Design and Analysis of Algorithms
BCS503
Unit-1: Comprehensive Notes

AKTU 5th Semester - CSE/IT
Prepared by: academicark
https://academicark-mvp8.onrender.com/

October 24, 2025

Contents

1 Introduction to Algorithms

1.1 What is an Algorithm?

An **algorithm** is a well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output. It is a sequence of computational steps that transform the input into the output.

Characteristics of a Good Algorithm:

- **Input:** Zero or more quantities are externally supplied
- **Output:** At least one quantity is produced
- **Definiteness:** Each instruction must be clear and unambiguous
- **Finiteness:** The algorithm must terminate after a finite number of steps
- **Effectiveness:** Every instruction must be basic enough to be carried out

Example 1: Algorithm to Find Maximum Element
Problem: Given an array of n elements, find the maximum element.

Algorithm 1 Find Maximum Element

```
1: procedure FindMax(A, n)
2:     max ← A[0]
3:     for i = 1 to n − 1 do
4:         if A[i] > max then
5:             max ← A[i]
6:         end if
7:     end for
8:     return max
9: end procedure
```

Example Execution:

- Input: $A = [5, 2, 9, 1, 7, 6]$, $n = 6$
- Step 1: $max = 5$
- Step 2: $i = 1$, $A[1] = 2 < 5$, no change
- Step 3: $i = 2$, $A[2] = 9 > 5$, $max = 9$
- Step 4: $i = 3$, $A[3] = 1 < 9$, no change
- Step 5: $i = 4$, $A[4] = 7 < 9$, no change
- Step 6: $i = 5$, $A[5] = 6 < 9$, no change
- Output: $max = 9$

1.2 Why Study Algorithms?

1. **Efficiency:** Different algorithms for solving the same problem can have drastically different performance characteristics
2. **Problem Solving:** Understanding algorithms improves problem-solving abilities
3. **Resource Optimization:** Helps in utilizing computational resources (time, memory) effectively
4. **Foundation:** Forms the basis for advanced computer science topics

2 Analyzing Algorithms

2.1 What is Algorithm Analysis?

Algorithm analysis is the process of determining the computational complexity of algorithms - the amount of time, storage, or other resources needed to execute them. The goal is to predict the behavior of an algorithm without implementing it on a specific computer.

2.2 Types of Analysis

2.2.1 Priori Analysis (Theoretical Analysis)

Analysis performed before the implementation of the algorithm. It is independent of:

- Programming language
- Computer hardware
- Compiler used

We use the following metrics:

- Number of comparisons
- Number of assignments
- Number of memory accesses

2.2.2 Posteriori Analysis (Empirical Analysis)

Analysis performed after implementing the algorithm by running it on a computer. It depends on:

- Hardware specifications
- Programming language
- Compiler efficiency

Why Prefer Priori Analysis?

- Machine-independent
- Provides insights before implementation
- Helps in algorithm selection

2.3 Time and Space Complexity

2.3.1 Time Complexity

Time complexity measures the amount of time taken by an algorithm to run as a function of the length of the input. It does not measure the actual time but counts the number of basic operations performed.

Example 2: Linear Search Time Complexity

Algorithm 2 Linear Search

```
1: procedure LinearSearch(A, n, key)
2:     for i = 0 to n − 1 do
3:         if A[i] == key then
4:             return i
5:         end if
6:     end for
7:     return −1
8: end procedure
```

Analysis:

- Best Case: Element found at first position - $O(1)$
- Worst Case: Element not present or at last position - $O(n)$
- Average Case: Element found at middle position - $O(n)$

2.3.2 Space Complexity

Space complexity measures the total amount of memory space required by an algorithm. It includes:

- Fixed part: Space for code, constants, variables (independent of input size)
- Variable part: Space that depends on input size (arrays, recursion stack)

Example: The Linear Search algorithm has space complexity $O(1)$ as it uses only a constant amount of extra space.

3 Complexity of Algorithms

3.1 Case Analysis

3.1.1 Best Case Complexity

The minimum time or resources required by an algorithm for any input of size n. It represents the optimistic scenario.

Example: In Linear Search, best case occurs when the target element is at the first position: $\Theta(1)$

3.1.2 Worst Case Complexity

The maximum time or resources required by an algorithm for any input of size n. It provides an upper bound on the running time.

Example: In Linear Search, worst case occurs when the element is not present: $\Theta(n)$

Why Worst Case Analysis is Important?

- Provides guarantee that algorithm will never take more time
- For many algorithms, worst case occurs fairly often
- Average case is often as bad as worst case

3.1.3 Average Case Complexity

The expected time or resources required considering all possible inputs and their probabilities.

Calculation: For Linear Search with n elements:

$$T_{avg} = \frac{1}{n+1}\left[\sum_{i=1}^{n} i + n\right] = \frac{1}{n+1}\left[\frac{n(n+1)}{2} + n\right] = \frac{n+2}{2}$$

Therefore, average case complexity is $\Theta(n)$.

3.2 Example 3: Insertion Sort Analysis

Algorithm 3 Insertion Sort

```
1: procedure InsertionSort(A, n)
2:     for i = 1 to n − 1 do
3:         key ← A[i]
4:         j ← i − 1
5:         while j ≥ 0 and A[j] > key do
6:             A[j + 1] ← A[j]
7:             j ← j − 1
8:         end while
9:         A[j + 1] ← key
10:    end for
11: end procedure
```

Example Execution:
Input Array: $[5, 2, 4, 6, 1, 3]$

- **Pass 1:** $i = 1$, $key = 2$, compare with 5, insert 2: $[2, 5, 4, 6, 1, 3]$
- **Pass 2:** $i = 2$, $key = 4$, compare with 5, insert 4: $[2, 4, 5, 6, 1, 3]$
- **Pass 3:** $i = 3$, $key = 6$, no change: $[2, 4, 5, 6, 1, 3]$

- **Pass 4:** $i = 4$, $key = 1$, shift all, insert 1: $[1, 2, 4, 5, 6, 3]$
- **Pass 5:** $i = 5$, $key = 3$, shift and insert: $[1, 2, 3, 4, 5, 6]$

Complexity Analysis:

- **Best Case:** Array already sorted - $\Theta(n)$
- **Worst Case:** Array in reverse order - $\Theta(n^2)$
- **Average Case:** Random order - $\Theta(n^2)$
- **Space Complexity:** $O(1)$ (in-place sorting)

4 Growth of Functions

4.1 Asymptotic Notations

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. They describe the limiting behavior of a function when the argument tends towards infinity.

4.1.1 Big-O Notation (O)

Big-O notation provides an **upper bound** on the growth rate of a function. It represents the worst-case scenario.

Definition:

$$f(n) = O(g(n)) \text{ if and only if } \exists \text{ constants } c > 0 \text{ and } n_0 \geq 0$$

such that

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Example 4: Prove that $f(n) = 3n^2 + 5n + 2 = O(n^2)$

Proof:

$$\begin{aligned} 3n^2 + 5n + 2 &\leq 3n^2 + 5n^2 + 2n^2 \quad (\text{for } n \geq 1) \\ &= 10n^2 \\ &= c \cdot n^2 \quad \text{where } c = 10, n_0 = 1 \end{aligned}$$

Therefore, $f(n) = O(n^2)$.

4.1.2 Big-Omega Notation (Ω)

Big-Omega notation provides a **lower bound** on the growth rate of a function. It represents the best-case scenario.

Definition:

$$f(n) = \Omega(g(n)) \text{ if and only if } \exists \text{ constants } c > 0 \text{ and } n_0 \geq 0$$

such that

$$0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0$$

Example 5: Prove that $f(n) = 5n^2 + 3n = \Omega(n^2)$
Proof:

$$\begin{aligned} 5n^2 + 3n &\geq 5n^2 \quad (\text{for all } n \geq 0) \\ &= c \cdot n^2 \quad \text{where } c = 5, n_0 = 0 \end{aligned}$$

Therefore, $f(n) = \Omega(n^2)$.

4.1.3 Big-Theta Notation (Θ)

Big-Theta notation provides a **tight bound** on the growth rate. It represents both upper and lower bounds.
Definition:

$$f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

Equivalently, $\exists$ constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0$$

Example 6: Prove that $f(n) = 4n^2 + 2n + 1 = \Theta(n^2)$
Proof:
For upper bound ($O(n^2)$):

$$\begin{aligned} 4n^2 + 2n + 1 &\leq 4n^2 + 2n^2 + n^2 \quad (\text{for } n \geq 1) \\ &= 7n^2 = c_2 \cdot n^2 \quad \text{where } c_2 = 7 \end{aligned}$$

For lower bound ($\Omega(n^2)$):

$$4n^2 + 2n + 1 \geq 4n^2 = c_1 \cdot n^2 \quad \text{where } c_1 = 4$$

Since both conditions are satisfied with $c_1 = 4, c_2 = 7, n_0 = 1$, we have $f(n) = \Theta(n^2)$.

4.2 Common Growth Rates

In increasing order of growth:

Notation	Name	Example
$O(1)$	Constant	Array access
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Linear search
$O(n \log n)$	Linearithmic	Merge sort, Heap sort
$O(n^2)$	Quadratic	Bubble sort, Insertion sort
$O(n^3)$	Cubic	Matrix multiplication
$O(2^n)$	Exponential	Fibonacci (naive)
$O(n!)$	Factorial	Traveling salesman (brute force)

Growth Rate Comparison:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

4.3 Properties of Asymptotic Notations

4.3.1 Transitivity

- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- Same applies for Ω and Θ

4.3.2 Reflexivity

- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$
- $f(n) = \Theta(f(n))$

4.3.3 Symmetry

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

4.3.4 Transpose Symmetry

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

5 Performance Measurements

5.1 Empirical Performance Measurement

Steps for Empirical Analysis:

1. Implement the algorithm
2. Prepare test data sets of varying sizes
3. Execute the algorithm on each test data set
4. Measure actual execution time
5. Plot the results (input size vs. time)
6. Analyze the trend

5.2 Benchmarking

Benchmarking involves comparing the performance of different algorithms on the same hardware and software environment.

Key Metrics:

- **Execution Time:** Actual time taken to execute
- **Memory Usage:** RAM consumed during execution
- **CPU Cycles:** Number of processor cycles used
- **Cache Misses:** Memory access patterns

5.3 Profiling

Profiling is the process of measuring the space (memory) and time complexity of a program, finding hotspots in the code, and identifying which parts of the code consume most resources.

Profiling Tools:

- gprof (GNU Profiler)
- Valgrind
- Python cProfile
- Java VisualVM

6 Sorting and Order Statistics

Sorting is the process of arranging data in a particular order (ascending or descending). It is one of the most fundamental operations in computer science.

6.1 Classification of Sorting Algorithms

6.1.1 Based on Stability

- **Stable:** Maintains relative order of equal elements (Merge Sort, Insertion Sort)
- **Unstable:** May change relative order (Quick Sort, Heap Sort)

6.1.2 Based on Space Complexity

- **In-place:** Uses $O(1)$ extra space (Quick Sort, Heap Sort)
- **Out-place:** Uses $O(n)$ or more extra space (Merge Sort)

6.1.3 Based on Comparison

- **Comparison-based:** Elements are compared (Quick Sort, Merge Sort)
- **Non-comparison-based:** Uses other properties (Counting Sort, Radix Sort)

7 Shell Sort

Shell Sort is an in-place comparison-based sorting algorithm. It is a generalization of insertion sort that allows the exchange of items that are far apart. The algorithm performs multiple passes through the array with decreasing gap sizes.

Algorithm 4 Shell Sort

```
1: procedure ShellSort(A, n)
2:     gap ← n/2
3:     while gap > 0 do
4:         for i = gap to n − 1 do
5:             temp ← A[i]
6:             j ← i
7:             while j ≥ gap and A[j − gap] > temp do
8:                 A[j] ← A[j − gap]
9:                 j ← j − gap
10:            end while
11:            A[j] ← temp
12:        end for
13:        gap ← gap/2
14:    end while
15: end procedure
```

7.1 Algorithm

7.2 Example 7: Shell Sort Execution

Input Array: $[35, 33, 42, 10, 14, 19, 27, 44]$, $n = 8$

Pass 1: gap = 4

- Compare elements 4 positions apart: $(35, 14), (33, 19), (42, 27), (10, 44)$
- After sorting: $[14, 19, 27, 10, 35, 33, 42, 44]$

Pass 2: gap = 2

- Compare elements 2 positions apart
- After sorting: $[14, 10, 27, 19, 35, 33, 42, 44]$

Pass 3: gap = 1

- Regular insertion sort
- Final sorted array: $[10, 14, 19, 27, 33, 35, 42, 44]$

7.3 Complexity Analysis

Case	Time Complexity
Best Case	$O(n \log n)$
Average Case	$O(n^{1.5})$ or $O(n(\log n)^2)$
Worst Case	$O(n^2)$
Space Complexity	$O(1)$

Note: The time complexity depends on the gap sequence used. Common sequences:

- Shell's original: $n/2, n/4, ..., 1$ - Worst case $O(n^2)$
- Knuth's: $1, 4, 13, ..., (3^k - 1)/2$ - Worst case $O(n^{1.5})$
- Sedgewick's: $1, 8, 23, 77, ...$ - Average case $O(n^{1.33})$

7.4 Advantages and Disadvantages

Advantages:

- Faster than insertion sort and selection sort for large arrays
- In-place sorting (uses constant extra space)
- Performs well on partially sorted arrays
- Simple to implement

Disadvantages:

- Not stable
- Complexity depends on gap sequence
- Slower than Quick Sort and Merge Sort for large datasets

8 Quick Sort

Quick Sort is a highly efficient, divide-and-conquer sorting algorithm. It works by selecting a 'pivot' element and partitioning the array around the pivot such that elements smaller than the pivot are on the left and elements greater than the pivot are on the right.

8.1 Algorithm

Algorithm 5 Quick Sort

```
1: procedure QuickSort(A, low, high)
2:     if low < high then
3:         pivotIndex ← Partition(A, low, high)
4:         QuickSort(A, low, pivotIndex − 1)
5:         QuickSort(A, pivotIndex + 1, high)
6:     end if
7: end procedure
```

8.2 Example 8: Quick Sort Execution

Input Array: $[10, 7, 8, 9, 1, 5]$

Step 1: Initial Call

- $low = 0, high = 5$, $pivot = 5$ (last element)
- After partition: $[1, 5, 8, 9, 7, 10]$, pivot at index 1

Step 2: Recursion on Left Sub-array $[1]$

- Single element, already sorted

Algorithm 6 Partition

```
1: procedure PARTITION(A, low, high)
2:     pivot ← A[high]
3:     i ← low − 1
4:     for j = low to high − 1 do
5:         if A[j] ≤ pivot then
6:             i ← i + 1
7:             SWAP(A[i], A[j])
8:         end if
9:     end for
10:    SWAP(A[i + 1], A[high])
11:    return i + 1
12: end procedure
```

Step 3: Recursion on Right Sub-array $[8, 9, 7, 10]$

- $pivot = 10$
- After partition: $[8, 9, 7, 10]$, pivot at index 5
- Further recursion on $[8, 9, 7]$

Step 4: Continue until sorted

- Final sorted array: $[1, 5, 7, 8, 9, 10]$

8.3 Complexity Analysis

Case	Time Complexity
Best Case	$O(n \log n)$ - Balanced partitions
Average Case	$O(n \log n)$
Worst Case	$O(n^2)$ - Already sorted or reverse sorted
Space Complexity	$O(\log n)$ - Recursion stack

8.4 Pivot Selection Strategies

1. **First element as pivot:** Simple but inefficient for sorted arrays
2. **Last element as pivot:** Commonly used, same issue as first element
3. **Random element as pivot:** Randomized Quick Sort, avoids worst case on average
4. **Median-of-three:** Choose median of first, middle, and last elements

8.5 Advantages and Disadvantages

Advantages:

- One of the fastest sorting algorithms in practice
- In-place sorting

- Cache-friendly (good locality of reference)
- Works well with virtual memory

Disadvantages:

- Unstable sorting algorithm
- Worst case $O(n^2)$ for sorted or reverse sorted arrays
- Not suitable for linked lists
- Recursive implementation uses stack space

9 Merge Sort

Merge Sort is a stable, divide-and-conquer sorting algorithm that divides the input array into two halves, recursively sorts them, and then merges the two sorted halves.

9.1 Algorithm

Algorithm 7 Merge Sort

```
1: procedure MergeSort(A, left, right)
2:     if left < right then
3:         mid ← ⌊(left + right)/2⌋
4:         MergeSort(A, left, mid)
5:         MergeSort(A, mid + 1, right)
6:         Merge(A, left, mid, right)
7:     end if
8: end procedure
```

9.2 Example 9: Merge Sort Execution

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

Divide Phase:

- Level 0: $[38, 27, 43, 3, 9, 82, 10]$
- Level 1: $[38, 27, 43, 3]$ and $[9, 82, 10]$
- Level 2: $[38, 27]$, $[43, 3]$, $[9, 82]$, $[10]$
- Level 3: $[38]$, $[27]$, $[43]$, $[3]$, $[9]$, $[82]$, $[10]$

Merge Phase:

- Merge $[38]$ and $[27] \rightarrow [27, 38]$
- Merge $[43]$ and $[3] \rightarrow [3, 43]$

Algorithm 8 Merge

```
1:  procedure MERGE(A, left, mid, right)
2:      n_1 ← mid − left + 1
3:      n_2 ← right − mid
4:      Create temp arrays L[0...n_1 − 1] and R[0...n_2 − 1]
5:      for i = 0 to n_1 − 1 do
6:          L[i] ← A[left + i]
7:      end for
8:      for j = 0 to n_2 − 1 do
9:          R[j] ← A[mid + 1 + j]
10:     end for
11:     i ← 0, j ← 0, k ← left
12:     while i < n_1 and j < n_2 do
13:         if L[i] ≤ R[j] then
14:             A[k] ← L[i]
15:             i ← i + 1
16:         else
17:             A[k] ← R[j]
18:             j ← j + 1
19:         end if
20:         k ← k + 1
21:     end while
22:     while i < n_1 do
23:         A[k] ← L[i]
24:         i ← i + 1, k ← k + 1
25:     end while
26:     while j < n_2 do
27:         A[k] ← R[j]
28:         j ← j + 1, k ← k + 1
29:     end while
30: end procedure
```

- Merge [9] and [82] $\rightarrow$ [9, 82]
- [10] remains
- Merge [27, 38] and [3, 43] $\rightarrow$ [3, 27, 38, 43]
- Merge [9, 82] and [10] $\rightarrow$ [9, 10, 82]
- Final merge: [3, 9, 10, 27, 38, 43, 82]

9.3 Complexity Analysis

Case	Time Complexity
Best Case	$O(n \log n)$
Average Case	$O(n \log n)$
Worst Case	$O(n \log n)$
Space Complexity	$O(n)$ - Temporary arrays

Recurrence Relation:

$$T(n) = 2T(n/2) + \Theta(n)$$

Using Master Theorem (Case 2):

$$T(n) = \Theta(n \log n)$$

9.4 Advantages and Disadvantages

Advantages:

- Stable sorting algorithm
- Guaranteed $O(n \log n)$ time complexity
- Predictable performance
- Suitable for linked lists
- Parallelizable

Disadvantages:

- Requires $O(n)$ extra space
- Slower in practice than Quick Sort for arrays
- Not in-place
- Overhead of copying to temporary arrays

10 Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It divides the input into a sorted and an unsorted region, and iteratively shrinks the unsorted region by extracting the largest element.

10.1 Binary Heap Properties

A binary heap is a complete binary tree that satisfies the heap property:

- **Max Heap:** Parent node child nodes
- **Min Heap:** Parent node child nodes

Array Representation: For a node at index i:

- Left child: $2i + 1$
- Right child: $2i + 2$
- Parent: $\lfloor (i - 1)/2 \rfloor$

10.2 Algorithm

Algorithm 9 Heap Sort

```
1: procedure HeapSort(A, n)
2:     BuildMaxHeap(A, n)
3:     for i = n − 1 downto 1 do
4:         Swap(A[0], A[i])
5:         MaxHeapify(A, 0, i)
6:     end for
7: end procedure
```

Algorithm 10 Build Max Heap

```
1: procedure BuildMaxHeap(A, n)
2:     for i = ⌊n/2⌋ − 1 downto 0 do
3:         MaxHeapify(A, i, n)
4:     end for
5: end procedure
```

10.3 Example 10: Heap Sort Execution

Input Array: $[4, 10, 3, 5, 1]$

Step 1: Build Max Heap

Initial array: $[4, 10, 3, 5, 1]$

After heapify operations:

- Heapify at index 1: $[4, 10, 3, 5, 1]$ (10 ¿ 5 and 1, no change)
- Heapify at index 0: Swap 4 and 10 $\rightarrow [10, 4, 3, 5, 1]$
- Further heapify: Swap 4 and 5 $\rightarrow [10, 5, 3, 4, 1]$

Max Heap: $[10, 5, 3, 4, 1]$

Step 2: Extract Elements

Algorithm 11 Max Heapify

```
1: procedure MaxHeapify(A, i, heapSize)
2:     left ← 2i + 1
3:     right ← 2i + 2
4:     largest ← i
5:     if left < heapSize and A[left] > A[largest] then
6:         largest ← left
7:     end if
8:     if right < heapSize and A[right] > A[largest] then
9:         largest ← right
10:    end if
11:    if largest ≠ i then
12:        Swap(A[i], A[largest])
13:        MaxHeapify(A, largest, heapSize)
14:    end if
15: end procedure
```

- Swap 10 and 1: $[1, 5, 3, 4, 10]$, heapify $\rightarrow [5, 4, 3, 1, 10]$
- Swap 5 and 1: $[1, 4, 3, 5, 10]$, heapify $\rightarrow [4, 1, 3, 5, 10]$
- Swap 4 and 3: $[3, 1, 4, 5, 10]$, heapify $\rightarrow [3, 1, 4, 5, 10]$
- Swap 3 and 1: $[1, 3, 4, 5, 10]$

Final Sorted Array: $[1, 3, 4, 5, 10]$

10.4 Complexity Analysis

Case	Time Complexity
Best Case	$O(n \log n)$
Average Case	$O(n \log n)$
Worst Case	$O(n \log n)$
Space Complexity	$O(1)$ - In-place
Building Heap	$O(n)$

10.5 Advantages and Disadvantages

Advantages:

- Guaranteed $O(n \log n)$ worst-case performance
- In-place sorting
- No recursion (can be implemented iteratively)
- Memory efficient

Disadvantages:

- Unstable sorting algorithm

- Slower than Quick Sort in practice
- Poor cache performance
- Complex implementation

11 Comparison of Sorting Algorithms

11.1 Comprehensive Comparison Table

Algorithm	Best	Average	Worst	Space	Stable	In-place
Shell Sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^2)$	$O(1)$	No	Yes
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	No
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes

11.2 When to Use Which Algorithm?

- **Quick Sort:**
  - General-purpose sorting
  - Average case performance is critical
  - Memory is limited
- **Merge Sort:**
  - Stability is required
  - Guaranteed $O(n \log n)$ performance needed
  - External sorting (sorting data on disk)
  - Linked list sorting
- **Heap Sort:**
  - Memory is very limited
  - Guaranteed $O(n \log n)$ without extra space
  - Finding k largest/smallest elements
- **Shell Sort:**
  - Medium-sized arrays
  - Simple implementation needed
  - Better than $O(n^2)$ required but space is limited
- **Insertion Sort:**

- Small arrays ($n < 50$)
- Nearly sorted data
- Online sorting (elements arrive one at a time)

12 Sorting in Linear Time

12.1 Lower Bound for Comparison-Based Sorting

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Proof Idea:

- Consider all possible permutations of n elements: $n!$ permutations
- Each comparison gives binary decision (yes/no)
- Decision tree has $n!$ leaves (one for each permutation)
- Height of binary tree with $n!$ leaves is at least $\log_2(n!)$
- Using Stirling's approximation: $\log_2(n!) = \Theta(n \log n)$

Therefore, comparison-based sorting has lower bound $\Omega(n \log n)$.

12.2 Counting Sort

Counting Sort is a non-comparison-based sorting algorithm that works by counting the occurrences of each distinct element. It assumes that input elements are integers in range $[0, k]$.

12.2.1 Algorithm

12.2.2 Example 11: Counting Sort

Input: $A = [2, 5, 3, 0, 2, 3, 0, 3]$, $n = 8$, $k = 5$ (range 0 to 5)

Step 1: Count Occurrences

- $C = [2, 0, 2, 3, 0, 1]$ (counts of 0,1,2,3,4,5)

Step 2: Cumulative Count

- $C = [2, 2, 4, 7, 7, 8]$

Step 3: Build Output Array

- Process from right to left: $A[7] = 3$, place at position $C[3] - 1 = 6$
- Continue for all elements
- Output: $B = [0, 0, 2, 2, 3, 3, 3, 5]$

Algorithm 12 Counting Sort

```
1: procedure CountingSort(A, n, k)
2:     Create arrays C[0...k] and B[0...n − 1]
3:     for i = 0 to k do
4:         C[i] ← 0
5:     end for
6:     for i = 0 to n − 1 do
7:         C[A[i]] ← C[A[i]] + 1
8:     end for
9:     for i = 1 to k do
10:        C[i] ← C[i] + C[i − 1]
11:    end for
12:    for i = n − 1 downto 0 do
13:        B[C[A[i]] − 1] ← A[i]
14:        C[A[i]] ← C[A[i]] − 1
15:    end for
16:    Copy B to A
17: end procedure
```

12.2.3 Complexity Analysis

Metric	Complexity
Time Complexity	$O(n + k)$
Space Complexity	$O(n + k)$
Stable	Yes

When to Use:

- $k = O(n)$ (range is not significantly larger than n)
- Integer sorting with small range
- Stability is required

12.3 Radix Sort

Radix Sort processes digits of numbers from least significant to most significant (or vice versa), using a stable sorting algorithm (typically Counting Sort) as a subroutine.

12.3.1 Algorithm

Algorithm 13 Radix Sort (LSD)

```
1: procedure RadixSort(A, n, d)
2:     for i = 1 to d do
3:         Use stable sort to sort A on digit i
4:     end for
5: end procedure
```

12.3.2 Example 12: Radix Sort

Input: $A = [170, 45, 75, 90, 802, 24, 2, 66]$

Sorting by Ones Place (digit 1):

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

Sorting by Tens Place (digit 2):

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

Sorting by Hundreds Place (digit 3):

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

12.3.3 Complexity Analysis

For n numbers with d digits, where each digit has b possible values:

Metric	Complexity
Time Complexity	$O(d(n+b))$
Space Complexity	$O(n+b)$
Stable	Yes

When to Use:

- Fixed-length integer or string sorting
- d is small
- Stability is required

12.4 Bucket Sort

Bucket Sort distributes elements into several buckets, sorts each bucket individually (using another algorithm or recursively), and then concatenates the sorted buckets.

12.4.1 Algorithm

Algorithm 14 Bucket Sort

```
procedure BucketSort(A, n)
    Create n empty buckets
    for i = 0 to n − 1 do
        Insert A[i] into bucket[⌊n · A[i]⌋]
    end for
    for i = 0 to n − 1 do
        Sort bucket[i] using Insertion Sort
    end for
    Concatenate all buckets in order
end procedure
```

12.4.2 Example 13: Bucket Sort

Input: $A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]$, $n = 10$

Step 1: Distribute into Buckets

- Bucket 0: []
- Bucket 1: $[0.17, 0.12]$
- Bucket 2: $[0.26, 0.21, 0.23]$
- Bucket 3: $[0.39]$
- Bucket 4: []
- Bucket 5: []
- Bucket 6: $[0.68]$
- Bucket 7: $[0.78, 0.72]$
- Bucket 8: []
- Bucket 9: $[0.94]$

Step 2: Sort Each Bucket

- Bucket 1: $[0.12, 0.17]$
- Bucket 2: $[0.21, 0.23, 0.26]$
- Bucket 7: $[0.72, 0.78]$

Step 3: Concatenate

- Output: $[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]$

12.4.3 Complexity Analysis

Case	Time Complexity
Average Case	$O(n + k)$
Worst Case	$O(n^2)$
Space Complexity	$O(n + k)$

Assumptions for Average Case:

- Input is uniformly distributed over $[0, 1)$
- Elements are distributed evenly among buckets

13 Order Statistics

Order statistics deal with finding the k^{th} smallest (or largest) element in an array.

13.1 Selection Problem

Problem: Given an unsorted array A of n elements, find the k^{th} smallest element.

Special Cases:

- $k = 1$: Minimum element
- $k = n$: Maximum element
- $k = \lceil n/2 \rceil$: Median

13.2 Randomized Select Algorithm

Based on QuickSort's partition method:

Algorithm 15 Randomized Select

```
1: procedure RandomizedSelect(A, low, high, k)
2:     if low == high then
3:         return A[low]
4:     end if
5:     pivotIndex ← RandomizedPartition(A, low, high)
6:     i ← pivotIndex − low + 1
7:     if k == i then
8:         return A[pivotIndex]
9:     else if k < i then
10:        return RandomizedSelect(A, low, pivotIndex − 1, k)
11:    else
12:        return RandomizedSelect(A, pivotIndex + 1, high, k − i)
13:    end if
14: end procedure
```

Complexity:

- Expected Time: $O(n)$
- Worst Case: $O(n^2)$

14 Important Questions for AKTU Exam

14.1 Short Answer Questions

1. Define algorithm and list its characteristics.
2. Explain Big-O, Big-Omega, and Big-Theta notations with examples.
3. What is the difference between time complexity and space complexity?
4. Compare stable and unstable sorting algorithms.
5. Why is Quick Sort preferred over Merge Sort in practice?
6. What is the lower bound for comparison-based sorting?

7. Explain in-place and out-of-place sorting algorithms.
8. What is the significance of pivot selection in Quick Sort?
9. Differentiate between best case, average case, and worst case analysis.
10. What are the applications of Heap Sort?

14.2 Long Answer Questions

1. Explain Quick Sort algorithm with example. Analyze its time complexity for all cases.
2. Describe Merge Sort algorithm. Trace the algorithm for the array [12, 11, 13, 5, 6, 7].
3. What is Heap Sort? Explain the algorithm with suitable example and analyze its complexity.
4. Explain Shell Sort with example. How does gap sequence affect its performance?
5. Compare all sorting algorithms studied in Unit-1 in terms of time complexity, space complexity, and stability.
6. Explain Counting Sort, Radix Sort, and Bucket Sort. When are they preferred over comparison-based sorting?
7. What is the order statistics problem? Explain randomized selection algorithm.
8. Derive the lower bound for comparison-based sorting using decision tree model.
9. Explain asymptotic notations with mathematical definitions and graphical representation.
10. Write and analyze algorithms for finding maximum and minimum elements in an array.

14.3 Numerical Problems

1. Sort the array [64, 34, 25, 12, 22, 11, 90] using:
   - Quick Sort
   - Merge Sort
   - Heap Sort
2. Prove that $f(n) = 5n^3 + 3n^2 + 2n + 1 = \Theta(n^3)$.
3. Given array [2, 1, 6, 0, 4, 2, 8, 3], perform Counting Sort.
4. Find the 4th smallest element in [7, 10, 4, 3, 20, 15] using selection algorithm.
5. Analyze the time complexity of the following code:

```
for(i=1; i<=n; i++)
    for(j=1; j<=i; j++)
        for(k=1; k<=100; k++)
            // O(1) operation
```

15 Summary and Key Takeaways

15.1 Algorithm Analysis

- Asymptotic analysis helps compare algorithms independent of hardware
- Big-O (upper bound), Big-Omega (lower bound), Big-Theta (tight bound)
- Worst case analysis is most commonly used

15.2 Sorting Algorithms

Comparison-Based:

- Quick Sort: Average $O(n \log n)$, in-place, unstable
- Merge Sort: Guaranteed $O(n \log n)$, stable, requires extra space
- Heap Sort: Guaranteed $O(n \log n)$, in-place, unstable
- Shell Sort: Better than $O(n^2)$, in-place, gap sequence dependent

Non-Comparison-Based:

- Counting Sort: $O(n + k)$, stable, requires range knowledge
- Radix Sort: $O(d(n + b))$, stable, for fixed-length integers
- Bucket Sort: $O(n)$ average, for uniformly distributed data

15.3 Important Formulas

- Arithmetic series: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Geometric series: $\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$
- Logarithm properties: $\log(ab) = \log a + \log b$, $\log(a^b) = b \log a$
- Master Theorem for recurrences
- Stirling's approximation: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

16 Tips for AKTU Exam

1. **Practice tracing algorithms:** Be able to trace sorting algorithms step-by-step on given arrays
2. **Understand complexity derivation:** Know how to derive time complexity from code
3. **Memorize comparison table:** Remember the complexity table for all sorting algorithms
4. **Master asymptotic notations:** Practice proving Big-O, Omega, and Theta relations
5. **Know when to use which algorithm:** Understand practical applications and trade-offs
6. **Write clear algorithms:** Use proper pseudocode format in exam
7. **Draw diagrams:** For heap sort, merge sort tree, decision tree
8. **Time management:** Allocate time based on marks distribution
9. **Revise previous year questions:** Many questions are repeated with variations
10. **Understand, don't memorize:** Focus on concepts rather than rote learning

Best Wishes for Your Exam!