Unit IV: Introduction to Computational Problems & Algorithms

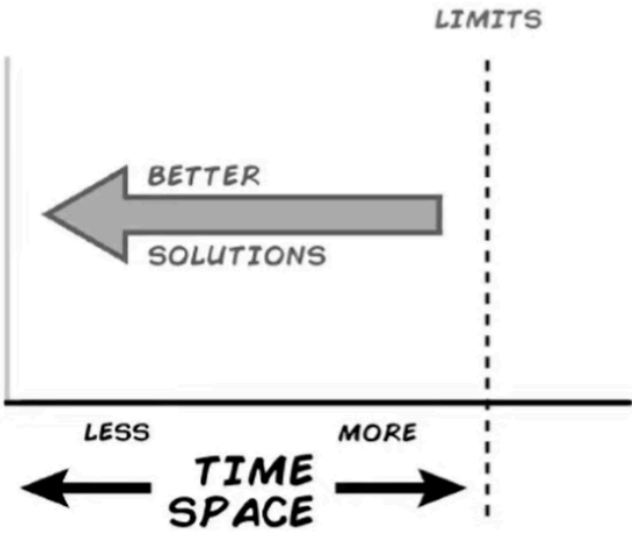
4.1 Space & Time Complexity, Asymptotic Notation

Generally, there can be multiple algorithms to solve a particular problem. Each algorithm may take a different amount of time to execute, depending on factors like the size of the input and the operations performed. Therefore, it is highly required to use the criteria to compare the solutions in order to judge which one is more optimal.

The two criterias are as follows:

1. **The time complexity** of an algorithm quantifies the amount of time taken by an
2. algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

3. Problem-solving using a computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve a given problem is called **space complexity** of the algorithm.



Asymptotic Notation will be used to represent both time and space complexity of an algorithm. It is like a unit of measurement. Whereby, liters are used to measure the liquid, meters to measure the distance. Likewise, the complexity of an algorithm can be measured using asymptotic notation.

For example, you are preparing for the exam. You are trying to revise the lecture over and over again, then explore different sources thinking the exam question might be tougher than you expected, and you should at least secure a pass mark. Here, you are preparing for the average. However, if you are preparing thinking that you might just flunk the exam, then you are

preparing for the worst case. Likewise, if you are preparing thinking that you should ace the exam, you are preparing for the best case. Similarly, there are three different notations to describe an algorithm:

1. big-O : For the worst case (Upper Bound)
   It will calculate the longest time of the algorithm and largest space.

2. big -Omega (Ω) : For the best case (Lower Bound)
   It will calculate the shortest time of the algorithm.

3. big-Theta (Θ): Tight bound i.e. it is between the best and the worst case (Average Bound)

Generally, we use big-O notation to represent all the three cases i.e. worst, best and average cases as big-O represents the upper bound, so best and average cases fall within its range. Additionally, big-O notation is simpler to denote.

(1< logn < n^(1/2) < n < n^2 < ….< n^n), O(1) is the best time and space complexity, whereas the rightmost one, O(n^n) is the worst time and space complexity. In other words, if an algorithm has time and space complexity of O(1), this particular algorithm is the best one.

Run time complexity

It is a relation between input and the number of operations performed in it. Run time is calculated only for executable statements and NOT for declarative statements.
Some of the factors affecting time complexity are:

1. Operations
2. Comparisons
3. Loops
4. Pointer References
5. Function Calls

```python
def sum_elements(arr):
    total = 0  # O(1) - Constant time complexity. Initializing a variable requires a fixed amount of time.
    for num in arr:
        total += num  # O(1) - Constant time complexity. Adding a number to 'total' takes a fixed amount of
        time.
    return total  # O(1) - Constant time complexity. Returning the final sum does not depend on the size of the
    input.

# Example usage:
my_list = list(map(int, input("Enter the numbers separated by whitespace: ").split()))  # O(n) - Linear time
complexity. Splitting the input string and converting each element to an integer.
print(sum_elements(my_list))  # Output: sum of elements entered by the user
```

Space Complexity

Auxiliary space is nothing but the space required by an algorithm/problem during the execution of that algorithm/problem and it is not equal to the space complexity because space complexity includes space for input values along with it also.

Space Complexity = Auxiliary Space + Space used for input values

Some of the factors affecting space complexity are:

1. Variables
2. Data Structures
3. Allocations
4. Function calls

```python
def duplicate_elements(arr):
    seen = set()  # O(n) - Linear space complexity. Creating a set to store seen elements.
    duplicates = []  # O(n) - Linear space complexity. Creating a list to store duplicate elements.

    for num in arr:  # O(n) - Linear space complexity. Iterating through each element of the input list.
        if num in seen:  # O(1) - Constant space complexity. Checking if the element has been seen before.
            duplicates.append(num)  # O(1) - Constant space complexity. Appending the duplicate element to the
            list.
        else:
            seen.add(num)  # O(1) - Constant space complexity. Adding the element to the set of seen elements.

    return duplicates  # O(n) - Linear space complexity. Returning the list of duplicate elements.
```

4.2 Searching & Sorting Algorithms; Linear Search, Binary Search, Bubble Sort, Quick Sort, Insertion Sort

Sorting

It is just arranging available data in some specific format.

Some of the importance of sorting are as follows:

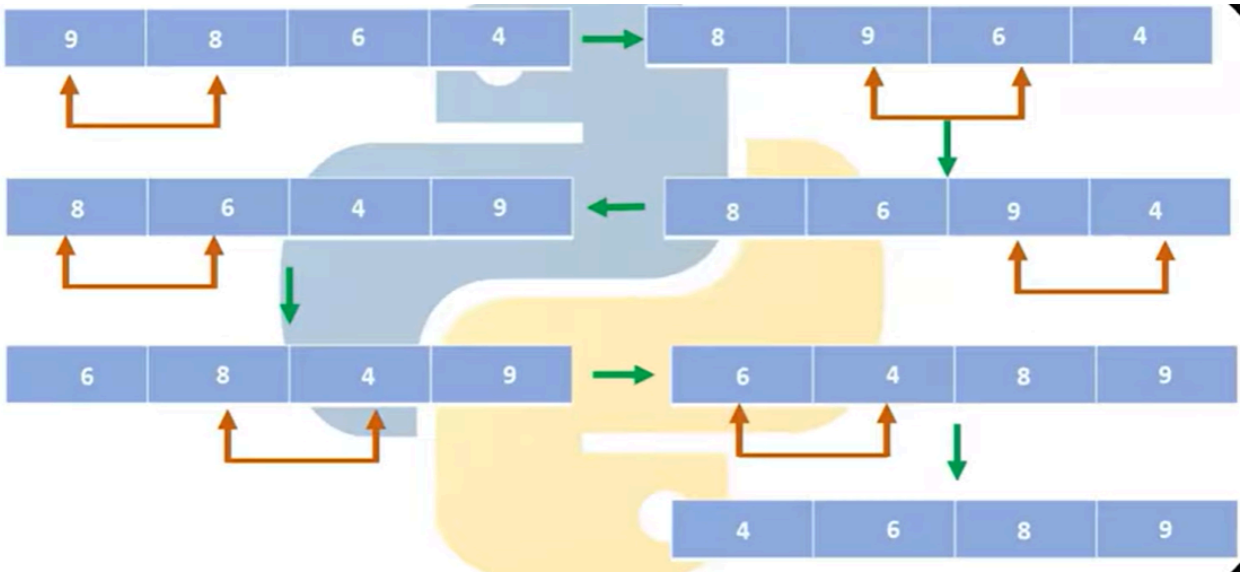
1. Searching is made faster
2. Helps to locate various patterns in data
3. Can easily track duplicate values

There are different types of sorting algorithms:

1. Bubble Sort

   The first element is compared with the adjacent element and swaps them if they aren't in the order. Then it again compares the second element with the adjacent element and swaps it if they are not in order. It continues this process until it reaches the last element.

   Eg: You are sorting the items in a given list [9,8,6,4] as follows:




```
# Define a function named bubbleSort that implements the bubble sort algorithm.
def bubbleSort(arr):
    # Iterate through the entire array.
    for i in range(len(arr)):
        # Iterate through the unsorted part of the array to swap its place in right order.
        # The range is from 0 to len(arr) - i - 1 because number of positions that can be swapped reduces after each iteration(i.e. in each iteration, an element is being sorted in its right position),
        # the largest element bubbles up to the end of the array, so we can ignore it in subsequent iterations.
        for j in range(0, len(arr) - i - 1):
            # Compare adjacent elements and swap them if they are in the wrong order.
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # Swap elements
                # Print the current state of the array after each swap
                print(f"i: {i}, j: {j} arr: {arr} ")
    # Return the sorted array.
    return arr

# Define an input array.
arr = [9, 8, 6, 4]
```

```
# Call the bubbleSort function to sort the input array.
print(bubbleSort(arr))
```

Time Complexity

● Best Case: O(n)

When an array is already sorted, in the first iteration no swaps are performed. Knowing that no swaps are required, we can stop the sorting. Hence, the best case time complexity is linear, i.e., O(n).

● Average Case: O(n^2)

For a random array, the average number of total swaps is (n^2)/4, thus the average case complexity is O(n^2).

● Worst Case: O(n^2)

When the array is sorted in reverse order, in the first pass (n-1) swaps are performed, in the second (n-2) swaps, and so on. So the total number of swaps is n*(n-1)/2, and the overall complexity is O(n^2).

Space Complexity

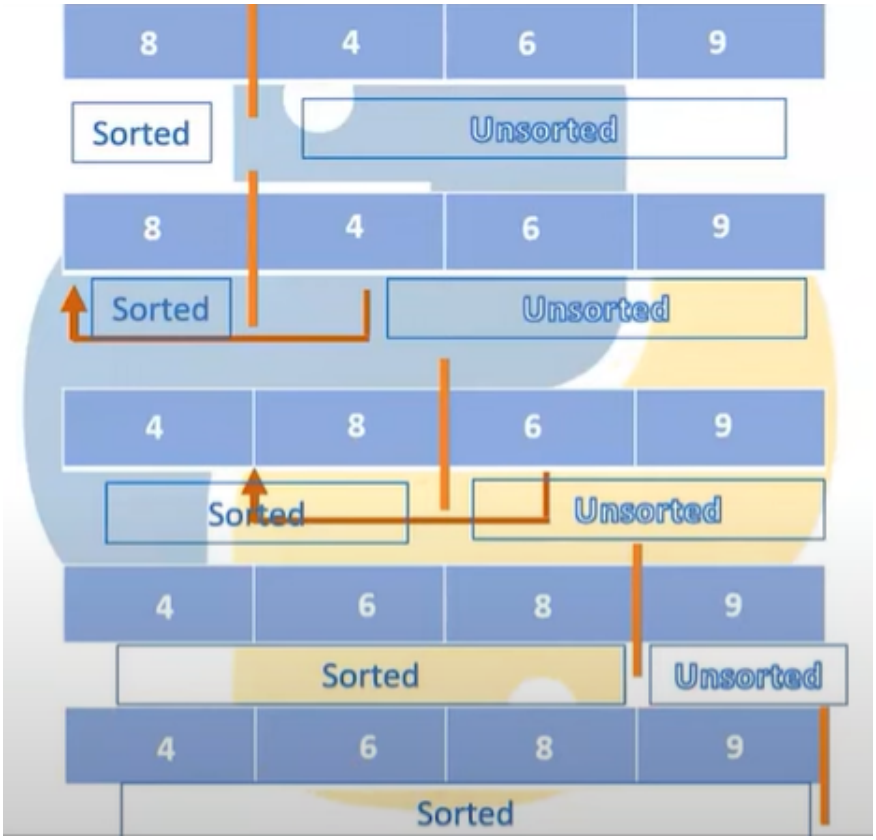
No extra memory is used by the algorithm apart from the original array, so the space complexity is O(1).

2. Insertion Sort

It places the given element at the right position in the sorted list.

In insertion sort, it is always considered that the first element is always sorted and the rest are unsorted.

Eg: sorting the elements in the list [8, 4, 6, 9]




```
# Define a function named insertion_sort that implements the insertion sort algorithm.
def insertion_sort(arr):
    # Iterate through the array starting from the second element (index 1).
    for i in range(1, len(arr)):
        # Initialize the variable j to the index before i. Which means that, j is the index of sorted part of the array.
        j = i - 1
            # Move elements of arr[0..i-1], that are greater than arr[i], to one position ahead of their current position.
        # This loop moves elements to the right until the correct position for arr[i] is found.
        while j >= 0 and arr[j + 1] < arr[j]:
            # Swap arr[j] and arr[j+1].
            arr[j + 1], arr[j] = arr[j], arr[j + 1]
            # Print the current state of the array after each swap
            print(f"i: {i}, j: {j} arr: {arr}"  )
```

```
                # Decrement j to continue comparing with previous
elements. That unsorted element should compare to every sorted
element to find its correct position.
            j -= 1
    # Return the sorted array.
    return arr

# Test the insertion_sort function with an input array.
print(insertion_sort([8, 4, 6, 9]))
```

Time Complexity

● Best Case: O(n)

When an array is already sorted, the algorithm picks the first element from the unsorted subarray and puts it at the end of the sorted subarray. So, the best-case complexity is O(n).

● Average Case: O(n^2)

For a random array, the average-case time complexity is O(n^2).

● Worst Case: O(n^2)

When the array is sorted in reverse order, the algorithm picks the first element from the unsorted subarray and puts it at the beginning of the sorted subarray. Since the total number of comparisons is n*(n-1)/2, the worst-case time complexity is O(n^2).

Space Complexity

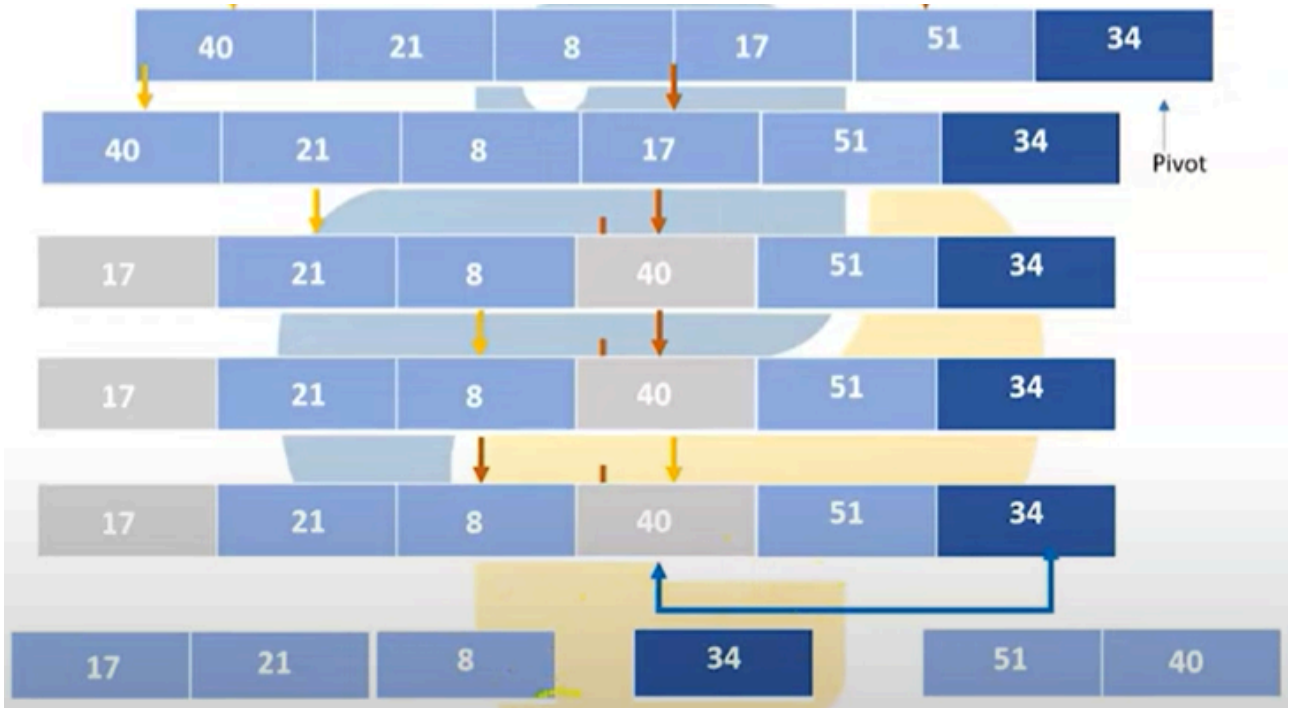
Since no extra memory is used apart from the original array, the space complexity is O(1).

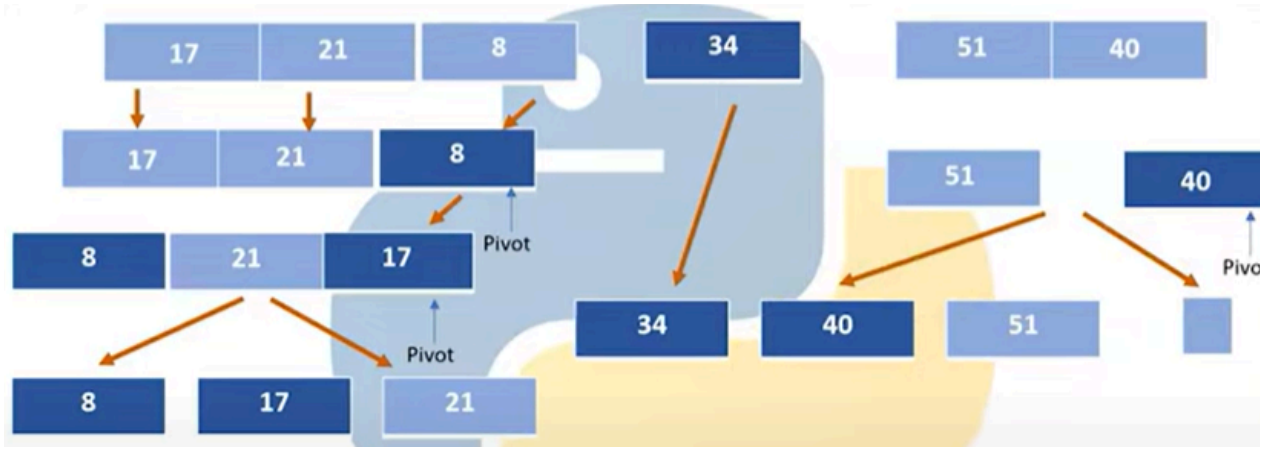
3. Quick Sort

   It is a divide and conquer algorithm.

   Firstly, a pivot element in the array is chosen. Then, the elements which are less than the pivot element are stored in the left sub array, likewise the elements that are greater than the pivot element are stored in the right subarray. Generally, pivot element is taken as the right most element (last element) in an array.

   Eg: Sorting the elements in the list [40, 21, 8, 17, 51, 34]






```
def quick_sort(arr):
    # Base case: If the length of the list is 0 or 1, it is already
sorted.
    if len(arr) <= 1:
        return arr
    else:
        # Choosing the pivot as the last element of the list.
        pivot = arr[-1]
        print(f"pivot: {pivot}")

        # Partitioning the list into two sublists:
           # - 'left' contains elements less than or equal to the
pivot.
```

```
        # - 'right' contains elements greater than the pivot.
        left = [x for x in arr[:-1] if x <= pivot]
        right = [x for x in arr[:-1] if x > pivot]
        print(f"left: {left}, right: {right}")

           # Recursively applying quick_sort to the left and right
sublists,
        # and concatenating the sorted sublists with the pivot.
        return quick_sort(left) + [pivot] + quick_sort(right)

# Example usage:
my_list = [40, 21, 8, 17, 51, 34]
sorted_list = quick_sort(my_list)
print("Sorted list:", sorted_list)
```

Time Complexity

● Best Case: O(n log n)

If every time we pick the median element as the pivot, then it creates O(log n) subarrays. In this case, the overall best-case time complexity is O(n log n).

● Average Case: O(n logn)

For a random array, the average-case time complexity is O(n log n).

● Worst Case: O(n^2)

When the array is already sorted, we select the rightmost element as the pivot. In this case, each partitioning step divides the array into one sub-array with n - 1 elements and another with 0 elements. Each subarray requires linear time for partitioning; hence, its overall worst-case complexity is O(n^2).

Space Complexity

Worst-case scenario: O(n) due to unbalanced partitioning leading to a skewed recursion tree requiring a call stack of size O(n).

Best-case scenario: O(log n) as a result of balanced partitioning leading to a balanced recursion tree with a call stack of size O(log n).

Searching Algorithm

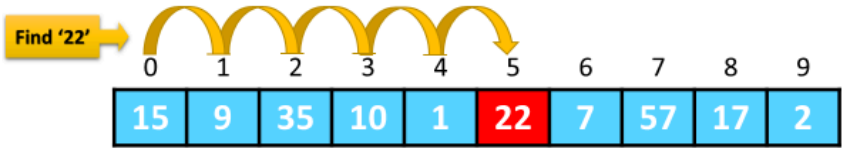
Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications.

The two types of searching algorithms are:

1. Linear Search

   Assume that item is in an array in random order and we have to find an item. Then the only way to search for a target item is, to begin with, the first position and compare it to the target. If the item is the same, we will return the position of the current item. Otherwise, we will move to the next position. If we arrive at the last position of an array and still can not find the target, we return -1. This is called the Linear search or Sequential search.

   Eg: To find the element “22” form the given list [15, 9, 35, 10, 1, 22, 7, 57, 17, 2]




```
# Define a function named linearSearch that performs a linear search
to find the index of a given element in an array.
def linearSearch(array, n, x):
    # Iterate over each element of the array.
    for i in range(0, n):
          # Check if the current element matches the target element
(x).
        if (array[i] == x):
            # If found, return the index of the element.
            return i
          # Print the current iteration index and the corresponding
element in the array
        print(f"i: {i}, array[i]: {array[i]}")
    # If the target element is not found in the array, return -1.
    return -1

# Call the linearSearch function with a sample array, its size, and
the element to search for.
print(linearSearch([15, 9, 35, 10, 1, 22, 7, 57, 17, 2], 10, 22))
```

Complexity Analysis

The time complexity of the Linear Search algorithm is O(n), where n is the number of elements in the array. The space complexity is O(1) as it requires a constant amount of extra space regardless of the input size.

Best Case Time Complexity of Linear Search Algorithm: O(1)

Best case is when the list or array's first element matches the target element. The time complexity is O(1) because there is just one comparison made. So the best case complexity is O(1).

Average Case Time Complexity of Linear Search Algorithm: O(n)

The average case time complexity of the linear search algorithm is O(n), where n is the number of elements in the array being searched.

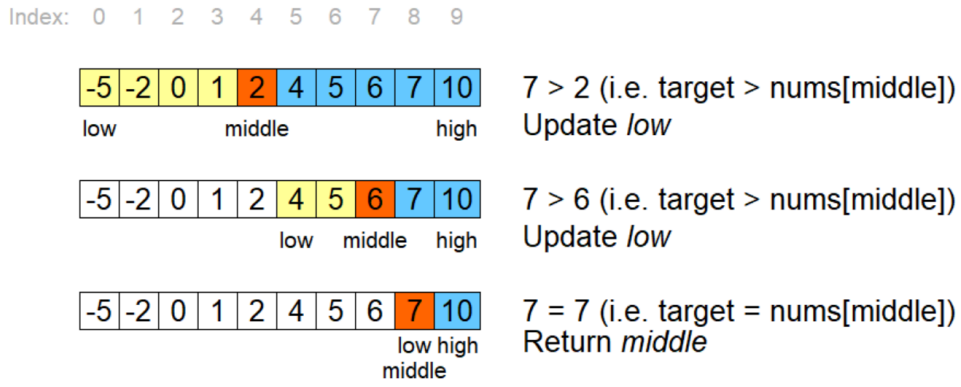
Worst Case Time Complexity of Linear Search Algorithm: O(n)

The worst-case will be when the target element is absent from the list or array or is located at the end of the list. O(n) time complexity results from the necessity to traverse the complete list and the n comparisons that are required.

2. Binary Search

In a binary search, however, cut down your search to half as soon as you find the middle of a sorted list. The middle element is looked at to check if it is greater than or less than the value to be searched. Accordingly, a search is done to either half of the given list.

Eg: Searching element "7" from the sorted list [-5, -2, 0, 1, 2, 4, 5, 6, 7, 10]




```
def binarySearch(array, x, low, high):
    # Repeat until the pointers low and high meet each other
    while low <= high:
        mid = low + (high - low)//2  # Calculate the middle index

         if array[mid] == x:  # If the middle element is the target
value, return its index
            print(f"The value {x} is at index:")
```

```
            return mid

        elif array[mid] < x:  # If the middle element is less than
the target value, search the right half by updating low
            low = mid + 1
         else:  # If the middle element is greater than the target
value, search the left half by updating high
            high = mid - 1

    return -1  # Return -1 if the target value is not found in the
array

# Example usage:
print(binarySearch([-5, -2, 0, 1, 2, 4, 5, 6, 7, 10], 7, 0, 9))
```

Complexity Analysis

Time complexity of Binary Search is O(log n), where n is the number of elements in the array. It divides the array in half at each step. Space complexity is O(1) as it uses a constant amount of extra space.

Best Case Time Complexity of Binary Search

The best case scenario of Binary Search occurs when the target element is in the central index. In this situation, there is only one comparison. Therefore, the Best Case Time Complexity of Binary Search is O(1).

Average Case Time Complexity of Binary Search

The average case arises when the target element is present in some location other than the central index or extremities. The time complexity depends on the number of comparisons to reach the desired element.

Therefore, the overall Average Case Time Complexity of Binary Search is O(logn).

Worst Case Time Complexity of Binary Search

The worst-case scenario of Binary Search occurs when the target element is the **smallest element or the largest element ** of the sorted array.

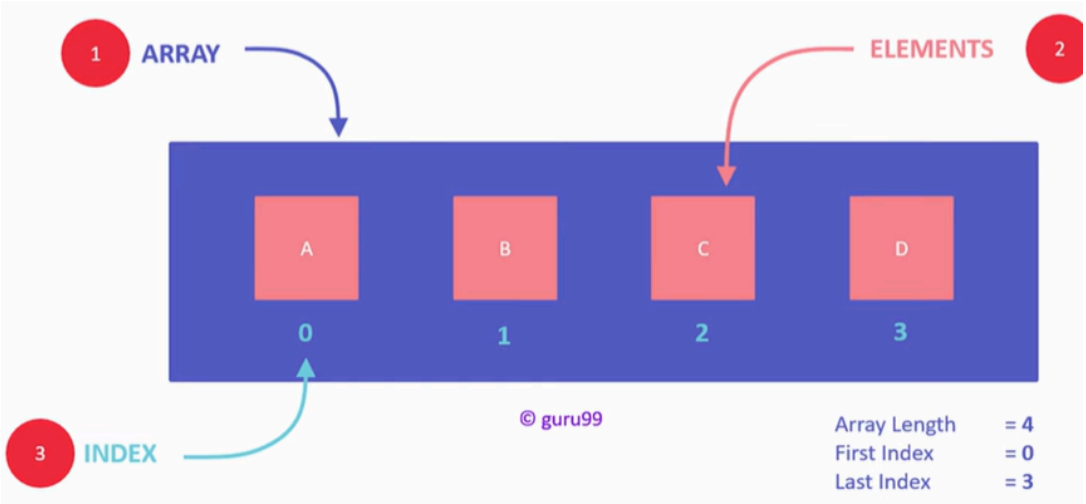
In each iteration or recursive call, the search gets reduced to half of the array.
So for an array of size n, there are at most log2n iterations or recursive calls.

Since the target element is present in the extremities (first or last index), there are logn comparisons in total. Therefore, the Worst Case Time Complexity of Binary Search is O(logn).

4.3 Arrays & Hashing; Contains Duplicate, Valid Anagram, Two Sums

Brief recapitulation of arrays:

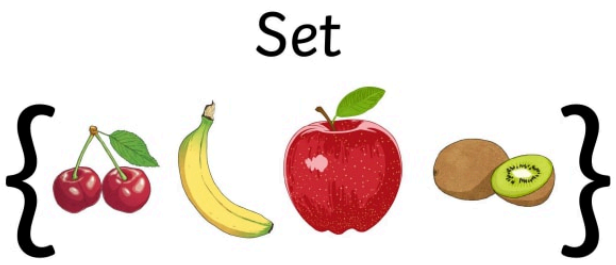
Arrays are the collection of the same type of elements that are stored in a contiguous memory location (adjacent to each other) of fixed size. Each element in an array has an index which makes it easier to access and manipulate the data.



However, arrays are not the best choice for frequent insertion and removal of the element.

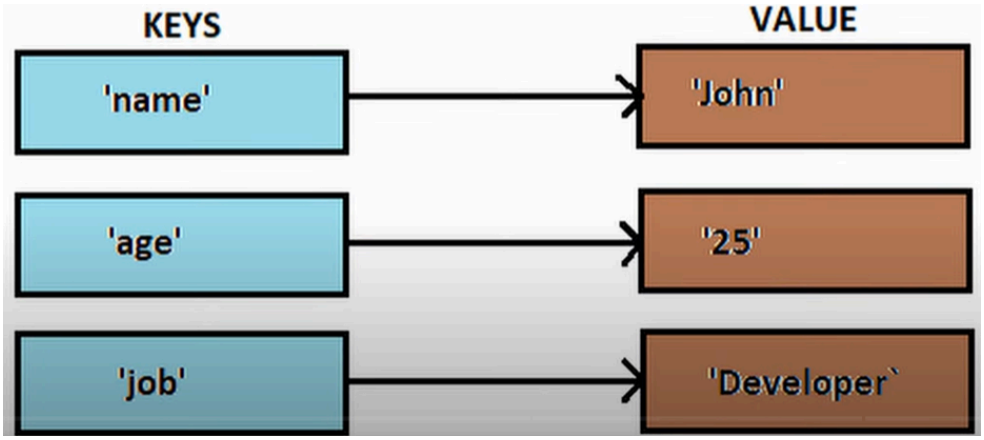
Brief recapitulation of Sets/Hashsets:

Set is also a linear data structure which stores the unique elements in it. Sets are also known as hashsets.



Brief recapitulation of Dictionary/Hashmap:

Dictionary is a linear data structure whereby it contains key-value pairs in it. Each value is associated with a unique key.



Sample python Code for LeetCode Problems (***Note that comments are just for your better understanding***):

1. **Contains-Duplicate:**

```
# Define a class named Solution. This class will contain a method to
determine if a list contains any duplicates.
class Solution(object):

      # Define a method named containsDuplicate that takes a list of
numbers (nums) as input.
    def containsDuplicate(self, nums):

          # Create an empty set to store unique elements encountered so
far.
        hash_set = set()

        # Iterate through each element (i) in the input list (nums).
        for i in nums:

                  # Check if the current element (i) is already in the
hash_set.
            if i in hash_set:

                    # If the current element is already in the hash_set,
return True, indicating that a duplicate is found.
                return True

              # If the current element is not in the hash_set, add it to
the hash_set.
            hash_set.add(i)
```

```
        # If the loop completes without finding any duplicates, return
False, indicating that no duplicates were found.
        return False
```

Time Complexity : O(n)
“n” is the number of elements in the input list. This is because the function iterates through each element in the input list once to check for duplicates.

Space Complexity: O(n)
This is because the function uses a set (hash_set) to store unique elements encountered.

2. **Valid Anagram:**

```
# Define a class named Solution. This class will contain a method to
determine if two strings are anagrams of each other.
class Solution(object):

    # Define a method named isAnagram that takes two strings, s and t, as
input.
    def isAnagram(self, s, t):

        # Check if the lengths of the two strings are different.
        if len(s) != len(t):
                # If the lengths are different, the strings cannot be
anagrams, so return False.
            return False

         # Create empty dictionaries to store the frequency of characters
in both strings. The character or a letter in string will be key, whereas
frequency number of that letter will be a value in dictionary.
        s_freq = {}
        t_freq = {}

        # Calculate the frequency of characters in string s.
        for char in s:
            if char in s_freq:
                s_freq[char] += 1 #value of that char gets incremented by
1
            else:
                    s_freq[char] = 1 #assigning a value as 1 for that
particular char(i.e. key) of that dictionary
```

```
        # Calculate the frequency of characters in string t.
        for char in t:
            if char in t_freq:
                t_freq[char] += 1
            else:
                t_freq[char] = 1

        # Check if the frequency dictionaries for both strings are equal.
        return s_freq == t_freq
```

Time Complexity: O(n)
The function iterates through each character of both input strings once, performing constant-time operations such as dictionary lookups and comparisons. Therefore, the overall time complexity is O(n).

Space Complexity: O(n)
The function uses extra two dictionaries (s_freq and t_freq) to store the frequencies of characters in the strings.

3. Two Sums:

```
# Define a class named Solution. This class will contain a method to find two numbers in a list that sum up to a given target.
class Solution(object):

    # Define a method named twoSum that takes a list of numbers (nums) and a target number as input.
    def twoSum(self, nums, target):

        # Create an empty dictionary to store the indices of numbers encountered so far.
        num_dict = {}

        # Iterate through the list of numbers (nums) along with their indices.
        for i, num in enumerate(nums):

            # Calculate the difference needed to achieve the target sum.
            difference = target - num

            # Check if the difference exists in the dictionary.
            if difference in num_dict:
                # If the difference exists, return the indices of the two numbers that sum up to the target.
```

```
                return [num_dict[difference], i]

                # Store the current number along with its index in the
dictionary.
            num_dict[num] = i

        # If no such pair is found, return an empty list.
        return []
```

Time Complexity: O(n)
The function iterates through each element of the input list once and performs constant-time operations inside the loop. Therefore, the overall time complexity is O(n), where n is the number of elements in the input list nums

Space Complexity: O(n)
The function uses a dictionary (num_dict) to store the indices of elements encountered so far.

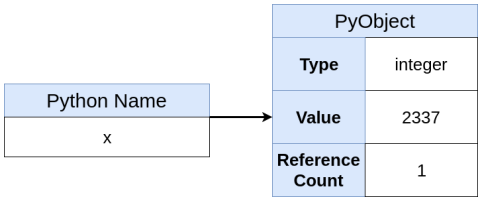
4.4 Two Pointers; Valid Palindrome, Three Sums
Pointer:
Essentially, pointers are variables that store the memory address of another variable. With pointers, you can access and modify data located in memory, pass data efficiently between functions, and create dynamic data structures like linked lists, trees, and graphs.

However, in python there's no pointer but we can mimic the behavior of the pointer in it.

```
>>> x = 2337
```

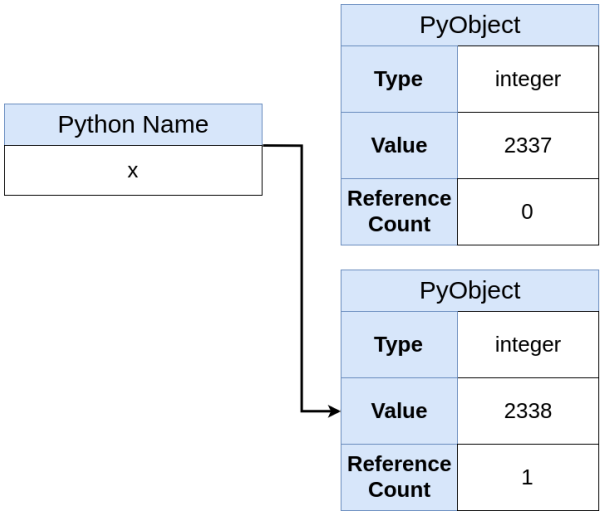
The above code is broken down into several distinct steps during execution:

1. Create a PyObject (PyObject' is a structure that only contains the reference count and the type pointer.)
2. Set the typecode to integer for the PyObject
3. Set the value to 2337 for the PyObject
4. Create a name called x
5. Point x to the new PyObject
6. Increase the refcount of the PyObject by 1



Now you are making following changes to the existing code in python:

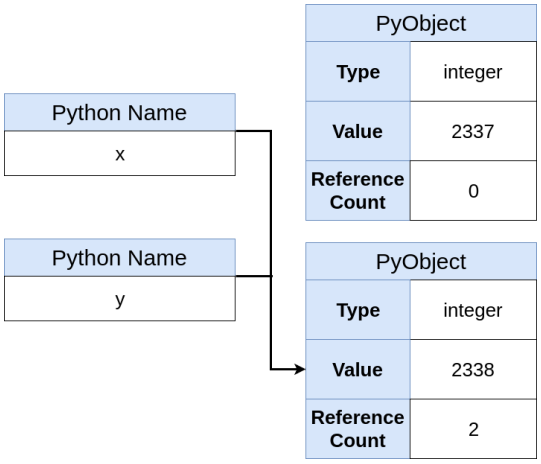
```
>>> x = 2338
```



This diagram helps illustrate that x points to a reference to an object and doesn't own the memory space. It also shows that the x = 2338 command is not an assignment, but rather binding the name x to a reference.

In addition, the previous object (which held the 2337 value) is now sitting in memory with a ref count of 0 and will get cleaned up by the garbage collector.

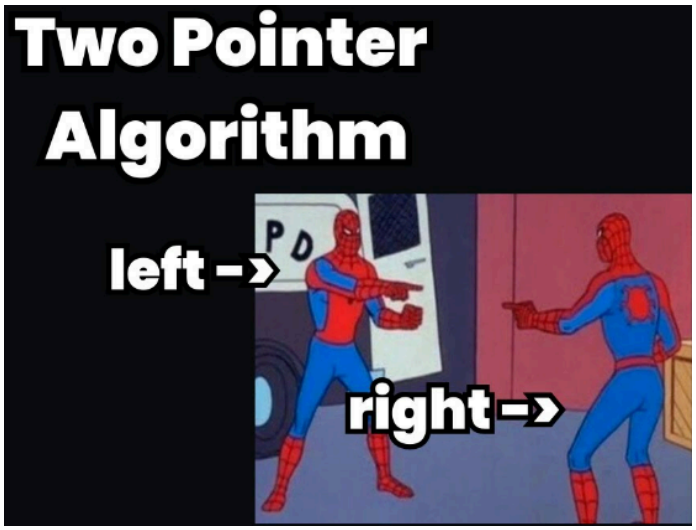
```
>>> y = x
```



Now you can see that a new Python object has not been created, just a new name that points to the same object. Also, the object's refcount has increased by one.

Two Pointers:

Two pointers is really an effective technique that is typically used for searching pairs in a sorted array. Imagine two people standing at two ends of an array and trying to find a particular pair of elements that fulfill a specific condition. These two individuals are our “pointers”. Depending on the condition, they might move towards each other or in the same direction.



Sample python code for LeetCode problems:

1. Valid Palindrome:

```
# Define a class named Solution. This class will contain a method to
check if a given string is a palindrome.
class Solution(object):

      # Define a method named isPalindrome that takes a string (s) as
input.
    def isPalindrome(self, s):

            # Transform the input string s by removing non-alphanumeric
characters and converting all characters to lowercase.
        s = ''.join(i.lower() for i in s if i.isalnum())

         # Initialize two pointers, left and right, to point to the start
and end of the transformed string, respectively.
        left, right = 0, len(s) - 1

        # Iterate through the string using the two pointers.
        while left < right:
             # Check if the characters at the left and right pointers are
different.
            if s[left] != s[right]:
```

```
                    # If the characters are different, return False,
indicating that the string is not a palindrome.
                return False
            # Move the left pointer to the right and the right pointer to
the left, towards the center of the string.
            left += 1
            right -= 1

        # If the loop completes without finding any differences, return
True, indicating that the string is a palindrome.
        return True
```

Time Complexity: O(n)
The function iterates through the string once, performing constant-time operations inside the loop.

Space Complexity: O(n)
The function creates a new string with alphanumeric characters only, which could potentially be the same length as the input string.

2. Three Sums

```
# Define a class named Solution. This class will contain a method to find
all unique triplets in an array that sum up to zero.
class Solution(object):

    # Define a method named threeSum that takes a list of integers (nums)
as input.
    def threeSum(self, nums):

        # Sort the input list of integers in ascending order.
        nums.sort()

        # Initialize an empty list to store the triplets that sum up to
zero.
        result = []

            # Iterate through the input list, excluding the last two
elements.
        for i in range(len(nums) - 2):

                # Skip duplicate values of nums[i] to avoid duplicate
triplets.
            if i > 0 and nums[i] == nums[i - 1]:
```

```
                continue

            # Initialize two pointers, left and right, to search for the
remaining two elements of the triplet.
            left = i + 1
            right = len(nums) - 1

                # Use a two-pointer technique to find the remaining two
elements that sum up to zero.
            while left < right:

                # Calculate the total sum of the current triplet.
                total = nums[i] + nums[left] + nums[right]

                    # If the total sum is less than zero, move the left
pointer to the right to increase the sum.
                if total < 0:
                    left += 1

                  # If the total sum is greater than zero, move the right
pointer to the left to decrease the sum.
                elif total > 0:
                    right -= 1

                # If the total sum is zero, add the triplet to the result
list.
                else:
                    result.append([nums[i], nums[left], nums[right]])

                    # Skip duplicate values of nums[left] and nums[right]
to avoid duplicate triplets.
                    while left < right and nums[left] == nums[left + 1]:
                        left += 1

                    while left < right and nums[right] == nums[right -
1]:
                        right -= 1

                    left += 1
                    right -= 1

        # Return the list of unique triplets that sum up to zero.
```

```
        return result
```

Time Complexity:O(n^2)
Sorting: The sorting step takes O(n log n) time. This is because the nums.sort() method internally uses an efficient sorting algorithm such as Timsort, which has a time complexity of O(n log n).

Nested Loops: After sorting, the function iterates through the input list nums in a nested loop structure. The outer loop iterates through each element of nums, and the inner loop (while loop) runs for each outer loop iteration, iterating through a subset of elements based on the current position of the outer loop. The combined effect of these nested loops results in O(n^2) time complexity.

Therefore, the overall time complexity is O(n log n) (for sorting) + O(n^2) (for nested loops), which simplifies to O(n^2) as the dominant term.

Space Complexity: O(n):
The function uses additional space to store the result list, which can potentially contain O(n) triplets

4.5 Sliding Window; Best Time to Buy And Sell Stock, Longest Substring Without Repeating Characters
4.6 Stacks & Queues; Valid Parentheses, Implement Stack using Queue(s)
4.7 Linked List; Reverse Linked List, Merge Two Sorted Lists, Remove Nth Node From End of List
4.8 Backtracking; Combination Sum, Word Search, Permutations
4.9 Recursions; Climbing Stairs, Tower of Hanoi
4.10 Bit Manipulation; Number of 1 Bits, Counting Bits, Reverse Bits, Missing Numbers

References

GfG. (2023, August 9). Time Complexity and Space Complexity. GeeksforGeeks. https://www.geeksforgeeks.org/time-complexity-and-space-complexity/

Trivedi, A. (2021, December 21). Space complexity in data structure. Scaler Topics. https://www.scaler.com/topics/data-structures/space-complexity-in-data-structure/

Real Python. (2023, November 27). Sorting algorithms in Python. https://realpython.com/sorting-algorithms-python/

Time and space complexity of sorting algorithms. shiksha. (n.d.). https://www.shiksha.com/online-courses/articles/time-and-space-complexity-of-sorting-algorithms-blogId-152755

GfG. (2024, February 22). Searching algorithms. GeeksforGeeks. https://www.geeksforgeeks.org/searching-algorithms/?ref=lbp

YouTube. (2023, October 25). EP.1 - Arrays & Hashing | Data Structures and algorithms | DSA in python. YouTube. https://www.youtube.com/watch?v=nET1jqI_Ntk

Real Python. (2023a, July 31). Pointers in python: What's the point? https://realpython.com/pointers-in-python/