# Introduction

▼ **TOC**



# Algorithm Design Strategies

**What is Algorithm Design Strategies?**

Algorithm design is the process of creating a step-by-step procedure to solve a problem or accomplish a task. In this article, we will discuss some of the common algorithm design strategies that can be used to create efficient and effective algorithms.

**What are the different types of approaches for ADS?**

1. **Brute force approach**: The brute force approach is the most straightforward algorithm design strategy. It involves trying every possible solution to a problem until the correct one is found. While this approach is simple, it can be very time-consuming and inefficient, especially for large problems.
2. **Divide and conquer:** The divide and conquer approach involves breaking a problem down into smaller sub-problems that can be solved independently. These sub-problems are then combined to solve the original problem. This approach is particularly useful for complex problems that can be broken down into simpler parts.
3. **Greedy algorithms**: Greedy algorithms make the locally optimal choice at each step in the hope of finding a global optimum. This approach can be very effective for some problems, but it does not always guarantee the best solution.
4. **Dynamic programming**: Dynamic programming is a method for solving complex problems by breaking them down into simpler sub-problems and storing the results of each sub-problem to avoid redundant calculations. This approach is particularly useful for problems that have overlapping sub-problems.
5. **Backtracking**: Backtracking is a general algorithm design strategy that involves exploring all possible solutions to a problem by incrementally building candidates

to the solutions, and abandoning a candidate ("backtracking") as soon as it determines that the candidate cannot possibly be completed to a valid solution. This approach is particularly useful for problems that involve searching through a large space of possible solutions.

# Algorithms

**What is an algorithm?**

An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.

algorithm is a step-by-step procedure that defines a set of instructions that must be carried out in a specific order to produce the desired result.

Algorithms are generally developed independently of underlying languages, which means that an algorithm can be implemented in more than one programming language.

**What is the need for algorithms?**

Algorithms are necessary for solving complex problems efficiently and effectively.

They help to automate processes and make them more reliable, faster, and easier to perform.

Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.

They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

**Advantages of algorithms.**

1. It is easy to understand.
2. An algorithm is a step-wise representation of a solution to a given problem.
3. In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

**Disadvantages of algorithms.**

1. Writing an algorithm takes a long time so it is time-consuming.
2. Understanding complex logic through algorithms can be very difficult.
3. Branching and Looping statements are difficult to show in Algorithms.

**What are the Characteristics of an Algorithm?**

- Clear and Unambiguous.
- Well-Defined Inputs
- Well-Defined Outputs
- Finite-ness
- Feasible
- Language Independent
- Input.
- Output
- Definiteness
- Finiteness
- Effectiveness

# Differentiate

**Differentiate between Recursive v. Non-recursive algorithms:**

| Recursive | Non- recursive |
|---|---|
| 1. Solves a problem by breaking it down into smaller subproblems that are similar to the original problem | 1. Solves a problem by repeatedly executing a set of instructions in a loop until a solution is reached |
| 2. A recursive sorting algorithm calls on itself to sort a smaller part of the array, then combining the partially sorted results | 2. A non-recursive algorithm does the sorting all at once, without calling itself |
| 3. Recursive algorithms involve a function that calls itself | 3. Non-recursive algorithms do not call themselves |

| Recursive | Non- recursive |
|---|---|
| 4. The recursive algorithm solves each sub-problem recursively until a base case is reached | 4. Non-recursive algorithm does not solve each sub-problem recursively until a base case is reached |
| 5. Recursive algorithm has a large amount of overhead as compared to non-recursive Algorithm | 5. Non-recursive algorithm has less amount of overhead as compared to recursive Algorithm |
| 6. Infinite repetition in recursion can lead to CPU crash | 6. In non-recursive algorithm, it will stop when memory is exhausted |
| 7. Finding the time complexity of recursive is more difficult than that of non-recursive | 7. Finding the time complexity of non-recursive is easier than that of recursive |
| 8. Memory usage is more in recursive algorithm as stack is used to store the current function state | 8. Memory usage is less in non-recursive algorithm as it doesn't use stack |
| 9. Recursive algorithms are comparatively slower | 9. Non-recursive algorithms are comparatively faster |
| 10. Example: quick-sort | 10. Example: bubble-sort |

**Differentiate between Comparison v. Non-comparison based sorting:**

| Comparison | Non-comparison |
|---|---|
| Slower compared to non-comparison based sorting | Faster compared to comparison based sorting |
| A comparator is required to sort elements | A comparator is not required to sort the elements |
| Best case for memory complexity O(1) | Best case for memory complexity O(n) |
| Comparison-based sorting algorithms are algorithms that sort a collection of elements by comparing each element with others in the collection | Non-comparison-based sorting algorithms, on the other hand, do not use comparisons to sort elements |
| Comparison-based algorithms have a time complexity of O(n*log(n)) | Non-comparison-based algorithms have a time complexity of O(n) |
| Comparison-based algorithms are more versatile and can be used to sort any collection of elements, regardless of their properties. | Non-comparison-based algorithms are not as versatile and cannot be used to sort any collection of elements |

| Comparison | Non-comparison |
| --- | --- |
| Comparison-based algorithms do not require additional information about the elements being sorted | Non-comparison-based algorithms require some additional information about the elements being sorted |
| Examples of comparison-based sorting algorithms are bubble sort, insertion sort, merge sort, quicksort, and heapsort | Examples of non-comparison-based sorting algorithms are counting sort, radix sort, and bucket sort. |