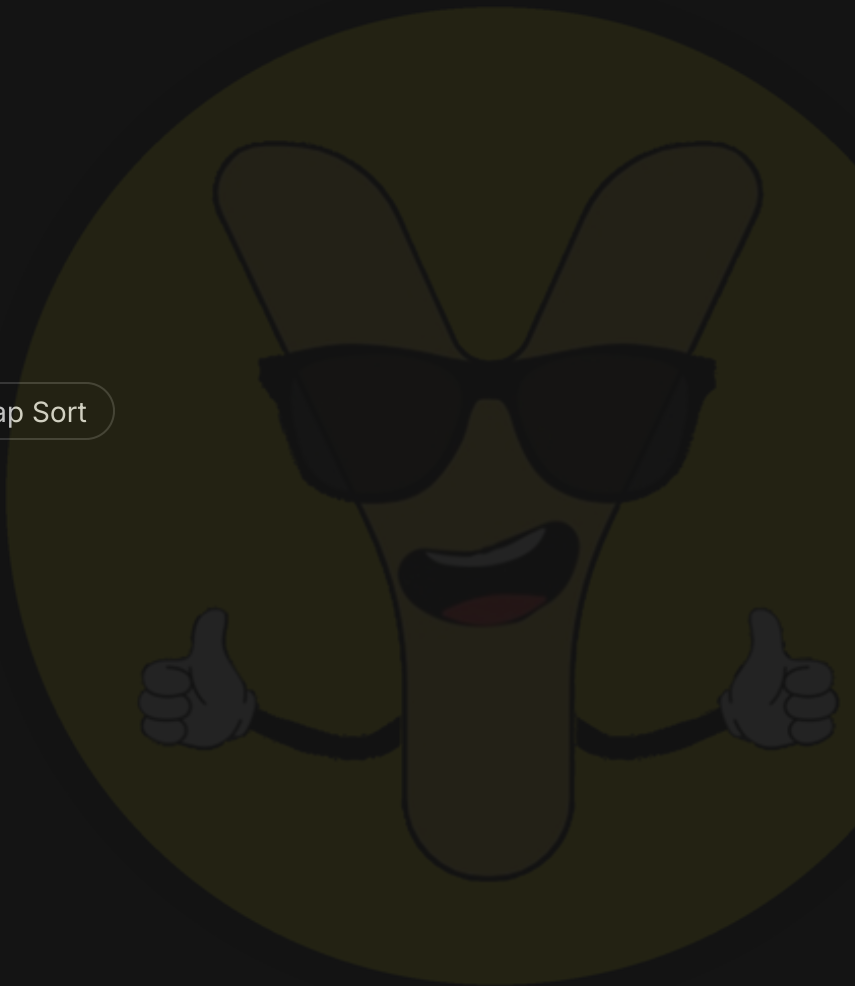
AKTU · 3RD YEAR · SEMESTER 5 · BCS503

Design and Analysis of Algorithm

Unit 1 — Introduction and Sorting Algorithms

- Introduction: Algorithms
- Analyzing Algorithms
- Complexity of Algorithms
- Growth of Functions
- Performance Measurements
- Sorting and Order Statistics - Shell Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Comparison of Sorting Algorithms
- Sorting in Linear Time



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

theycotes.com

Introduction to Algorithms

ALGORITHM

An algorithm is a well-defined computational procedure that takes a set of values as input and produces a set of values as output to solve a specific problem in a finite number of steps.

- Unambiguous: Every instruction must be clear and distinct.
- Finiteness: The algorithm must terminate after a finite number of steps.
- Input & Output: Must accept zero or more inputs and produce at least one output.
- Effectiveness: Operations must be basic enough to be executed in practice.

Analyzing Algorithms

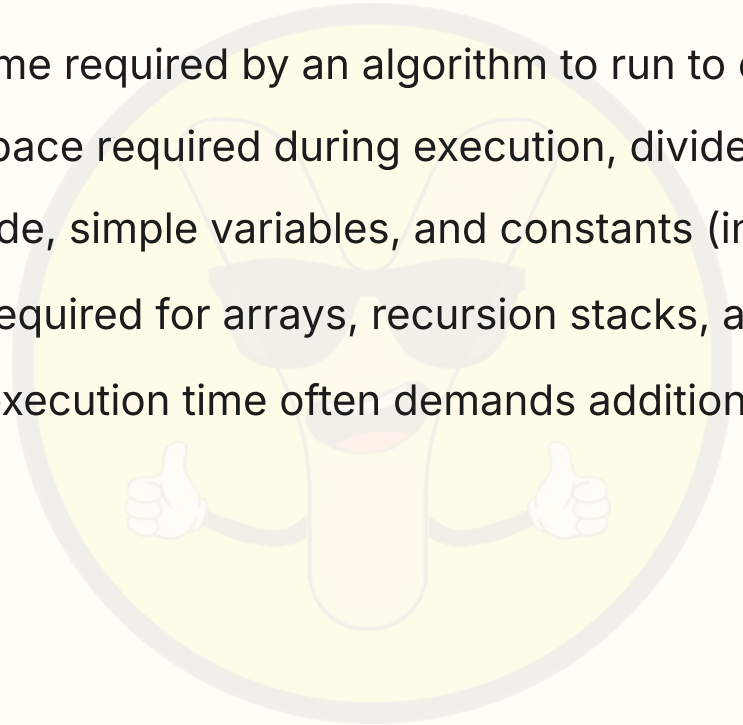
Algorithm analysis evaluates the resource requirements (execution time and memory) as input size grows.

- RAM Model of Computation: Assumes sequential execution of instructions with constant time per basic operation.
- Input Size n: Resource requirements are expressed as a function of the input size n.
- Primitive Operations: Counts basic operations like additions, comparisons, and variable assignments.
- Best-Case Scenario: Minimum running time required over all inputs of size n.
- Worst-Case Scenario: Maximum running time over all inputs of size n; primary focus in algorithm design.
- Average-Case Scenario: Expected running time over a statistical distribution of inputs of size n.

Complexity of Algorithms

Complexity quantifies how the running time and memory footprint scale relative to input size.

- Time Complexity: Total execution time required by an algorithm to run to completion.
- Space Complexity: Total memory space required during execution, divided into fixed and variable space.
- Fixed Space: Space required for code, simple variables, and constants (independent of n).
- Variable Space: Dynamic memory required for arrays, recursion stacks, and inputs (dependent on n).
- Space-Time Tradeoff: Decreasing execution time often demands additional memory allocation.



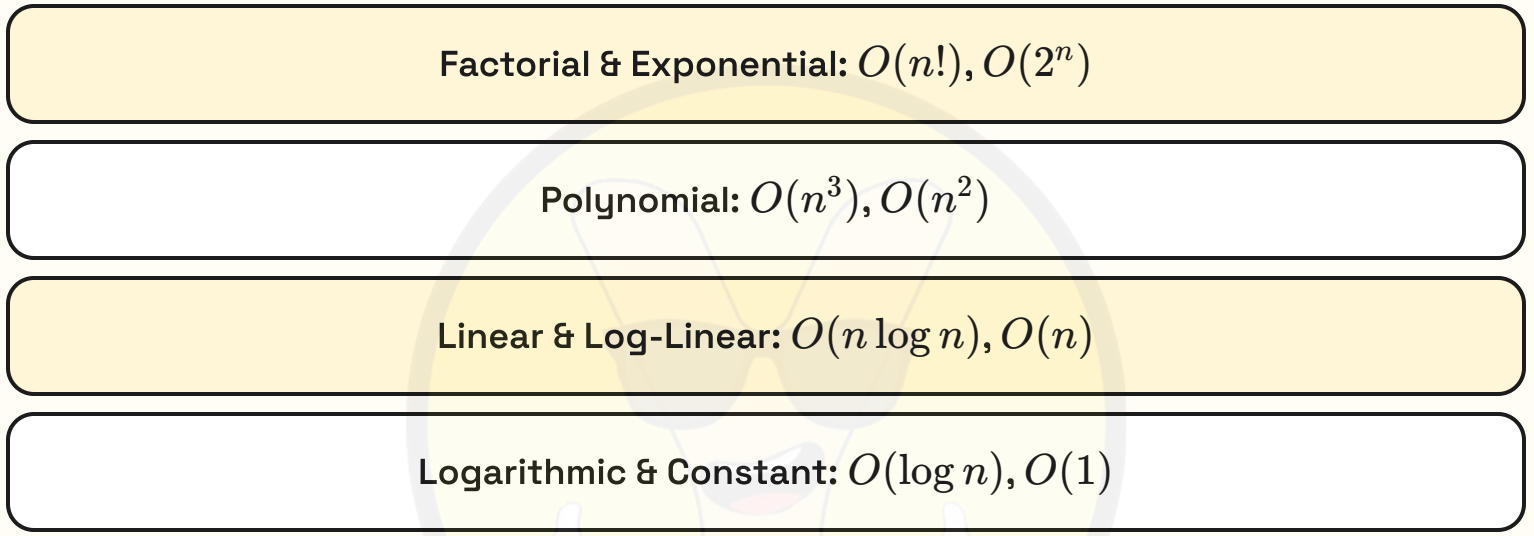
Growth of Functions: Asymptotic Notations

ASYMPTOTIC NOTATION

Mathematical tools used to describe the limiting behavior and growth rate of an algorithm's running time function as input size n approaches infinity.

- Big-O Notation $O(g(n))$: Asymptotic upper bound. $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
- Omega Notation $\Omega(g(n))$: Asymptotic lower bound. $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.
- Theta Notation $\Theta(g(n))$: Asymptotically tight bound. $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
- Little-o Notation $o(g(n))$: Strict upper bound where $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.
- Little-omega Notation $\omega(g(n))$: Strict lower bound where $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.

Asymptotic Growth Rate Hierarchy



Order of complexity growth from highest consumption (top) to lowest consumption (bottom).

Performance Measurements

Evaluating algorithm efficiency involves both theoretical bounding and practical experimental testing.

- Theoretical Analysis: Machine-independent evaluation using mathematical recurrence relations and asymptotic bounds.
- Empirical Analysis: Real-world profiling measuring actual wall-clock execution time on hardware.
- Software Profiling: Uses tools to track CPU cycle counts, cache misses, and heap allocation.
- Benchmark Datasets: Execution testing using structured best-case, worst-case, and uniform random inputs.
- Hardware Dependencies: Empirical results vary based on CPU architecture, clock speed, compiler optimizations, and OS.

Shell Sort: Diminishing Increment Sort

SHELL SORT

An in-place comparison sort that generalizes insertion sort by exchanging elements located multiple positions apart using a decreasing gap sequence.

- Gap Sequence: Uses a sequence of decreasing gaps h (e.g., Shell's original sequence $\lfloor n/2^k \rfloor$).
- h-Sorted Array: Array is divided into sub-lists containing elements spaced h positions apart.
- Final Pass: Ends with gap $h = 1$, which becomes a standard insertion sort operating on a nearly sorted array.
- Time Complexity: Varies between $O(n^{1.3})$ and $O(n^2)$ depending on the chosen gap sequence.
- Space Complexity: In-place algorithm with auxiliary space complexity of $O(1)$.

Shell Sort Worked Example

Sort array $A = [35, 33, 42, 10, 14, 19, 27, 44]$ of size $n = 8$ using gap sequence $h = \lfloor n/2 \rfloor$.

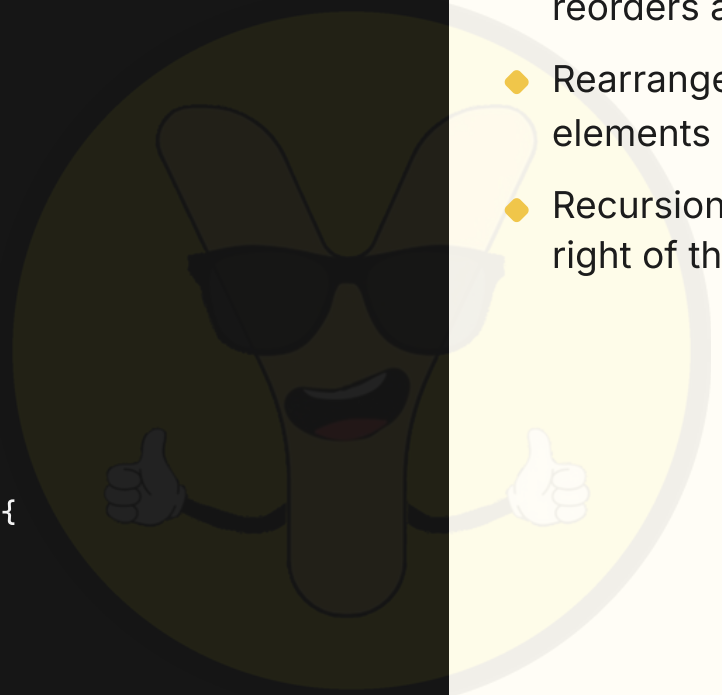
1. Initial state: [35, 33, 42, 10, 14, 19, 27, 44], Gap h = 4.
2. Compare index pairs at distance 4: (35,14), (33,19), (42,27), (10,44). Swap out-of-order pairs.
3. Array after Gap 4 pass: [14, 19, 27, 10, 35, 33, 42, 44].
4. Reduce Gap to h = 2. Sub-lists at distance 2: [14, 27, 35, 42] and [19, 10, 33, 44]. Sort each sub-list.
5. Array after Gap 2 pass: [14, 10, 27, 19, 35, 33, 42, 44].
6. Reduce Gap to h = 1. Perform final Insertion Sort pass on the nearly sorted array.

Final Sorted Array: $[10, 14, 19, 27, 33, 35, 42, 44]$.

Quick Sort Algorithm & Partitioning

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] ⩽ pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```



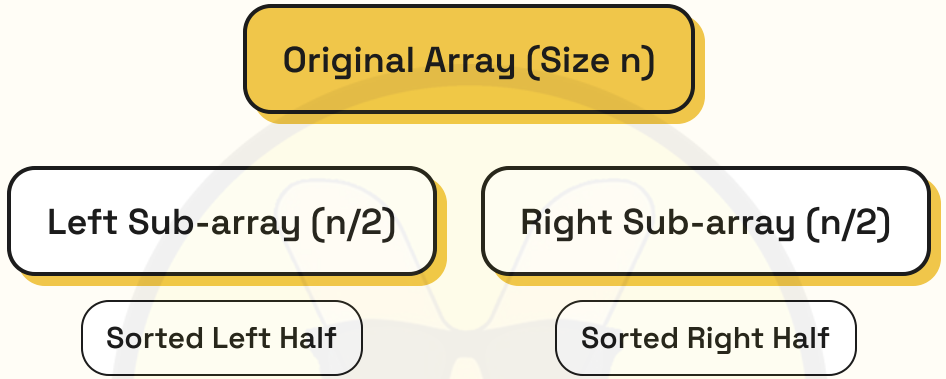
- Lomuto Partitioning: Selects last element as pivot and reorders array.
- Rearrangement: Elements $\leq$ pivot move left; elements $>$ pivot move right.
- Recursion: Recursively sorts partitions to the left and right of the pivot index.

Quick Sort Performance Analysis

Quick Sort efficiency depends on partition balance determined by pivot selection.

- Best-Case Recurrence: $T(n) = 2T(n/2) + \Theta(n) \implies \Theta(n \log n)$ when pivot splits array evenly.
- Worst-Case Recurrence: $T(n) = T(n-1) + \Theta(n) \implies \Theta(n^2)$ when array is already sorted.
- Average-Case Time: $O(n \log n)$ when assuming random pivot choices.
- Space Complexity: $O(\log n)$ auxiliary stack space for recursion (worst-case $O(n)$).
- Randomized Quick Sort: Randomly selecting a pivot prevents systemic worst-case behavior.
- Stability: Non-stable sort due to long-distance element swaps during partitioning.

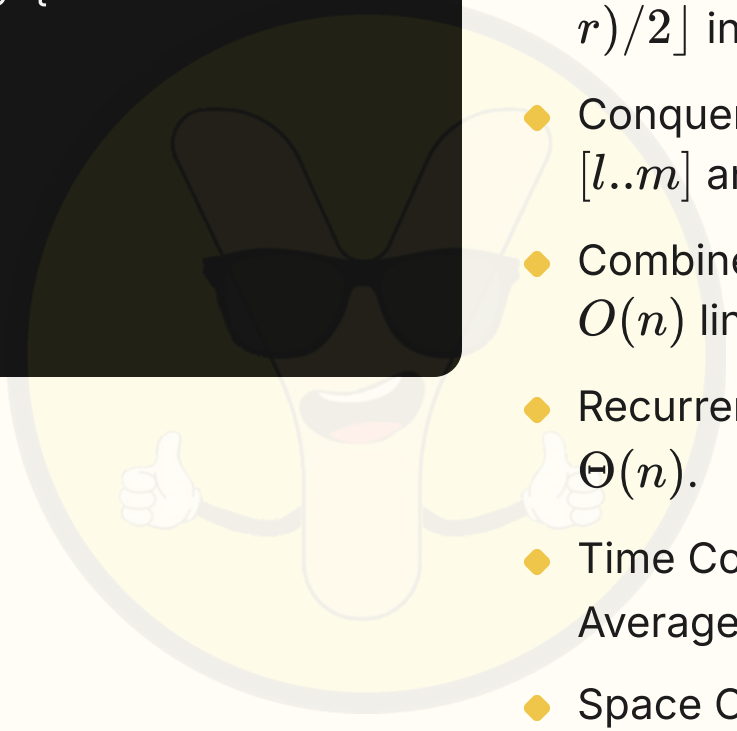
Divide-and-Conquer Framework (Merge Sort)



Merge Sort workflow: Divide array into halves, recursively sort halves, and merge sorted results.

Merge Sort Algorithm & Analysis

```
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```



- Divide: Calculates middle index $m = \lfloor (l + r)/2 \rfloor$ in $O(1)$ time.
- Conquer: Recursively sorts left sub-array $[l..m]$ and right sub-array $[m+1..r]$.
- Combine: Merges two sorted sub-arrays in $O(n)$ linear time.
- Recurrence Relation: $T(n) = 2T(n/2) + \Theta(n)$.
- Time Complexity: $\Theta(n \log n)$ in Best, Average, and Worst cases.
- Space Complexity: $O(n)$ auxiliary memory needed for temporary merging arrays.

Heap Sort: Binary Heap Concepts

MAX-HEAP

A complete binary tree stored in an array where every parent node has a key value greater than or equal to its children.

- Array Representation: For element at index i, parent is at $\lfloor (i-1)/2 \rfloor$, left child at $2i+1$, right child at $2i+2$.
- Heapify Operation: Restores the Max-Heap property downwards in $O(\log n)$ time.
- Build Max-Heap: Converts an unsorted array of size n into a Max-Heap in $O(n)$ time.
- In-Place Sorting: Operates within array storage without auxiliary linked nodes.

Heap Sort Algorithm Procedure

Heap Sort uses a max-heap structure to extract elements in descending order.

- Build Heap: Construct a Max-Heap from the unsorted input array of size n.
- Swap Root: Exchange root element (maximum value) with the last element in the heap.
- Reduce Size: Decrement active heap size by 1 to isolate the maximum element at the array end.
- Heapify Root: Call Max-Heapify on the root index to restore max-heap property.
- Iterate: Repeat swap and heapify steps until heap size shrinks to 1.
- Performance: Time complexity is $O(n \log n)$ across all cases with $O(1)$ auxiliary space.

Comparison of Major Sorting Algorithms

Algorithm	Best Time	Worst Time	Space	Stable
Shell Sort	$O(n \log n)$	$O(n^2)$	$O(1)$	No
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$	Yes
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	Yes
Radix Sort	$O(d(n + k))$	$O(d(n + k))$	$O(n + k)$	Yes
Bucket Sort	$O(n + k)$	$O(n^2)$	$O(n)$	Yes

Sorting in Linear Time: Lower Bound Theory

Comparison-based sorting algorithms have a fundamental lower bound on running time.

- Decision Tree Model: Models comparison decisions as a binary decision tree for input size n.
- Leaf Node Count: Every unique permutation of n elements must appear as a leaf; total leaves $L \geq n!$.
- Tree Height h: Represents the worst-case number of comparisons executed.
- Height Inequality: A binary tree of height h has at most 2^h leaves, so $2^h \geq n!$.
- Logarithmic Bound: Taking logs yields $h \geq \log_2(n!)$.
- Stirling's Approximation: $\log_2(n!) = \Omega(n \log n)$. Any comparison sort requires at least $\Omega(n \log n)$ operations.

Counting Sort Worked Example

Sort array $A = [4, 2, 2, 8, 3, 3, 1]$ with maximum element key $k = 8$.

1. Initialize frequency array C of size k+1 (index 0 to 8) with all zeros.
2. Count frequencies of elements in A: C[1]=1, C[2]=2, C[3]=2, C[4]=1, C[8]=1.
3. Compute cumulative prefix sums in C: C[1]=1, C[2]=3, C[3]=5, C[4]=6, C[8]=7.
4. Iterate input array A from right to left, place A[i] into output array B at index C[A[i]]-1, then decrement C[A[i]].
5. Placing elements yields sorted array B.

Output Array: $[1, 2, 2, 3, 3, 4, 8]$. **Time Complexity:** $O(n + k)$.

Non-Comparison Sorts: Radix and Bucket Sort

Non-comparison sorting techniques achieve linear time by placing keys into numeric buckets.

- Radix Sort Principle: Sorts keys digit by digit starting from Least Significant Digit (LSD) to Most Significant Digit (MSD).
- Subroutine Requirement: Requires a stable sorting algorithm (typically Counting Sort) for each digit pass.
- Radix Time Complexity: $O(d \cdot (n + k))$ where d is digit count and k is numeric base.
- Bucket Sort Principle: Distributes elements into k sub-buckets based on value range, sorts each bucket individually.
- Bucket Performance: Average time is $O(n + k)$; worst-case is $O(n^2)$ if items cluster into one bucket.
- Space Tradeoff: Achieves linear time performance by increasing memory consumption.

AKTU Unit 1 Exam Practice Questions

Q1 Define Big-O, Omega, and Theta notations. Formally prove that $3n^2 + 5n + 7 = \Theta(n^2)$.

Q2 Write the Quick Sort algorithm. Trace it for array $[38, 27, 43, 3, 9, 82, 10]$ and derive its worst-case time complexity.

Q3 Explain Merge Sort using Divide-and-Conquer. Solve the recurrence relation $T(n) = 2T(n/2) + n$ using the Master Method.

Q4 Describe Heap Sort. Build a Max-Heap step-by-step for $[4, 10, 3, 5, 1]$ and perform one delete-max operation.

Q5 Prove that any comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparisons in the worst case.

Q6 Explain Counting Sort algorithm. Trace it for array $[2, 5, 3, 0, 2, 3, 0, 3]$ and state its time and space complexity.

Q7 Compare Quick Sort, Merge Sort, Heap Sort, and Shell Sort in terms of time complexity, space complexity, and stability.

Unit 1 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 2 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

