**History of Algorithms**

1. Ancient Greek Problem Solvers:

Procedures for solving geometric and arithmetic problems date back to ancient Greece. Visionaries like Euclid, who lived over two thousand years ago, were instrumental in formulating techniques that laid the foundation for algorithmic thinking. Euclid's most celebrated contribution was the discovery of the algorithm for finding the greatest common divisor (gcd), a fundamental concept in number theory.

2. The Birth of the Algorithm:

The term "algorithm" owes its existence to the 9th-century Persian mathematician Abu Abdullah Muhammad ibn Musa al-Khwarizmi. His groundbreaking works introduced algebraic concepts that would revolutionize mathematics. Al-Khwarizmi conducted his research in the vibrant city of Baghdad, which served as the epicenter of scientific studies and trade during his time.

3. From Al-Khwarizmi to Algorithm:

Al-Khwarizmi's name was Latinized as "Algoritmi," which eventually evolved into the term "algorithm." Initially, the word referred primarily to rules for performing arithmetic operations, reflecting its early roots. Over time, its scope expanded to encompass a wide range of definite procedures for solving problems and performing tasks.

4. The Modern Era: D.E. Knuth and "The Art of Computer Programming"

In the mid-twentieth century, a pivotal moment in the history of algorithms occurred when Donald E. Knuth undertook an in-depth study and analysis of algorithms. His monumental work culminated in the creation of the comprehensive book series, "The Art of Computer Programming." This series serves as the cornerstone of modern algorithmic study, providing a vast and influential resource for computer scientists and mathematicians worldwide.

**Definition of Algorithm:** An algorithm is any well-defined computational procedure that takes some values or set of values, as input and produces some value, or set of values, as output.

**Representation of Algorithm:**

Algorithm can be expressed/represented through natural language, pseudocode, a

nd mathematical notation.

1. The **natural language** representation makes the algorithm easy to understand.
2. The **pseudocode** representation provides a structured and language-independent format.
3. The **mathematical notation** offers a concise and formal description.

Let's take a simple algorithm for finding the largest number in a list and express it in natural language, pseudocode, and mathematical notation:

Algorithm: Find the Largest Number in a List

**Natural Language:**

1. Start by assuming the first number in the list is the largest.
2. Go through each number in the list one by one.
3. Compare the current number with the assumed largest number.
4. If the current number is larger, update the assumed largest number to be the current number.
5. Continue this process until you have checked all numbers in the list.
6. The final assumed largest number is the largest number in the list.

**Pseudocode:**

function findLargestNumber(list):

largest = list[0] // Assume the first number is the largest

for i from 1 to length(list) - 1:

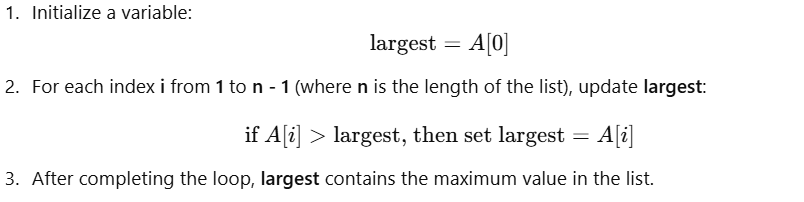
if list[i] > largest:

largest = list[i]

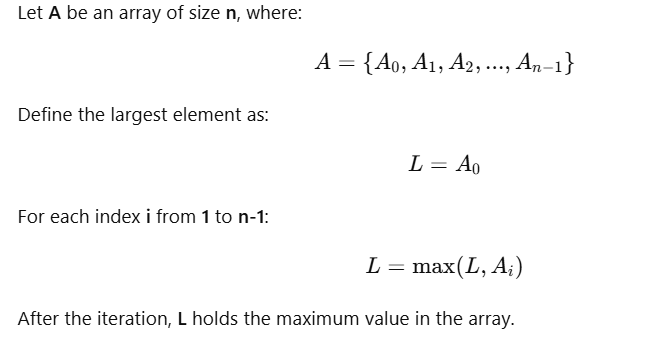
return largest

**Mathematical Notation Representation:**

Let's represent the list as an array A, where A[0] represents the first element, A[1] represents the second element, and so on. We can use mathematical notation to express the algorithm as follows:



Or



The same algorithm is expressed in natural language for easy understanding, in pseudocode for a more structured representation that is still language-independent, and in mathematical notation to provide a concise and formal description.

**Difference between Algorithm and Program**

|  |  |
| --- | --- |
| **Algorithm** | **Program** |
| Created at design time | Created at implementation time |
| Domain expert-person with domain knowledge write algorithm | Programmer writes program using certain programming language. |
| H/W and OS independent | H/W and OS dependent |
| Mathematical analysis of algorithm to measure its efficiency | Testing (run and debug |

Think of software development like building a house. Before you start construction, you need a clear blueprint—this is the **design phase**. It helps you understand what you're going to build, ensuring everything fits together properly.

Once the design is ready, you move to the **implementation phase**, where you actually build the software—just like constructing the house based on the blueprint.

Unlike physical construction, where mistakes can be costly and hard to fix, software engineers have the advantage of flexibility. If they make a mistake or want to improve something, they can simply delete the code and rewrite it without wasting materials or resources. This ability to easily modify and refine code makes software development more adaptable compared to building physical objects.

An **algorithm** is a step-by-step method for solving a problem. A **program** is also a step-by-step method for solving a problem, but it is written in a programming language so that a computer can execute it.

When developing software, there are several stages. Two important ones are **design** and **implementation**.

* **Design phase:** This is the planning stage, where you carefully think about how your program will work. You don’t write code yet; instead, you describe the steps in simple language, like writing instructions in plain English. You can use pen and paper or basic tools like MS Word or Notepad.
* **Implementation phase:** After designing, you write the actual program in a programming language based on your plan.

In contrast to physical construction, where mistakes can waste time and resources, software engineers can easily rewrite programs without any material loss. That's why proper design comes first—it ensures you know exactly what to build before you start coding.

In short, an **algorithm** is the plan or blueprint, and a **program** is the final product written in a programming language.

Algorithm is a step-by-step procedure for solving computational problems. Program is a step-by-step procedure for solving computational problems. In the software development lifecycle, which encompasses the phases of a software project's development, two crucial phases are the design and implementation phases. For instance, when you intend to create or engineer something, the first essential step is designing. Designing is the key to ensuring that you fully comprehend what you are going to construct or develop. Here's the crux: you cannot embark on development or construction through trial and error, as you would with physical objects. In software engineering, engineers can easily write a program, change their minds, delete the program, and start over. This flexibility eliminates the concern of wasting time on useless code. Therefore, the sequence is clear: first, you design, and then you write the program. During the design phase, you don't write code in a programming language; rather, you express it in simple, understandable English statements. You may even use pen and paper or software like MS Word or Notepad for this purpose. This process helps you gain a clear understanding of how your program will function, and this is essentially what an algorithm is.

* Algorithm creation occurs during the design phase, while program implementation takes place once the design is finalized.
* The person responsible for designing the program is typically referred to as a designer. However, whether it's a designer or a domain expert with knowledge of the specific problem domain, they can create the algorithm. Additionally, a programmer, when taking on the role of a designer, can also design an algorithm. The key factor is having the necessary domain knowledge to create an effective algorithm for the given problem. For example, when developing accounting software, an accountant's expertise is invaluable, and for a hospital application, doctors or administrative staff with domain knowledge are essential contributors. Therefore, those with domain knowledge about the application's purpose are the ones best suited to write the Algorithm. Programmers can also possess domain knowledge and play dual roles as designers and programmers. Now, let's talk about the language used to write algorithms. You have the flexibility to use any language, be it English or mathematical notations, as long as it is understandable to both designers and programmers on the project. However, when it comes to actually writing the program, programming languages like C, C++, Java, or Python are commonly employed.
* It's important to note that an algorithm is hardware and operating system-independent, meaning it doesn't rely on the specific machine or configuration. Conversely, writing a program is dependent on hardware and operating systems, which can vary.
* After creating an algorithm, the next step involves analyzing it for efficiency in terms of time and space. This analysis ensures that the algorithm is both logically sound and efficient. In contrast, with a program, you don't need to study it in this manner; you simply run it and check for errors. This process is known as testing.

**Properties of Algorithm**

The five properties of an algorithm:

**Input:** An algorithm can have zero or more inputs. It's not mandatory to have at least one input. For example, printing "hello world" has zero inputs, while finding the sum of two numbers has two inputs.

**Output:** An algorithm must have at least one output, although it can have more. Output represents the result or outcome of the algorithm's computation.

**Unambiguous (Definiteness):** Every instruction in an algorithm must be clear and unambiguous. Instructions should have a single, precise meaning without multiple interpretations.

**Finiteness:** An algorithm must contain a finite number of steps, and each instruction within it must take a finite amount of time for execution. It should not result in an infinite loop or take an infinite amount of time.

**Effectiveness:** An algorithm should perform the intended task for which it was created, without containing unnecessary or irrelevant statements. Every instruction should be feasible and contribute to achieving the algorithm's goal.

These properties ensure that an algorithm is well-defined, efficient, and capable of producing meaningful results.

// Ambiguous Statement: "Read the value."

int x;

cin >> x;

// Ambiguous Statement: "Add 5."

int total = 10;

total += 5;

// Ambiguous Statement: "Multiply by 2."

int value = 7;

value \*= 2;

In these ambiguous statements, the intent or context of the operation is not clear without additional information.

**What is Algorithm**

An algorithm is any **well-defined computational procedure** that takes some values, or set of values, as input and produces some value, or set of values, as output. It terminates after finite steps.

An algorithm is thus a **sequence of computational steps** that transform the input into output.

Unlike a program, an algorithm is a **mathematical entity**, which is **independent of** a specific programming language**, machine, or compiler**.

**Algorithm design** is all about the **mathematical theory** behind the design of good programs.

A **good understanding of algorithms** is essential for a good understanding of the most basic element of computer science: **programming.**

Sometimes, there is often a **small critical portion** of the **software**, which may involve only tens **to hundreds of lines** of code, but where the great **majority of computational time** is spent (80% of the execution time takes place in 20% of the code.) The micro issues in programming involve how best to deal with these small critical sections.

It may be very important for the success of the overall project that these sections of code be written in the most efficient manner possible.

An unfortunately **common approach to this problem** is to first design an **inefficient algorithm** and **data structure** to solve the problem, and then take this poor design and attempt to **fine-tune** its performance by applying **clever coding tricks** or by **implementing** it on the most expensive and **fastest machines** around to boost performance as much as possible.

The problem is that if the **underlying design is bad**, then often no amount of **fine-tuning** is going to make a **substantial difference**.

Before you **implement**, first be sure you have **a good design**.

Most of the fastest algorithms are fast because they use fast data structures, and vice versa.

**Data Structures**

A **data structure** is a way to store and organize data to facilitate efficient access and modification.

**Key Considerations for Data Structures**

1. **Data Organization:**  
   Data structures determine how data is arranged and stored in memory. This organization directly impacts the efficiency of data access, manipulation, and searching within a program.
2. **Data Retrieval & Efficiency:**  
   Different data structures are optimized for different operations:
   * **Arrays** provide fast random access but can be inefficient for frequent insertions and deletions.
   * **Linked lists** are efficient for dynamic insertions and deletions but have slower random access compared to arrays.
   * **Hash tables** allow fast lookups using keys, making them ideal for databases and caching.
   * **Trees and graphs** efficiently represent hierarchical or networked relationships between data elements.
3. **Choosing the Right Data Structure:**  
   The choice of data structure depends on the program's specific needs. Examples:
   * **Spreadsheet program:** Data should be stored in a grid-like structure.
   * **Bank-account database:** Customer accounts, each with a unique ID, should be stored in a hash table or set for fast lookups.
   * **File-system manager:** Files and folders should be organized in a tree-like structure to represent hierarchical relationships.

**Caching** to the technique of **storing frequently accessed data in a fast-access memory (cache) to reduce retrieval time** and improve performance.

For example, in a **hash table**, caching allows quick lookups of previously stored key-value pairs, avoiding slower database queries or computations. Caching is commonly used in:

* **Web applications** (e.g., storing frequently accessed pages to reduce server load).
* **Databases** (e.g., keeping recently queried records in memory).
* **Processors** (e.g., storing recently used instructions for quick execution).

**Data structure**

When you **solve a problem** with a **computer program**, always ask first,

***How should the program store the information upon which it computes?***

A ***data structure*** is a way to store and organize data in order to facilitate access and modifications.

additional points to consider about data structures:

Data Organization: Data structures determine how data is organized and stored in memory. This organization affects how efficiently the program can access, manipulate, and search for data.

Data Retrieval: Different data structures excel in different operations. For instance, arrays are great for random access, while linked lists are more efficient for insertion and deletion operations. Choosing the appropriate data structure depends on the specific requirements of your program.

For example:

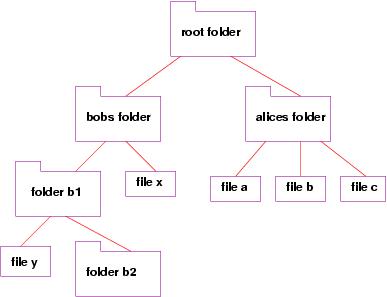
* If the program is a **spreadsheet program**, then the **information** should be held in a data structure that is a **grid**.
* If the program is **a bank-account database**, then the **information** should be grouped into customer accounts, each with a **unique ID**, saved in an **array or set**.
* If the program is a **file-system manager**, then the information are **files and folders** that are organized in **a tree-like structure**.

Each of these problems required a distinct *data structure* in the solution.

It helps to draw a picture of the structure. For example, if you are writing a **vote-counting program for the US presidential election**, you might draw this picture of the model:



On the other hand, if you are writing the **file-system manager for Linux**, then your program must hold folders and files, and the picture of the model might look like this:



The picture should suggest to you the kind of computer variables and data structures you will require to build the solution. In fact, many of the courses in the computer science program deal with **efficient algorithms and data structures**, but just as they **apply to various applications:** compilers, operating systems, databases, artificial intelligence, computer graphics and vision, etc.

Thus, a **good understanding of algorithm design** is a central element to a good understanding of **computer science** and **good programming**.

If more than one algorithm exists for a certain problem, then we will mathematically prove that one algorithm is better than another.

In order to design good algorithms, we must define a criteria for measuring the efficiency of algorithms.

We will measure algorithms in terms of the **amount of computational resources** that the algorithm requires. These resources include mostly **running time and memory.**

Example: Insertion sort and merge sort are two algorithms for sorting problem (Same Problem)

|  |  |  |
| --- | --- | --- |
| **Computer A Faster Computer** | | **Computer B Slower Computer** |
| Running insertion sort | | Running merge sort |
| Execute 1 billion instructions/sec | | Execute 10 million instructions/sec |
| 1billion=1000 million | | 1 million=106 |
| Compute A is 100 times faster than Computer B | | |
| Programmer-codes insertion sort—2n2instructions to sort n numbers (c1=2) | | Merge sort---average programmer—inefficient compiler—code takes 50 nlgn instructions (C2=50) |
| To sort one million numbers | | |
|  | |  |
| Using algorithm on Computer B whose running times grow slowly even with poor compiler, Comuter B runs 20 times faster than computer A | | |
| To sort 10 million numbers | | |
| Insertion sort takes approx. 2.3days | Merge sort takes 20 minutes | |