Chapter Summary: Algorithm Design and Performance

Recursion

- A function that calls itself, directly or indirectly.
- Useful for solving problems where iterative solutions are not intuitive.
- Comparison of recursive and iterative approaches to highlight when each is preferable.

Searching and Sorting

- Aim is to choose the most efficient algorithm in terms of speed and memory.
- Although all correct algorithms yield the same result, their performance can vary significantly.
- In large-scale or big data applications, algorithms that support parallel processing are preferred.

Algorithm Performance

- Introduction to Big O notation for classifying algorithm efficiency.
- Simple, obvious algorithms can be inefficient.
- Better-designed algorithms can provide significant performance improvements.

Visualization (Optional)

- Example: Animated selection sort used to demonstrate algorithm behavior.
- Visualizations help in understanding and improving algorithm design.

Recursive Problem-Solving

Recursive problem-solving approaches have several elements in common.

- A recursive function can directly solve only the simplest case(s), known as **base cases**.
- If called with a base case, the function immediately returns a result.
- For more complex inputs, the function divides the problem into two parts:
  - One part it can solve directly.
  - Another part that is a **simpler version** of the original problem.

To handle the unsolved part:

- The function **calls itself** with the smaller problem. This is known as the **recursion step**.
- This approach is an example of the **divide-and-conquer** strategy.

How it Works:

- The recursion step occurs **while the original function call is still active**.
- Each recursive call may result in further recursive calls, continuing to divide the problem.
- The process continues until the problem size **reduces to the base case**.
- Once the base case is reached, results are returned **back through the call stack**.
- Eventually, the original function call receives the final result.

Indirect Recursion

- Occurs when a function calls another function, which then calls the original function.
- Example: Function A calls Function B, which then calls Function A.
- It is considered recursion because the second call to Function A is made while the first call is still active (has not finished executing).

Stack Overflow and Infinite Recursion

- Computer memory is finite; only a limited number of activation records can be stored on the function-call stack.
- If too many recursive calls are made, a **stack overflow** error occurs.
- Stack overflow is typically caused by **infinite recursion**.
- Infinite recursion happens when:
  - The base case is missing, or
  - The recursion step is written incorrectly and does not move toward the base case.
- This is similar to an **infinite loop** in an iterative solution.

Recursion and the Function-Call Stack

- Each recursive function call gets its own **stack frame** on the function-call stack.
- When a call completes, its stack frame is **popped** from the stack.
- Control returns to the caller, which may be another instance of the same function.
- Recursive function calls use the same stack mechanism as regular function calls.

IMP: To make recursion feasible, the recursion step in a recursive solution must resemble the original problem, but be a slightly smaller or simpler version of it.

Fibonacci Series

- The Fibonacci series starts with **0 and 1**.

- Each subsequent number is the **sum of the previous two**: `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`

- The series occurs **in nature** and describes a **spiral pattern**.

- The **ratio** of successive Fibonacci numbers converges to **1.618...**, known as the **golden ratio** or **golden mean**.

- The golden ratio is considered **aesthetically pleasing**.

- It is used in **architecture and design**, such as in windows, rooms, buildings, and postcards.

Recursive Definition

- `fibonacci(0) = 0`
- `fibonacci(1) = 1`

- `fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)`

Base Cases

- `fibonacci(0)` is **0**
- `fibonacci(1)` is **1**

Recursive Fibonacci: A Word of Caution

- Recursive Fibonacci functions that don't hit the base cases (`fibonacci(0)` or `fibonacci(1)`) make **two more recursive calls**.
- This leads to a rapid explosion in the number of calls.

Examples of Call Explosion:

- `fibonacci(20)` → **21,891 calls**

- `fibonacci(30)` → **2,692,537 calls**

- `fibonacci(31)` → **4,356,617 calls**

- `fibonacci(32)` → **7,049,155 calls**

- The number of calls increases **exponentially**:

  - From `30 → 31`: **+1,664,080 calls**
  - From `31 → 32`: **+2,692,538 calls**
  - The increase from `31 → 32` is **1.5× more** than from `30 → 31`

Key Points:

- Recursive Fibonacci grows exponentially in time and number of calls.
- Such growth can overload even the **most powerful computers**.
- This behavior falls under **complexity theory**, which analyzes how many operations algorithms perform.
- Recursive Fibonacci is an example of **exponential time complexity**.
- In practice, **avoid naive recursive Fibonacci implementations** due to inefficient performance.

IMP: Iteration and recursion can occur infinitely.

Recursion vs. Iteration

Key Comparisons:

- **Control Structure**:

  - **Iteration** uses `for` or `while` loops.
  - **Recursion** uses conditional statements like `if`, `if…else`, etc.

- **Iteration Mechanism**:

  - Iteration explicitly loops.

  - Recursion loops through repeated **function calls**.

- **Termination**:

  - **Iteration** stops when the **loop condition** fails.
  - **Recursion** stops when it reaches a **base case**.

- **Approach to Termination**:

  - Iteration modifies a **counter** until the condition fails.
  - Recursion calls itself with **smaller problems** until the base case is reached.

- **Risk of Infinite Execution**:

  - **Infinite Loop**: Loop condition never becomes false.
  - **Infinite Recursion**: Fails to reduce the problem or **missing base case**.

Negatives of Recursion:

- **Overhead of Function Calls**:

  - Each call adds a new **stack frame**, consuming memory.
  - More **processor time** and **memory space** than iteration.

- **Memory Usage**:

  - Each call stores function variables → higher **stack memory** consumption.

- **Iteration is More Efficient**:

  - Avoids repeated function calls.
  - Saves **CPU and memory**.

Final Note:

- Use **recursion** when it provides a **clearer, simpler solution**.
- Use **iteration** for **efficiency** in time and memory when possible.

Searching Algorithms & Big O Notation

What is Searching?

Searching algorithms are used to find an element (or elements) matching a search key in a data structure.

Big O Notation

Big O notation describes the worst-case time complexity of an algorithm in terms of input size `n`.

O(1) – Constant Time

- Time does not depend on the size of the array.
- Example: Comparing the first two elements in an array.
- Number of operations remains constant regardless of input size.

O(n) – Linear Time

- Time increases linearly with the input size.
- Example: Linear search – check every element until a match is found.
- Worst case: The element is at the end or not present.

$O(n^2)$ – Quadratic Time

- Time grows proportionally to the square of the input size.
- Example: Checking for duplicate elements using nested loops.
- Total comparisons: $(n - 1) + (n - 2) + ... + 1 = n^2/2 - n/2$
- Examples:
  - 4 elements → 16 comparisons
  - 8 elements → 64 comparisons
  - 100,000 elements → runs for several minutes
  - 1 billion elements → approximately 13.3 years

Linear Search – Big O Analysis

- Time Complexity: O(n)
- Best case: Element is at the beginning.
- Worst case: Element is at the end or not present.
- Linear search is simple to implement but inefficient for large arrays.
- A better alternative for sorted data is binary search.

Summary Table

Big O	Name	Example Use Case
O(1)	Constant time	First-element comparison
O(n)	Linear time	Linear search
$O(n^2)$	Quadratic time	Duplicate checking

Note: As input size increases, the performance impact of higher complexity algorithms becomes significant. Prefer optimized algorithms for large datasets.

Binary Search Notes

Overview

- Binary search is more efficient than linear search.
- It requires the array to be **sorted**.
- Time complexity: **O(log n)**.
- It reduces the search space by half with each iteration.

Working of Binary Search

1. Check the middle element of the array.
2. If it matches the search key, return its index.
3. If the key is **less than** the middle element, search in the **left half**.
4. If the key is **greater than** the middle element, search in the **right half**.
5. Repeat the process until the element is found or the subarray size becomes zero.

Example

Given a sorted array: [2, 3, 5, 10, 27, 30, 34, 51, 56, 65, 77, 81, 82, 93, 99]

Search key: `65`

Steps:

1. Check middle element: `51` → 65 > 51 → search right half.
2. New subarray: `[56, 65, 77, 81, 82, 93, 99]`
3. Middle element: `81` → 65 < 81 → search left half.
4. New subarray: `[56, 65, 77]`
5. Middle element: `65` → Match found.

Total comparisons made: **3**

If linear search were used, it would have taken **10 comparisons** in the worst case.

Note

- For arrays with an even number of elements, binary search typically chooses the **higher of the two middle elements** when calculating the midpoint.

Minimum Comparisons in Binary Search

With binary search, the **smallest number of comparisons** needed to find a matching element in a **1,000,001-element** array is:

Answer: One.

This occurs if, on the **first comparison**, the search key matches the **middle element** of the array.

Binary Search – Key Points

- **Binary search** efficiently finds an element in a **sorted array**.
- It works by repeatedly dividing the array in half and comparing the middle element to the key.
- In the **worst case**, it takes **$\log_2(n)$** comparisons, where `n` is the number of elements.

Examples:

- Array of **1023 elements (2^{10} - 1)** → max **10 comparisons**
- Array of **1,048,575 (2^{20} - 1)** → max **20 comparisons**
- Array of **1 billion elements (≈ 2^{30})** → max **30 comparisons**

Big O Time Complexity:

- **Binary Search:** `O(log n)` – *logarithmic time*
- **Linear Search:** `O(n)` – *linear time*

Key Advantage:

- Binary search offers a **huge performance gain** over linear search for large datasets.

- Binary search only works if the array is **already sorted**.

Sorting – Key Concepts

- **Sorting** means arranging data in **ascending or descending order**.
- It's a core task in computing and used across **almost every organization**.
- Sorting is **algorithm-dependent**, but the final result is the **same sorted array**.
- Choice of algorithm affects **speed and memory usage**, not the outcome.

Common Sorting Algorithms

1. **Selection Sort**

    - Simple to implement
    - Inefficient for large datasets
    - Time Complexity: `O(n²)`

2. **Insertion Sort**

    - Easy to code
    - Efficient for **small or nearly sorted** datasets
    - Time Complexity: `O(n²)`

3. **Merge Sort**

    - More complex to implement
    - **Much faster** than Selection and Insertion Sort
    - Time Complexity: `O(n log n)`
    - Uses **Divide and Conquer** approach

- Note: These algorithms are demonstrated using arrays of primitive types like `int`.

Selection Sort – Key Points

- A **simple but inefficient** sorting algorithm.
- Works by **repeatedly selecting the smallest element** and swapping it to its correct position.

How It Works (for ascending order):

1. Find the **smallest element** in the array.
2. Swap it with the **first element**.
3. Find the **second-smallest** in the remaining array.
4. Swap it with the **second element**.
5. Repeat until the array is sorted.

- After the `i-th` iteration, the **first `i` elements are sorted**.

Time Complexity

- **Worst/Average/Best Case:** `O(n²)`
- **Space Complexity:** `O(1)` (in-place sorting)
- Not suitable for large datasets.

Selection Sort – Time Complexity

- Uses two nested loops:
  - Outer loop: selects the position to fill.
  - Inner loop: finds the smallest element in the unsorted part.
- Iteration pattern: $(n - 1) + (n - 2) + ... + 1 = n(n - 1)/2$
- Time Complexity: **$O(n^2)$** (same for best, average, and worst case).
- Doesn't depend on initial order of the array.

Insertion Sort – Overview

- Simple but inefficient sorting algorithm.
- At each iteration, takes the next element and inserts it into its correct position in the sorted portion of the array.
- After the *i-th* iteration, the first *i* elements are sorted.

Insertion Sort – Time Complexity

- Runs in **$O(n^2)$** time.
- Contains two nested loops:
  - Outer `for` loop runs **$n – 1$** times.
  - Inner `while` loop compares and shifts elements to insert in correct position.
- In the worst case, both loops run **$O(n)$** times → overall time complexity is **$O(n^2)$**.
- Performance does **not** improve on already sorted or partially sorted data.

Merge Sort – Overview

- **Efficient but more complex** than selection or insertion sort.
- Uses **divide-and-conquer**:

  - Splits the array into two halves.
  - Recursively sorts each half.
  - Merges the sorted halves.
- **Base case**: A single-element array (already sorted).
- **Recursive step**: Split → Sort → Merge.
- Handles arrays with odd lengths by allowing one subarray to have one extra element.

Merge Sort – Time Complexity

- **Much faster** than selection or insertion sort.
- Each recursive call:
  - Splits the array into halves.
  - Merges using at most **O(n)** comparisons.
- Recursion continues until one-element subarrays.
- Each level of recursion takes O(n), and there are **$\log_2 n$ levels**.
- **Total time complexity:** `O(n log n)`
- Scaling pattern:
  - Doubling array size → +1 recursion level.
  - Quadrupling → +2 levels.
- Merge Sort is efficient and ideal for large datasets.