ITECH WORLD AKTU

SUBJECT NAME: DESIGN AND ANALYSIS OF ALGORITHM (DAA) SUBJECT CODE: BCS503

UNIT 1: INTRODUCTION

Syllabus

1. Introduction: Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions.
2. Performance Measurements.
3. Sorting and Order Statistics:
    - Shell Sort
    - Quick Sort
    - Merge Sort
    - Heap Sort
4. Comparison of Sorting Algorithms.
5. Sorting in Linear Time.

1 Introduction to Algorithms

1.1 Definition of an Algorithm

An **algorithm** is a clear and precise sequence of instructions designed to solve a specific problem or perform a computation. It provides a step-by-step method to achieve a desired result. .

1.2 Difference Between Algorithm and Pseudocode

Algorithm	**Pseudocode**

Algorithm to find the GCD of two numbers:
• Start with two numbers, say a and b.
• If $b = 0$, the GCD is a. Stop.
• If $b \neq 0$, assign $a = b$, $b = a$ mod b.
• Repeat the above steps until $b = 0$.
	Pseudocode for GCD:
	`function GCD(a, b)`
	`    while b  0`
	`        temp := b`
	`        b := a mod b`
	`        a := temp`
	`    return a`
	`end function`

1.3 Characteristics of an Algorithm

An effective algorithm must have the following characteristics:

1. **Finiteness:** The algorithm must terminate after a finite number of steps.
2. **Definiteness:** Each step must be precisely defined; the actions to be executed should be clear and unambiguous.
3. **Input:** The algorithm should have zero or more well-defined inputs.
4. **Output:** The algorithm should produce one or more outputs.
5. **Effectiveness:** Each step of the algorithm must be simple enough to be carried out exactly in a finite amount of time.

1.4 Difference Between Algorithm and Pseudocode

Algorithm	**Pseudocode**
A step-by-step procedure to solve a problem, expressed in plain language or mathematical form.	A representation of an algorithm using structured, human-readable code-like syntax.
Focuses on the logical sequence of steps to solve a problem.	Focuses on illustrating the algorithm using a syntax closer to a programming language.
Language independent; can be written in natural language or mathematical notation.	Language dependent; mimics the structure and syntax of programming languages.
More abstract and high-level.	More concrete and closer to actual code implementation.
No need for specific formatting rules.	Requires a consistent syntax, but not as strict as actual programming languages.

1.5 Analyzing Algorithms

Analyzing an algorithm involves understanding its **time complexity** and **space complexity**. This analysis helps determine how efficiently an algorithm performs, especially in terms of execution time and memory usage.

What is Analysis of Algorithms?

- **Definition:** The process of determining the computational complexity of algorithms, including both the time complexity (how the runtime of the algorithm scales with the size of input) and space complexity (how the memory requirement grows with input size).
- **Purpose:** To evaluate the efficiency of an algorithm to ensure optimal performance in terms of time and space.
- **Types:** Analyzing algorithms typically involves two main types of complexities:
  - **Time Complexity:** Measures the total time required by the algorithm to complete as a function of the input size.
  - **Space Complexity:** Measures the total amount of memory space required by the algorithm during its execution.

Example:

- Analyzing the time complexity of the Binary Search algorithm.

2 Complexity of Algorithms

2.1 Time Complexity

Time Complexity is the computational complexity that describes the amount of time it takes to run an algorithm as a function of the length of the input.

Cases of Time Complexity:

- **Best Case:** The minimum time required for the algorithm to complete, given the most favorable input. Example: In Binary Search, the best-case time complexity is $O(1)$ when the target element is the middle element.
- **Average Case:** The expected time required for the algorithm to complete, averaged over all possible inputs. Example: For Quick Sort, the average-case time complexity is $O(n \log n)$.
- **Worst Case:** The maximum time required for the algorithm to complete, given the least favorable input. Example: In Linear Search, the worst-case time complexity is $O(n)$ when the element is not present in the array.

Example:

- Time complexity of the Merge Sort algorithm is $O(n \log n)$.

2.2 Space Complexity

Space Complexity refers to the total amount of memory space required by an algorithm to complete its execution.

Cases of Space Complexity:

- **Auxiliary Space:** Extra space or temporary space used by an algorithm.
- **Total Space:** The total space used by the algorithm, including both the input and auxiliary space.

Example:

- Space complexity of the Quick Sort algorithm is $O(n)$.

3 Growth of Functions

Growth of Functions describes how the time or space requirements of an algorithm grow with the size of the input. The growth rate helps in understanding the efficiency of an algorithm.

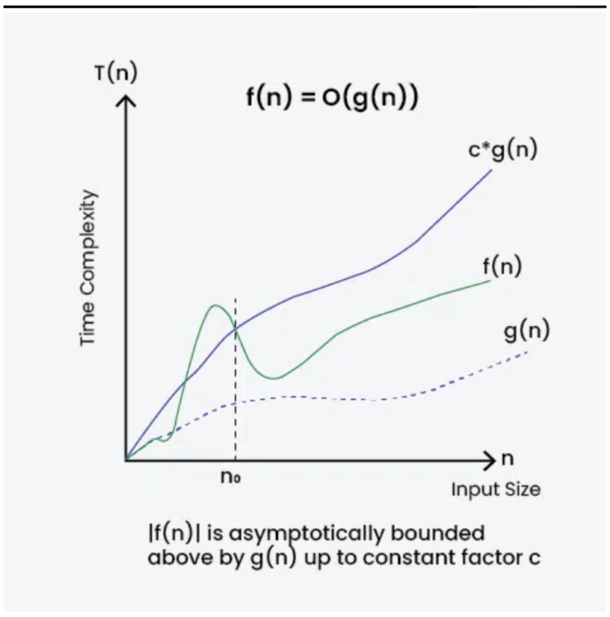
Examples:

- **Polynomial Growth:** $O(n^2)$ - Example: Bubble Sort algorithm.
- **Exponential Growth:** $O(2^n)$ - Example: Recursive Fibonacci algorithm.

3.1 Big-O Notation

Big-O notation, denoted as $O(f(n))$, describes the upper bound of an algorithm's time or space complexity. It gives the worst-case scenario of the growth rate of a function.

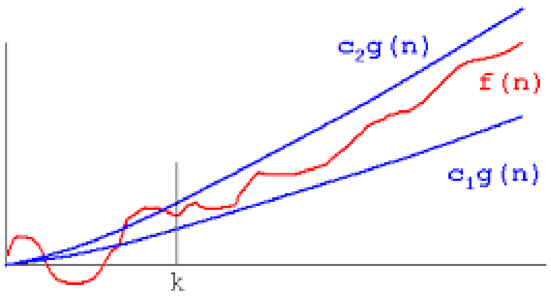
Graphical Representation:


T(n)
f(n) = O(g(n))
c*g(n)
f(n)
g(n)
Time Complexity
n₀
n
Input Size
|f(n)| is asymptotically bounded
above by g(n) up to constant factor c

3.2 Theta Notation

Theta notation, denoted as $\Theta(f(n))$, describes the tight bound of an algorithm's time or space complexity. It represents both the upper and lower bounds, capturing the exact growth rate.

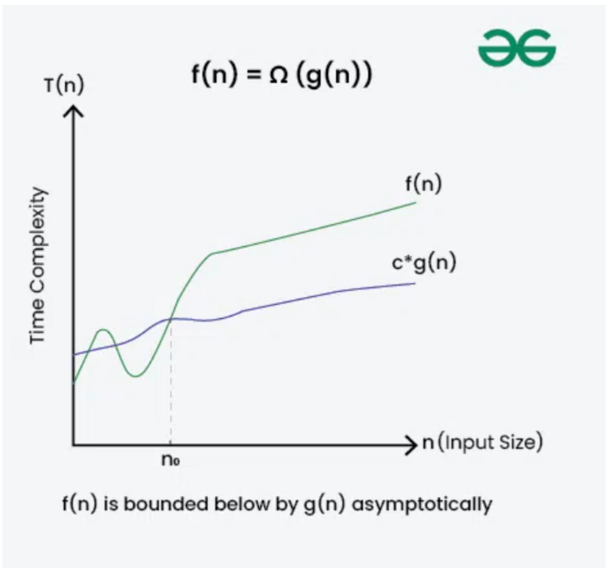
Graphical Representation:



3.3 Omega Notation

Omega notation, denoted as $\Omega(f(n))$, describes the lower bound of an algorithm's time or space complexity. It gives the best-case scenario of the growth rate of a function.

Graphical Representation:



3.4 Numerical Example

Problem: If $f(n) = 100 \cdot 2^n + n^5 + n$, show that $f(n) = O(2^n)$.
Solution:

- The term $100 \cdot 2^n$ dominates as $n \to \infty$.
- n^5 and n grow much slower compared to 2^n.
- Therefore, $f(n) = 100 \cdot 2^n + n^5 + n = O(2^n)$.

3.5 Recurrences

Recurrence relations are equations that express a sequence in terms of its preceding terms. In the context of Data Structures and Algorithms (DAA), a recurrence relation often represents the time complexity of a recursive algorithm. For example, the time complexity $T(n)$ of a recursive function can be expressed as:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

where:

- a is the number of subproblems in the recursion,
- b is the factor by which the subproblem size is reduced in each recursive call,
- $f(n)$ represents the cost of the work done outside of the recursive calls.

There are several methods to solve recurrence relations:

1. **Substitution Method:** Guess the form of the solution and use mathematical induction to prove it.
2. **Recursion Tree Method:** Visualize the recurrence as a tree where each node represents the cost of a recursive call and its children represent the costs of the subsequent subproblems.

3. **Master Theorem:** Provides a direct way to find the time complexity of recurrences of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ by comparing $f(n)$ to $n^{\log_b a}$.

3.6 Master Theorem

The **Master Theorem** provides a solution for the time complexity of divide-and-conquer algorithms. It applies to recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $a \geq 1$ and $b > 1$ are constants.
- $f(n)$ is an asymptotically positive function.

Cases of the Master Theorem:

1. **Case 1:** If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. **Case 2:** If $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$.
3. **Case 3:** If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and large n, then $T(n) = \Theta(f(n))$.

Example:

- Consider the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$.
- Here, $a = 2$, $b = 2$, and $f(n) = n$.
- $\log_b a = \log_2 2 = 1$.
- $f(n) = n = \Theta(n^{\log_2 2}) = \Theta(n^1)$, so it matches Case 2.
- Therefore, $T(n) = \Theta(n \log n)$.

Question 1.6

Solve the recurrence relation:

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2$$

Now, consider another algorithm with the recurrence:

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2$$

Find the largest integer a such that the algorithm T' runs faster than the first algorithm.
Solution:

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \quad \text{and} \quad T'(n) = aT'\left(\frac{n}{4}\right) + n^2$$

By comparing, we have:

$$a = 7, \quad b = 2, \quad f(n) = n^2$$

Using Master's theorem:

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

Case 1 of Master's theorem applies:

$$f(n) = O(n^{\log_b a - \epsilon}) = O(n^{2.81-\epsilon}) = O(n^2)$$

Thus,

$$T(n) = \Theta(n^{2.81})$$

Now for the second recurrence:

$$\frac{\log 7}{\log 2} = \frac{\log a}{\log 4} \quad \Rightarrow \quad \log a = \frac{\log 7}{\log 2} \times \log 4 = 1.6902$$

Taking antilog, we get:

$$a = 48.015$$

Thus, for $a = 49$, algorithm A' will have the same complexity as A. The largest integer a such that A' is faster than A is:

$$a = 48$$

Question 1.7

Solve the recurrence relation:

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2$$

Now, consider another algorithm with the recurrence:

$$S(n) = aS\left(\frac{n}{9}\right) + n^2$$

Find the largest integer a such that the algorithm S runs faster than the first algorithm.

Solution:

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2 \quad \text{and} \quad S(n) = aS\left(\frac{n}{9}\right) + n^2$$

Comparing the equations:

$$a = 7, \quad b = 3, \quad f(n) = n^2$$

Using Master's theorem:

$$n^{\log_b a} = n^{\log_3 7} = n^{1.771}$$

Case 3 of Master's theorem applies:

$$f(n) = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{1.771+\epsilon}) = \Omega(n^2)$$

Thus, the complexity is:

$$T(n) = \Theta(n^2)$$

For algorithm B, we get:

$$n^{\log_9 a} = n^{\log_9 81} = n^2$$

Thus, for $a = 81$, both algorithms have the same complexity.

If $a > 81$, algorithm B has a higher complexity than A:

$$S(n) = \Theta(n^2 \log n) > T(n)$$

Therefore, algorithm B can never be faster than A.

Question 1.8

Solve the recurrence relation:

$$T(n) = T(\sqrt{n}) + O(\log n)$$

Solution:
Let:

$$m = \log n \quad \text{and} \quad n = 2^m \quad \Rightarrow \quad n^{1/2} = 2^{m/2}$$

Then:

$$T(2^m) = T(2^{m/2}) + O(\log 2^m) \quad \text{Let} \quad x(m) = T(2^m)$$

Substituting into the equation:

$$x(m) = x\left(\frac{m}{2}\right) + O(m)$$

The solution is:

$$x(m) = \Theta(m \log m) \quad \Rightarrow \quad T(n) = \Theta(\log n \log \log n)$$

Recursion:

Recursion is a process where a function calls itself either directly or indirectly to solve a problem. In recursion, a problem is divided into smaller instances of the same problem, and solutions to these smaller instances are combined to solve the original problem. Recursion typically involves two main parts:

- **Base Case:** A condition under which the recursion stops.
- **Recursive Case:** The part where the function calls itself to break the problem into smaller instances.

Example of Recursion: Let's take the example of calculating the factorial of a number n, which is defined as:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

The recursive definition of factorial is:

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times \text{factorial}(n-1), & \text{if } n > 0 \end{cases}$$

In this example, the base case is factorial(0) = 1, and the recursive case is $n \times \text{factorial}(n-1)$.

Recursion Tree:

A recursion tree is a tree representation of the recursive calls made by a recursive algorithm. Each node represents a function call, and its children represent the subsequent recursive calls. The depth of the tree represents the depth of recursion.

3.7 Sorting Algorithms

3.7.1 Shell Sort

Shell sort is an in-place comparison sort algorithm that extends the basic insertion sort algorithm by allowing exchanges of elements that are far apart. The main idea is to rearrange the elements so that elements that are far apart are sorted before doing a finer sort using insertion sort.

Shell sort improves the performance of insertion sort by breaking the original list into sublists based on a gap sequence and then sorting each sublist using insertion sort. This allows the algorithm to move elements more efficiently, especially when they are far apart from their correct positions.

Algorithm:

1. Start with a large gap between elements. A commonly used gap sequence is to divide the length of the list by 2 repeatedly until the gap is 1.
2. For each gap size, go through the list and compare elements that are that gap distance apart.
3. Use insertion sort to sort the sublists created by these gaps.
4. Continue reducing the gap until it becomes 1. When the gap is 1, the list is fully sorted by insertion sort.

Shell Sort Algorithm (Pseudocode):

```
shellSort(arr, n):
    gap = n // 2  # Initialize the gap size
    while gap > 0:
        for i = gap to n-1:
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j = j - gap
            arr[j] = temp
        gap = gap // 2
```

Example: Let's sort the array $[12, 34, 54, 2, 3]$ using Shell sort.

Step 1: Initial array

$$[12, 34, 54, 2, 3]$$

1. Start with gap $= 5//2 = 2$, meaning the array will be divided into sublists based on the gap 2.

- Compare elements at index 0 and 2: $[12, 54]$. No change since $12 < 54$.
- Compare elements at index 1 and 3: $[34, 2]$. Swap since $34 > 2$, resulting in:

$$[12, 2, 54, 34, 3]$$

- Compare elements at index 2 and 4: $[54, 3]$. Swap since $54 > 3$, resulting in:

$$[12, 2, 3, 34, 54]$$

Step 2: After first pass with gap 2

$$[12, 2, 3, 34, 54]$$

2. Reduce gap to 1: gap $= 2//2 = 1$. Now we perform insertion sort on the whole array:

- Compare index 0 and 1: $[12, 2]$. Swap since $12 > 2$, resulting in:

$$[2, 12, 3, 34, 54]$$

- Compare index 1 and 2: $[12, 3]$. Swap since $12 > 3$, resulting in:

$$[2, 3, 12, 34, 54]$$

- Compare index 2 and 3: $[12, 34]$. No change.
- Compare index 3 and 4: $[34, 54]$. No change.

Step 3: After final pass with gap 1

$$[2, 3, 12, 34, 54]$$

At this point, the array is sorted.
Key Insights:

- Shell sort is more efficient than insertion sort for large lists, especially when elements are far from their final positions.
- The efficiency depends on the choice of the gap sequence. A commonly used sequence is gap $= n//2$, reducing until gap equals 1.

3.7.2 Quick Sort

Quick Sort is a divide-and-conquer algorithm that sorts an array by partitioning it into two sub-arrays around a pivot element. The sub-arrays are then sorted recursively.
Algorithm:

1. **Choose a Pivot:** Select an element from the array to act as the pivot.
2. **Partition:** Rearrange the array such that elements less than the pivot come before it, and elements greater come after it.

3. **Recursively Apply:** Apply the same process to the sub-arrays formed by the partition.

Pseudocode:

```
QuickSort(arr, low, high):
    if low < high:
        pivotIndex = Partition(arr, low, high)
        QuickSort(arr, low, pivotIndex - 1)
        QuickSort(arr, pivotIndex + 1, high)

Partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j = low to high - 1:
        if arr[j] < pivot:
            i = i + 1
            swap arr[i] with arr[j]
    swap arr[i + 1] with arr[high]
    return i + 1
```

Example: Consider the array:

$$[10, 7, 8, 9, 1, 5]$$

- Choose pivot: 5
- Partition around pivot 5:
$$[1, 5, 8, 9, 7, 10]$$
- Recursively apply Quick Sort to [1] and [8, 9, 7, 10]
- Continue until the entire array is sorted:
$$[1, 5, 7, 8, 9, 10]$$

Visualization:

Initial Array:

$$[10, 7, 8, 9, 1, 5]$$

After Partitioning:

$$[1, 5, 8, 9, 7, 10]$$

Final Sorted Array:

$$[1, 5, 7, 8, 9, 10]$$

Advantages of Quick Sort:

- **Efficient Average Case:** Quick Sort has an average-case time complexity of $O(n \log n)$.

- **In-Place Sorting:** It requires minimal additional space.

Disadvantages of Quick Sort:

- **Worst-Case Performance:** The worst-case time complexity is $O(n^2)$, typically occurring with poor pivot choices.
- **Not Stable:** Quick Sort is not a stable sort.

3.7.3 Merge Sort

Merge Sort is a stable, comparison-based divide-and-conquer algorithm that divides the array into smaller sub-arrays, sorts them, and then merges them back together.

Algorithm:

1. **Divide:** Recursively divide the array into two halves until each sub-array contains a single element.
2. **Merge:** Merge the sorted sub-arrays to produce sorted arrays until the entire array is merged.

Pseudocode:

```
MergeSort(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        MergeSort(arr, left, mid)
        MergeSort(arr, mid + 1, right)
        Merge(arr, left, mid, right)

Merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid
    L = arr[left:left + n1]
    R = arr[mid + 1:mid + 1 + n2]
    i = 0
    j = 0
    k = left
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i = i + 1
        else:
            arr[k] = R[j]
            j = j + 1
        k = k + 1
    while i < n1:
        arr[k] = L[i]
        i = i + 1
        k = k + 1
    while j < n2:
        arr[k] = R[j]
        j = j + 1
        k = k + 1
```

Example: Consider the array:

$$[38, 27, 43, 3, 9, 82, 10]$$

- Divide the array into:
$$[38, 27, 43, 3]$$
and
$$[9, 82, 10]$$
- Recursively divide these sub-arrays until single elements are obtained.
- Merge the single elements to produce sorted arrays:
$$[27, 38, 43, 3]$$
and
$$[9, 10, 82]$$
- Continue merging until the entire array is sorted:
$$[3, 9, 10, 27, 38, 43, 82]$$

Visualization:

Initial Array:
$$[38, 27, 43, 3, 9, 82, 10]$$

After Dividing and Merging:

$$[3, 9, 10, 27, 38, 43, 82]$$

Advantages of Merge Sort:

- **Stable Sort:** Merge Sort maintains the relative order of equal elements.
- **Predictable Performance:** It has a time complexity of $O(n \log n)$ in the worst, average, and best cases.

Disadvantages of Merge Sort:

- **Space Complexity:** It requires additional space for merging.
- **Slower for Small Lists:** It may be slower compared to algorithms like Quick Sort for smaller lists.

3.7.4 Heap Sort

Heap Sort is a comparison-based sorting algorithm that utilizes a binary heap data structure. It works by building a max heap and then repeatedly extracting the maximum element to build the sorted array.

Algorithm:

1. **Build a Max Heap:** Convert the input array into a max heap where the largest element is at the root.
2. **Extract Max:** Swap the root of the heap (maximum element) with the last element of the heap and then reduce the heap size by one. Heapify the root to maintain the max heap property.
3. **Repeat:** Continue the extraction and heapify process until the heap is empty.

Pseudocode:

```
HeapSort(arr):
    n = length(arr)
    BuildMaxHeap(arr)
    for i = n - 1 down to 1:
        swap arr[0] with arr[i]
        Heapify(arr, 0, i)

BuildMaxHeap(arr):
    n = length(arr)
    for i = n // 2 - 1 down to 0:
        Heapify(arr, i, n)

Heapify(arr, i, n):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        swap arr[i] with arr[largest]
        Heapify(arr, largest, n)
```

Advantages of Heap Sort:

- **In-Place Sorting:** Heap Sort does not require additional space beyond the input array.
- **Time Complexity:** It has a time complexity of $O(n \log n)$ for both average and worst cases.

Disadvantages of Heap Sort:

- **Not Stable:** Heap Sort is not a stable sort, meaning equal elements may not retain their original order.
- **Performance:** It can be slower compared to algorithms like Quick Sort due to the overhead of heap operations.

4 Comparison of Sorting Algorithms

Comparison Table:

Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Space Complexity
Shell Sort	$O(n \log n)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$

5 Sorting in Linear Time

5.1 Introduction to Linear Time Sorting

Linear time sorting algorithms such as Counting Sort, Radix Sort, and Bucket Sort are designed to sort data in linear time $O(n)$.

Example:

- Counting Sort: Efficient for small range of integers.

5.1.1 Bucket Sort

Bucket Sort is a distribution-based sorting algorithm that divides the input into several buckets and then sorts each bucket individually. It is particularly useful when the input is uniformly distributed over a range.

Algorithm:

1. **Create Buckets:** Create an empty bucket for each possible range.
2. **Distribute Elements:** Place each element into the appropriate bucket based on its value.
3. **Sort Buckets:** Sort each bucket individually using another sorting algorithm (e.g., Insertion Sort).
4. **Concatenate Buckets:** Combine the sorted buckets into a single sorted array.

Pseudocode:

```
BucketSort(arr):
    minValue = min(arr)
    maxValue = max(arr)
    bucketCount = number of buckets
    buckets = [[] for _ in range(bucketCount)]
    for num in arr:
        index = (num - minValue) // bucketWidth
        buckets[index].append(num)
    sortedArray = []
    for bucket in buckets:
        InsertionSort(bucket)
        sortedArray.extend(bucket)
    return sortedArray

InsertionSort(arr):
    for i from 1 to length(arr):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

Example: Consider the array:

$$[0.78, 0.17, 0.39, 0.26, 0.72]$$

- **Buckets Creation:** Create 5 buckets.
- **Distribute Elements:** Place elements into buckets.
- **Sort Buckets:** Sort each bucket using Insertion Sort.
- **Concatenate Buckets:** Combine the sorted buckets:

$$[0.17, 0.26, 0.39, 0.72, 0.78]$$

5.1.2 Stable Sort

Stable Sort maintains the relative order of equal elements. Examples include Insertion Sort and Merge Sort.

Algorithm for Stable Sort:

- **Insertion Sort:** Maintain the order of elements with equal keys by inserting each element into its correct position relative to previously sorted elements.

Pseudocode:

```
InsertionSort(arr):
    for i from 1 to length(arr):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

Example: Consider the array:

$$[4, 3, 2, 1]$$

- Sort the array:

$$[1, 2, 3, 4]$$

5.1.3 Radix Sort

Radix Sort is a non-comparative integer sorting algorithm that processes digits of numbers. It works by sorting numbers digit by digit, starting from the least significant digit to the most significant digit.

Algorithm:

1. **Determine Maximum Digits:** Find the maximum number of digits in the array.
2. **Sort by Digit:** Sort the array by each digit using a stable sort (e.g., Counting Sort).
3. **Repeat:** Continue until all digits are processed.

Pseudocode:

```
RadixSort(arr):
    maxValue = max(arr)
    exp = 1
    while maxValue // exp > 0:
        CountingSort(arr, exp)
        exp = exp * 10

CountingSort(arr, exp):
    n = length(arr)
    output = [0] * n
    count = [0] * 10
    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1
    for i in range(1, 10):
        count[i] += count[i - 1]
    for i in range(n - 1, -1, -1):
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1
    for i in range(n):
        arr[i] = output[i]
```

Example: Consider the array:

$$[170, 45, 75, 90, 802, 24, 2, 66]$$

- Sort by least significant digit:

$$[170, 90, 802, 2, 24, 45, 75, 66]$$

- Sort by next digit:
$$[802, 24, 45, 66, 75, 90, 170, 2]$$
- Continue until all digits are processed.

Question: Among Merge Sort, Insertion Sort, and Quick Sort, which algorithm performs the best in the worst case? Apply the best algorithm to sort the list

E, X, A, M, P, L, E

in alphabetical order.

Answer: - Merge Sort has a worst-case time complexity of $O(n \log n)$. - Insertion Sort has a worst-case time complexity of $O(n^2)$. - Quick Sort has a worst-case time complexity of $O(n^2)$, though its average-case complexity is $O(n \log n)$.

In the worst case, Merge Sort performs the best among these algorithms.

Sorted List using Merge Sort:

Step-by-Step Solution:

1. **Initial List:**
   - Given List:

     E, X, A, M, P, L, E
2. **Divide the List:**
   - Divide the list into two halves:

     E, X, A and M, P, L, E
3. **Recursive Division:**
   - For the first half E, X, A:
     - Divide further into:

       E and X, A
     - For X, A:
       * Divide into:

         X and A
   - For the second half M, P, L, E:
     - Divide further into:

       M, P and L, E
     - For M, P:
       * Divide into:

         M and P
     - For L, E:
       * Divide into:

         L and E
4. **Merge the Sorted Sublists:**

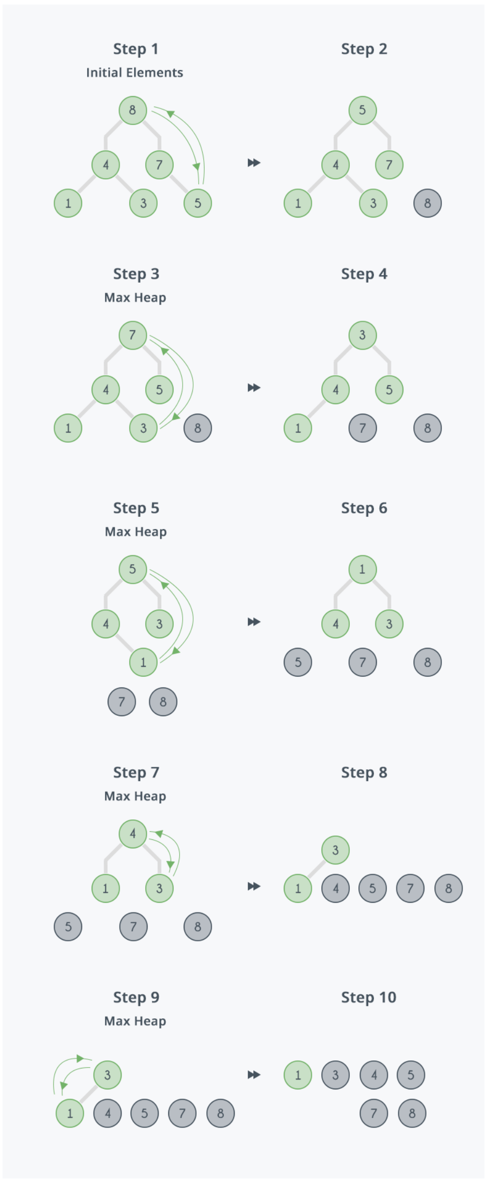
- Merge E and X, A:
  - Merge X and A to get:
    $$\text{A, X}$$
  - Merge E and A, X to get:
    $$\text{A, E, X}$$
- Merge M and P to get:
  $$\text{M, P}$$
- Merge L and E to get:
  $$\text{E, L}$$
- Merge M, P and E, L:
  - Merge M and E, L, P to get:

    $$\text{E, L, M, P}$$

5. **Merge Final Sublists:**
   - Merge A, E, X and E, L, M, P:
     - Final merge results in:

       $$\text{A, E, E, L, M, P, X}$$

6. **Sorted List:**
   - Sorted List:
     $$\text{A, E, E, L, M, P, X}$$


Step 1
Initial Elements
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10