Assignment -1

CSE-320

Lovely Professional University
Jalandhar, Punjab, India.

Delivered by:	**Received by:**
Names of the student: **Vivek Kumar Vinay** Reg. No.: **12203779** Roll No.: **RK22UNA37** Signature:	Name of the faculty: **Waseem Ud Din Wani** UID: **63869** Signature:

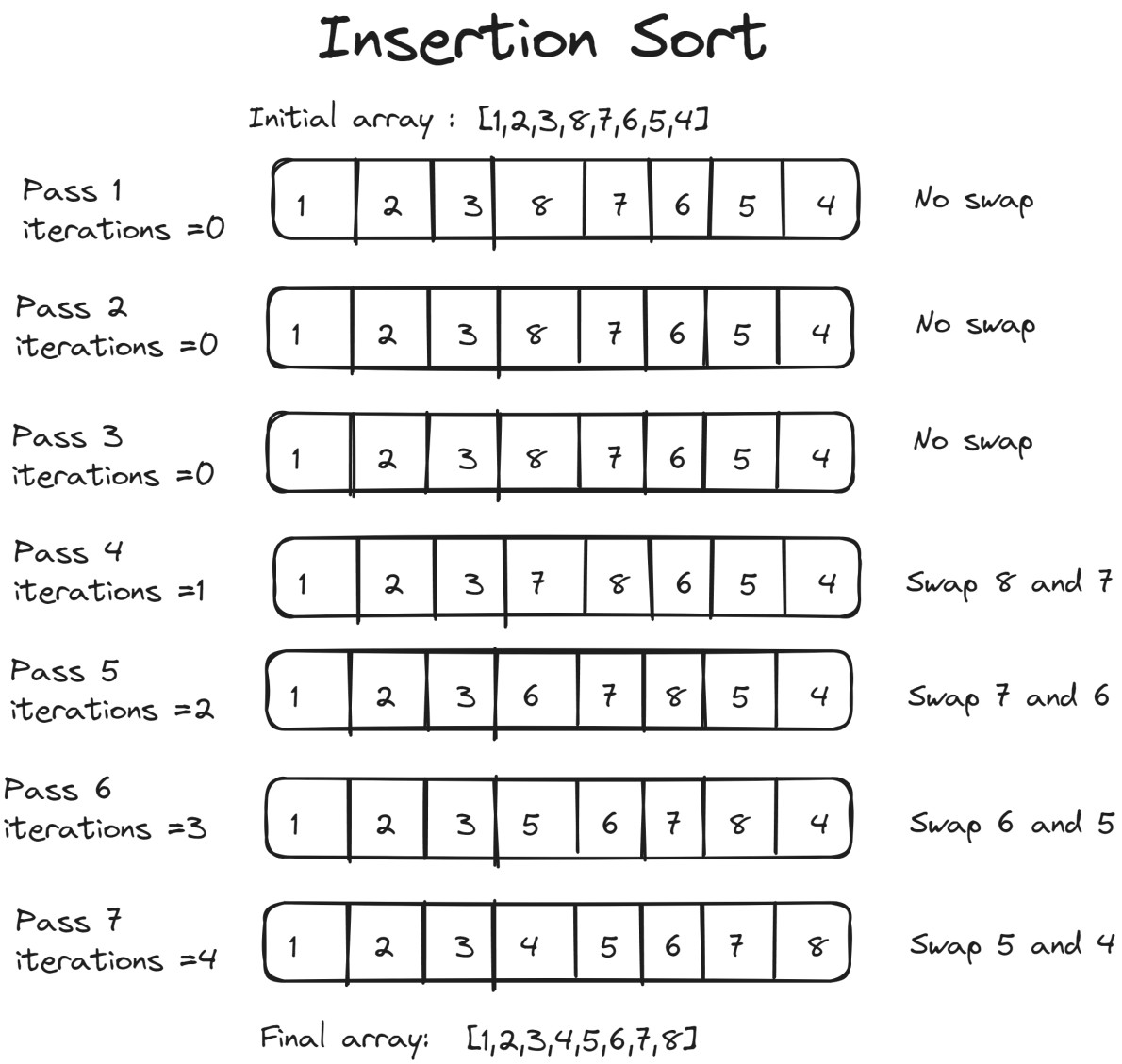
Q1. Among the three sorting algorithms (Bubble, Insertion, Selection), explain which one would be most suitable for sorting an array that increases until midway and then decreases. Support your explanation with an example.

Out of the three given sorting algorithms, **I believe insertion sort would be the best sorting algorithm because it is adaptive when the data is partially sorted**. So, it only checks for the elements that are not placed in the correct order and then efficiently inserts elements into their correct positions within the sorted part of the array. While, bubble sort compares and swaps adjacent elements regardless of their initial order and similar to bubble sort, selection sort is not adaptive as well, it consistently searches for the minimum element and places it at the beginning of the sorted part.

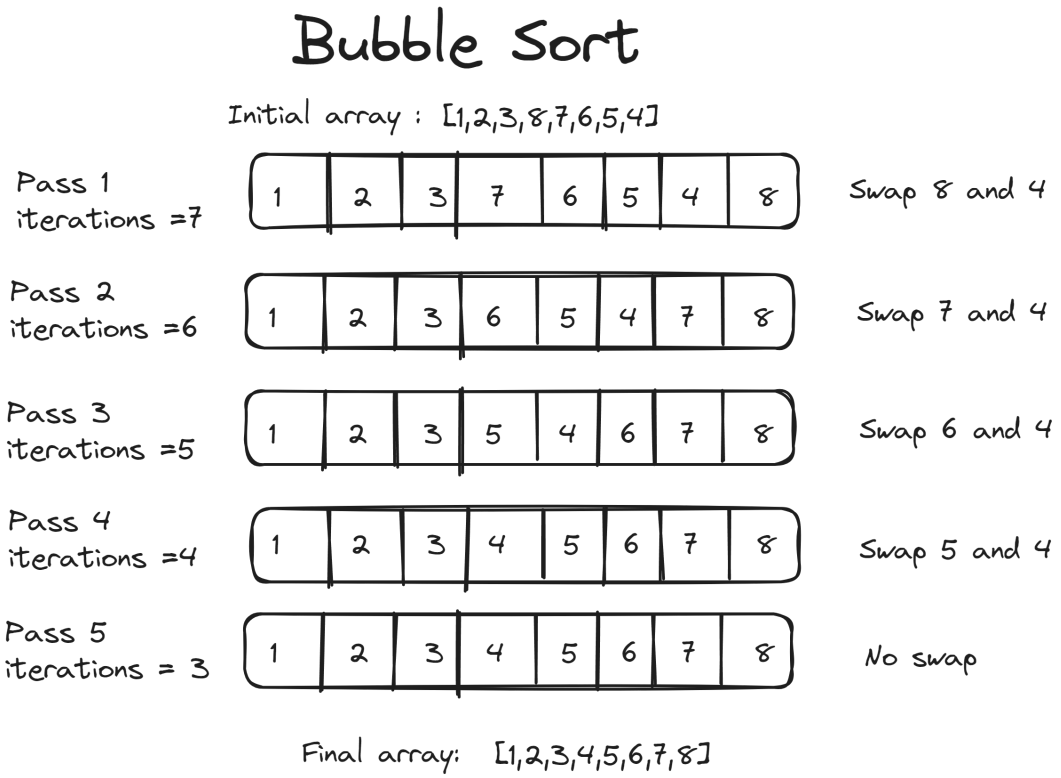
Example:

Initial Array: [1, 2, 3, 8, 7, 6, 5, 4]

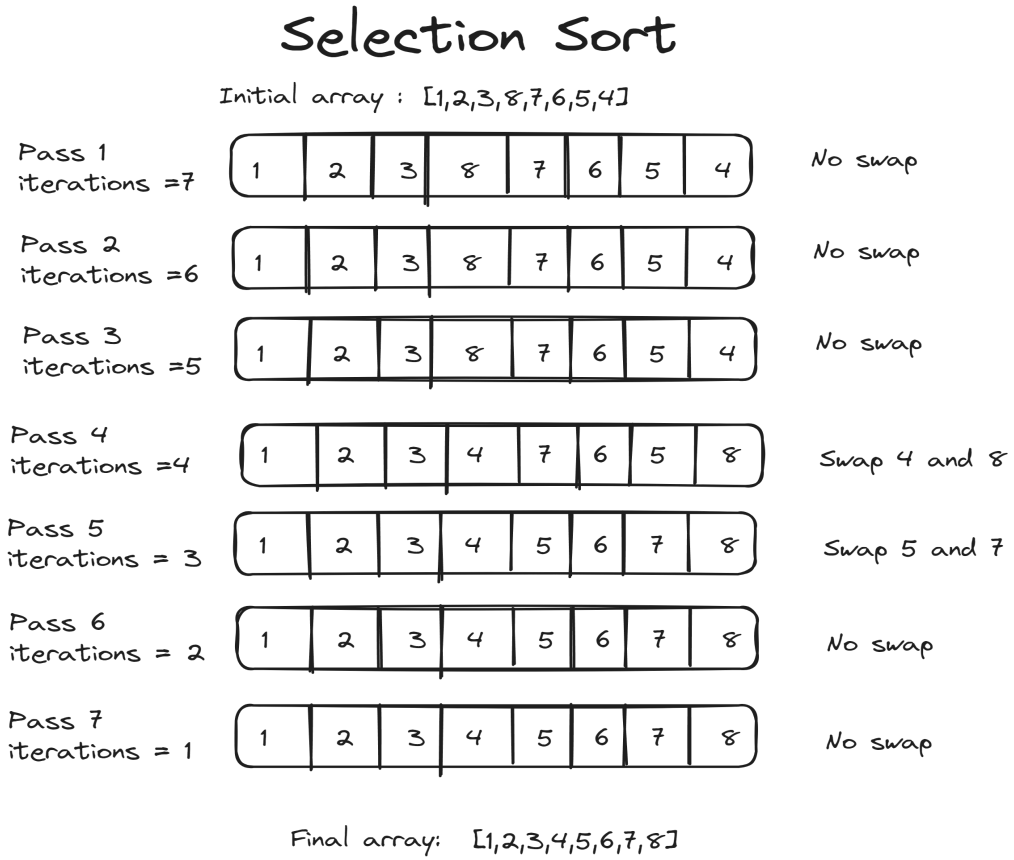
In 'Insertion Sort', when this array is sorted, the first **outer loop(pass)** has 0 **iteration(inner loop)** inside it because it checks the condition 2>1 which evaluates to false and then moves on, in pass 2 also it has 0 iteration as the condition it checks is 3>2 which again evaluates to false, the same thing happens with pass 3 but in pass 4 the first condition evaluates to true which is 8>7 .This process goes on in passes 5,6 and 7 where they have 2, 3 and 4 iterations respectively. Bringing the total iteration count to 17.



Now, in Bubble Sort in every **pass(outer loop)** there will be 7 **iterations(inner loop)** in the first pass and then 6 and so on and in each pass the largest element will be shifted to the end respective of their proper places and this will continue until the array is sorted and in this example the array is sorted in the 4th pass but it still goes into the 5th pass to check if the array is fully sorted thus taking the iteration count to 30, which is almost twice of what insertion sort took.



Finally, in Selction Sort the first three **passes(outer loops)** will have no swaps made in all their **iterations(inner loops)**, and then in the following passes the minimum element which would first be 4 will be swapped with 8 the first index of the unsorted part of the array, then 5 will be swapped with 7 thus sorting the array but since there is no exit condition the loop will continue till the end while making no swaps. The total iteration count here becomes 35.



Q2. Implement Bubble Sort, Selection Sort, and Insertion Sort on datasets (different arrays) of various sizes. Analyze the results, highlighting the distinctions in their time complexities. Define the intermediate state of a given array being sorted, providing an example. Explore practical scenarios where one sorting algorithm might outperform the others. Discuss the total number of passes, swaps, and comparisons required for all three sorting techniques based on the array's state.

Implementation:

```
import java.util.Arrays;
import java.util.Random;

public class SortingComparison {

    public static int[] bubbleSort(int[] arr) {
        int n = arr.length;
        int passes = 0, swaps = 0, comparisons = 0;

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                comparisons++;
                if (arr[j] > arr[j + 1]) {
                    swaps++;
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
            passes++;
        }

        return new int[]{passes, swaps, comparisons};
    }

    public static int[] selectionSort(int[] arr) {
        int n = arr.length;
        int passes = 0, swaps = 0, comparisons = 0;

        for (int i = 0; i < n; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                comparisons++;
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            if (i ≠ minIndex) {
                swaps++;
                int temp = arr[i];
                arr[i] = arr[minIndex];
```

```
                arr[minIndex] = temp;
            }
            passes++;
        }

        return new int[]{passes, swaps, comparisons};
    }

    public static int[] insertionSort(int[] arr) {
        int n = arr.length;
        int passes = 0, swaps = 0, comparisons = 0;

        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j ≥ 0 && key < arr[j]) {
                comparisons++;
                swaps++;
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
            passes++;
        }

        return new int[]{passes, swaps, comparisons};
    }

    public static void main(String[] args) {
        Random rand = new Random();

        int[] sizes = {10, 100, 1000, 50000};

        for (int size : sizes) {
            int[] dataset = new int[size];
            for (int i = 0; i < size; i++) {
                dataset[i] = rand.nextInt(1000);
            }

            // Bubble Sort
            long startTime = System.nanoTime();
           int[] bubbleSortResult = bubbleSort(Arrays.copyOf(dataset, dataset.
length));
            long elapsedTime = System.nanoTime() - startTime;
            System.out.println("Bubble Sort - Size: " + size +
                    ", Passes: " + bubbleSortResult[0] +
                    ", Swaps: " + bubbleSortResult[1] +
                    ", Comparisons: " + bubbleSortResult[2] +
```

```
                    ", Time: " + elapsedTime / 1e6 + " milliseconds");

            // Selection Sort
            startTime = System.nanoTime();
             int[] selectionSortResult = selectionSort(Arrays.copyOf(dataset,
dataset.length));
            elapsedTime = System.nanoTime() - startTime;
            System.out.println("Selection Sort - Size: " + size +
                    ", Passes: " + selectionSortResult[0] +
                    ", Swaps: " + selectionSortResult[1] +
                    ", Comparisons: " + selectionSortResult[2] +
                    ", Time: " + elapsedTime / 1e6 + " milliseconds");

            // Insertion Sort
            startTime = System.nanoTime();
             int[] insertionSortResult = insertionSort(Arrays.copyOf(dataset,
dataset.length));
            elapsedTime = System.nanoTime() - startTime;
            System.out.println("Insertion Sort - Size: " + size +
                    ", Passes: " + insertionSortResult[0] +
                    ", Swaps: " + insertionSortResult[1] +
                    ", Comparisons: " + insertionSortResult[2] +
                    ", Time: " + elapsedTime / 1e6 + " milliseconds");

            System.out.println("\n");
        }
    }
}
```

```
Bubble Sort - Size: 10, Passes: 10, Swaps: 25, Comparisons: 45, Time: 0.000018 seconds
Selection Sort - Size: 10, Passes: 10, Swaps: 7, Comparisons: 45, Time: 0.000016 seconds
Insertion Sort - Size: 10, Passes: 9, Swaps: 25, Comparisons: 25, Time: 0.000017 seconds


Bubble Sort - Size: 100, Passes: 100, Swaps: 2125, Comparisons: 4950, Time: 0.001036 seconds
Selection Sort - Size: 100, Passes: 100, Swaps: 90, Comparisons: 4950, Time: 0.000607 seconds
Insertion Sort - Size: 100, Passes: 99, Swaps: 2125, Comparisons: 2125, Time: 0.000502 seconds


Bubble Sort - Size: 1000, Passes: 1000, Swaps: 246462, Comparisons: 499500, Time: 0.123687 seconds
Selection Sort - Size: 1000, Passes: 1000, Swaps: 993, Comparisons: 499500, Time: 0.064069 seconds
Insertion Sort - Size: 1000, Passes: 999, Swaps: 246462, Comparisons: 246462, Time: 0.064810 seconds


Bubble Sort - Size: 50000, Passes: 50000, Swaps: 623266820, Comparisons: 1249975000, Time: 369.415240 seconds
Selection Sort - Size: 50000, Passes: 50000, Swaps: 49943, Comparisons: 1249975000, Time: 178.295078 seconds
Insertion Sort - Size: 50000, Passes: 49999, Swaps: 623266820, Comparisons: 623266820, Time: 186.388004 seconds
```

```
Selection Sort - Size: 100000, Passes: 100000, Swaps: 99893, Comparisons: 4999950000, Time: 791.274048 seconds
Insertion Sort - Size: 100000, Passes: 99999, Swaps: 2510385021, Comparisons: 2510385021, Time: 816.152717 seconds
```

First let's talk about thier individual **Time Complexities** in each case:

1. Bubble Sort:

- Best case: O(n) [When array is already sorted]
- Average and worst case: O(n^2)

2. Selection Sort:

- Best , average and worst case: O(n^2)

3. Insertion Sort:

- Best case: O(n) [When array is already sorted]
- Average and worst case: O(n^2)

Space complexities:

O(1) for all three

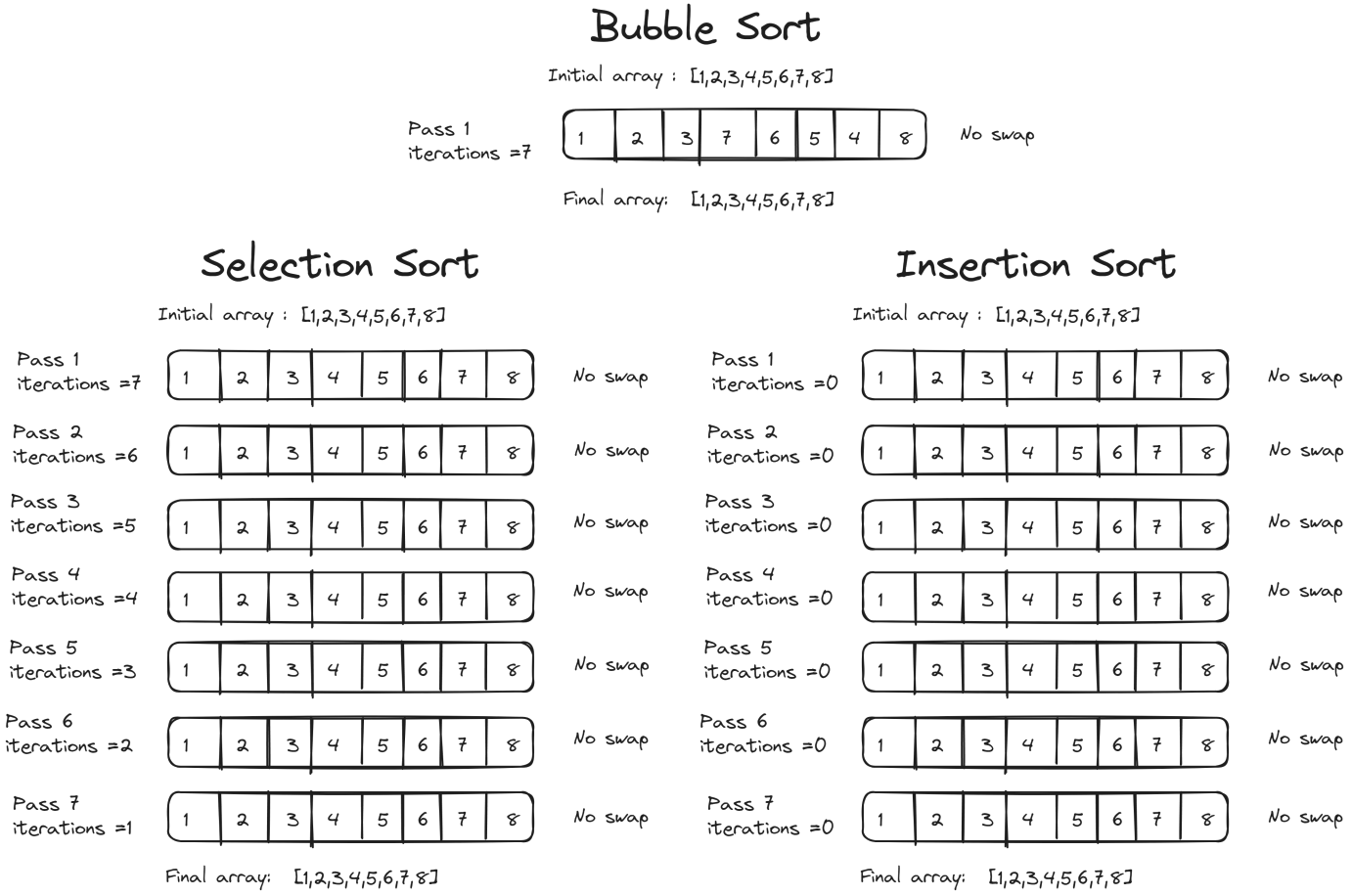
From just this data alone we can see that none of these three are efficient sorting algorithms, Selection Sort always takes O(n^2) time and Bubble Sort along with Insertion Sort are also the same except in their best cases which is an already sorted array. Bubble Sort is only better than Selection Sort in this scenario alone and when an array is nearly sorted, even then Insertion Sort outperforms Bubble Sort for a partially sorted array.

Now, from the above implementation of the question we can verify that Insertion Sort is indeed the fastest among the three for almost any type of array, with the exception that Selection Sort is better for really large datasets, Bubble Sort being slowest. We can also see that Insertion Sort makes one less pass than the size of the array, the number of swaps in Bubble Sort and Insertion Sort are the same, the number of comparisons in Bubble Sort and Selection Sort are same. While Selection Sort makes the least number of swaps, Insertion Sort makes the least number of comparisons meaning the inner loop is executed a lesser number of times, but the number of swaps made by Insertion Sort is exponentially higher than Selection Sort whereas the number of comparisons made by Selection Sort is just twice of Insertion Sort.

Now we can take three examples for this, one would be a completely sorted array, a partially sorted array and an unsorted array.

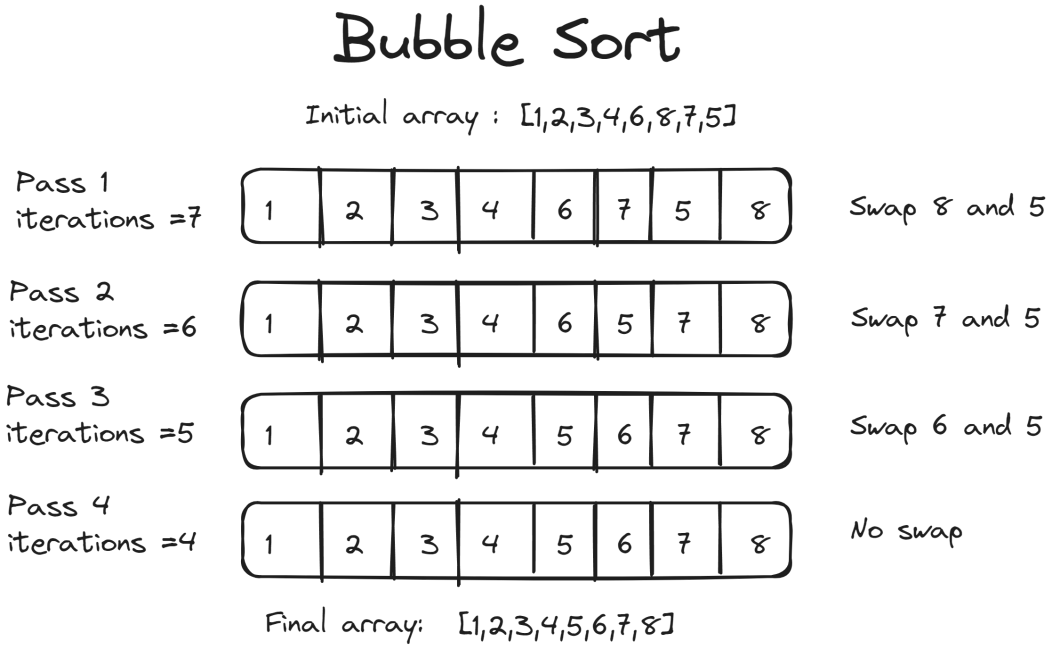
- *Sorted array:*

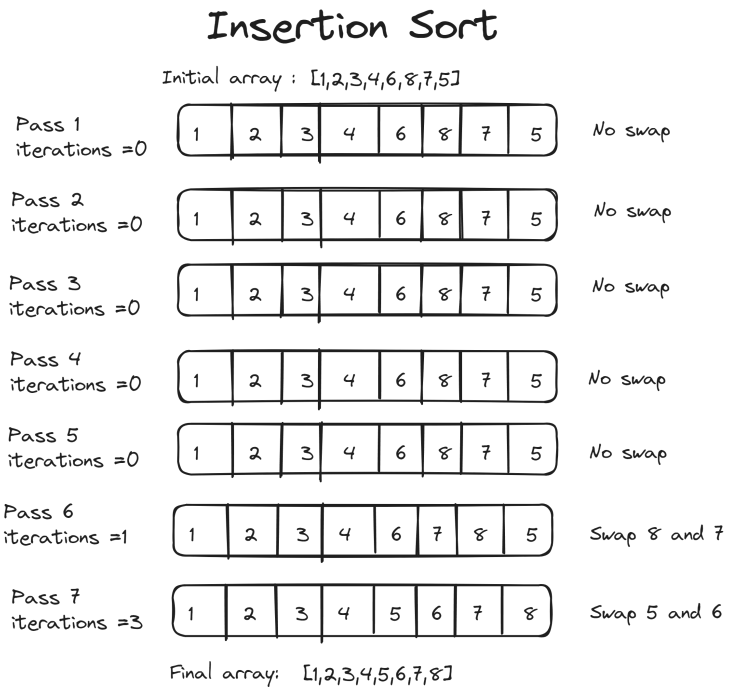
[1,2,3,4,5,6,7,8]

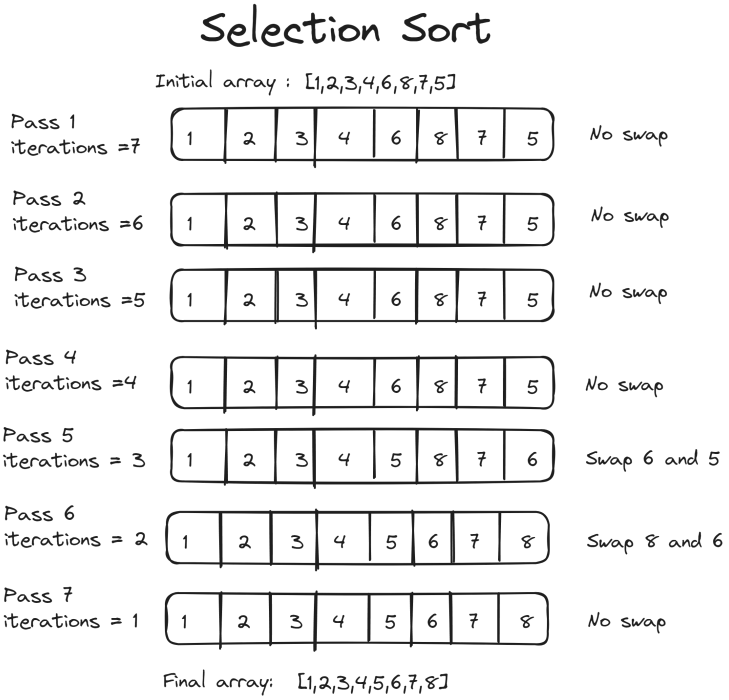


As discussed above and shown through this example in best case, insertion sort and bubble sort have O(n) and selection sort still has O(n^2).

-Partially Sorted array:
[1,2,3,4,6,8,7,5]



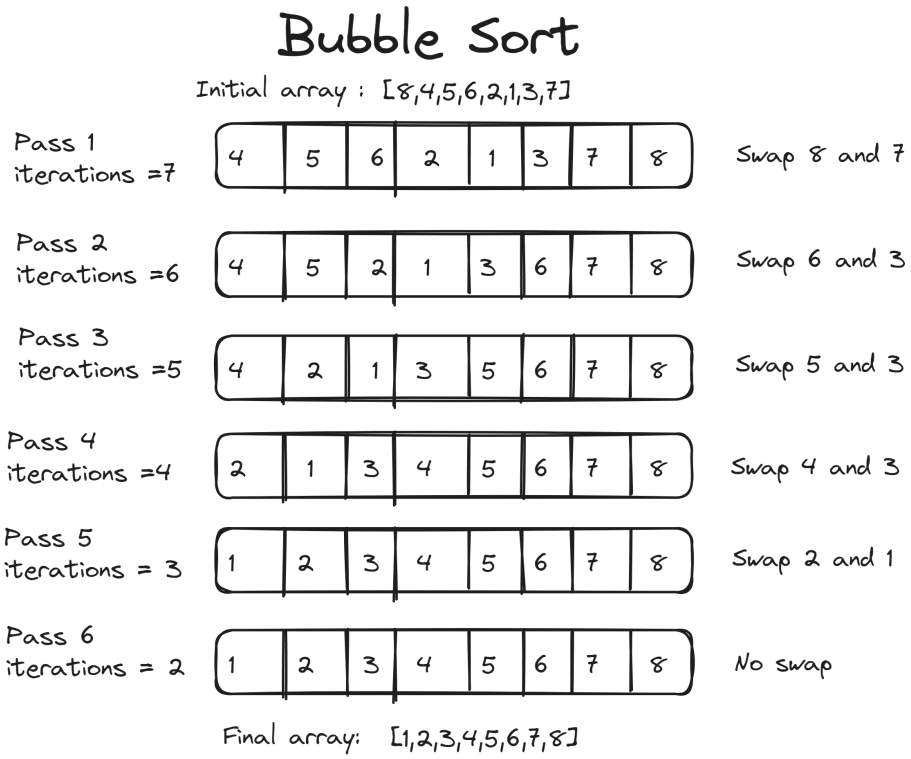


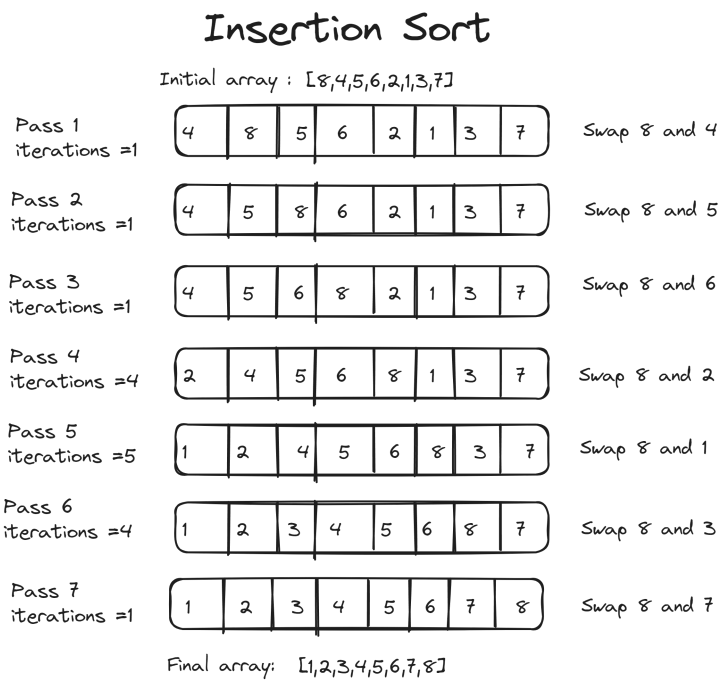


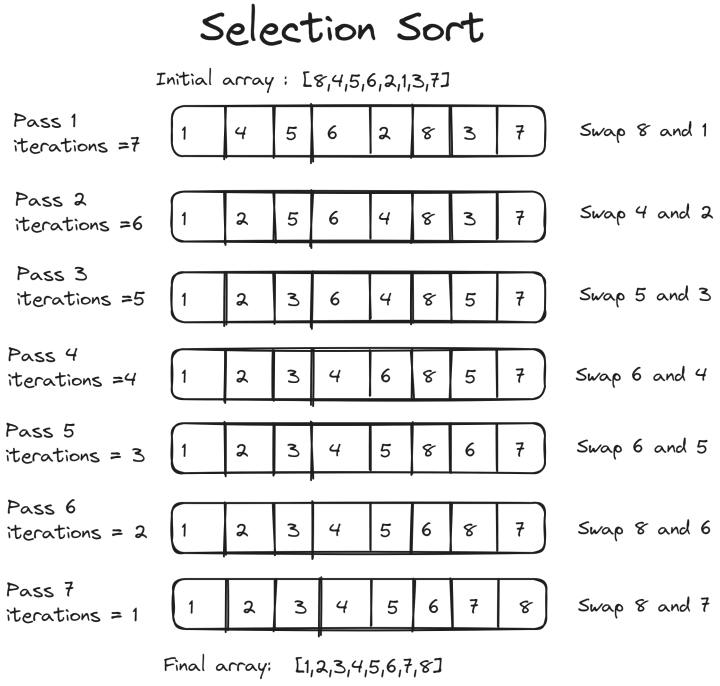
As discussed in the previous question and further in this question we can see that insertion sort is better than bubble and selection sort when the array is partially sorted. Selection sort irrrespective of how much the array is sorted performs in time complexity of O(n^2). Insertion Sort only performs the required swaps.

-Unsorted Array

[8,4,5,6,2,1,3,7]







From this data we can visualize that Insertion Sort is the fastest for an array of smaller size, while bubble sort performs a little better than selection sort in this scenario, in most cases bubble sort performs the worst also the reason it is known as **'Sinking Sort'.**

Q3. Explore Asymptotic Analysis and Amortised Analysis in detail, accompanied by an example. Provide insights into both and discuss when to use each. Determine which one is best suited for specific scenarios.

Amortized Analysis

Introduction:
Amortized analysis is a technique used to determine the average time complexity of an algorithm over a sequence of operations, rather than just for a single operation. In some cases, an individual operation of an algorithm may be very costly, but it can be shown that the overall cost of a sequence of operations is much lower than the sum of the costs of the individual operations. The amortized analysis aims to capture this phenomenon by providing an upper bound on the average cost per operation over a sequence of operations.

Significance in Specific Scenarios:
Amortized analysis is significant when dealing data structures or algos that are a mix of few cheap/inexpensive and a few occasional expensive operations. In such cases, asymptotic analysis is not precise as it only focuses on the worst case scenarios.

Real-life Example:
For example, a shop has 500 items priced at ₹1 each and a single expensive item costing ₹500. If we use an asymptotic approach, taking the maximum price (₹500) multiplied by the number of items (501), the estimate would be ₹250,500, leading to a significant overestimation, while amortized analysis would add 500 x 1 and 1 x 500 thus the estimate would be ₹1000.

Agrregate Method in Amortized Analysis:

Cost of Insertions in Dynamic Tables:

Let's explore an example involving dynamic tables. Consider a scenario where inserting elements has an occasional expensive operation, such as resizing the table.

Cost Analysis:

- Cost of ith Insertion:
  - `i` if `i-1` is an exact power of 2 (due to copying during overflow).
  - `1` otherwise (no overflow).

- Total Cost of n Insertions:
  - Order of `n` (summation formula with geometric series).

- Amortized Cost per Insertion:
  - Order of `1` (total cost divided by the number of insertions).

Comparison with Asymptotic Analysis:

While asymptotic analysis might suggest a cost of `O(n^2)` for n insertions, amortized analysis reveals a lower and more accurate cost of `O(n)`. This highlights the benefit of considering the average cost across operations.

Accounting Method in Amortized Analysis (Augmented Stack Example):

The accounting method is one of the approaches within amortized analysis, and its application can be demonstrated through an augmented stack example.

Key Principles:

1. Assigning Costs:
- Each operation is assigned an “amortized cost” representing the average expected cost.
- Example: Push operation might have an amortized cost of 2 units.

2. Tracking Actual Costs:
- Each operation also has an actual cost, reflecting the real-time taken by the operation.

3. Accounting Principle:
- If the actual cost is less than the amortized cost, the difference is saved as “balance.”

4. Example with Push Operations:
- Push operations with an actual cost of 1 unit each might be allocated 2 units as amortized cost, creating a balance.

5. Using the Balance:
- The saved balance can be used to cover the underestimated costs of other operations.

Potential Method in Amortized Analysis:

The potential method is a complementary approach to accounting and aggregate methods. It particularly excels in analyzing specific data structures, providing another layer of understanding in amortized analysis.

Core Concept:

- Each data structure has a “potential” value, initially set to zero.
- Each operation transforms the data structure and changes its potential (always non-negative).
- Amortized cost of an operation is the sum of the actual cost and the change in potential.

Two Scenarios:

1. Overcharging:
   - `delta(phi_i) > 0` - the assigned amortized cost exceeds the actual cost.

2. Undercharging:
   - `delta(phi_i) < 0` - the assigned amortized cost is less than the actual cost.

In conclusion, amortized analysis, through methods like accounting and potential, provides a more detailed and accurate understanding of algorithmic efficiency, especially in scenarios where operations have varying costs.

Asymptotic Analysis

Asymptotic analysis is a method of describing the limiting behavior of a function or algorithm when the argument tends towards a particular value or infinity, usually in terms of computational complexity.

Notations: Commonly used notations in asymptotic analysis include Big O (O), Omega (Ω), and Theta (Θ).

Example:
Consider a simple algorithm that iterates through an array of size n and prints each element.

```
procedure print_array(arr):
    for element in arr:
        print(element)
```

The time complexity of this algorithm is O(n), where n is the size of the array.

When to Use:
Asymptotic analysis is useful for understanding the overall efficiency of an algorithm and comparing the growth rates of different algorithms. It is suitable for scenarios where you want to know how the algorithm's performance scales with input size.

Q4. Compare the time complexity of the recursive and iterative methods for calculating the Fibonacci sequence. Demonstrate their behaviors using growth functions and graphs. Illustrate the Recursion Tree for the recursive approach and discuss the number of calls it makes for fib(5).

1. Recursive Method:

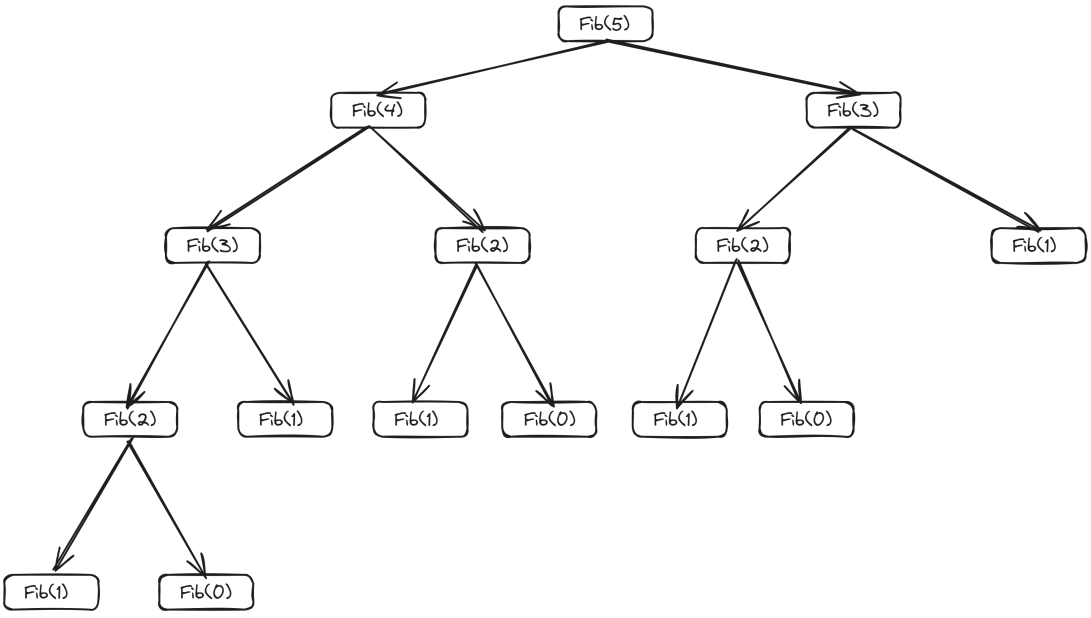
The recursive method for calculating the Fibonacci sequence is straightforward, but it can lead to exponential time complexity due to redundant calculations.

```
procedure fibonacci_recursive(n):
    if n ≤ 1:
        return n
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

Time Complexity:
The time complexity of the recursive approach is O(2^n), where n is the input to the Fibonacci function. This is because, for each call, two more recursive calls are made, resulting in an exponential growth in the number of function calls.

Growth Function:
The growth function for the recursive approach is exponential, and it quickly becomes impractical for larger values of n.

Recursion Tree for fib(5):



In the recursion tree for fib(5), you can observe redundant calculations, such as fib(3) being computed twice. The number of calls made for fib(5) is 15.

2. Iterative Method:

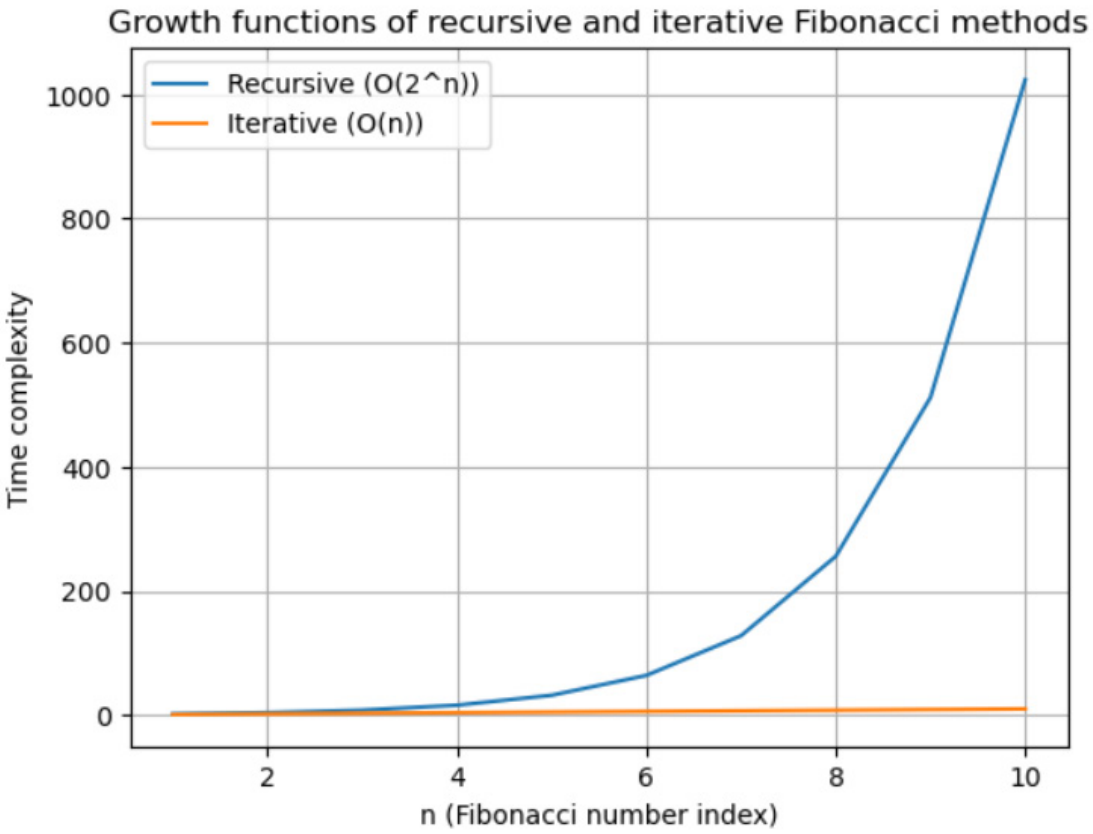
The iterative method for calculating the Fibonacci sequence avoids the exponential time complexity by storing and reusing intermediate results.

```
procedure fibonacci_iterative(n):
      if n ≤ 1:
        return n
      fib = [0] * (n + 1)
      fib[1] = 1
      for i in range(2, n + 1):
            fib[i] = fib[i - 1] + fib[i - 2]
      return fib[n]
```

Time Complexity:
The time complexity of the iterative approach is O(n). It iterates through the Fibonacci sequence only once, calculating each Fibonacci number in a linear manner.

Growth Function:
The growth function for the iterative approach is linear, making it more efficient for larger values of n compared to the recursive approach.

Graphical Comparison:



This graphical representation will clearly demonstrate the difference in growth rates between the recursive and iterative methods.

In summary, the iterative method is more efficient in terms of time complexity, especially for larger values of n, as it avoids the redundant calculations present in the recursive approach.

Q. 5 Discuss how changes in the input size can influence both the time and space complexity of an algorithm. Provide examples of at least three different algorithms and explain how they may behave differently with varying inputs. Illustrate their behaviors using growth functions and graphs (input size vs. time) and determine their order of growth using Big O Notation.

Input size can influence botht the time and space complexity of an algorithm drastically. Understanding these influences can help us better utilize algorithms.

1. Linear Search

Algorithm:

Starts at one end of the array and goes towards the other end until the desired output is found.

Pseudocode:

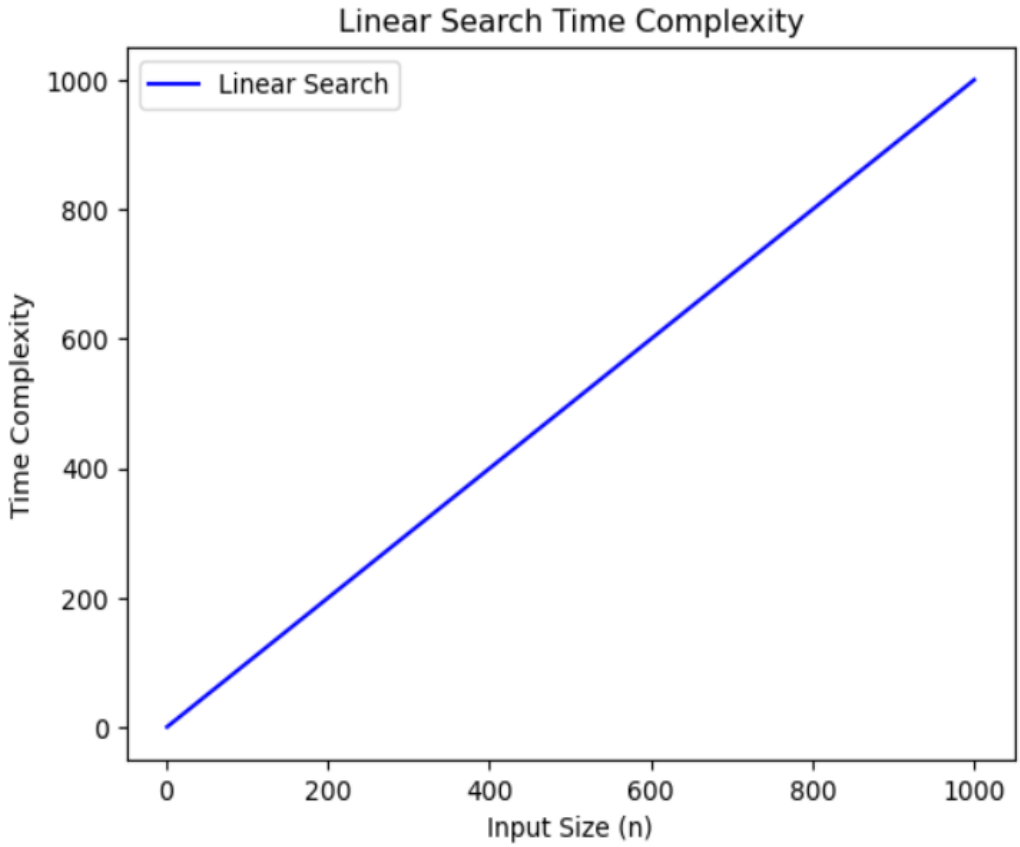
```
procedure linearSearch(arr, target):
      for i in arr:
            if i equals target:
                  return true
      return false
```

Complexities:

Time Complexity (Big O): O(n) - Linear
Space Complexity: O(1) - Constant

Behavior:

As the input size increases, the time taken for linear search grows linearly. The graph of input size vs. time will be a straight line.



2. Binary Search

Algorithm:

It starts with the middle element and compares it with the target, if the target is smaller than the mid then it searches in the left half otherwise in the right half and if the middle value is the target it returns the value, this algorithm repeats this process until the target is found or the array ends.

Pseudocode:

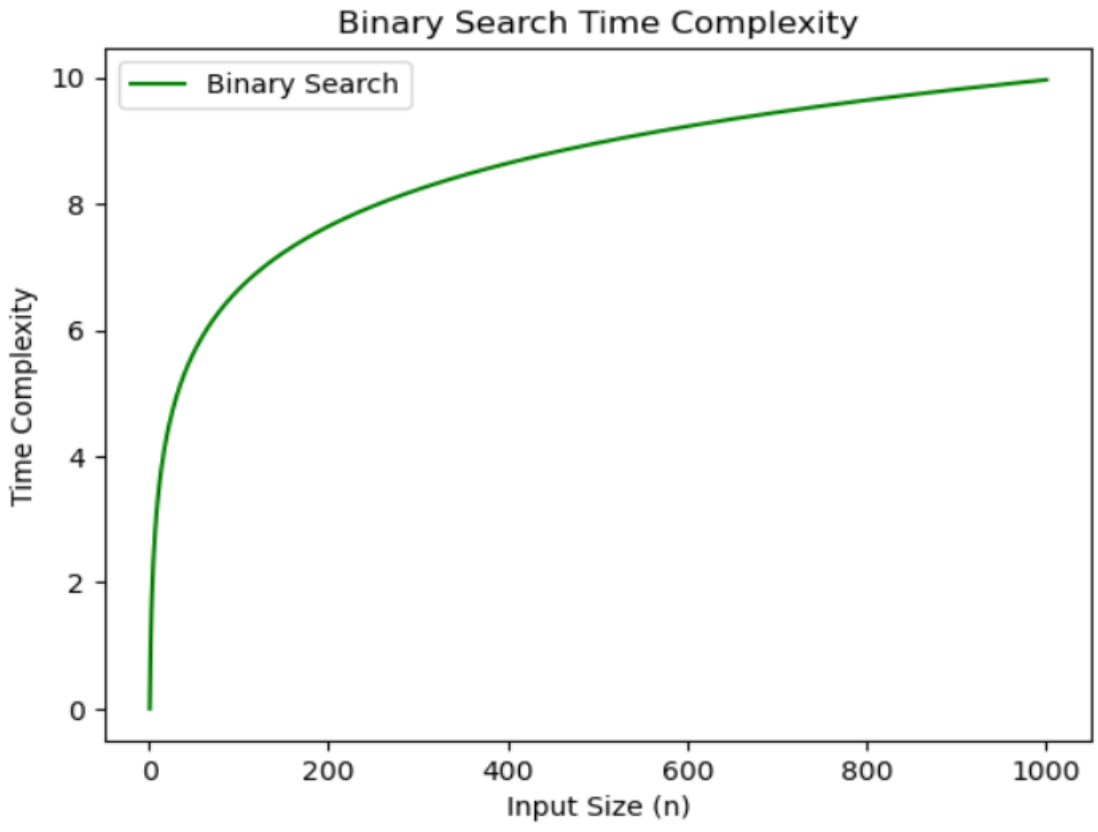
```
procedure BinarySearch(arr, target):
      low =0, high = arr.length -1
      while low ≤ high:
            mid = (low + high) // 2
            if arr[mid] == target:
                  return true
            elif arr[mid] < target:
                  low = mid + 1
            else:
                  high = mid - 1
      return false
```

Complexities:

Time Complexity (Big O): O(log n) - Logarithmic
Space Complexity: O(1) - Constant

Behavior:

As the input size increases, the time taken for binary search grows logarithmically. The graph of input size vs. time will resemble a logarithmic curve.



3. Bubble Sort

Algorithm:

1. Start with the first element and compare it with the next element.
2. If the first element is greater than the next element, swap them.
3. Move to the next pair of elements and repeat step 2.
4. Continue this process until the end of the list is reached.
5. After the first pass, the largest element is guaranteed to be at the end.
6. Continue until no more swaps are needed, indicating that the list is sorted.

Pseudocode:

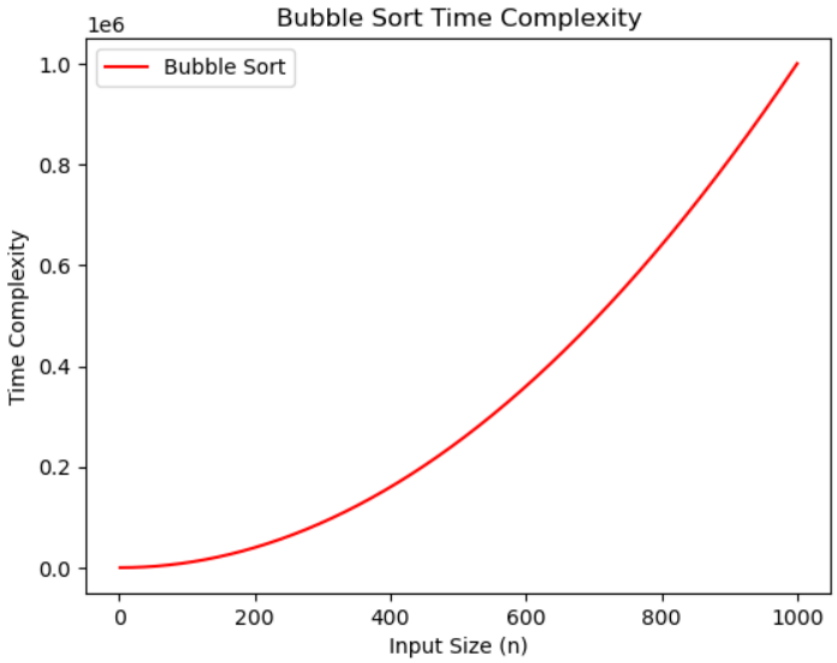
```
procedure bubbleSort(arr):
      n = arr.length
      swapped = true
      while swapped:
            swapped = false:
            for i from 0 to n-2:
                  if arr[i] > arr[i+1]
                        swap(arr, i, i+1)
                        swapped = true

procedure swap(arr, i, j):
      temp = arr[i]
      arr[i] = arr[j]
      arr[j] = temp
```

Complexities:

Time Complexity (Big O): O(n^2) - Quadratic
Space Complexity: O(1) - Constant

Behavior:

As the input size increases, the time taken for bubble sort grows quadratically. The graph of input size vs. time will form a parabolic curve.



Q6. **Identify at least three scenarios where you can deduce that the Order of Growth of the algorithm/ code/program is O(logN). Explain how you arrived at this conclusion and discuss methods to determine the order of growth for any given code/algorithm.**

Three scenarios where the order of growth is O(logN) are: Binary Search, Finding the exponentiation and Tree Traversals in BSTs.

1. Binary Search:

Algorithm:

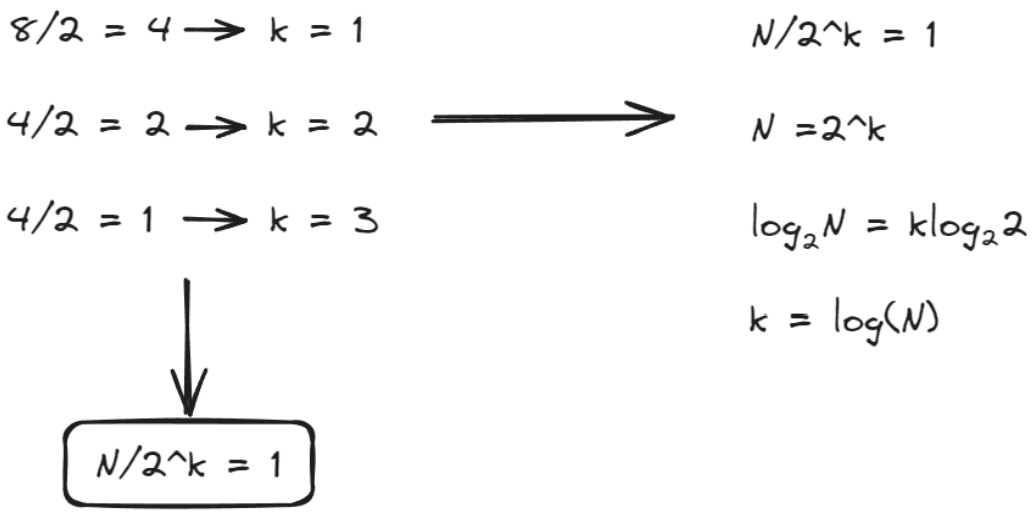
It starts with the middle element and compares it with the target, if the target is smaller than the mid then it searches in the left half otherwise in the right half and if the middle value is the target it returns the value, this algorithm repeats this process until the target is found or the array ends.

Pseudocode:

```
procedure BinarySearch(arr, target):
      low =0, high = arr.length -1
      while low ≤ high:
            mid = (low + high) // 2
            if arr[mid] == target:
                  return true
            elif arr[mid] < target:
                  low = mid + 1
            else:
                  high = mid - 1
      return false
```

Reasoning:

The original array is divided into two halves, so if the array is of size 8 then first it will be 8/2 = 4, then 4/2 =2, then 2/2 = 1. So mathematically this can be expressed as N/2^k =1, where N is the size of the array and k is the power of 2 needed to find the target value in worst case. Then we take logarithm on both sides which evaluates to the end result of log(N).



2. Finding the exponentiation:

Algorithm:

It checks if the exponent is greater than 0 and then it checks if the bits of the exponent are odd or even if the bit is odd then the result is multiplied with the base and if not then this step is skipped and the base is multiplied with itself effectively squaring it and then the exponent is halved. When the exponent reaches 0 then the result is returned.

Pseudocode:

```
Function binaryExponentiation(base, exponent):
    result = 1

    while exponent > 0:
        if exponent is odd:
            result = result * base

        base = base * base
        exponent = exponent / 2

    return result
```

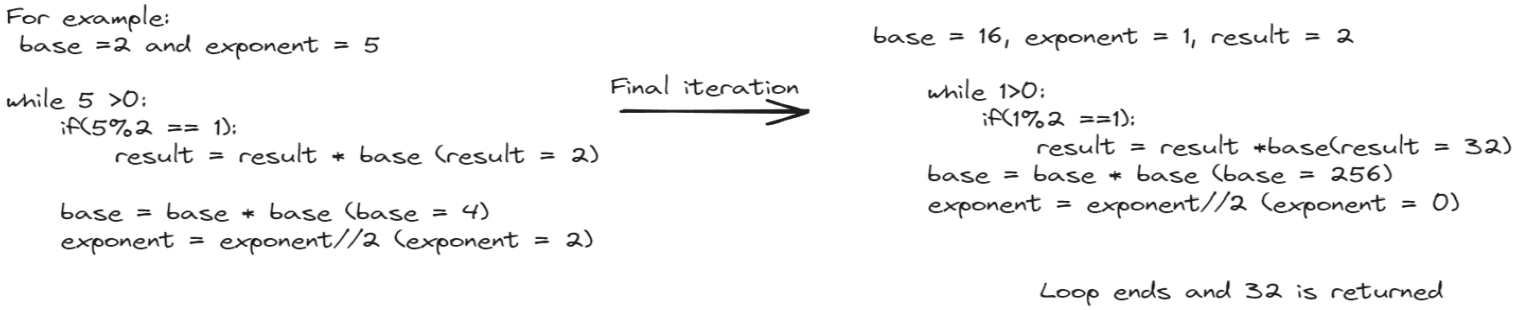
Reasoning:

In each iteration, the exponent is halved (exponent = exponent / 2), which takes O(1) time.

The multiplication (result = result * base) or squaring (base = base * base) operation is also O(1).

The loop runs for log N iterations because the exponent is halved in each iteration.

Therefore, the overall time complexity is O(log N), making binary exponentiation an efficient algorithm for exponentiation.



3. Tree traversal in BST:

Agorithm:

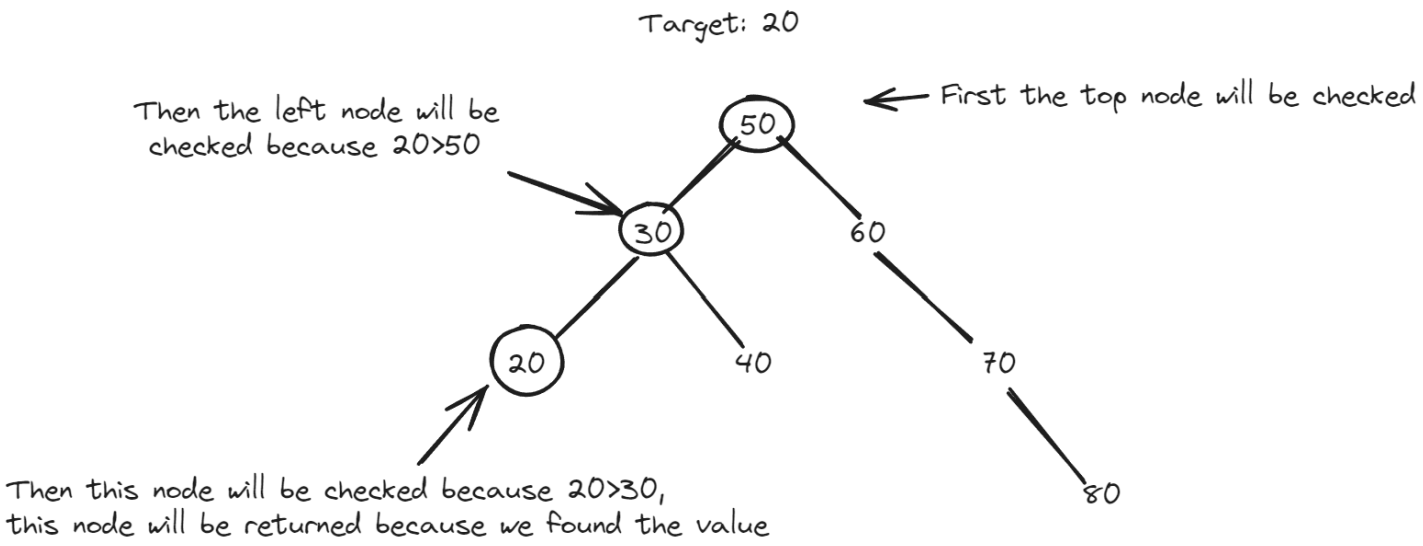
Start at the root node of the BST.
Compare the target data with the data of the current node.
If the target data is less than the current node's data, move to the left child node.
If the target data is greater than the current node's data, move to the right child node.
If the target data is equal to the current node's data, the element is found.
If the current node is None, the element is not found.

Pseudocode:

```
Function searchBST(root, target):
    if root is null:
        return null  # Element not found

    if target is equal to root.key:
        return root  # Element found

    if target is less than root.key:
        return searchBST(root.left, target)

    if target is greater than root.key:
        return searchBST(root.right, target)
```

Reasoning:

First the topmost is node is checked and if the target is less than the root then the left node is checked, if the target is greater than root then the right node is checked and so on until the target is found is which the root or that particular node is returned.



Methods to determine the order of growth:

1. Observation and Reasoning:

- We dry run the algorithms and check how many timesthe loops will run.
- We can aso modify the code to give the number of passes and iterations.
- We can conclude by minor calculations taking the worst case.

2. Mathematical Analysis:

- Break down the algorithm into basic operations.
- Express the number of operations as a mathematical function of the input size.
- Analyze the worst case to determine the growth rate.

3. Empirical Analysis:

- Implement the algorithm with varying input sizes.
- Calculate the time taken for each size.
- Use tableau or matplotlib to plot the data and see the growth.

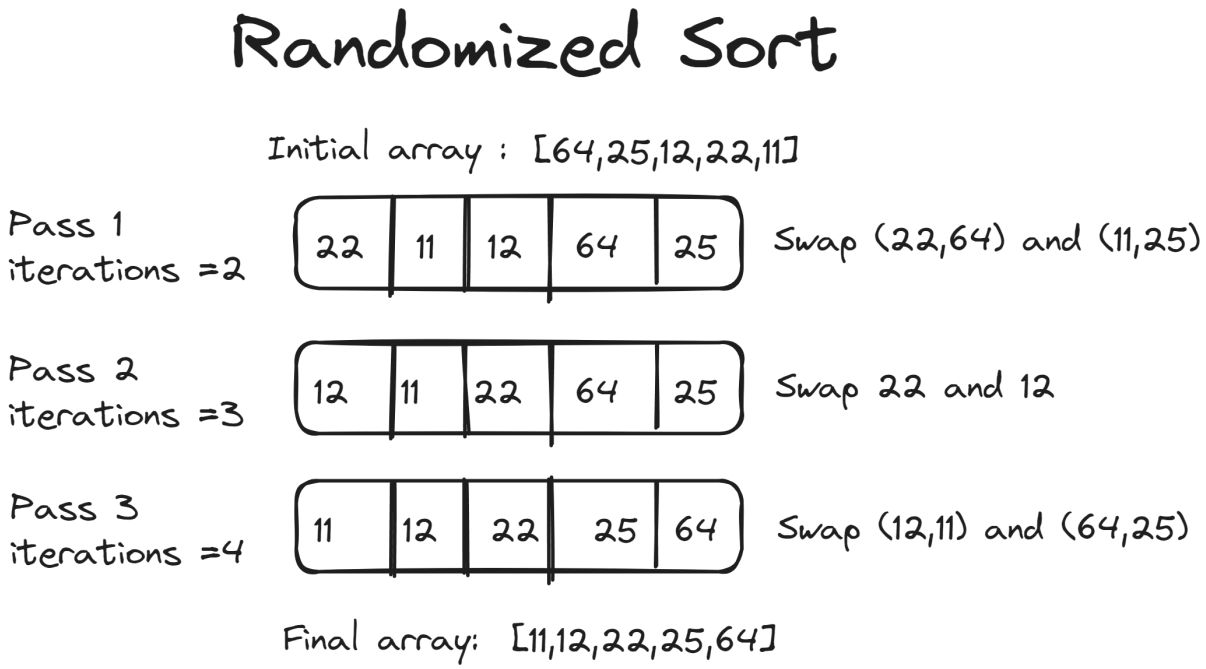
Q7.**Based on the above sorting algo, how many passes will be required to completely sort the given array? What will be the total no. of comparisons and swaps made?**

```java
import java.util.Arrays;

public class RandomizedSortAlgorithm {

    public static void randomizedSort(int[] arr) {
        int n = arr.length;
        int gap = n;
        boolean swapped = true;
        double shrinkFactor = 1.3;

        while (gap > 1 || swapped) {
            gap = (int) (gap / shrinkFactor);

            if (gap < 1) {
                gap = 1;
            }

            swapped = false;

            for (int i = 0; i + gap < n; i++) {
                if (arr[i] > arr[i + gap]) {
                    int temp = arr[i];
                    arr[i] = arr[i + gap];
                    arr[i + gap] = temp;
                    swapped = true;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] array = {64, 25, 12, 22, 11};

        System.out.println("Original Array: " + Arrays.toString(array));

        randomizedSort(array);

        System.out.println("Sorted Array: " + Arrays.toString(array));
    }
}
```

The given code seems to be *'Comb Sort algorithm'* or a variation of it, in this a gap variable is used to compare elements, the variable is initialized to the length of the array and a shrink factor is used to reduce the gap between the elements, the loop runs until the gap between the elements become less than 1 or there are no swaps made. The comparison is made between the i^{th} element and $i+gap^{th}$ element and if the ith element is greater a swap is made. The time complexity of this algorithm is $O(n^2)$.

For the particular array:

Passes = 3
Comparisons = 9
Swaps = 5



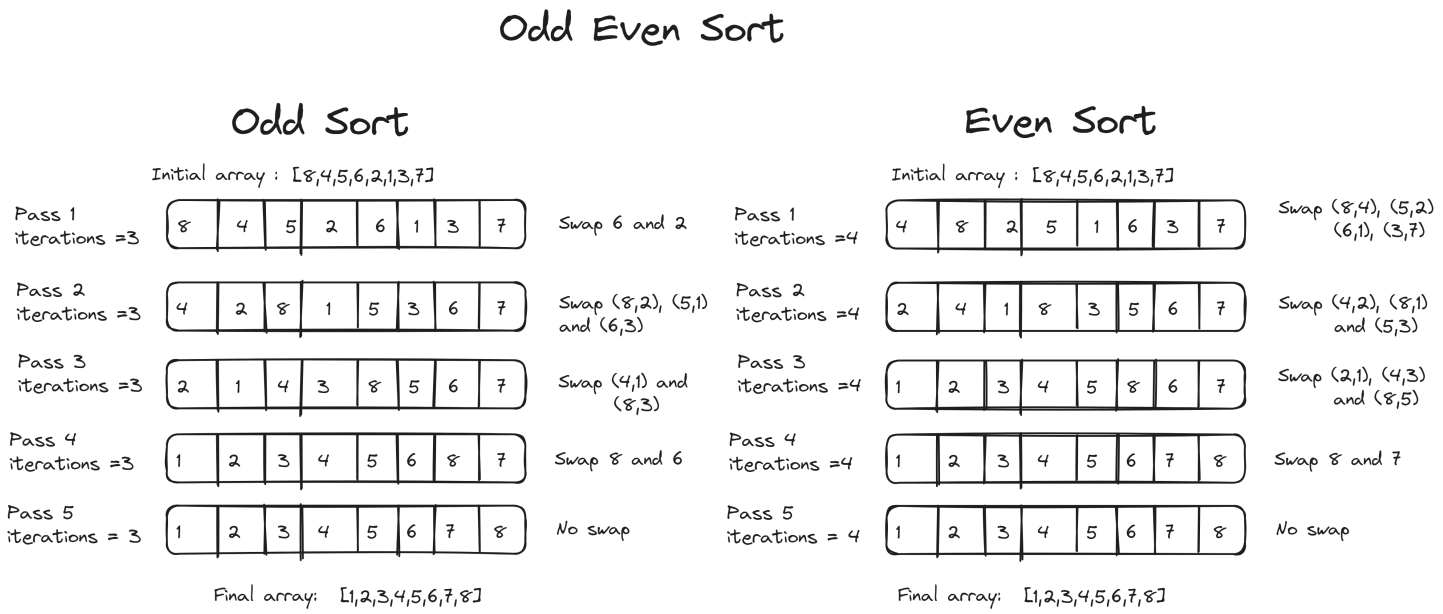
Q8. Google about Odd Even Sort. Discuss how it compares with Bubble sort. Write Pseudo code for this Sorting Algo and sort an example array using this while also showing intermediate states after each pass.

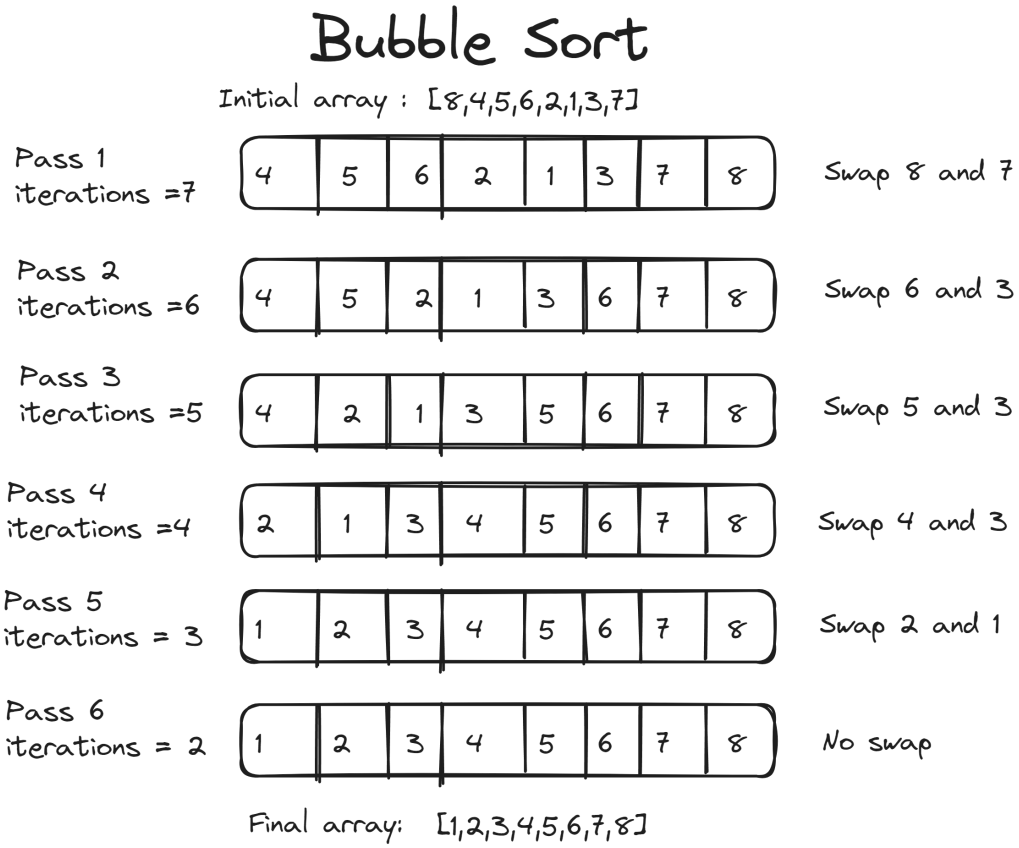
Odd Even Sort is a variation of Bubble Sort, it is also known as *'Brick Sort'*. The sorting is done in two phases in each iteration, first the odd elements are sorted using Bubble sort and then the even elements are sorted using Bubble sort. The iteration continues until the array is sorted.

Pseudocode:

```
Procedure OddEvenSort(arr):
    n = length(arr)
    sorted = false

    while not sorted:
        sorted = true

        // Odd Phase
        for i from 1 to n-2 step 2:
            if arr[i] > arr[i+1]:
                swap(arr[i], arr[i+1])
                sorted = false

        // Even Phase
        for i from 0 to n-2 step 2:
            if arr[i] > arr[i+1]:
                  swap(arr[i], arr[i+1])
                  sorted = false
```

Example Array: [8,4,5,6,2,1,3,7]





For this particular array we can see that Bubble Sort has lesser iterations than Odd Even Sort, but the advantage of Odd Even Sort lies in its parallelization and adaptability in some cases when the array is partially sorted or nearly sorted.