:mid]  right = nums[mid:]    # Solve the problem for each half recursively  left\_sorted, right\_sorted = merge\_sort(left), merge\_sort(right)    # Combine the results of the two halves  sorted\_nums = merge(left\_sorted, right\_sorted)    return sorted\_numsdef merge\_sort(nums):  # Terminating condition (list of 0 or 1 elements)  if len(nums) <= 1:  return nums    # Get the midpoint  mid = len(nums) // 2    # Split the list into two halves  left = nums[:mid]  right = nums[mid:]    # Solve the problem for each half recursively  left\_sorted, right\_sorted = merge\_sort(left), merge\_sort(right)    # Combine the results of the two halves  sorted\_nums = merge(left\_sorted, right\_sorted)    return sorted\_nums |
| --- |

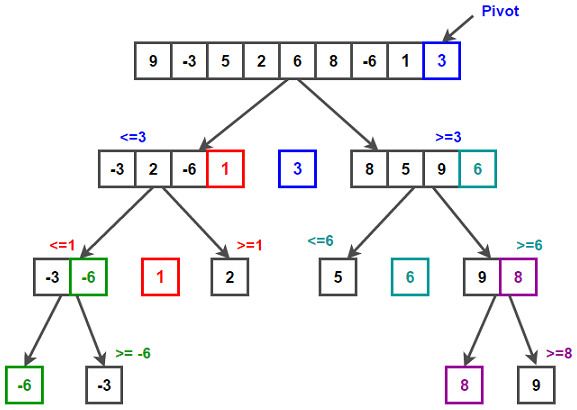
| def merge(nums1, nums2):   # List to store the results  merged = []    # Indices for iteration  i, j = 0, 0    # Loop over the two lists  while i < len(nums1) and j < len(nums2):     # Include the smaller element in the result and move to next element  if nums1[i] <= nums2[j]:  merged.append(nums1[i])  i += 1  else:  merged.append(nums2[j])  j += 1    # Get the remaining parts  nums1\_tail = nums1[i:]  nums2\_tail = nums2[j:]    # Return the final merged array  return merged + nums1\_tail + nums2\_tail |
| --- |

**Time Complexity:** O(N\*logN)), because to cover all N elements, the last depth k = log(N)

**Space Complexity**: O(N), because in recursion, a copy of all original list is made at the last depth level. Why not O(N\*logN) ? because intermediate space is freed, while splitting and merging.

#### Quick Sort

This is again application of divide and conquer method. We use a strategy to convert a large problem into 2 small problems, and the same strategy can be recursively applied until to the end of list.



| ## approach  # if the list is empty or has 1 element, return it as it is sorted. (BASE CONDITION) # Another way to write BASE CONDITION is when pivot is None # else we do PARTITIONING # AS [ # we keep the last element as pivot element # on the basis of pivot element we partition the other elements in list such that all the # elements smaller than pivot element are arranged on left side (order unimportant) # elements larger than pivot element are arranged on right side (order unimportant) # finally we bring the pivot element in between the smaller elements on the left side and larger elements on the right side. # Now we have [unsorted left list]-[pivot element at correct position]-[unsorted right list] # ]  # To do PARTITIONING- what is required? # 1. list # 2. start\_index ( 0 at first level of recursion, at BASE CONDITION and at all recursion levels for left sublist # AND pivot+1 for all other level of recursion for right sublist) # 3. end\_index ( len(nums)-1 at first level of recursion, at BASE CONDITION and at all recursion levels of # right sublist AND pivot-1 for all other recursion levels of left sublist)  # PARTITIONING returns the index of pivot in original list  # This is a good strategy, because now we can apply the same strategy recursively to smaller unsorted left and right sublists. # In each recursion we are correctly locating the pivot element. # The recursion ends when BASE CONDITION is reached. |
| --- |



| def quicksort(nums, start=0, end=None):  # print('quicksort', nums, start, end)  if end is None:  nums = list(nums)  end = len(nums) - 1    if start < end:  pivot = partition(nums, start, end)  quicksort(nums, start, pivot-1)  quicksort(nums, pivot+1, end)   return nums |
| --- |

| def partition(nums, start=0, end=None):  # print('partition', nums, start, end)  if end is None:  end = len(nums) - 1    # Initialize right and left pointers  l, r = start, end-1    # Iterate while they're apart  while r > l:  # print(' ', nums, l, r)  # Increment left pointer if number is less or equal to pivot  if nums[l] <= nums[end]:  l += 1    # Decrement right pointer if number is greater than pivot  elif nums[r] > nums[end]:  r -= 1    # Two out-of-place elements found, swap them  else:  nums[l], nums[r] = nums[r], nums[l]  # print(' ', nums, l, r)  # Place the pivot between the two parts  if nums[l] > nums[end]:  nums[l], nums[end] = nums[end], nums[l]  return l # return the index of pivot element  else:  return end # return the index of pivot element |
| --- |

**Time Complexity:**

1. **For Best Case Partitioning condition**: ( When the pivot index after partitioning is roughly in the middle of list/sublist) **O(N\*logN)** - To sort a list completely, we have to make every element in the list as a pivot element - guaranteed correct location in the original list. So, for a list of size N, we need N pivot elements and a divide and conquer method that translates into a recursive tree of k levels, where k = log(N+1) (why? Because we are calling two recursive calls in each recursion level so N = 1 + 2^1 + 2^2 + . . . + 2^(k-1) )
2. **For Worst Case Partitioning condition**: ( When the pivot index is at extreme position in partitioned list/sublist). **O(n^2)**

**Space Complexity:** **O(1)**