# Introduction

Madiba Hudson-Quansah

# Contents

# Chapter 1

# Introduction

## 1.1 What is an Algorithm

**Definition 1.1.1: Algorithm**

A sequence of unambiguous instructions for solving a problem i.e. for obtaining a required output for any legitimate input in a finite amount of time.

Using the task of finding the greatest common divisor (GCD) of two numbers we can explicitly illustrate the general notions of an algorithm. Namely:

- The requirement of non-ambiguity in describing each step of an algorithm.
- The specified range of inputs for which an algorithm works.
- The different ways the same algorithm can be expressed / implemented.
- The non uniqueness of algorithms for a particular problem
- Different algorithms for the same problem may be based on different ideas and have different time and space complexities.

The GCD of two non-negative, non-zero integers $m$ and $n$, denoted $\gcd(m, n)$ is defined as the largest integer that divides both $m$ and $n$ evenly, i.e., with a remainder of zero. Euclid of Alexandria, a Greek athematician, outlined an algorithm for solving this problem in his book *Elements*. Defined as follows:

$$\gcd(m, n) = \begin{cases} m & \text{if } n = 0 \\ \gcd(n, m \mod n) & \text{otherwise} \end{cases}$$

Where $m \mod n$ is the remainder of the division of $m$ by $n$, and when n is zero the GCD is $m$ as it is the only and largest number that divides $m$ evenly between $m$ and $0$. In pseudocode:

**Algorithm 1** gcd $(m, n)$

▷

▷ Computes the greatest common divisor of two numbers
▷ Input: Two non-negative integers $m$ and $n$
▷ Output: The greatest common divisor of $m$ and $n$

**while** $n \neq 0$ **do**
  $r \leftarrow m \mod n$
  $m \leftarrow n$
  $n \leftarrow r$
**end while**
**return** $r$

This algorithm eventually terminates as the value of $n$ decreases with each iteration as its value is set to $m \mod n$ every iteration, and the value of $m$ is always greater than $n$.

Another algorithm for solving this problem is based on the definition of the GCD as the greatest common divisor of $m$ and $n$ as the largest integer that divides both number evenly. This leads us to the conclusion that such a common divisor cannot be greater than the smallest of the two numbers, denoted $t = \min\{m, n\}$. We can then check if $t$ divides both numbers $m$ and $n$ evenly, if it does then $t$ is the GCD of $m$ and $n$. If it does not, we decrease $t$ by 1 and then check if $t - 1$ divides both $m$ and $n$ evenly. This process is repeated until we find a number that divide both $m$ and $n$ evenly or until $t = 1$ in which case the GCD is 1 as 1 divides all numbers evenly. In pseudocode:

**Algorithm 2** Consecutive Integer Checking $(m, n)$

$t \leftarrow \min\{m, n\}$
**while** $t > 1$ **do**
  **if** $m \mod t == 0$ and $n \mod t == 0$ **then**
    **return** $t$
  **end if**
  $t \leftarrow t - 1$
**end while**
**return** $t$

The final algorithm considers the prime factorization of the two numbers $m$ and $n$. The GCD of two numbers is the product of the common prime factors of the two numbers. As there is no unambiguous way to find the prime factorization of a number we simply layout the steps for the algorithm for now.

1. Find the prime factorization of $m$
2. Find the prime factorization of $n$
3. Identify the common prime factors of $m$ and $n$
4. Multiply the common prime factors and return it as the GCD

A simple way of generating consecutive prime numbers not exceeding a given integer $n > 1$ is the Sieve of Eratosthenes. This algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to $n$. Then each iteration of the algorithm selects the next number in the list as a prime number and removes all multiples of that prime number in the list. This continues until no more numbers can be selected as prime numbers i.e. no number can be removed from the list. The remaining numbers in the list are the prime numbers not exceeding $n$. In pseudocode:

**Algorithm 3** SieveNaïve ($n$)

```
▷ Generates a list of prime numbers not exceeding n > 1
▷ Input: A positive number to not exceed n
▷ Output: A list of prime numbers not exceeding n

function ZeroMultiples(L, p, n)
    for i ← 1 to n do
        if L_i ≠ 0 and p ≠ L_i and L_i mod p == 0 then
            L_i ← 0
        end if
    end for
    return L
end function

P[n]                                              ▷ P is an array of size n
for i ← 2 to n do
    P_i ← i
end for
for i ← 1 to n do
    if P_i ≠ 0 then
        prime ← P_i
        P ← ZeroMultiples (P, prime, n)
    end if
end for

T [n]                                             ▷ Temporary array to store prime numbers
i ← 0
for j ← 1 to n do
    if P_j ≠ 0 then
        T_i ← P_j
        i ← i + 1
    end if
end for
                                                  ▷ Remove trailing zeros
return T
```

But through observation, we can deduce that the largest number $p$ whose multiples can still remain to warrant further iterations of the algorithm. We must first note that if $p$ is a number whose multiples are being eliminated on the current pass, then the first multiple of $p$ we must consider is $p \times p$ because all of its smaller multiples $2p, \ldots, (p-1)$ have been eliminated on earlier passes. This prevents us from eliminating the same number more than once. As $p \times p$ should not be greater than $n$, $p$ cannot exceed $\sqrt{n}$ floored i.e. $\lfloor \sqrt{n} \rfloor$. We assume in the following pseudocode that there is function available for computing $\lfloor \sqrt{n} \rfloor$.

**Algorithm 4** Sieve of Eratosthenes ($n$)

▷

▷ Generates a list of prime numbers not exceeding $n > 1$
▷ Input: A positive number to not exceed $n$
▷ Output: A list of prime numbers not exceeding $n$

```
for p ← 2 to n do
    A[p] ← p
end for

for p ← 2 to ⌊√n⌋ do
    if A_p ≠ 0 then                         ▷ p hasn't been eliminated on previous passes
        j ← p × p
        while j ⩽ n do
            A_j ← 0
            j ← j + p
        end while
    end if
end for
                                            ▷ Copy remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A_p ≠ 0 then
        L_i ← A_p
        i ← i + 1
    end if
end for
return L
```

### 1.1.1 Exercises

**Question 1**

Design an algorithm for computing $\lfloor \sqrt{n} \rfloor$ for any positive integer $n$. Besides assignment and comparison, your algorithm may only use the four basic arithmetical operations.

***Solution:***

**Algorithm 5** FloorSqrt $(n)$

▷

▷ Computes $\lfloor \sqrt{n} \rfloor$ for any positive integer $n$
▷ Input: Integer $n$
▷ Output: The floored square root of $n$

```
function ABS( x)
    if x < 0 then
        return −1 × x
    else
        return x
    end if
end function
                                        ▷ Newtons Method for calculating square root
x ⟵ 10
while ABS (x × x − n) > 0.001 do
    x ⟵ (x + n/x) /2
end while

r ⟵ x                                   ▷ Approximate definition of the floor function
while r ⩾ 1 do
    r ⟵ r − 1
end while
return x − r
```

**Algorithm 6** FloorSqrtBinarySearch $(n)$

▷

▷ Finds the floored square root of a positive integer $n$ using binary search
▷ Input: A positive integer $n$
▷ Output: The floored square root of the integer $n$

```

low ⟵ 0
high ⟵ n
while high > low do
    mid ⟵ (high + low) / 2
    if mid × mid ⩽ n and (mid + 1) × (mid + 1) > n then
        return mid
    end if
    if mid × mid < n then
        low ⟵ mid + 1
    else
        high ⟵ mid − 1
    end if
end while
```

**Question 2**

Design an algorithm to find all the common elements in two sorted lists of numbers. For example, for the lists 2, 5, 5, 5 and 2, 2, 3, 5, 5, 7, the output should be 2, 5, 5. What is the maximum number of comparisons your algorithm makes if the lengths of the two given lists are $m$ and $n$, respectively?

***Solution:***

**Algorithm 7** Common $(A, m, B, n)$

▹

▹ Finds the common elements in two lists $A$ and $B$ keeping their multiplicities
▹ Input:
▹ Output:

$i \leftarrow 1$
$j \leftarrow 1$
$k \leftarrow 1$
**while** $i \leqslant m$ and $j \leqslant n$ **do**
  **if** $A_i == B_j$ **then**
    $L_k \leftarrow A_i$ ▹ $L$ is the list of common elements
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
    $k \leftarrow k + 1$
  **else if** $A_j > B_i$ **then**
    $j \leftarrow j + 1$
  **else**
    $i \leftarrow i + 1$
  **end if**
**end while**

## 1.2 Fundamentals of Algorithmic Problem Solving

An algorithm is not a solution to a problem itself, instead it is a step by step procedure for arriving at a solution to a problem. In designing an algorithm for a problem one usually goes through the following steps:

### 1.2.1 Understanding The Problem

- What is the problem?
- What are the inputs to the problem?
- What does an answer to the problem look like?
- What are special cases to consider?
- What are some examples of the problem with their solutions?
- What are the constraints on the problem?
- What are the requirements of the problem?

An input to a problem specifies an instance of the problem an algorithm solves, therefore it is important to define exactly the set of instances an algorithm is expected to solve. Failing to this can lead to an algorithm failing on valid inputs. This requires specifically defining the input space of the problem. For example in the problem of finding the greatest common divisor of two numbers, the input space is the set of all pairs of non-negative integers.

### 1.2.2 Ascertain the Capabilities of the Computational Device

The computational capabilities of an algorithm's target device determines the computational means available to the algorithm, i.e. exact or approximate solving. Most algorithms designed today are intended to operate on a computer using a variant of the von Neumann architecture, i.e. the Random Access Machine (RAM) model. This model assumes that the computer has a memory that can store data and instructions, a processor that can execute instructions, and a control unit that can interpret and execute instructions.

A central assumption of algorithm design is that instructions are executed one after another, one operation after another, sequentially, in what is known as **sequential algorithm design**. However, with the proliferation of parallel processing, computers can now execute instructions concurrently. .This has lead to algorithm design taking advantage of this leading to the development of **parallel algorithms**.

### 1.2.3 Choosing between Exact and Approximate Problem Solving

**Definition 1.2.1: Exact Algorithm**

An algorithm that solves a problem exactly, i.e. it returns the exact solution to the problem.

**Definition 1.2.2: Approximation Algorithm**

An algorithm that solves a problem approximately, i.e. it returns an approximate solution to the problem.

The next thing to consider is whether the problem can be feasibly solved exactly or if an approximate solution is more feasible. The question of feasibility is apparent when considering the time and space complexity of the problem, for example in computing square roots, solving non-linear equation, and evaluating definite integrals.

### 1.2.4 Algorithm Design Techniques

**Definition 1.2.3: Algorithm Design Technique**

A general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Some common algorithm design techniques include:

- Brute Force
- Exhaustive Search
- Decrease and Conquer
- Divide and Conquer
- Transform and Conquer

### 1.2.5 Designing an Algorithm and Data Structures

When designing an algorithm it is important to consider the data structures that will be used to store the data that the algorithm will operate on. The choice of data structure can have a significant impact on the time and space complexity of the algorithm as the data structure determines how the data is stored and accessed.

### 1.2.6 Methods of Specifying an Algorithm

- Natural Language
- Pseudocode
- Flowcharts
- Programming Languages

### 1.2.7 Proving an Algorithm's Correctness

**Definition 1.2.4: Correctness**

Proving an algorithm always produces the correct output for any legitimate input in a finite amount of time.

In order to prove an algorithm's correctness, one must prove that the algorithm terminates and that it produces the correct output for any legitimate input. This can be done using a variety of techniques, including:

- Mathematical Induction - As an algorithm's iterations provide a sequence of steps needed for such proofs.
- Loop Invariants - A condition that is true before and after each iteration of a loop.
- Precondition and Postcondition - Conditions that must be true before and after the algorithm is executed.
- Invariants - Conditions that are true at all times during the execution of the algorithm.
- Assertions - Statements that are true at specific points in the algorithm's execution.

The notion of correctness for approximate algorithms is less straightforward. Instead for an approximate algorithm, one must show the error of the algorithm does not exceed a predefined limit.

### 1.2.8 Exercises

**Question 3**

A peasant finds himself on a riverbank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room for only the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Solve this problem for the peasant or prove it has no solution. (Note: The peasant is a vegetarian but does not like cabbage and hence can eat neither the goat nor the cabbage to help him solve the problem. And it goes without saying that the wolf is a protected species.)

***Solution:*** The peasant can first bring the goat to the other side of the bank. Then return for the wolf and head to the other side of the bank. As the wolf and goat cannot be together alone the peasant can then leave the wolf and take the goat back with him to the other side. Then the peasant drops off the goat but picks up the cabbage and moves it over to the wolf. Then he finally comes back for the goat.

# Chapter 2

# Algorithm Design

For a time function $T(n)$ if the assumptions of the RAM model cannot be confirmed, then the actual time function of an algorithm is:

$$a \times T(n) \leqslant T(n) \leqslant b \times T(n)$$

Where $a$ is the least time taken by any primitive operation and $b$ is the most time taken by any primitive operation.

## 2.1 Primitive Operations

**Definition 2.1.1: Primitive Operation**

Any operation that takes a constant amount of time to execute. Usually primitive operations are implemented as machine instructions.

Examples of primitive operations include:

- Evaluating an expression
- Variable Assignment
- Array indexing
- Comparisons
- Arithmetic operations
- Function calls
- Returning from a function

## 2.2 Growth Rate of Running Time / Asymptotic Order of Growth

### 2.2.1 Big-$O$

The growth rate of a function can also described informally as:

$$\lim_{n \to \infty} T(n) \approx O(n)$$

For the formal definitions see `2_Algorithm_Analysis.pdf` in the `CompSci`/`Data_Structures` folder.

**Example 2.2.1**

**Question 4**

Show that $T(n) = 5n + 3$ is $O(n)$ using the big-$O$ notation

***Solution:*** Let $f(n) = 5n + 3$ and $g(n) = n$. $O$ is defined as for a given function $g(n)$:

$$O(g(n)) = \{f(n) : \exists c > 0, n_0 \in \mathbb{N} \text{ such that } 0 \leqslant f(n) \leqslant c \cdot g(n)\ \forall n \geqslant n_0\}$$

$\therefore$

$$5n + 3 \leqslant cn\ \forall n \geqslant n_0$$

$$\begin{aligned}
5n &\leqslant cn \\
c &= 5, 6, 7, \ldots \\
\text{Since } 5n + 3 \leqslant 5n = F \text{ we chose } c &= 6 \\
5n + 3 &\leqslant 6n \\
3 &\leqslant 6n - 5n \\
3 &\leqslant n \\
\therefore c &= 6\ n_0 = 3
\end{aligned}$$

$\therefore T(n) = 5n + 3$ is $O(n)$.

**Example 2.2.2**

**Question 5**

Show that $T(n) = 5n^2 + 3n - 7$ is not $O(n)$

***Solution:*** Let $f(n) = T(n)$ and $g(n) = n$. From the definition of $O(n)$:

$$O(g(n)) = f(n) \leqslant cg(n)\ \forall n \geqslant n_0$$

$$\begin{aligned}
5n^2 + 3n - 7 &\leqslant cn \\
5n^2 + 3n &\leqslant cn \\
5n + 3 &\leqslant c
\end{aligned}$$

As $c$ is a constant but is dependent on $n$ there is no $c$ that satisfies the inequality
$\therefore T(n)$ is not $O(n)$

### 2.2.2 Big-$\Omega$

**Example 2.2.3**

**Question 6**

Show that $f(n) = 3n^2 + 2n + 1$ is $\Omega(n^2)$

***Solution:*** Let $g(n) = n^2$. From the definition of $\Omega\left(n^2\right)$:

$$\Omega\left(g(n)\right) = cg(n) \leqslant f(n)\ , \forall n \geqslant n_0 \land n > 0$$

$$\begin{aligned} cg(n) &\leqslant f(n) \\ cn^2 &\leqslant 3n^2 + 2n + 1 \\ cn &\leqslant 3n + 2 + \frac{1}{n} \\ c &\leqslant 3 + \frac{2}{n} + \frac{1}{n^2} \\ \text{Let } n &= 1 \\ c &\leqslant 6 \end{aligned}$$

Or

$$\begin{aligned} cg(n) &\leqslant f(n) \\ cn^2 &\leqslant 3n^2 + 2n + 1 \\ &\quad \text{Make everything on the right side have a variable of } n^2 \\ cn^2 &\leqslant 3n^2 + 2n^2 + n^2 \\ cn^2 &\leqslant 6n^2 \\ c &\leqslant 6 \end{aligned}$$

### 2.2.3 Big-Θ

**Example 2.2.4**

**Question 7**

Shot that $4n^2 + 3n + 2$ is $\Theta\left(n^2\right)$

***Solution:*** Let $f(n) = 4n^2 + 3n + 2$ and $g(n) = n^2$. From the definition of $\Theta$:

$$\Theta(n) = c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

$$
\begin{aligned}
c_1 n^2 &\leqslant 4n^2 + 3n + 7 \leqslant c_2 n^2 \\
c_1 n^2 &\leqslant 4n^2 + 3n + 7 \\
c_2 n^2 &\geqslant 4n^2 + 3n + 7 \\
\\
c_1 n^2 &\leqslant 4n^2 + 3n^2 + 7n^2 \\
c_1 &\leqslant 14 \\
\\
c_2 n^2 &\geqslant 4n^2 + 3n^2 + 7n^2 \\
c_2 &\geqslant 14 \\
c_1 &= 14 \, , c_2 = 14 \\
\text{For } n_0 &= 1
\end{aligned}
$$

### 2.2.4 Establishing order of growth using limits

Using the limit:

$$\lim_{n \to \infty} \frac{T(n)}{g(n)}$$

If the function approaches:

- $0$ - Order of growth of $T(n)$ < order of growth $g(n)$ - $O$
- $k > 0$ - Order of growth of $T(n)$ = Order of growth of $g(n)$ - $\Theta$
- $\infty$ - Order of growth of $T(n) > g(n)$ - $\Omega$

**Note:-**

Trial and error also works