The Importance of Algorithms

Manjunath.R

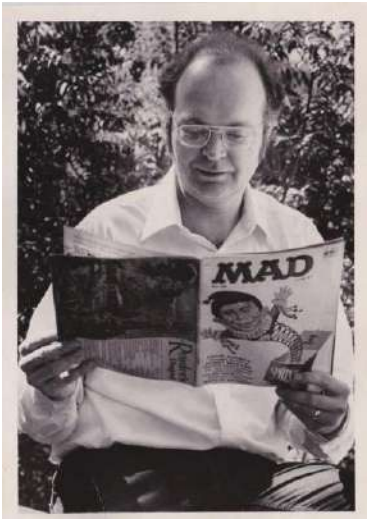
#16/1, 8th Main Road, Shivanagar, Rajajinagar, Bangalore560010, Karnataka, India

*Corresponding Author Email: manjunath5496@gmail.com

*Website: http://www.myw3schools.com/

Abstract

A Computer Program can be viewed as an elaborate algorithm and algorithms are very important in Computer Science for solving a problem -- based on conducting a sequence of specified actions. The best chosen algorithm usually means a small procedure that solves a recurrent problem and makes sure computer will do the given task at best possible manner. In cases where efficiency matters -- a proper algorithm is really vital to be used. An algorithm is important in optimizing a computer program according to the available resources – often play a very significant part in the structure of artificial intelligence, where simple algorithms are used in simple applications, while more complex ones help frame strong artificial intelligence.



> The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct.
>
> **Donald Knuth**

Introduction

You might have an algorithm for getting from office to home, for making a chunk of code that calculates the terms of the Fibonacci sequence, or for finding what you're looking for in a retail store. Algorithms are the building blocks of computer programs or sequence of unambiguous instructions (the term 'unambiguous' indicates that there is no room for subjective interpretation) that tells how the problem could be addressed and solved -- which is definitely overblown in their importance like road maps for accomplishing a given, well-defined automated reasoning task -- which always have a clear stopping point.

Long division and column addition are examples that everyone is familiar with -- even a simple function for adding two numbers is implementation of a particular algorithm. Online grammar checking uses algorithms. Financial computations use algorithms. Robotic field uses algorithms for controlling their robot using algorithms. An **encryption algorithm** transforms data according to specified actions to protect it. A search engine like Google uses search engine algorithms (such as, takes search strings of keywords as input, searches its associated database for relevant web pages, and returns results). In fact, it is difficult to think of a task performed by your computer that does not use computer rules that are a lot like a recipes (**called algorithms**).

The use of computer algorithms (step-by-step techniques used for Problem-solving) plays an essential role in space search programs. Scientists have to use enormous calculations, and they are managed by high-end supercomputers, which are enriched with detailed sets of instructions that computers follow to arrive at an answer. Algorithms have applications in many different disciplines from **science** to **math** to **physics** and, of course, computing -- and provide us the most ideal option of accomplishing a task. Here is some importance of algorithms in computer programming.

- **To improve the effectiveness of a computer program:** An algorithm (procedure or formula for solving a problem, based on conducting a sequence of specified actions) can be used to improve

the speed at which a program executes a problem and has the potential of reducing the time that a program takes to solve a problem.

- **Proper usage of resources:** The right selection of an algorithm will ensure that a program consumes the least amount of memory. Apart from memory, the algorithm can determine the amount of processing power that is needed by a program.

The algorithm for a child's morning routine could be the following:

Step 1: Wake up and turn off alarm

Step 2: Get dressed

Step 3: Brush teeth

Step 4: Eat breakfast

Step 5: Go to school

The algorithm to add two numbers entered by user would look something like this:

Step 1: Start

Step 2: Declare variables num1, num2 and sum

Step 3: Read values num1 and num2

Step 4: Add num1 and num2 and assign the result to sum

sum ← num1 + num2

Step 5: Display sum

Step 6: Stop

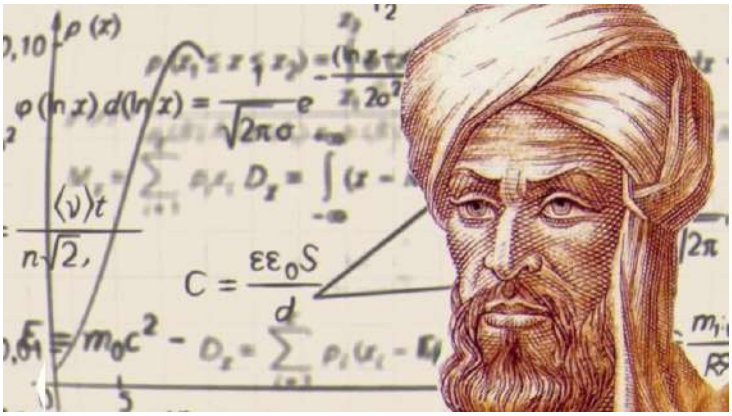
Two of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way to achieve the required output or to accomplish our task. In computer programming, there are often many different ways – algorithms (any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output) -- to accomplish any given task. Each algorithm has credits and demerits in different situations. If you have a million integer values between **-2147483648** and **+2147483647** and you need to sort them, the bin sort is the accurate algorithm to use. If you have a million book titles, the quick sort algorithm might be the best choice. By knowing the toughness and weaknesses of the different algorithms, you pick the best one to accomplish a specific task or to solve a specific problem.

One of the most important aspects of an algorithm is how fast it can manipulate data in various ways, such as inserting a new data item, searching for a particular item or sorting an item. It is often easy to come up with a list of rules to follow in order to solve a problem, but if the algorithm is too slow, it's back

to the drawing board. Efficiency of an algorithm depends on its design and implementation. Since every procedure or formula for solving a problem based on conducting a sequence of specified actions -- uses computer resources to run -- execution time and internal memory usage are important considerations to analyze an algorithm.

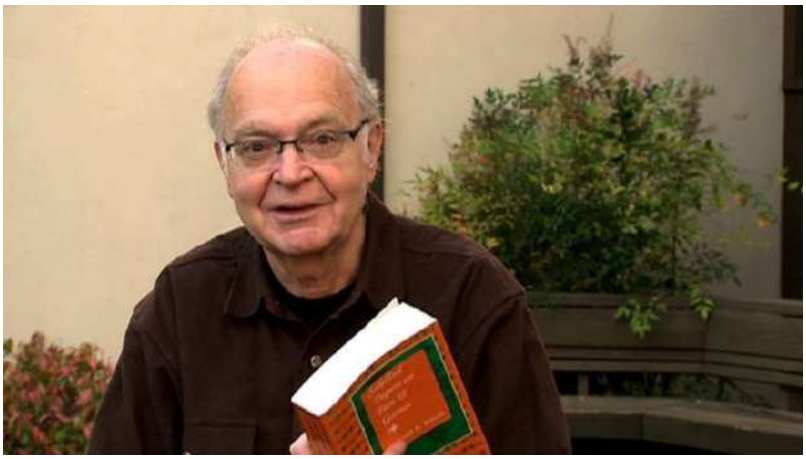
Why Study Algorithms?

Algorithms are the heart of computer science (usually means a procedure or basically instance of logic written in software that solves a recurrent problem of finding an item with specific properties among collection of items or transforming data according to specified actions to protect it), and the subject has countless practical applications as well as intellectual depth that is widely used throughout all areas of information technology including solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that often involves repetition of an operation. The word algorithm -- a mathematical concept whose roots date back to 600 AD with invention of the decimal system -- derives from the name of the ninth century Persian mathematician and geographer,



Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850. On the other hand, it turns out algorithms (widely recognized as the foundation of modern computer coding) have a long and distinguished history stretching back as far as the Babylonians.

Although there is some available body of facts or information about early multiplication algorithms in Egypt (around **1700-2000** BC) the oldest algorithm is widely recognized to be valid or correct to have been found on a set of **Babylonian clay tablets** that date to around **1600 - 1800 BC**. Their exact significance only came to be revealed or exposed around 1972 when an American computer scientist, mathematician, and professor emeritus at Stanford University



Donald E. Knuth published the first English translations of various Babylonian **cuneiform** mathematical tablets.

Here are some short extracts from his 1972 manuscript that explain these early algorithms:-

"The calculations described in Babylonian tablets are not merely the solutions to specific individual problems; they are actually general procedures for solving a whole class of problems." - Pages 672 to 673 of "Ancient Babylonian Algorithms**".**

The wedge-shaped marks on clay tablets also seem to have been an early form of instruction manual:-

"Note also the stereotyped ending, 'This is the procedure,' which is commonly found at the end of each section on a table. Thus the Babylonian procedures are genuine algorithms, and we can commend the Babylonians for developing a nice way to explain an algorithm by example as the algorithm itself was being defined...." - Pages 672 to 673 of "Ancient Babylonian Algorithms**".**

The use of computers, however, has raised the use of algorithms in daily transactions (like accessing an automated teller machine (ATM), booking an air or train or buying something online) to unprecedented levels of real-world problems with solutions requiring advanced algorithms abounds. From **Google**

search to morning routines, algorithms are ubiquitous in our everyday life -- and their use is only likely to grow to break down tasks into chunks that can be solved through specific implementations. Many of the problems, though they may not seem realistic, need the set of well-defined algorithmic knowledge that comes up every day in the real world. By developing a good understanding of a series of logical steps in an algorithmic language, you will be able to choose the right one for a problem and apply it properly. Different algorithms play different roles in programming – and algorithms are used by computer programs **where a program –**

- Get input data.
- Process it using the complex logics.
- Stop when it finds an answer or some conditions are met.
- Produce the desired output.

To give you a better picture, here is the most common type of algorithms:

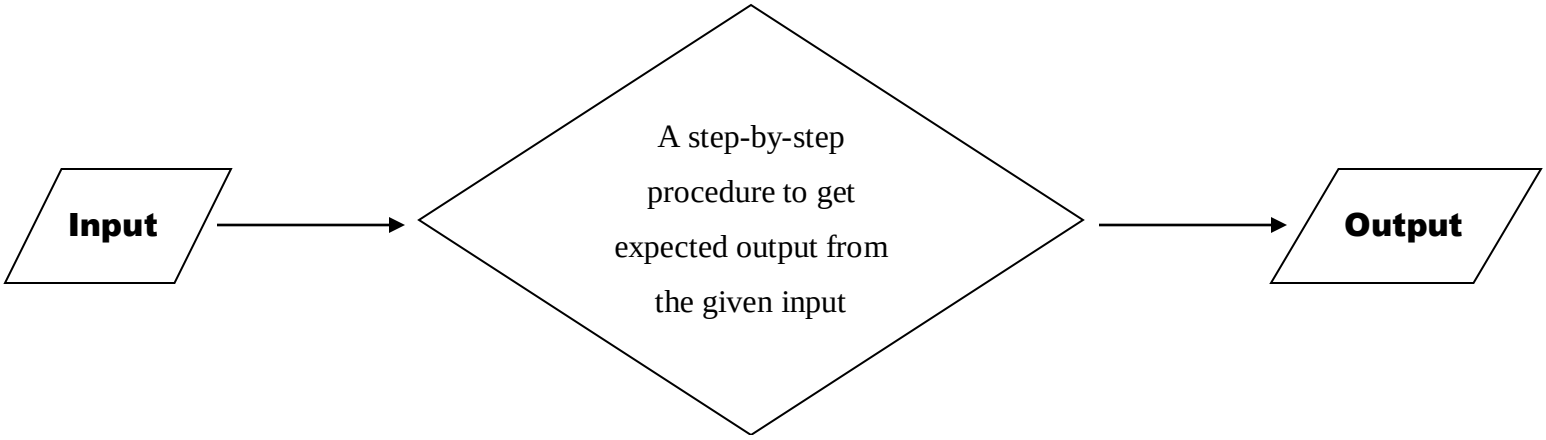
- Searching Algorithms
- Sorting Algorithms
- Path finding Algorithms
- Tree and graph based algorithms
- Approximate Algorithms
- Compression Algorithms
- Random Algorithms
- Pattern Matching
- Sequence Finding and a lot more

You only need to define your problem then select the right algorithm to use. The word **algorithm** may not appear closely connected to kids, but the truth is that -- for kids -- understanding the process of building a step by step method of solving a problem helps them build a strong foundation in logical thinking and problem solving. Here are some problems you can ask your kid to discuss algorithmic solutions with you:

- How do we know if a number is odd or even?
- How do we calculate all of the factors of a number?
- How can we tell if a number is prime?
- Given a list of ten numbers in random order, how can we put them order?

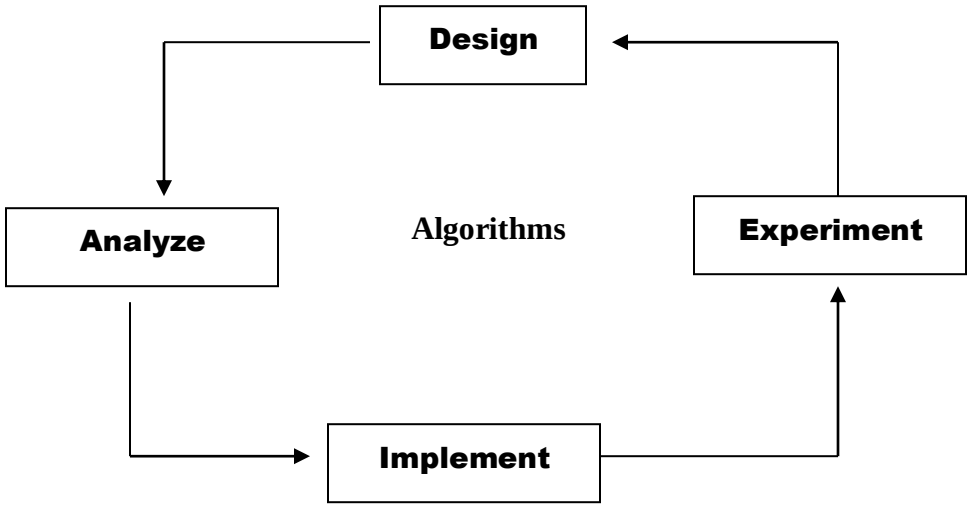
Algorithms has shown it can yield results in all industries — from predicting insurance sales opportunities and generating the millions of search inquiries every day to automating medicine research, optimizing transportation routes, and much more. While algorithms help companies like **Master Card** and **Visa** to keep their users' information, such as card number, password, and bank statement safely -- **algorithms aren't perfect**. They fail and some fail spectacularly. Over the past few years, there have been some serious fails with algorithms, which are the formulas or sets of rules used in digital decision-making processes. Now people are questioning whether we're putting too much trust in the algorithms. When algorithms go bad: Online failures show humans are still needed. Disturbing events at **Facebook**, **Instagram** and **Amazon** reveal the importance of context.

Scripting language	Programming language
Platform-specific	Platform-agnostic (**cross-platform**)
Interpreted	Compiled
Faster at runtime	Slower at runtime
More code-intensive	Less code-intensive
Creates standalone applications	Creates applications as part of a stack



Algorithm

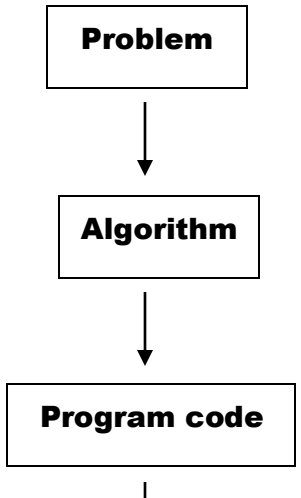
Priori Analysis	Posterior Analysis
checking the algorithm before its implementation	checking the algorithm after its implementation



Algorithm + Data structure = Program

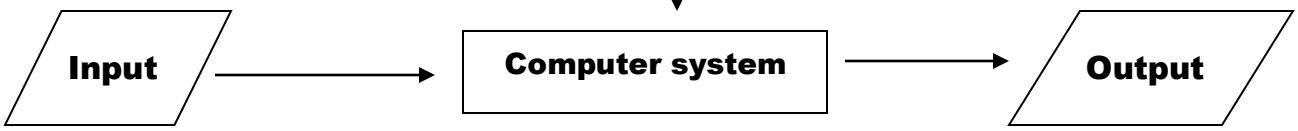
The way in which various **program data elements** are organized and stored into the memory so that the data can be used efficiently

Characteristics of an Algorithm
• Well-defined input
• Desired output
• Finiteness (**not end up in an infinite loops**)
• Effectiveness (**executed in finite time**)
• Definiteness (**precisely defined**)



Algorithms are not arbiters of objective truth and fairness simply because they're math.

— Zoe Quinn



Performance analysis of an algorithm depends on 2 factors:

- **Space Complexity:** The amount of memory space required by an algorithm to complete its task
- **Time Complexity:** The amount of time required by an algorithm complete its task

The number of operations an algorithm performs to complete its task (**considering that each operation takes the same amount of time**)

Linear Data Structure	Non-linear Data Structure
Data items are arranged in sequential order (**one after the other**)	Data items are arranged in non-sequential order (**hierarchical manner**)
Memory is not utilized in an efficient way	Memory is utilized in an efficient way
Time complexity increase with the data	Time complexity remains the same

The algorithm to find the largest number among 3 numbers:

```
Step 1: Start

Step 2: Declare variables a, b and c.

Step 3: Read variables a, b and c.

Step 4: If a > b

          If a > c
```

```
            Display a is the largest number.
         Else
            Display c is the largest number.
      Else
         If b > c
            Display b is the largest number.
         Else
            Display c is the greatest number.
Step 5: Stop
```

The time taken by the computer to run code = number of instructions × time to execute each instruction

Factors that affect run time of an algorithm:

- The hardware platform used
- Representation of abstract data types
- Efficiency of compiler
- Implementer programming skill
- Complexity of underlying algorithm
- Size of the input

- **Worst Case Complexity:** The maximum time taken by an algorithm to complete its task.
- **Best Case Complexity:** The minimum time taken by an algorithm to complete its task.
- **Average Case Complexity:** The average time taken by an algorithm to complete its task.

Run time

- Polynomial time
- Superpolynomial time

Polynomial time → run time that does not increase faster than n^k, which includes:

- constant time (n^0)
- logarithmic time (log n)
- linear time (n^1)
- quadratic time (n^2) and other higher degree polynomials (like n^3)

n → input size

Superpolynomial time → run time that does increase faster than n^k, which includes:

- exponential time (2^n)
- factorial time (n!) and anything else faster.

An algorithm is said to take **constant time** (n^0), if

- The run time of an algorithm → **constant** (doesn't increase) − no matter how large the input size increases

An algorithm is said to take **logarithmic time** (log n), if

- The run time of an algorithm increases in direct proportion to the logarithm of the input size

An algorithm is said to take **linear time** (n^1), if

- The run time of an algorithm increases in direct proportion to the input size

↓

Whenever input size doubles, the running time increases twofold

An algorithm is said to take **quadratic time** (n^2), if

- The run time of an algorithm increases in direct proportion to the input size squared

↓

Whenever input size doubles, the running time increases fourfold

An algorithm is said to take **cubic time** (n^3), if

- The run time of an algorithm increases in direct proportion to the cube of the input size

↓

Whenever input size doubles, the running time increases eightfold

An algorithm is said to take **factorial time** (!n), if

- The run time of an algorithm increases in direct proportion to the factorial of the input size

↓

Whenever input size increases by 1, the running time increases by a factor of input size

The better the time complexity of an algorithm is, the faster the algorithm will complete its task

An algorithm is said to take **linearithmic time** (n log n), if

- The run time of an algorithm increases in direct proportion to the input size times the logarithm of the input size

The total amount of memory space used by the algorithm to execute and produce the result

Space Complexity = Auxiliary space + Memory space used by input values

The extra memory space used by the algorithm during its execution

For any program, **memory space** is required for the following purposes:

- To store compiled version of instructions (**Instruction Space**)
- To store information of partially executed functions at the time of function call (**Environmental Stack**)
- To store all the variables and constants (**Data Space**)

Constant space complexity	takes the same amount of memory space regardless of the input size (n)
Logarithmic space complexity	takes memory space proportional to log n
Linear space complexity	takes memory space directly proportional to n
Linearithmic space complexity	takes memory space directly proportional to n log n
Quadratic space complexity	takes memory space directly proportional to n^2
Cubic space complexity	takes memory space directly proportional to n^3
Exponential space complexity	takes memory space directly proportional to 2^n
Factorial space complexity	takes memory space directly proportional to !n

```
#include<stdio.h>
int main()
{
  int x = 4, y = 6, z;
  z = x + y;
  printf("%d", z);
   return 0;
}
```

- In the above program, 3 integer variables are used, hence they will take up **4** bytes each, so the total space occupied by the above-given program is **4** × 3 = 12 bytes.

- In this program, we have three integer variables. Therefore, this program always takes 12 bytes of memory space to complete its execution. And because this memory space requirement is fixed for the above program, hence space complexity is said to be **constant space complexity** or O(1) space complexity.

```
public int sumArray(int[] array) {
    int size = 0;
    int sum = 0;

    for (int iterator = 0; iterator < size; iterator++) {
        sum += array[iterator];
    }

    return sum;
}
```

In the above program:

- **array** – the function's only argument – the space taken by the array is equal to 4n bytes (where n is the length of the array)
- **size** – a 4-byte integer
- **sum** – a 4-byte integer
- **iterator** – a 4-byte integer

The total memory space needed for this program to execute is 4n + 4 + 4 + 4 = 4n + 12 bytes − which will increasing linearly with the increase in the input value n, hence it is called as **linear space complexity** or O(n) space complexity.

To search an element in a given array, it can be done in two ways:

- Linear search
- Binary search

Linear Search:

Linear search is a very basic and simple search algorithm. In this type of search, a sequential search is made over all elements one by one. Every element is checked and if a match is found then that particular element is returned, otherwise the search continues till the end of the data collection.

For Example:

To search the element 17 it will go step by step in a sequence order:

8	10	12	15	17	20	25
17						

Match not found

8	10	12	15	17	20	25
	17					

Match not found

8	10	12	15	17	20	25
		17				

Match not found

8	10	12	15	17	20	25
			17			

Match not found

8	10	12	15	17	20	25
				17		

Match found

Element 17 is returned.

Linear search (whose running time increases linearly with the number of elements in the array. For example if number of elements is doubled then, on average, the search would take twice as long) is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to linear search.

Binary Search:

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

First, we shall determine half of the array by using this formula −

$$\text{mid} = \text{low} + \frac{(\text{high} - \text{low})}{2}$$

Here it is, $0 + \frac{(9-0)}{2} = 4$ (integer value of 4.5). So, 4 is the mid of the array.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

We change our low to mid + 1 and find the new mid value again.

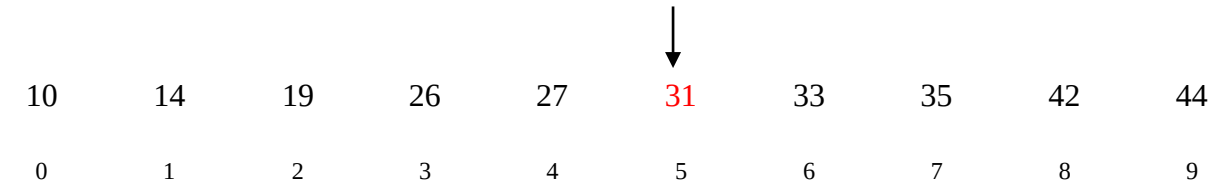
```
low = mid + 1
```

$$mid = low + \frac{(high - low)}{2}$$

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match; rather it is more than what we are looking for. So, the value must be in the lower part from this location.

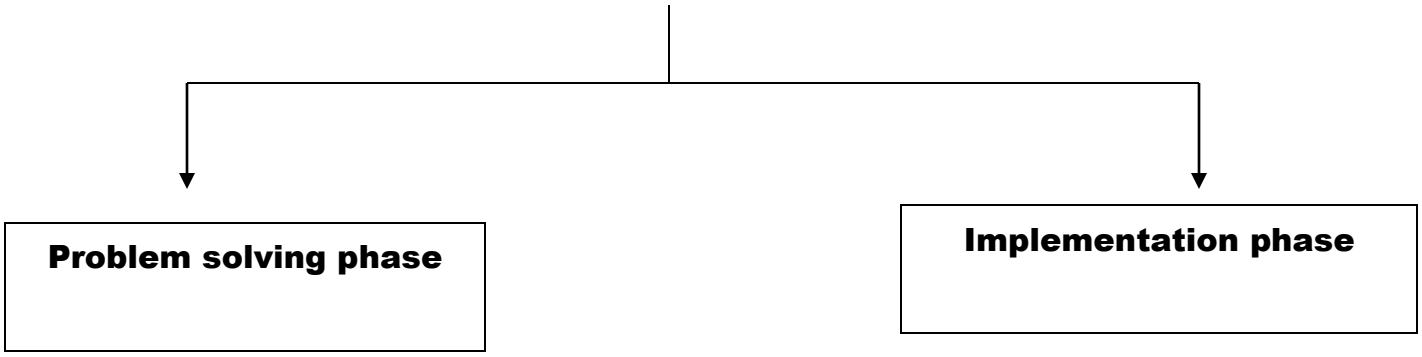
Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

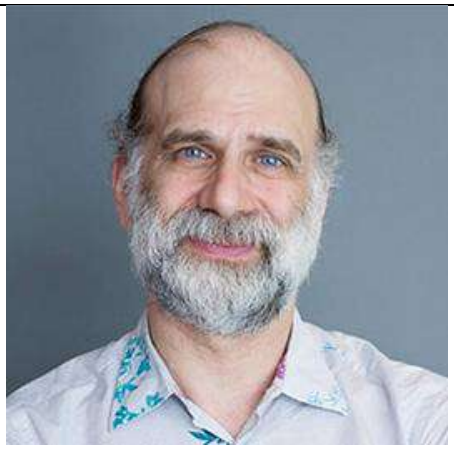
2 phases of Computer Programming Task

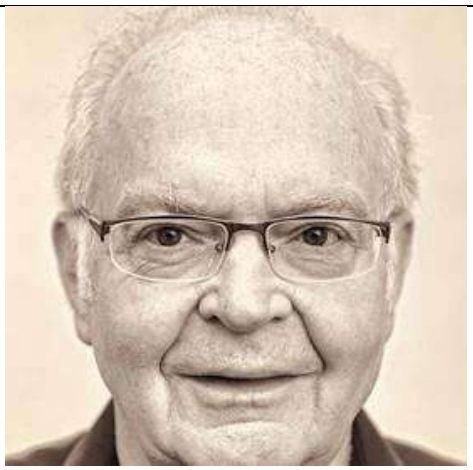


Pseudocode	Algorithm
A method of developing an algorithm	A finite sequence of well-defined, computer-implementable instructions, typically help to simplify and understand the problem
Easy to understand, interpret and easier ease of construction	Quite hard to understand and complex ease of construction

Pseudocode to calculate the area of a circle	Algorithm to calculate the area of a circle

AreaofCircle()
{
BEGIN
Read: Number radius, Area;
Input r;
Area = 3.14 * r * r;
Output Area;
END
}
	1. Start.
	2. Read the radius value r as the input given by the user.
	3. Calculate the area as Area: 3.14 * r * r.
	4. Display the Area.
	5. End.

 **Alexander Stepanov**	Object-oriented programming aficionados think that everything is an object.... this [isn't] so. There are things that are objects. Things that have state and change their state are objects. And then there are things that are not objects. A binary search is not an object. **It is an algorithm.**
 **Bruce Schneier**	Anyone, from the most clueless amateur to the best cryptographer, can create an **algorithm** that he himself can't break.
 **Carl Benjamin Boyer**	Mathematics is as much an aspect of culture as it is a collection of algorithms.

 **Donald Knuth**	[**The Euclidean algorithm is**] the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day.
 **John Green**	You cannot invent an algorithm that is as good at recommending books as a good bookseller, and that's the secret weapon of the bookstore - is that no algorithm will ever understand readers the way that other readers can understand readers.
 **John von Neumann**	The emphasis on mathematical methods seems to be shifted more towards **combinatorics** and **set theory** - and away from the algorithm of differential equations which dominates mathematical physics.

 **Rob Pike**	Data dominates. If you've chosen the right data structures and organized things well, the **algorithms** will almost always be self-evident. Data structures, not algorithms, are central to programming.
 **Peter Norvig**	More data beats clever algorithms, but better data beats more data.
 **Tim Berners-Lee**	The Google algorithm was a significant development. I've had thank-you emails from people whose lives have been saved by information on a medical website or who have found the love of their life on a dating website.

References

- **What's the Importance of Algorithms in Computer Programming?** By Cleophas Mulongo
- **15 of the Most Important Algorithms That Helped Define Mathematics, Computing, and Physics** By Christopher McFadden
- **The Importance of Algorithms for Kids** | Juni Learning
- **Algorithms** By Jeff Erickson