Algorithms part I

Billy Langlet & Alen Lovric

Introductory Knowledge

Basic

- What is an algorithm?
- Where are algorithms used?

Advanced

- What should you keep in mind when designing algorithms?
- How do you know if an algorithm is effective?

<Think about it>

Introduction

The what, where and how of algorithms

What is an Algorithm?

The name Muḥammad ibn Mūsā **al-Khwārizmī** gave rise to the word

Algorithms in math:

- A step-by-step procedure for solving the problem

Algorithms in programming:

- A set of instructions for a computer to follow to perform a task

What is an Algorithm?

Key characteristics of algorithms include:

- Input
- Output
- Finiteness
- Definiteness (Unambiguity)
- Effectiveness
- Feasibility and language independence (sometimes included in effectiveness)

Algorithms are a fundamental concept in computer science

Why are Algorithms Important?

Algorithms are the backbone of essentially all programming languages and are used for things such as:

- Problem solving
- Efficiency
- Reusability
- Correctness
- Scalability
- Maintainability

Why are Algorithms Important?

Algorithms are the backbone of essentially all programming languages and are used for things such as :

- **Problem solving** – Providing a structured and systematic approach to problem-solving. Breaking down complex task into manageable steps.
- Efficiency
- Reusability
- Correctness
- Scalability
- Maintainability

Why are Algorithms Important?

Algorithms are the backbone of essentially all programming languages and are used for things such as :

- Problem solving
- **Efficiency** – A well-designed algorithm can make a program run faster and consume less resources.
- Reusability
- Correctness
- Scalability
- Maintainability

Why are Algorithms Important?

Algorithms are the backbone of essentially all programming languages and are used for things such as :

- Problem solving
- Efficiency
- **Reusability** – Once an algorithm is developed, it can be reused in different parts of a program or in entirely different programs.
- Correctness
- Scalability
- Maintainability

Why are Algorithms Important?

Algorithms are the backbone of essentially all programming languages and are used for things such as :

- Problem solving
- Efficiency
- Reusability
- **Correctness** – By following a well-defined algorithm, programmers can ensure consistency in their code under various conditions.
- Scalability
- Maintainability

Where are Algorithms Used?

Algorithms are the backbone of essentially all programming languages and are used for things such as :

- Problem solving
- Efficiency
- Reusability
- Correctness
- **Scalability** – Algorithms can handle small and large datasets efficiently, which is essential for building scalable applications and systems.
- Maintainability

Why are algorithms important?

Algorithms are the backbone of essentially all programming languages and are used for things such as:

- Problem solving
- Efficiency
- Reusability
- Correctness
- Scalability
- **Maintainability** – Algorithms with clear and well-documented steps are easier to maintain and debug.

Where are algorithms used?

Algorithms are used in essentially all software, but serving different purposes:

- Scientific Research
- Computer science and programming
- Computer graphics
- Healthcare
- And more…

Where are Algorithms Used?

Algorithms are used in essentially all software, but serving different purposes:

- **Scientific Research** – Researchers use algorithms for data analysis, simulations, and modelling in fields like physics, chemistry, biology, and astronomy.
- Computer science and programming
- Computer graphics
- Healthcare
- And more…

Where are Algorithms Used?

Algorithms are used in essentially all software, but serving different purposes:

- Scientific Research
- **Computer Science and Programming** – Algorithms are used to create programs that perform various tasks, from simple calculations to complex data processing. Algorithms are also applied to organize and manipulate data efficiently using structures like arrays, linked lists, trees, and graphs.
- Data analysis and machine learning
- Computer graphics
- Healthcare
- And more…

Where are Algorithms Used?

Algorithms are used in essentially all software, but serving different purposes:

- Scientific Research
- Computer Science and Programming
- **Data Analysis and Machine Learning** – Algorithms are used to discover patterns, trends, and insights from large datasets, as well as to train models to make predictions and decisions.
- Computer graphics
- Healthcare
- And more…

Where are Algorithms Used?

Algorithms are used in essentially all software, but serving different purposes:

- Scientific Research
- Computer Science and Programming
- Data Analysis and Machine Learning
- **Computer Graphics** – Algorithms are employed to render graphics in video games, simulations, and computer-aided design (CAD) applications. Algorithms also analyze images and videos for tasks, such as facial recognition.
- Healthcare
- And more…

Where are Algorithms Used?

Algorithms are used in essentially all software, but serving different purposes:

- Scientific Research
- Computer Science and Programming
- Data Analysis and Machine Learning
- Computer Graphics
- **Healthcare** – Medical algorithms assist in diagnosis, treatment planning, and monitoring patient health. Genetic algorithms can optimize DNA sequences and predict protein structures.
- And more…

Where are Algorithms Used?

Algorithms are used in essentially all software, but serving different purposes:

- Scientific Research
- Computer Science and Programming
- Data Analysis and Machine Learning
- Computer Graphics
- Healthcare
- **And more…** – Cryptography, artificial intelligence, search engines, networking, optimization, finance, gaming, transportation and logistics, weather forecasting, social media and recommendations, and E-commerce.

Where are Algorithms Used in Science?

Algorithms are used in all sciences, but the ones most relevant to the course may be:

- Simulation and modelling
- Genomics and bioinformatics
- Medical research
- Social sciences
- Neuroscience

Where are Algorithms Used in Science?

Algorithms are used in all sciences, but the ones most relevant to the course may be:

- **Simulation and modelling** – Algorithms are used to simulate complex physical, chemical, biological, and environmental systems, aiding researchers understand phenomena that are difficult to study directly.
- Genomics and bioinformatics
- Medical research
- Social sciences
- Neuroscience

Where are Algorithms Used in Science?

Algorithms are used in all sciences, but the ones most relevant to the course may be:

- Simulation and modelling
- **Genomics and bioinformatics** – Algorithms are vital for DNA sequencing and genome assembly. Bioinformatics algorithms analyse DNA, RNA, and protein sequences, identifying genes, regulatory elements, and potential disease markers.
- Medical research
- Social sciences
- Neuroscience

Where are Algorithms Used in Science?

Algorithms are used in all sciences, but the ones most relevant to the course may be:

- Simulation and modelling
- Genomics and bioinformatics
- **Medical research** – Algorithms assist in medical image analysis, such as MRI, CT scans, and X-rays, aiding in diagnosis and treatment planning. Machine learning algorithms analyse patient data for disease diagnosis and prognosis, drug discovery, and personalized medicine.
- Social sciences
- Neuroscience

Where are Algorithms Used in Science?

Algorithms are used in all sciences, but the ones most relevant to the course may be:

- Simulation and modelling
- Genomics and bioinformatics
- Medical research
- **Social sciences** – Social science researchers use algorithms for social network analysis, sentiment analysis of text data, and modelling human behaviour in various contexts.
- Neuroscience

Where are Algorithms Used in Science?

Algorithms are used in all sciences, but the ones most relevant to the course may be:

- Simulation and modelling
- Genomics and bioinformatics
- Medical research
- Social sciences
- **Neuroscience** - Algorithms analyse brain imaging data (e.g., fMRI, EEG) to study brain function, connectivity, and neurological disorders. Machine learning algorithms assist in decoding brain signals for brain-computer interfaces.

Where will You Use Algorithms?

<Think about it>

How are Algorithms Created?

Algorithm creation is a skill that requires knowledge in multiple areas, among the most rudimentary are:

- Understanding the problem
- Problem-solving and algorithmic thinking
- Programming knowledge
- Data structures

How are Algorithms Created?

Algorithm creation is a skill that requires knowledge in multiple areas, among the most rudimentary are:

- **Understanding the problem** – Before you can design an algorithm, you need a clear and complete understanding of the problem you are trying to solve. This includes defining the problem, its constraints, and the desired outcomes.
- Problem-solving and algorithmic thinking
- Programming knowledge
- Data structures

How are Algorithms Created?

Algorithm creation is a skill that requires knowledge in multiple areas, among the most rudimentary are:

- Understanding the problem
- **Problem-solving and algorithmic thinking** – You should be able to break down complex problems into smaller, more manageable subproblems and devise a plan to solve them. You should also be able to abstract a real-world problem into a computational problem and devise a step-by-step solution.
- Programming knowledge
- Data structures

How are Algorithms Created?

Algorithm creation is a skill that requires knowledge in multiple areas, among the most rudimentary are:

- Understanding the problem
- Problem-solving and algorithmic thinking
- **Programming knowledge** – You should have a strong grasp of programming concepts and at least one programming language. This includes knowledge of variables, data types, control structures (e.g., loops and conditionals), functions, and libraries or modules.
- Data structures

How are Algorithms Created?

Algorithm creation is a skill that requires knowledge in multiple areas, among the most rudimentary are:

- Understanding the problem
- Problem-solving and algorithmic thinking
- Programming knowledge
- **Data structures** – Understanding data structures (e.g., vectors, lists, data frames) and how to use them effectively is essential because many algorithms manipulate data.

Fundamental Concepts of Algorithms

Structure and design

Algorithm Characteristics – Basics

Let us use the analogy of baking a cake as using an algorithm in programming.

- Input
- Output
- Finiteness
- Definiteness
- Effectiveness

Algorithm Characteristics – Input

In our analogy **input** is the list of ingredients you need to make the cake (e.g., eggs, sugar, flour).

Just like the recipe, an algorithm needs specific information to work with. This information is provided as input (e.g., numbers, text, data frames).

The input is essential because it tells the algorithm what it should work on or process. Just as you need the right ingredients to bake a cake, the algorithm needs the **right input data** to do its job correctly.

Algorithm Characteristics – Output

Output is the result or what you get after following the recipe's instructions. When you bake a cake using a recipe, the **finished cake** is the output.

Just like the recipe, when an algorithm finishes its job, it produces an output in the form of, for example **numbers, text, or data frames**.

The output is what you are trying to achieve by using the algorithm. Just as you follow a recipe to get a delicious cake, you use an algorithm to get the **desired information or result**.

Algorithm Characteristics – Overview

An algorithm takes the **input** (like ingredients in a recipe), processes it **step by step**, and finally gives you the **output**, which is the answer or solution to the problem you wanted to solve.

Algorithm Characteristics – Finiteness

Finiteness means that the instructions in the algorithm must come to an end. Just as a recipe has a final step where you are done cooking, an algorithm has a last step where it finishes its work.

Just like you stop cooking when the dish is ready, an algorithm can not go on forever. It must complete its task and stop at some point. This ensures that it will not run indefinitely and use up all your resources.

This characteristic makes algorithms predictable and reliable.

Algorithm Characteristics – Definiteness

Definiteness means that each step in the algorithm is clear and leaves no room for confusion. Just like a recipe tells you exactly what to do, an algorithm's instructions are specific and unambiguous.

When you follow a recipe, you do not have to guess or make assumptions. Likewise, an algorithm's steps are precisely defined. It tells the computer exactly what to do at every point, without any vagueness or uncertainty.

This characteristic ensures that anyone who follows the algorithm, including a computer, can understand and execute each step without confusion.

Algorithm Characteristics – Effectiveness

Effectiveness means that the algorithm does the job it is supposed to do and gets you the result you want. When you follow a recipe, you expect it to help you make a tasty cake, and if it does, then it is effective.

Similarly, when you use an algorithm, it's designed to solve a specific problem or task, and it should do so correctly.

This characteristic ensures that the algorithm is valuable and trustworthy.

Algorithm Analysis

- **Time complexity**
- **Space complexity**
- Worst-case, best-case, and average-case analysis
- Amortized analysis
- Comparisons and swaps
- And more…

Algorithm Analysis

- **Time complexity** – Measures how the algorithm's runtime grows as the size of the input increases. It is often expressed using Big O notation (e.g., O(1), O(log n), O(n), O(n^2)), which provides an upper bound on the algorithm's running time.
- Space complexity
- Worst-case, best-case, and average-case analysis
- Amortized analysis
- Comparisons and swaps
- And more…

Algorithm Analysis

- Time complexity
- **Space complexity** – Measures the amount of memory or space an algorithm requires to execute. It helps assess the algorithm's memory efficiency and is also expressed using Big O notation.
- Worst-case, best-case, and average-case analysis
- Amortized analysis
- Comparisons and swaps
- And more...

Algorithm Analysis

- Time complexity
- Space complexity
- **Worst-case, best-case, and average-case analysis** - Algorithms can perform differently in various scenarios. Analyzing the worst-case, best-case, and average-case time complexities helps provide a comprehensive view of algorithm performance.
- Amortized analysis
- Comparisons and swaps
- And more…

Algorithm Analysis

- Time complexity
- Space complexity
- Worst-case, best-case, and average-case analysis
- **Amortized analysis** – This metric evaluates the average time or space consumption of an algorithm over a sequence of operations, rather than analyzing individual operations.
- Comparisons and swaps
- And more…

Algorithm Analysis

- Time complexity
- Space complexity
- Worst-case, best-case, and average-case analysis
- Amortized analysis
- **Comparisons and swaps** – For sorting and searching algorithms, counting the number of key comparisons and data swaps can be useful metrics to assess their efficiency.
- And more…

Algorithm Analysis

- Time complexity
- Space complexity
- Worst-case, best-case, and average-case analysis
- Amortized analysis
- Comparisons and swaps
- **And more…** – Stability, recursion depth, input sensitivity, constant factors and hidden constants, I/O operations, cache complexity, parallelization and concurrency, energy efficiency, quantum complexity.

Big O Notation

Conducting a Big O notation analysis involves evaluating the time or space complexity of an algorithm to understand how it scales with input size.

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Clearly define the algorithm you want to analyze. Ensure you understand how it works.

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Identify the input parameters or data size that affect the algorithm's performance. For example, in a sorting algorithm, the input size may be the number of elements to be sorted.

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Break down the algorithm into its basic operations or steps. These operations are usually in the form of loops, conditionals, assignments, and arithmetic operations. Count how many times each basic operation is executed in terms of the input size.

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

In most cases, you want to express the operation count in terms of the input size, which is typically denoted as 'n.' For example, if a loop runs 'n' times, the operation count would be O(n). If a nested loop runs 'n' times within another loop that also runs 'n' times, you might have O(n^2) operations.

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Determine which term in your operation count dominates as 'n' becomes very large. This term represents the primary factor in the algorithm's complexity. For example, in an algorithm with O(3n^2 + 2n + 1), the dominant term is O(n^2).

Big O Notation

Identify the algorithm
Determine the relevant input
Count basic operations
Express the operations in terms of n
Identify the dominant term
Apply Big O notation
Consider best, worst, and average cases
Validate the analysis
Document the analysis

Express the algorithm's complexity using Big O notation. It is often written as "O(f(n))," where 'f(n)' represents the dominant term identified in the previous step. For example, if the dominant term is 'n^2,' you would express the complexity as O(n^2).

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Analyze the algorithm's performance in different scenarios, such as best-case, worst-case, and average-case scenarios. This helps provide a more complete understanding of the algorithm's behavior.

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Validate your Big O analysis through testing and benchmarking. Run the algorithm with various input sizes and measure its actual runtime or memory usage. Compare the observed performance with your theoretical analysis.

Big O Notation

Identify the algorithm

Determine the relevant input

Count basic operations

Express the operations in terms of n

Identify the dominant term

Apply Big O notation

Consider best, worst, and average cases

Validate the analysis

Document the analysis

Document your findings, including the algorithm's complexity in Big O notation, any assumptions made, and the reasoning behind your analysis. This documentation can be valuable for others who use or maintain the algorithm.

Big O Examples

- O(1)
- O(log n)
- O(n)
- O(n^2)

Big O Examples – O(1)

An example of a Big O O(1) algorithm is a simple array (or list) **element access or retrieval** operation. In this type of algorithm, the **time complexity remains constant** regardless of the size of the data structure.

Big O Examples – O(1)

<example “Algorithm” in R>

Reflection - O(1) time complexity is considered the best-case scenario in terms of efficiency because the algorithm's performance remains constant, making it highly efficient for tasks that require quick and predictable access to specific data elements.

Big O Examples – O(log n)

An example of a Big O O(log n) algorithm is the **binary search algorithm**. Binary search is a highly efficient way to search for a specific element in a sorted list or array.

The time complexity of the binary search algorithm is O(log n), where 'n' is the number of elements in the sorted array.

Big O Examples – O(log n)

<example “Algorithm” in R>

Reflection – The algorithm is significantly more efficient than a linear search (O(n)), especially for large datasets, as it reduces the search space by half with each comparison.

Big O Examples – O(n)

An example of a Big O O(n) algorithm is **linear search**, also known as sequential search. In a linear search algorithm, you look through a list of elements one by one until you find the target element you are searching for or determine that it does not exist in the list.

The time complexity of the linear search algorithm is O(n), where 'n' is the number of elements in the input list. This means that the number of operations (comparisons) grows linearly with the size of the input list.

Big O Examples – O(n)

<example “Algorithm” in R>

- Reflection – In the worst case, you may have to check every element in the list before finding the target or determining that it is not present.

Big O Examples – O(n^2)

An example of a Big O O(n^2) algorithm is the **bubble sort algorithm**. Bubble sort is a simple sorting algorithm that repeatedly selects the minimum (or maximum) element from an unsorted part of the list and moves it to the beginning (or end) of the sorted part of the list.

The time complexity of the bubble sort algorithm is O(n^2), where 'n' is the number of elements in the list.

Big O Examples – O(n^2)

<example “Algorithm” in R>

Reflection – The number of comparisons and swaps grows quadratically with the size of the input, making it inefficient for large datasets compared to more efficient sorting algorithms like merge sort or quicksort, which have better time complexity.

Big O Notations

<Discuss>

Basic Algorithms

Sort and search

Sorting Algorithms

Sorting algorithms are a fundamental part of computer science and data processing. They are a set of techniques used to arrange elements in a specific order within a data structure, such as an array or a list.

Common types of sorting algorithms include:

- **Bubble sort, insertion sort**, selection sort, merge sort, quick sort, heap sort, counting sort, and radix sort

The choice of a sorting algorithm depends on the specific requirements of the task and the characteristics of the data being sorted.

Sorting Algorithms

Bubble Sort

Insertion Sort

Selection Sort

Merge Sort

Quick Sort

Heap Sort

Counting Sort

Radix Sort

Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the entire list is sorted.

Sorting Algorithms

Bubble Sort

Insertion Sort

Selection Sort

Merge Sort

Quick Sort

Heap Sort

Counting Sort

Radix Sort

Insertion sort builds the sorted array one item at a time. It takes each element from the input list and inserts it into its correct position within the sorted part of the list.

Sorting Algorithms

- Bubble Sort
- Insertion Sort
- **Selection Sort**
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort

Selection sort finds the minimum (or maximum) element from the unsorted part of the list and moves it to the beginning (or end) of the sorted part. It repeats this process until the entire list is sorted.

Sorting Algorithms

- Bubble Sort
- Insertion Sort
- Selection Sort
- **Merge Sort**
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort

Merge sort is a divide-and-conquer algorithm that divides the input list into smaller sublists, sorts them, and then merges them back together to create a sorted list. It has a time complexity of O(n log n).

Sorting Algorithms

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- **Quick Sort**
- Heap Sort
- Counting Sort
- Radix Sort

Quick sort is another divide-and-conquer algorithm that selects a "pivot" element, partitions the list into elements less than the pivot and elements greater than the pivot, and then recursively sorts the sublists. It has an average time complexity of O(n log n).

Sorting Algorithms

Bubble Sort
Insertion Sort
Selection Sort
Merge Sort
Quick Sort
Heap Sort
Counting Sort
Radix Sort

Heap sort converts the input list into a binary heap, a special data structure with the largest (or smallest) element at the root. It repeatedly removes the root element and rebuilds the heap until the list is sorted.

Sorting Algorithms

Bubble Sort
Insertion Sort
Selection Sort
Merge Sort
Quick Sort
Heap Sort
Counting Sort
Radix Sort

Counting sort is a non-comparison-based sorting algorithm that works for integers within a specific range. It counts the occurrences of each element and uses this information to place the elements in order.

Sorting Algorithms

Bubble Sort
Insertion Sort
Selection Sort
Merge Sort
Quick Sort
Heap Sort
Counting Sort
Radix Sort

Radix sort is a non-comparison-based sorting algorithm that sorts integers by their individual digits, from the least significant digit to the most significant digit (or vice versa).

Bubble Sort Algorithm

<Example "Algorithm" in R>

Insertion Sorting Algorithm

<Example "Algorithm" in R>

Bubble Sort vs Insertion Sort

- Are there instances where one algorithm performs better than the other.
- How would you express this in a mathematical formula and with Big O notations?

<Discuss>

What Do I Now Know About Algorithms?

Basic

- What is an algorithm?
- Where are algorithms used?

Advanced

- What should you keep in mind when designing algorithms?
- How do you know if an algorithm is effective?

Karolinska
Institutet