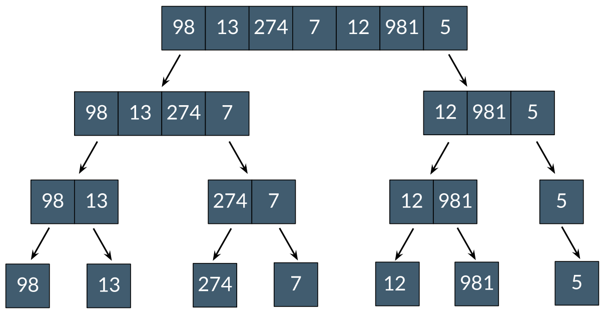
Cheatsheets / **Learn Data Structures and Algorithms with Python**

Divide and Conquer

Divide and Conquer Algorithms

- Divide-and-conquer solves a large problem by recursively breaking it down into smaller subproblems until they can be solved directly.
- Divide-and-conquer works in three steps: divide, conquer, and combine.
- It is more efficient than brute force approaches.
- It is prone to stack overflow error due to the use of recursion.



Complexity of Binary Search

A dataset of length n can be divided *log n* times until everything is completely divided. Therefore, the search complexity of binary search is O(log n).

Iterative Binary Search

A binary search can be performed in an iterative approach. Unlike calling a function within the function in a recursion, this approach uses a loop.

```
function binSearchIterative(target, array,
left, right) {
  while(left < right) {
    let mid = (right + left) / 2;
    if (target < array[mid]) {
      right = mid;
    } else if (target > array[mid]) {
      left = mid;
    } else {
      return mid;
    }
  }
  return -1;
}
```

Base case in a binary search using recursion

In a recursive binary search, there are two cases for which that is no longer recursive. One case is when the middle is equal to the target. The other case is when the search value is absent in the list.

```
binary_search(sorted_list, left_pointe
  if (left_pointer >= right_pointer)
    base case 1
  mid_val and mid_idx defined here
  if (mid_val == target)
    base case 2
  if (mid_val > target)
    recursive call with left pointer
  if (mid_val < target)
    recursive call with right pointer
```

Updating pointers in a recursive binary search

In a recursive binary search, if the value has not been found then the recursion must continue on the list by updating the left and right pointers after comparing the target value to the middle value.
If the target is less than the middle value, you know the target has to be somewhere on the left, so, the right pointer must be updated with the middle index. The left pointer will remain the same. Otherwise, the left pointer must be updated with the middle index while the right pointer remains the same. The given code block is a part of a function binarySearchRecursive() .

```
function binarySearchRecursive(array,
first, last, target) {
  let middle = (first + last) / 2;
  // Base case implementation will be in
here.

  if (target < array[middle]) {
    return binarySearchRecursive(array,
first, middle, target);
  } else {
    return binarySearchRecursive(array,
middle, last, target);
  }
}
```

Binary Search Sorted Dataset

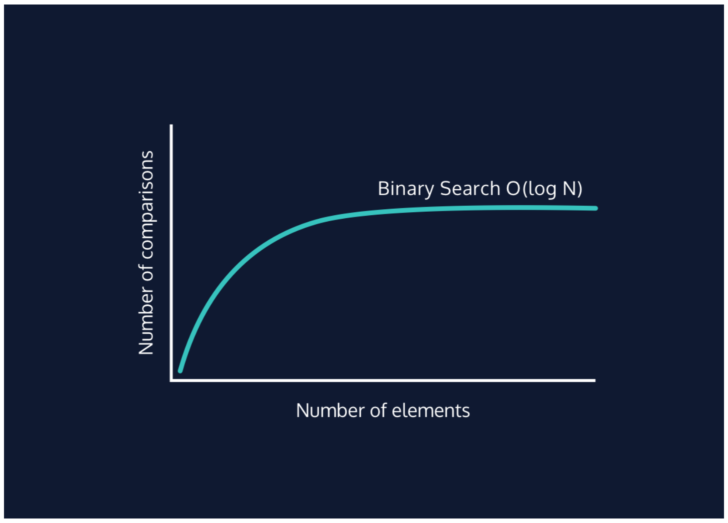
Binary search performs the search for the target within a sorted array. Hence, to run a binary search on a dataset, it must be sorted prior to performing it.

Operation of a Binary Search

The binary search starts the process by comparing the middle element of a sorted dataset with the target value for a match. If the middle element is equal to the target value, then the algorithm is complete. Otherwise, the half in which the target cannot logically exist is eliminated and the search continues on the remaining half in the same manner.
The decision of discarding one half is achievable since the dataset is sorted.

Binary Search Performance

The binary search algorithm takes time to complete, indicated by its time complexity. The worst-case time complexity is $O(\log N)$. This means that as the number of values in a dataset increases, the performance time of the algorithm (the number of comparisons) increases as a function of the base-2 logarithm of the number of values.
Example: Binary searching a list of 64 elements takes at MOST $\log 2(64)$ = 6 comparisons to complete.



Binary Search

The binary search algorithm efficiently finds a goal element in a sorted dataset. The algorithm repeatedly compares the goal with the value in the middle of a subset of the dataset. The process begins with the whole dataset; if the goal is smaller than the middle element, the algorithm repeats the process on the smaller (left) half of the dataset. If the goal is larger than the middle element, the algorithm repeats the process on the larger (right) half of the dataset. This continues until the goal is reached or there are no more values.

1
2
7
8
22
28
41
58
67
71
94

Recursive Binary Search

Binary search can be implemented using recursion by creating a function that takes in the following arguments: a sorted list, a left pointer, a right pointer, and a target. The base cases must account for when the left and right pointers are equal, as well as when the target is found, in which case the index of the array is returned.
Initially, set the left pointer to index 0 of the list and right pointer to the last index. If middle value > target, right pointer = middle value. If middle value < target, left pointer = middle value. Call the binary search function with the properly adjusted pointers.

```
def binary_search(sorted_list,
left_pointer, right_pointer, target):
  if left_pointer >= right_pointer:
    #base case 1
        #mid_val and mid_idx defined here
  if mid_val == target:
    #base case 2
  if mid_val > target:
    #recursive call with left pointer
  if mid_val < target:
    #recursive call with right pointer
```

Depth First Traversal

The BinarySearchTree Python class has a .depth_first_traversal() instance method that prints the in-order depth-first traversal of the tree. The output will always be in ascending order. It takes no variables and returns nothing.

```
def depth_first_traversal(self):
  if (self.left is not None):
    self.left.depth_first_traversal()
  print(f'Depth={self.depth}, Value=
{self.value}')
  if (self.right is not None):
    self.right.depth_first_traversal()
```

Getting a Node by Value

The BinarySearchTree Python class has a .get_node_by_value() instance method that takes in a value and returns the corresponding BinarySearchTree node, or None if it doesn't exist. The method uses recursion to search through the sides of the tree. On an averagely balanced binary search tree with N nodes, the performance would be $O(\log N)$, just like the Binary Search algorithm.

```
def get_node_by_value(self, value):
  if (self.value == value):
    return self
  elif ((self.left is not None) and
(value < self.value)):
    return
self.left.get_node_by_value(value)
  elif ((self.right is not None) and
(value >= self.value)):
    return
self.right.get_node_by_value(value)
  else:
    return None
```

Insertion

The BinarySearchTree Python class has an .insert() method that takes in a value and uses recursion to add a new node to the tree while maintaining the binary tree property. The method returns nothing. On an averagely balanced binary search tree with N nodes, the performance would be $O(\log N)$.

```
def insert(self, value):
  if (value < self.value):
    if (self.left is None):
      self.left = BinarySearchTree(value, self.depth + 1)
      print(f'Tree node {value} added to the left of {self.value} at depth {self.depth + 1}')
    else:
      self.left.insert(value)
  else:
    if (self.right is None):
      self.right = BinarySearchTree(value, self.depth + 1)
      print(f'Tree node {value} added to the right of {self.value} at depth {self.depth + 1}')
    else:
      self.right.insert(value)
```

Constructor

The Python implementation of the BinarySearchTree class should contain value and depth instance variables, as well as left and right pointers. The constructor has the following parameters:

- value
- depth , which has a default value of 1

The left and right pointers are set to None in the constructor.

```
def __init__(self, value, depth=1):
  self.value = value
  self.depth = depth
  self.left = None
  self.right = None
```

Print Share