Fundamentals of Algorithms

3.1 Algorithms

3.1.1 Introduction

There are many general classes of problems that arise in discrete mathematics. For instance: given a sequence of integers, find the largest one; given a set, list all its subsets; given a set of integers, put them in increasing order; given a network, find the shortest path between two vertices. When presented with such a problem, the first thing to do is to construct a model that translates the problem into a mathematical context. Discrete structures used in such models include sets, sequences, and functions—structures discussed in **Chapter 2**—as well as such other structures as permutations, relations, graphs, trees, networks, and finite state machines—concepts that will be discussed in later chapters.

Setting up the appropriate mathematical model is only part of the solution. To complete the solution, a method is needed that will solve the general problem using the model. Ideally, what is required is a procedure that follows a sequence of steps that leads to the desired answer. Such a sequence of steps is called an **algorithm**.

Definition 1

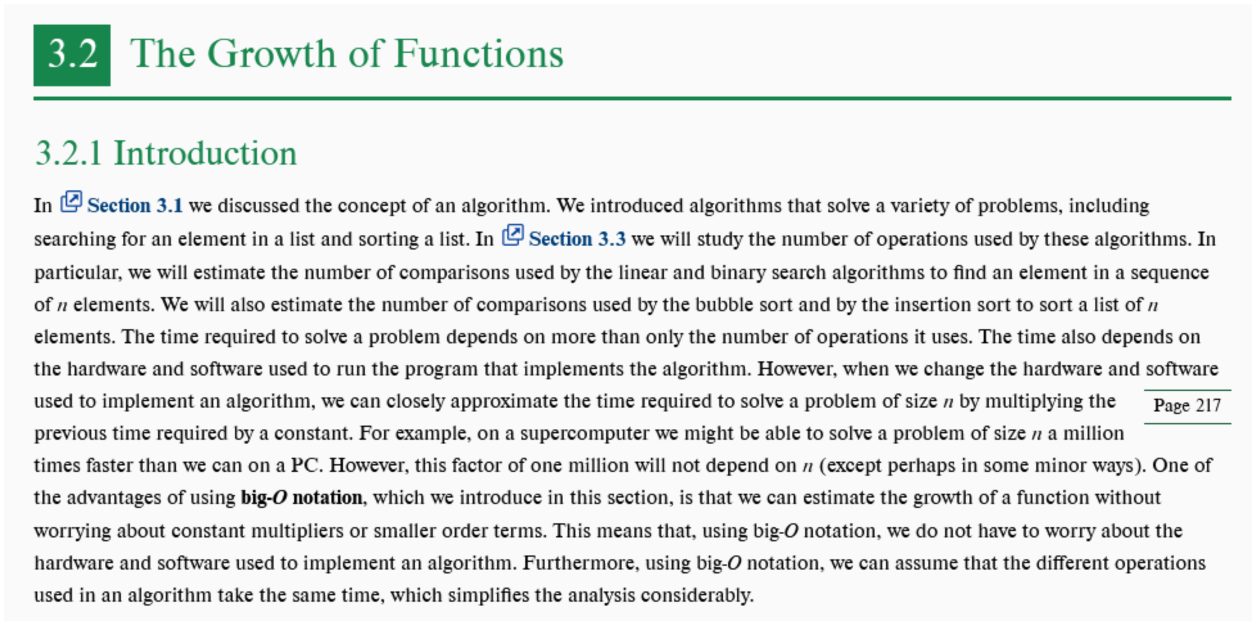
An *algorithm* is a finite sequence of precise instructions for performing a computation or for solving a problem.

PROPERTIES OF ALGORITHMS

There are several properties that algorithms generally share. They are useful to keep in mind when algorithms are described. These properties are:

- *Input.* An algorithm has input values from a specified set.
- *Output.* From each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem.
- *Definiteness.* The steps of an algorithm must be defined precisely.
- *Correctness.* An algorithm should produce the correct output values for each set of input values.
- *Finiteness.* An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
- *Effectiveness.* It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- *Generality.* The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

Page 204



3.2 The Growth of Functions

3.2.1 Introduction

In Section 3.1 we discussed the concept of an algorithm. We introduced algorithms that solve a variety of problems, including searching for an element in a list and sorting a list. In Section 3.3 we will study the number of operations used by these algorithms. In particular, we will estimate the number of comparisons used by the linear and binary search algorithms to find an element in a sequence of n elements. We will also estimate the number of comparisons used by the bubble sort and by the insertion sort to sort a list of n elements. The time required to solve a problem depends on more than only the number of operations it uses. The time also depends on the hardware and software used to run the program that implements the algorithm. However, when we change the hardware and software used to implement an algorithm, we can closely approximate the time required to solve a problem of size n by multiplying the previous time required by a constant. For example, on a supercomputer we might be able to solve a problem of size n a million times faster than we can on a PC. However, this factor of one million will not depend on n (except perhaps in some minor ways). One of the advantages of using **big-*O* notation**, which we introduce in this section, is that we can estimate the growth of a function without worrying about constant multipliers or smaller order terms. This means that, using big-*O* notation, we do not have to worry about the hardware and software used to implement an algorithm. Furthermore, using big-*O* notation, we can assume that the different operations used in an algorithm take the same time, which simplifies the analysis considerably.

Page 217

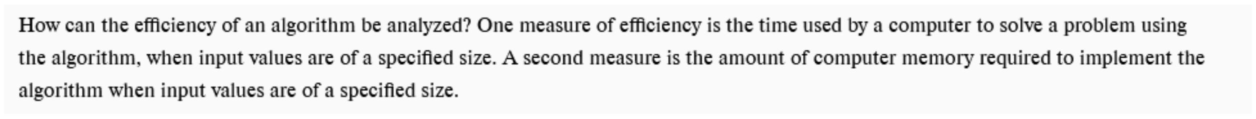
Big-O notation is kind of a simplification of a function. It does not worry about details such as constants or terms with a low degree since as n gets larger and larger, the function acts more and more like a simple form of n.
For instance: the function $6n^3 + 2n^2 - 50$ can be simplified into n^3. Because the change of n becomes more visible with just n^3, as opposed to other terms. Thus, the *big-O notation* of this function would be $O(n^3)$.

We do not want to mess with the formal definition of *big-O* it is already really complicated :)

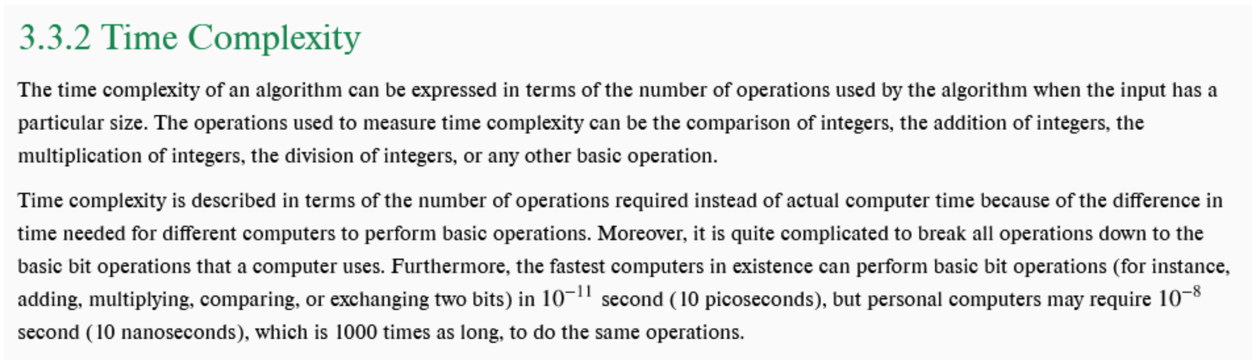
💡 You can check the Big Oh page for a more detailed explanation of *big-O* notation.

Big Oh

Complexity of Algorithms



How can the efficiency of an algorithm be analyzed? One measure of efficiency is the time used by a computer to solve a problem using the algorithm, when input values are of a specified size. A second measure is the amount of computer memory required to implement the algorithm when input values are of a specified size.

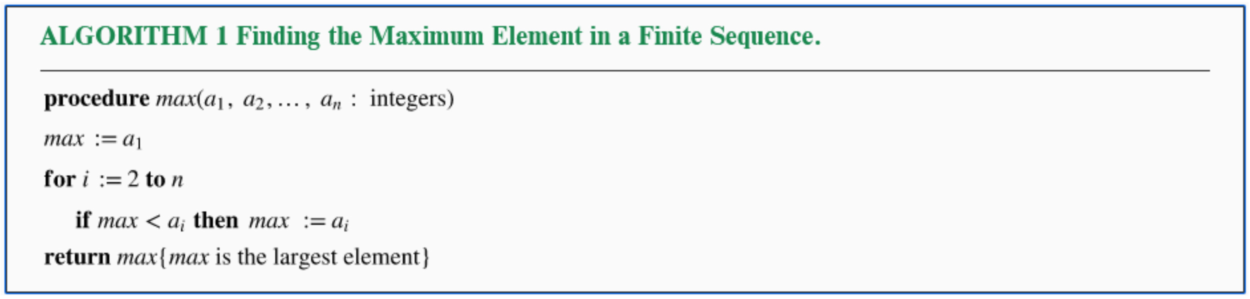


3.3.2 Time Complexity

The time complexity of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size. The operations used to measure time complexity can be the comparison of integers, the addition of integers, the multiplication of integers, the division of integers, or any other basic operation.

Time complexity is described in terms of the number of operations required instead of actual computer time because of the difference in time needed for different computers to perform basic operations. Moreover, it is quite complicated to break all operations down to the basic bit operations that a computer uses. Furthermore, the fastest computers in existence can perform basic bit operations (for instance, adding, multiplying, comparing, or exchanging two bits) in 10^{-11} second (10 picoseconds), but personal computers may require 10^{-8} second (10 nanoseconds), which is 1000 times as long, to do the same operations.

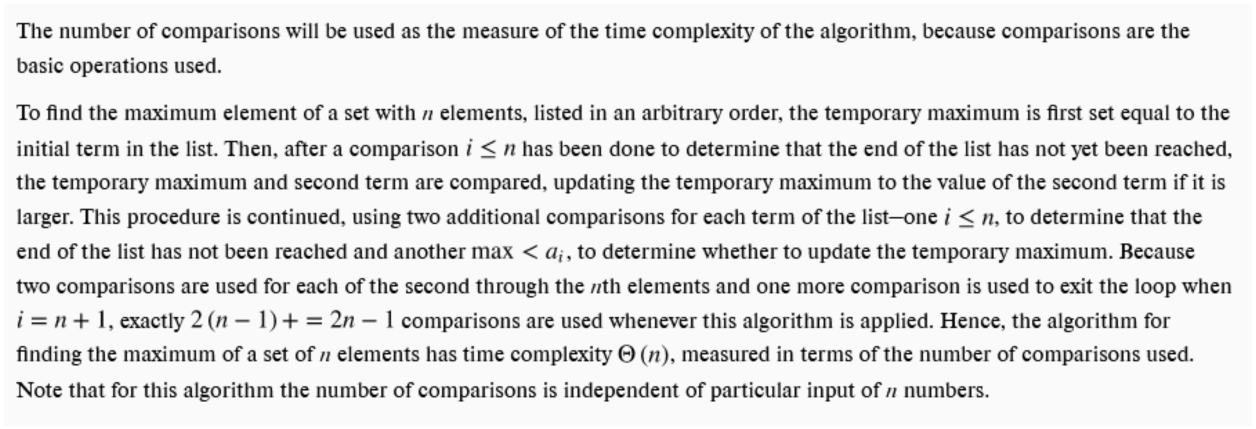
An example time complexity analysis



ALGORITHM 1 Finding the Maximum Element in a Finite Sequence.

```
procedure max($a_1, a_2, \dots, a_n$: integers)
max := $a_1$
for $i := 2$ to $n$
    if $max < a_i$ then $max := a_i$
return max{max is the largest element}
```

This is an algorithm example written with pseudocode. Pseudocode is not a programming language. It is a human-readable form of commands that can be represented in any programming language.



The number of comparisons will be used as the measure of the time complexity of the algorithm, because comparisons are the basic operations used.

To find the maximum element of a set with n elements, listed in an arbitrary order, the temporary maximum is first set equal to the initial term in the list. Then, after a comparison $i \leq n$ has been done to determine that the end of the list has not yet been reached, the temporary maximum and second term are compared, updating the temporary maximum to the value of the second term if it is larger. This procedure is continued, using two additional comparisons for each term of the list–one $i \leq n$, to determine that the end of the list has not been reached and another max $< a_i$, to determine whether to update the temporary maximum. Because two comparisons are used for each of the second through the nth elements and one more comparison is used to exit the loop when $i = n + 1$, exactly $2(n - 1) + = 2n - 1$ comparisons are used whenever this algorithm is applied. Hence, the algorithm for finding the maximum of a set of n elements has time complexity $\Theta(n)$, measured in terms of the number of comparisons used. Note that for this algorithm the number of comparisons is independent of particular input of n numbers.

Do not worry if this seems a bit complicated. It basically means that we have to compare the maximum value found so far with the current element of the sequence and do this for each element once so that in the end, we obtain the maximum element in the sequence. Since we do a constant number of operations for n elements, the time complexity of this algorithm would be O(n).

We can analyse and obtain the Space Complexity of the algorithm in a similar manner, only focusing on the extra variables we keep during the runtime of our algorithm.
In the case of the previous example, we have only one extra variable: the maximum value. During the runtime, this value gets updated for each element of the sequence, but no other variable is introduced. Since the extra space is not dependent on the size of the input n, the *space complexity* of the above algorithm would be constant, that is, O(1).

Things to keep in mind

- $O(1)$ $\Rightarrow$ constant complexity (cost remains unchanged as N increases)
- $O(log(N))$ $\Rightarrow$ logarithmic complexity (cost increases proportional to the increase in $log(N)$)
- $O(N)$ $\Rightarrow$ linear complexity (cost increases proportional to the increase in N)
- $O(N^2)$ $\Rightarrow$ quadratic complexity (cost increases proportional to the increase in N^2)

You can check this website for more complexity analysis examples. They are not written in Python, but the algorithms are well explained and you are smart 💡

Understanding Time Complexity with Simple Examples - GeeksforGeeks
A lot of students get confused while understanding the concept of time complexity, but in this article, we will explain it with a very simple example. Q. Imagine a classroom of 100 students in which you gave your pen to one person. You have to
https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/