Types of Algorithms

In this topic, we study different **ways (strategies)** to solve problems using algorithms.
Each type of algorithm follows a **different thinking approach**.

In these notes, we will discuss:

1. Brute Force Algorithm

Definition

A **Brute Force Algorithm** is a simple approach that **tries all possible solutions** to find the correct answer.

👉 It does **not use clever tricks** or optimizations.
👉 It checks **every possibility one by one**.

Basic Intuition

- "Check everything until you get the answer."
- Works on the idea: **Nothing is skipped**.

Real-Life Analogy

🔐 **Opening a Lock Without Knowing the Code**

Suppose a lock has a 3-digit code (000 to 999), and you don't know the code.

What will you do?

- Try **000**
- Then **001**
- Then **002**
- …
- Keep trying **all combinations** until the lock opens.

✔ This is **Brute Force** — slow but guaranteed to work.

Simple Example (Searching a Number)

Problem:
Find whether number `7` exists in this list:
`[2, 4, 1, 9, 7, 5]`

Brute Force Approach:

- Compare 7 with 2 ❌
- Compare 7 with 4 ❌
- Compare 7 with 1 ❌
- Compare 7 with 9 ❌

- Compare 7 with 7 ✅ (Found!)

👉 We check **each element one by one**.

Algorithm Steps

- Start from the first element.
- Check if it matches the required value.
- If not, move to the next element.
- Continue until:
    - The element is found, or
    - The list ends.

Key Characteristics

- ✅ Very easy to understand
- ✅ Simple to implement
- ❌ Takes **more time** for large inputs
- ❌ Not efficient for big problems

Time Complexity (Basic Idea)

- Usually **high**
- For searching n elements → **O(n)**
- For some problems (like password cracking) → **$O(2^n)$** or worse

When to Use Brute Force?

✔ When:

- Input size is **small**
- You want a **simple solution**
- Efficiency is not very important
- You want a **guaranteed correct answer**

2. Randomized Algorithm

Definition

A **Randomized Algorithm** uses **random choices** while solving a problem.

👉 The algorithm's behavior can **change each time** it runs, even on the same input.

Basic Intuition

- Instead of checking everything in order,
- The algorithm **randomly picks a path or choice**.

Real-Life Analogy

🎯 Finding a Name in a Crowd

Imagine you are searching for your friend in a large crowd.

Two ways:

- **Brute Force:** Check every person one by one.
- **Randomized:** Randomly look in different directions hoping to spot your friend faster.

Sometimes you get lucky and find them quickly 😄

Simple Example (Guessing a Number)

Problem:
Guess a secret number between **1 and 10**.

Randomized Approach:

- First guess: 3 ❌
- Second guess: 8 ❌
- Third guess: 6 ✅ (Found!)

👉 The guesses are **random**, not in order.

Why Use Randomness?

- Avoids **worst-case situations**
- Often **faster on average**
- Useful when:
    - Input is very large
    - Deterministic (fixed) methods are slow

Types of Randomized Algorithms

1. Las Vegas Algorithm

- Always gives the **correct answer**
- Time taken may vary

Example:
Randomized Quick Sort
✔ Correct output
⏱ Time may differ each run

2. Monte Carlo Algorithm

- Gives the answer **quickly**
- Answer **may be incorrect with small probability**

Example:
Primality testing

✔ Very fast
❌ Small chance of wrong result

Key Characteristics

- ✅ Faster on average
- ✅ Useful for large problems
- ❌ Output or time may vary
- ❌ Slight chance of incorrect result (in some cases)

Time Complexity (Basic Idea)

- Depends on probability
- Usually analyzed as **expected time**
- Often better than brute force

When to Use Randomized Algorithms?

✔ When:

- Deterministic algorithms are too slow
- Input size is very large
- A small chance of error is acceptable
- Average performance matters more than worst case

3. Sorting Algorithm

Definition

A **Sorting Algorithm** is an algorithm that **arranges data in a particular order**.

The order can be:

- **Ascending order** (small to large)
- **Descending order** (large to small)

Why Do We Need Sorting?

Sorting makes data:

- Easier to **search**
- Easier to **understand**
- Faster to **process**

Real-Life Analogy

📚 **Arranging Books on a Shelf**

- If books are arranged randomly → difficult to find one.
- If books are arranged **alphabetically** → very easy to find a book.

✔ This arrangement is **sorting**.

Simple Example

Unsorted list:

```
45, 12, 9, 33, 5
```

Sorted list (Ascending):

```
5, 9, 12, 33, 45
```

Basic Idea of Sorting

- Compare two elements
- Decide their correct order
- Swap if necessary
- Repeat until all elements are in order

Common Sorting Algorithms (Beginner View)

1. Bubble Sort

- Compares **adjacent elements**
- Repeatedly swaps them if they are in the wrong order

Analogy:
🫧 Bubbles rise to the top in water
Larger elements slowly move to the end

✔ Very easy
❌ Very slow

2. Selection Sort

- Selects the **smallest element**
- Places it at the correct position

Analogy:
🏆 Selecting the smallest number first and placing it correctly

3. Insertion Sort

- Picks one element at a time
- Inserts it into its correct position

Analogy:
🃏 Arranging playing cards in your hand

Key Characteristics

- Sorting improves efficiency of searching
- Different algorithms have different speeds
- Choice depends on data size

Time Complexity (Basic Idea)

Algorithm	Time Complexity
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$

When to Use Sorting Algorithms?

✔ When:

- Data needs to be organized
- Searching is required frequently
- Output must be in order

4. Searching Algorithm

Definition

A **Searching Algorithm** is used to **find a specific element** in a list or dataset.

Real-Life Analogy

📞 **Searching a Contact in Mobile Phone**

- Scroll one by one → Linear Search
- Type name directly (sorted list) → Binary Search

Simple Example

List:

```
[3, 8, 12, 20, 25, 40]
```

Find element: `20`

Types of Searching Algorithms

1. Linear Search

Definition

A **Linear Search** checks **each element one by one** until the required element is found.

How It Works

- Start from the first element
- Compare with the target
- Move to the next element
- Stop when found or list ends

Example

Search `12` in:

`[3, 8, 12, 20]`

- Check 3 ❌
- Check 8 ❌
- Check 12 ✅ Found

Key Points

- Works on **sorted and unsorted** lists
- Very simple
- Slow for large data

Time Complexity

- Best Case: O(1)
- Worst Case: O(n)

2. Binary Search

Definition

A **Binary Search** works only on **sorted data** and repeatedly divides the list into two halves.

Real-Life Analogy

📖 **Finding a Word in a Dictionary**

- You don’t start from page 1
- You open the middle
- Decide left or right
- Repeat until found

How It Works

- Find the middle element
- Compare it with the target
- If target is smaller → search left half
- If target is larger → search right half
- Repeat

Example

Search `20` in:

`[3, 8, 12, 20, 25, 40]`

- Middle = 12 → go right
- Middle = 25 → go left
- Middle = 20 → Found

Key Points

- Very fast
- Requires sorted data
- Slightly complex

Time Complexity

- Best Case: O(1)
- Worst Case: O(log n)

Sorting vs Searching

Feature	Sorting	Searching
Purpose	Arrange data	Find data
Output	Ordered list	Found / Not Found
Example	Bubble Sort	Binary Search
Requirement	No	Binary search needs sorted data

5. Recursive Algorithm

Definition

A **Recursive Algorithm** is an algorithm that **calls itself** to solve a problem.

👉 The problem is divided into **smaller subproblems of the same type**.

Basic Intuition

- Solve a big problem by first solving **smaller versions** of the same problem.
- Stop when the problem becomes **simple enough** (called the **base case**).

Real-Life Analogy

🪞 **Mirror Reflection**

A mirror facing another mirror shows **smaller and smaller reflections**, stopping when the view ends.

Simple Example (Factorial)

`n! = n × (n−1)!`

Base case:

`1! = 1`

To find `4!`:

```
4! = 4 × 3!
3! = 3 × 2!
2! = 2 × 1!
1! = 1
```

Key Components

- **Base Case** – stops recursion
- **Recursive Case** – calls itself

Advantages

- Simple and clean logic
- Easy to understand for divide-type problems

Disadvantages

- Uses more memory
- Can be slow if not designed carefully

6. Divide and Conquer Algorithm

Definition

A **Divide and Conquer Algorithm** solves a problem in **three steps**:

- **Divide** the problem into smaller subproblems
- **Conquer** by solving each subproblem
- **Combine** the results

👉 It is usually implemented using **recursion**.

Real-Life Analogy

🍕 **Cutting a Pizza**

- Divide pizza into slices
- Eat each slice
- Combine satisfaction 😄

Simple Example (Merge Sort Idea)

List:

```
[38, 27, 43, 3, 9]
```

Steps:

- Divide into smaller lists
- Sort each list
- Merge them back in order

Common Divide and Conquer Algorithms

- Merge Sort
- Quick Sort
- Binary Search

Why Use Divide and Conquer?

- Reduces problem size
- Faster for large inputs
- Efficient time complexity

Time Complexity (General Idea)

Often **O(n log n)**

7. Dynamic Programming Algorithm

Definition

A **Dynamic Programming (DP) Algorithm** solves a problem by:

- Breaking it into **overlapping subproblems**
- Solving each subproblem **only once**
- **Storing the result** for future use

Basic Intuition

👉 "**Don't solve the same problem again and again**."

Real-Life Analogy

📒 **Remembering Exam Answers**

- If you remember an answer, you don't need to think again.
- You simply **reuse it**.

Simple Example (Fibonacci Series)

Recursive approach:

```
fib(5) = fib(4) + fib(3)
```

Here `fib(3)` is calculated multiple times ❌

DP approach:

- Store results in a table
- Use stored values ✔

Two DP Approaches

- **Top-Down (Memoization)** – recursion + memory

- **Bottom-Up (Tabulation)** – table first, then build

Key Characteristics

- Efficient
- Uses extra memory
- Best for optimization problems

Common DP Problems

- Knapsack Problem
- Fibonacci
- Shortest Path

8. Greedy Algorithm

Definition

A **Greedy Algorithm** makes the **best choice at each step**, hoping it will lead to the best final solution.

👉 It does **not look back**.

Basic Intuition

"Take the **best option available right now**."

Real-Life Analogy

💰 **Giving Change**

To give ₹87:

- Take ₹50
- Then ₹20
- Then ₹10
- Then ₹5
- Then ₹2

Each step takes the **largest possible value**.

Simple Example (Activity Selection)

Choose activities with:

- **Earliest finish time**
- So that maximum activities can be done

Key Characteristics

- Simple
- Fast

- May not always give optimal solution

Common Greedy Algorithms

- Kruskal's Algorithm
- Prim's Algorithm
- Huffman Coding

9. Backtracking Algorithm

Definition

A **Backtracking Algorithm** builds a solution **step by step** and **removes (backtracks)** a step if it leads to a wrong solution.

Basic Intuition

👉 "Try → Check → Undo if wrong → Try again"

Real-Life Analogy

🧩 **Solving a Maze**

- Take a path
- If dead end ❌ → go back
- Try another path ✔

Simple Example (N-Queens Problem)

Place queens on a chessboard:

- One queen per row
- No two queens attack each other
- If conflict occurs → remove queen and try next position

Key Characteristics

- Guarantees correct solution
- Explores all possibilities
- Can be slow for large problems

Common Backtracking Problems

- N-Queens
- Sudoku
- Subset generation

Summary: Types of Algorithms

Algorithms are different **ways or strategies** to solve problems.
Each type follows a **different thinking approach**.

1. Brute Force Algorithm

- **Idea:** Try **all possible solutions**
- **Thinking:** “Check everything”
- **Key Point:** Simple but slow
- **Example:** Linear Search
- **Best Used When:** Input size is small

2. Randomized Algorithm

- **Idea:** Uses **random choices**
- **Thinking:** “Let luck help speed up the solution”
- **Key Point:** Faster on average, result/time may vary
- **Example:** Randomized Quick Sort
- **Best Used When:** Large inputs, average performance matters

3. Sorting Algorithm

- **Idea:** Arrange data in order
- **Thinking:** “Make data organized”
- **Key Point:** Improves searching efficiency
- **Examples:** Bubble Sort, Selection Sort, Insertion Sort
- **Best Used When:** Data needs to be ordered

4. Searching Algorithm

- **Idea:** Find an element in a list
- **Thinking:** “Where is the required value?”
- **Key Point:** Faster searching saves time
- **Examples:** Linear Search, Binary Search
- **Best Used When:** Data retrieval is required

5. Recursive Algorithm

- **Idea:** Algorithm calls itself
- **Thinking:** “Solve smaller version of same problem”
- **Key Point:** Needs a base case
- **Example:** Factorial, Fibonacci

- **Best Used When:** Problem can be broken into similar subproblems

6. Divide and Conquer Algorithm

- **Idea:** Divide → Solve → Combine
- **Thinking:** “Break big problem into small problems”
- **Key Point:** Efficient for large inputs
- **Examples:** Merge Sort, Quick Sort, Binary Search
- **Best Used When:** Problem size is large

7. Dynamic Programming Algorithm

- **Idea:** Store and reuse results
- **Thinking:** “Don’t solve the same problem again”
- **Key Point:** Uses extra memory to save time
- **Examples:** Fibonacci, Knapsack
- **Best Used When:** Overlapping subproblems exist

8. Greedy Algorithm

- **Idea:** Best choice at each step
- **Thinking:** “Choose what looks best now”
- **Key Point:** Fast but may not always be optimal
- **Examples:** Prim’s, Kruskal’s, Huffman Coding
- **Best Used When:** Local optimum leads to global optimum

9. Backtracking Algorithm

- **Idea:** Try → Check → Undo
- **Thinking:** “Explore all possibilities carefully”
- **Key Point:** Guarantees solution if exists
- **Examples:** N-Queens, Sudoku
- **Best Used When:** All solutions or correct solution is required

One-Line Memory Trick

- **Brute Force:** Try everything
- **Randomized:** Use randomness
- **Sorting:** Arrange data
- **Searching:** Find data
- **Recursive:** Call yourself

- **Divide & Conquer:** Break and solve
- **Dynamic Programming:** Remember past work
- **Greedy:** Best choice now
- **Backtracking:** Try and undo

Quick Comparison Table

Algorithm Type	Core Strategy	Speed	Memory
Brute Force	All possibilities	Slow	Low
Randomized	Random choice	Fast (avg)	Medium
Sorting	Arrange data	Medium	Low
Searching	Find element	Fast	Low
Recursive	Self-calling	Medium	High
Divide & Conquer	Divide problem	Fast	Medium
Dynamic Programming	Store results	Very Fast	High
Greedy	Best local choice	Fast	Low
Backtracking	Try & undo	Slow	Medium