

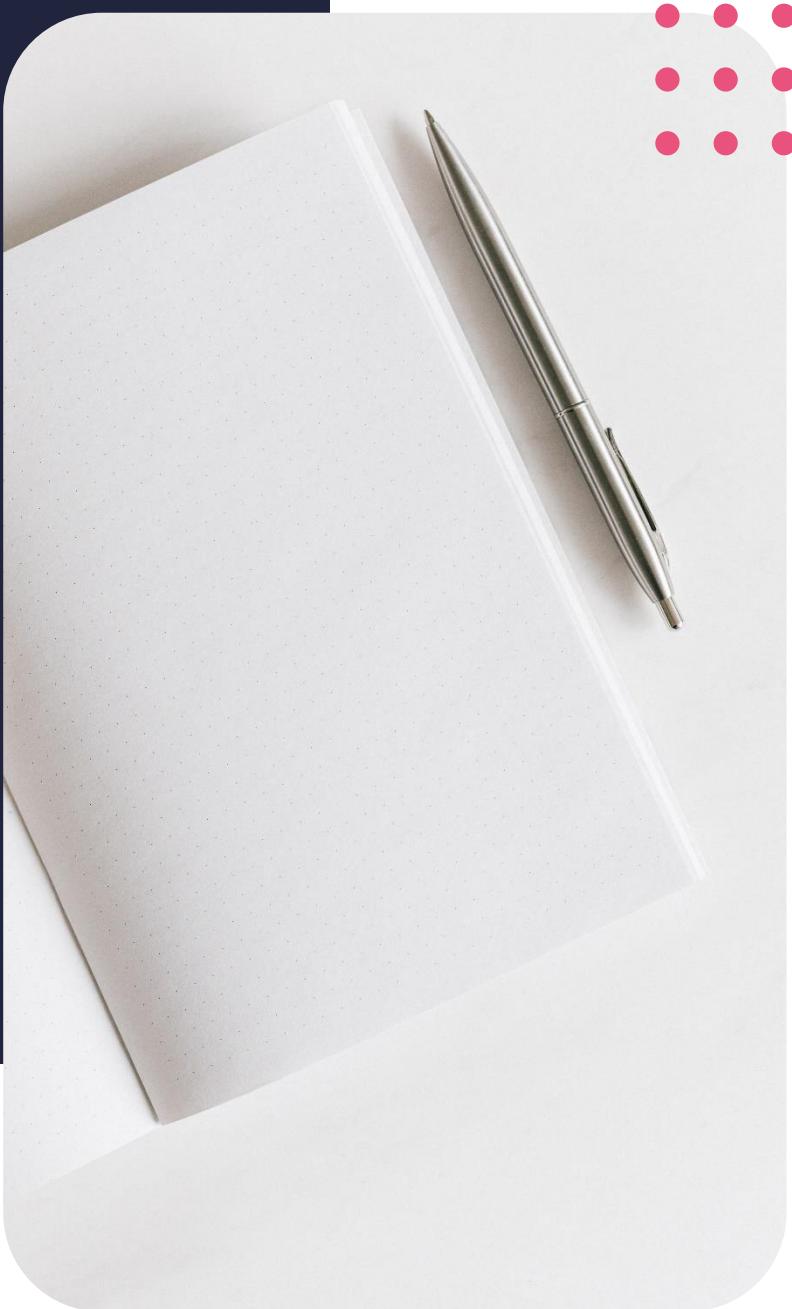
Diploma in Computer Science

Algorithms



Contents

- 3 Lesson outcomes
- 3 Digital business innovation
- 5 Commerce on the web
- 6 Marketing on the web
- 11 References



Lesson outcomes

By the end of this lesson, you should be able to:

- understand what algorithms really are
- Appreciate the different types of algorithm complexities
- Understand how popular searching and sorting algorithms work
- Discuss the attributes of a good algorithm

Understanding the algorithm

Algorithms are the building blocks of all the amazing things that computers can do. The key to writing good software lies in understanding the underlying logic, instead of regurgitating code that you learnt from someone else. There are many factors that affect algorithm design and performance.

What is an algorithm

This is naturally the place to start, right? Well, an algorithm is a step by step sequence for doing something. A more formal definition would be “a process or set of rules to be followed in calculations or other problem-solving operations” (Oxford) you probably have already figured out why this is such a big deal from previous lessons- a computer uses instructions, and instructions have to be issued step by step for clarity. An algorithm is a finite list of instructions

You may be wondering “well, that’s simple, why do we need to study that?” well, that is because designing algorithms isn’t always straightforward. Some algorithms are so complex that they haven’t even been solved efficiently yet! Studying algorithms will upgrade you from merely just writing code; you will not just be writing code because anyhow, as long as it works, you will write code with a great understanding of the underlying logic. This has the direct result of producing way better code.

If computers were fast and memory was infinite, we would still have reason to study algorithms; we still need to prove that the proposed solution is viable, can be executed and can terminate. Even if computers were phenomenal and could do anything you throw at them, it would still make sense to draw up them most optimal solution for a problem. Software still needs to produced according to the proper standards of software development, that includes being well designed and implemented. Back to reality, because computers are not as fast as we would wish them to be, and memory can only be added to a certain reasonable point according to the architecture of the computer, there is even more reason to make our algorithms as efficient as possible.

Underneath all those pretty user interface elements on your phone, PC or tablet exists tons of algorithms that run the show. Fun fact, proficiency in algorithms is one of the most sought-after skills in the world; it is in serious demand globally and can land you at the very top of the food chain!

Relationship of an algorithm to a problem

Generally speaking, an algorithm is a tool for solving a well specified computational problem. Notice here, we say well specified. Remember that lesson we had on problems? Specification is everything when it comes to computing problems.

A computer will take the input and apply each step of the algorithm to that input in order to produce output. Take Google for example. It will take your search query as input, run the algorithm, and return search results as output.

With a flowchart, you can easily follow the path of an algorithm and track the input as it is handled and transformed by the algorithm. A problem statement is the first step in developing an algorithm.

Formulating an algorithm

An algorithm generally consists of three parts: the input, the process then the output. The input is the set of data that the algorithm will work on, the process is the steps that the computer will follow to produce the desired result, then the output is the result of the process followed by the computer.

When writing your algorithm, it is important to ensure that its desired goal does not get lost in lines and lines of code. This includes building in efficiency into your algorithm. Your problem should always follow the optimal route. You should always choose the best way to execute a task. You may be thinking, computers these days are fast, why does it matter to make a madly efficient algorithm? Well, it turns out, those fractions of a second actually matter! A study by marketwatch.com shows that nearly 88% of mobile users will abandon an app if it is slow. Yes, you heard right, 88%! Imagine racking up one million users for your app, only to lose eight hundred thousand users because your app is slow! In computer science, optimisation is everything, it is gold, computer scientists are paid really good money to optimise algorithms!

Complexities

We cannot talk about the efficiency of algorithms without talking about the term “algorithm complexity”. This typically is a mathematical definition, but we’re going to try and avoid that and define it more simply. Algorithm complexity is a measure which evaluates the order of the count of operations, performed by a given or algorithm as a function of the size of the input data. In simpler English, this approximates the number of steps that our algorithm will take to produce output, given input of a certain size. We have already established from previous lessons that computers perform tasks step by step, so given an approximation of the number of steps that the algorithm will take, we can then calculate the approximate time in which execution of the algorithm will take. Take for example, you are creating an online shopping site. You need to find out which search algorithm takes the least time to complete the search. You can calculate the complexities side by side and easily determine which algorithm is best for your site. It is worth noting, though, that when we calculate complexity, we do not calculate exact count, but the order of operation count, for example, an order of N^2 operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order. Just for an extra bit of clarity, “ N ” here refers to the number of items that the computer receives as input.

The big O notation

The big O notation is a way of expressing the efficiency of algorithm without having to dive deep into mathematical equations. It is a metric used to evaluate how fast your code will run, regardless of the platform on which it will be run. The big O notation is used to determine the advantages and disadvantages of an algorithm,

especially where there are rival algorithms for executing the same task. Generally, the big O notation classifies algorithms from fastest to slowest in the following order

$O(1)$ — constant

$O(\log n)$ — logarithmic

$O(n)$ — linear

$O(n \log n)$ — $n \log n$

$O(n^2)$ — quadratic

$O(n^3)$ — cubic

$O(2^n)$ — exponential

$O(n!)$ — factorial

When dealing with Big O: we drop constants in our expression.

The first type of algorithms takes a constant number of steps regardless of the input size, that is, no matter how large or small the input is, the algorithm will execute in the same number of steps all the time.

The next type of algorithm is the logarithmic algorithm. It is named as such because it takes the order of $\log(N)$ steps. The base of the logarithm is often 2, and these are the same logarithms that you learnt about in math classes. For example, if your algorithm has an input of 5 elements then the algorithm would take $O(\log_2(5))$, which would be approximately 2 steps. Similarly, for $N=1,000,000$ we would say $O(\log_2(1000000))$, which is approximately 20 steps.

Another type of algorithm is the linear algorithm. It takes approximately the same number of steps as the number of elements presented as the input. This is denoted as $O(N)$. This is fairly straightforward, if you have 100 input elements, it takes approximately 100 steps. If you have 60000 input elements, you have approximately 60000 steps. You probably remember from mathematics that this type of relationship is called a linear relationship.

For Linearithmic/quasilinear complexity $O(n * \log(n))$ It takes $N * \log(N)$ steps for performing a given operation on N elements. For example, if you have 1,000 elements, it will take about 10,000 steps. It is somewhere in between linear complexity and logarithmic complexity.

In quadratic complexities, it takes the order of N^2 numbers of steps, if $N = 100$, it takes about 10,000 steps.

In cubic functions, it takes the order of N^3 numbers of steps, if we have 100 elements, it takes about 1,000,000 steps.

Things escalate really quickly in relation to the size of the input. Take for example, if N is 10, the value of $O(2^N)$ will be 1024. Now if we add just 10 more input elements to the processing queue, the value of $O(2^N)$ will be 1 048 576! Now if we bump it to 120, then the value of $O(2^N)$ will be $1.33 * 10^{36}$!

Here's a table that summarises the types of complexities

Complexity	Number of steps
Constant	$O(1)$
Logarithmic	$O(\log(N))$
Linear	$O(N)$
Linearithmic /quasilinear	$O(n * \log(n))$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n), O(N!), O(n^k), \dots$

Best Case – this describes the behaviour of algorithm under optimal conditions.

Average Case – this is midway between the minimum number of elements and the maximum number of elements

Worst Case – in this case maximum number of operations are executed.

The actual time that an algorithm will take in execution depends on the complexity that we have been calculating. Low complexity means faster execution, and higher complexity means slower execution, and vice versa. To put this into perspective, let's take a random computer that can perform 50 million operations per second. This table shows how algorithms of various complexities will perform on that computer.

Algorithm	10	20	50	100	1,000	10,000	100,000
O (1)	< 1 sec.	< 1 sec.					
O(log(n))	< 1 sec.	< 1 sec.					
O(n)	< 1 sec.	< 1 sec.					
O(n*log(n))	< 1 sec.	< 1 sec.					
O(n ²)	< 1 sec.	2 sec.	3-4 min.				
O(n ³)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	20 sec.	5.55 hours	231.5 days
O(2 ⁿ)	< 1 sec.	< 1 sec.	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 sec.	hangs	hangs	hangs	hangs	hangs	hangs
O(n ⁿ)	3-4 min.	hangs	hangs	hangs	hangs	hangs	hangs

In general, from this table we can say

-Algorithms with a constant, logarithmic, or linear complexity are so fast we won't even notice any delay, even with a large data set.

-algorithms of complexity O(n*log(n)) are similar to the linear and works nearly as fast, the difference is imperceptible.

- Quadratic algorithms work very well up to several thousand elements.

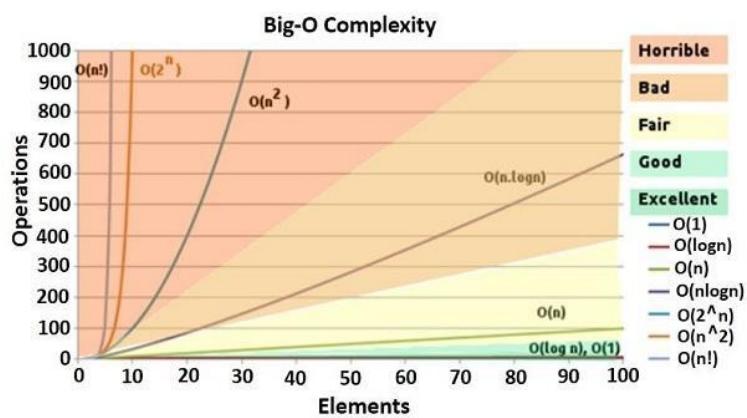
- things start to get ugly for cubic algorithms as you go beyond 1000

- Generally, polynomial algorithms are considered to be fast and work well for thousands of elements.
- Generally, exponential algorithms do not work well and should be avoided whenever possible for an exponential solution, we are only good for about 20 elements, after which things get impossibly huge. Modern cryptography is based on exactly this – there aren't any non-exponential algorithms for breaking current encryption standards
- If you solve a given problem with an exponential complexity this means that you have solved it for a small amount of input data and generally your solution does not work.

There are also a number of points to note when exploring algorithm complexity.

- It is possible that constants in an algorithm with a low complexity may be large and this could eventually make the algorithm slow.

Here's a graph that summarises these observations



For an in-depth visualisation of the big O notation, head over to <https://www.bigocheatsheet.com/>

Algorithms in the wild

There are a number of types of algorithms, each with a specific goal. We will look at these and see what characterises each type, then have a look at one of the most important ones, searching and sorting.

Types of algorithms

Recursive algorithms are ones that call themselves again with a smaller value as inputs which it gets after processing the current input. In simpler words, it's an Algorithm that calls itself repeatedly until the problem is solved. One example of this type of algorithms is the factorial algorithm.

The divide and conquer algorithm works in 2 parts. It first divides the problem into 2, then each of these problems are then solved and merged to produce the final result. One example of this is the merge sorting algorithm, which we are going to look at in a bit.

Dynamic programming algorithms work by recalling the results of the past run and using them to find new results. In other words, it breaks a problem into several simple subproblems, stores them for future reference. An example of this type of algorithm is the Fibonacci number algorithm:

$\text{fib}(N) = 0$ (for $n=0$)

$= 0$ (for $n=1$)

$= \text{fib}(N-1) + \text{fib}(N-2)$ (for $n>1$)

Here, the previous number is used to find the next number.

In greedy algorithms, we attempt to find the optimal solution in a set of solution.

This is done using the 5 components of the algorithm.

The first one is a candidate set from which we try to find a solution.

A selection function which helps choose the best possible candidate.

A feasibility function which helps in deciding if the candidate can be used to find a solution.

An objective function which assigns value to a possible solution or to a partial solution

Solution function that determines when we have found a solution to the problem.

Dijkstra's algorithm is a good example of a greedy algorithm.

A brute force algorithm is literally what it sounds like. It blindly searches for a solution through a set of possible solutions until it finds a suitable one. An example for this is sequential search.

The backtracking algorithm attempts to solve a problem by solving one piece at a time. If a piece of the problem cannot be solved, think of how you would solve a puzzle, sudoku for instance. If a number that you have filled in does not work, you remove it and try another number. Fun fact, sudoku is actually solved in a computer using a backtracking algorithm!

Search algorithms

A lot of times, we need the computer to search for something. For this, a search algorithm is typically used. Popular search algorithms include binary search, Fibonacci search and linear search.

Binary search

Binary search works by comparing the middle item of the list with the search item. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This is repeated until either the search item is found or there are no more sub lists to search. This requires a sorted list.

					↓				
10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

We set our x as 23, the number that we are looking for.

```
Procedure binary_search
    A ← sorted array
    n ← size of array
    x ← value to be searched

    Set lowerBound = 1
    Set upperBound = n

    while x not found
        if upperBound < lowerBound
            EXIT: x does not exist.

            set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

            if A[midPoint] < x
                set lowerBound = midPoint + 1

            if A[midPoint] > x
                set upperBound = midPoint - 1

            if A[midPoint] = x
                EXIT: x found at location midPoint
        end while

    end procedure
```

We then compare our midpoint to x .

Next, we pick a new midpoint.

We then compare our midpoint to x , and there, we find our search item!

There's nothing left to do.

So we exit the loop and end the procedure.

Our search item has been found at location 5.

Linear search

Linear search works by going through the list item by item until the item is found. This doesn't need the list to be sorted.

We set our value as 23, the number that we are looking for.

We then compare our search item to the first item.

We see that they don't match.

So we run the loop again for the next item.

Again, they don't match.

We run the loop again for the next item.

Again they don't match.

We run the loop again for the next item, and again they don't match.

We run the loop again for the next item, again they don't match.

On the 5th run, the search item matches the list item.

So the loop exits, and the procedure ends.

The algorithm for linear search looks like this:

```
procedure linear_search (list, value)
    for each item in the list
        if match item == value
            return the item's location
        end if
    end for
end procedure
```

10	13	16	18	19	23	27	34	36	40
0	1	2	3	4	5	6	7	8	9

Sort algorithms

Back to our different types of algorithms. We are now going to talk about sort algorithms.

There are five popular sorting algorithms.

They are:

Insertion sort

Selection sort, Quicksort, Bubble sort and Merge sort.

They all operate in fairly different ways, and we are going to look at how some of them operate.

If, for example, you are making a search engine application to search for books in a library system. You want the results to be in descending order of relevance, so you need to sort the results according to how close they are to the search term, so you would create an array, store the returned books in the array and sort the array using the built-in sort algorithm. Pretty straightforward right? Well, yes, and no. Yes, because it saves you a lot of coding to just use inbuilt search functions. Then a big BUT, there is a downside. If you didn't know about algorithms, you would think searching is searching, sorting is sorting. This will literally kill your application, because it's not always that the sort algorithm will be appropriate for your use case.

Bubble sort

The bubble sort algorithm goes through an entire array and compares each neighbouring number. It then swaps the numbers and keeps doing this until the list is in ascending order.

Bubble sort is based on the concept of comparing a pair of adjacent elements. If the elements are not in order, then they exchange places.

This algorithm isn't good for large data sets as it has a complexity of $O(N^2)$

```

procedure bubbleSort( list : array of items )
    loop = list.count;
    for i = 0 to loop-1 do:
        swapped = false
        for j = 0 to loop-1 do:
            /* compare the adjacent elements */
            if list[j] > list[j+1] then
                /* swap them */
                swap( list[j], list[j+1] )
                swapped = true
            end if
        end for
        /*if no number was swapped that means
        array is sorted now, break the loop.*/
        if(not swapped) then
            break
        end if
    end for
end procedure return list

```

So, we compare item 0 and 1.

Item 1 is larger, so we swap the items.

We then set the swap flag to TRUE.

We compare items 1 and 2.

Item 1 is larger so we swap the items.

We then set the swap flag to TRUE

We compare Items 2 and 3.

No swap occurs here, so we proceed with the FOR loop.

We compare items 3 and 4.

No swap occurs here, so we proceed with the FOR loop.

We compare items 4 and 5. Item 5 is larger so we swap the items.

We then set the swap flag to TRUE.

Our list is not yet sorted, so we redo the loop.

Item 1 is larger so we swap the items.

We then set the swap flag to TRUE.

Our list is not yet sorted, so we redo the loop.

Item 1 is larger so we swap the items.

We then set the swap flag to TRUE.

For the rest of the items, no swap occurs.

So the loop retains a FALSE swap flag and exits.

From pseudocode to algorithm

In our previous lesson, we discussed how pseudocode helps kickstart your problem-solving quest. In order for that to be as streamlined as possible, there are a few things to be kept in mind.

Pseudocode conventions

As we covered in our previous lesson, when you are writing your program in pseudocode, it needs to follow pseudocode conventions. This makes your program much easier to follow and shortens the time you need to convert it to actual code. As we shall see shortly, the properties of a good algorithm are strongly reminiscent of the properties of good pseudocode. You remember that while we said pseudocode doesn't exactly follow any syntax, it does have a few ground rules that make it universally readable and not confusing. You may have noticed this in the pseudocode that we had in our search and sort algorithms.

Breaking down the code to the smallest units

Another important note to take when you draw up your algorithm is to break down your algorithm into the smallest bits possible. This has the advantage of making the problem a lot easier to solve, while also making your algorithm easier to understand. This cuts down on algorithm development time and results in faster turnaround times. Breaking your algorithm into the smallest bits is probably the first step you would take, identifying parts that can be solved at once and parts that can be further

broken down into sub problems, this ensures that the problem is completely solved and that you will not run into unexpected, overlooked portions.

Properties of a good algorithm

A good algorithm should generally conform to these rules.

- The input should be specified. The data must be precise, that is, input precision requires that you know what kind of data, how much and what form the data should be.

- The output of the algorithm must be specified. It is also good practice to include the output in the algorithm's name. Output precision also requires that you know what kind of data, how much and what form the output should be

- The algorithm must specify every step and specify the order in which the steps will be taken. There should be as much detail as possible in the algorithm, including a way handle any encountered errors.

- The algorithm should be effective, that is, it should be possible to execute all the steps required to get to a solution.

- the algorithm must eventually stop running, either after getting the required solution or establishing that it is not possible to get the required solution. Typically, algorithms that repeat instructions with modified data have finiteness problems.

In short, an algorithm needs to:

Specify input

Specify output

Be well defined

Be effective

Be finite

Other than that, other than that, algorithms should be efficient, that is, run in the least time possible while using as little resources (RAM, etc) as possible. Algorithms should also be understandable.

References

Cusanelli, M. (2014). Study Shows Mobile App Performance is Critical to Success – Channel Futures. [online] Channel Futures. Available at: <https://www.channelfutures.com/mobility/study-shows-mobile-app-performance-is-critical-to-success>

EDUCBA. (2019). Types of Algorithms | Learn The Top 6 Important Types of Algorithms. [online] Available at: <https://www.educba.com/types-of-algorithms/>

ThinkAutomation. (2019). What is an algorithm? An ‘in a nutshell’ explanation - ThinkAutomation. [online] Available at: <https://www.thinkautomation.com/eli5/what-is-an-algorithm-an-in-a-nutshell-explanation/>

GeeksforGeeks. (2019). Why Data Structures and Algorithms Are Important to Learn? - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/why-data-structures-and-algorithms-are-important-to-learn/>

Runestone.academy. (2014). 1.6. Why Study Algorithms? — Problem Solving with Algorithms and Data Structures. [online] Available at: <https://runestone.academy/runestone/books/published/pythonds/Introduction/WhyStudyAlgorithms.html>

HackerEarth Blog. (2016). Why study data structures and algorithms? | HackerEarth Blog. [online] Available at: <https://www.hackerearth.com/blog/developers/study-data-structures-algorithms/>

Svetlin Nakov (2020). Introduction to Programming with C# / Java Books » Chapter 19. Data Structures and Algorithm Complexity. [online] Introprogramming.info. Available at: <https://introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity>

Appendix A: Pseudo-code Conventions. (2006). Algorithms and Networking for Computer Games, [online] pp.227–246. Available at: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/0470029757.app1>