Algorithm Analysis

Madiba Hudson-Quansah

Contents

Chapter 1

Foundations

1.1 Assumptions of the RAM Model

- Each simple operation (+, -, *, /, =, ==, <, >, <=, >=) takes a constant amount of time.
- Each memory access takes a constant amount of time.
- Each instruction takes a constant amount of time.
- The input size is the number of bits needed to represent the input.

1.2 Analysing Algorithms

Definition 1.2.1: Time Complexity

How an algorithm's execution time or running time as a function of the input size grows as the input size grows.

Definition 1.2.2: Space Complexity

The amount of memory an algorithm needs to solve a problem as a function of the input size.

Analysing an algorithm involves prediction the resources that the algorithm will require. The resources that are of interest are:

- Memory
- Computation Time
- Bandwidth
- Energy Consumption

In analysing an algorithm represented in pseudocode we can determine how long the algorithm takes to run by examining how many times each line of the code is executed and how long each line takes to execute. To do this we first derive a precise formula for the running time, then simplify the formula using a convenient notation that allows us to compare running times of different algorithms. For this example we will use the insertion-sort algorithm with the pseudocode defined below:

Line	Cost	Times
1	c_1	n
2	c_2	$n-1$
3	c_3	$n-1$
4	c_4	$\sum_{i=2}^{n} t_i$
5	c_5	$\sum_{i=2}^{n} (t_i - 1)$
6	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7	0	$n-1$
8	c_8	$n-1$
9	0	1

Algorithm 1 Insertion-Sort (A, n)

```
1: for index = 2 to n do
2:     prevIndex := index −1
3:     nextElement := A[index]                ▷ Insert A[index] into the sorted array A[1 : index − 1]
4:     while A[prevIndex] > nextElement and prevIndex ⩾ 0 do
5:         A[prevIndex +1] := A[prevIndex]
6:         prevIndex := prevIndex −1
7:     end while
8:     A[prevIndex +1] = nextElement
9: end for
```

We first must acknowledge that the running time of an algorithm depends on the input which we will call A. Also we must acknowledge that the insertion-sort algorithm can take a different amount of time sorting two arrays of the same size depending on how already sorted the arrays are. We then describe the running time of an algorithm as a function of the size of the input n. To do so we must clearly define the terms "running time", "size of input", and be clear about what scenario we are considering, whether it be best-case, worst-case, or average-case.

The best description of the *size of input* depends on the problem being solved. In this case for a sorting algorithm the best notion of the size of input is the number of elements in the array to be sorted. The *running time* of an algorithm is the number of instructions and data accesses made by the algorithm, how this is accounted for differs from computer to computer but using the **RAM model** in which each simple operation takes a constant amount of time, we can generalize the running time of an algorithm as the number of comparisons and data movements made by the algorithm. Therefore in this framework each execution of the kth line of pseudocode takes c_k time units where c_k is a constant.

First for each $i = 2, 3, \ldots, n$, let t_i denote the number of times the `WHILE` loop test in line 5 is executed for that value of i. Below is a table of lines, the number of times each line is executed and cost of each line.

Therefore:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_8 (n-1)$$

However even for inputs of the same size the running time of an algorithm can vary depending on the input. For example the insertion-sort algorithm can take a different amount of time sorting two arrays of the same size depending on how already sorted the input arrays are. Therefore we must consider the *best-case*, in which the input is already completely sorted, the *worst-case*, in which the input is sorted in reverse order, and the *average-case*, in which the input is a random permutation of the integers $1, 2, \ldots, n$.

In the best-case the `WHILE` loop on line 4 exits every time the condition is evaluated as the next element is always greater than the previous element. Therefore the lines inside the loop are not executed and only the loop header is executed

making the time of line 4 $(n-1)$ and lines 5 and 6 0, hence $T(n)$ is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) \\ &= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 n - c_4 + c_8 n - c_8 \\ &= n(c_1 + c_2 + c_3 + c_4 + c_8) - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

This can be expressed as

$$T(n) = an + b$$

Where a and b are constants. This results in the best-case running time being a linear function of n, the size of the input.

In the worst-case, where the entries are sorted in reverse order, the function must compare each element $A[i]$ to each element in the sorted sub array $A[1 : i-1]$ (the elements already gone through), therefore the `WHILE` loop runs $i-1$ for $i = 2, 3, \ldots, n$. Noting that:

$$\begin{aligned} \sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i\right) - 1 \\ &= \frac{n(n+1)}{2} - 1 \end{aligned}$$

And

$$\begin{aligned} \sum_{i=2}^{n} (i-1) &= \sum_{i=1}^{n-1} \\ &= \frac{n(n-1)}{2} \end{aligned}$$

$T(n)$ is :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= c_1 n + c_2 n - c_2 + c_3 n - c_3 + \frac{c_4 n^2 + c_4 n}{2} - c_4 + \frac{c_5 n^2 - c_5 n}{2} + \frac{c_6 n^2 - c_6 n}{2} + c_8 n - c_8 \\ &= c_1 n + c_2 n - c_2 + c_3 n - c_3 + \frac{c_4 n^2}{2} + \frac{c_4 n}{2} + \frac{c_5 n^2}{2} - \frac{c_5 n}{2} + \frac{c_6 n^2}{2} - \frac{c_6 n}{2} + c_8 n - c_8 \\ &= n^2\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) + n\left(\frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_1 + c_2 + c_3 + c_8\right) - (c_2 + c_3 + c_8) \end{aligned}$$

This can be expressed as

$$T(n) = an^2 + bn + c$$

Where a, b, and c are constants. This results in the worst-case running time being a quadratic function of n.

Usually when analysing the running times of algorithms we are mainly interested in the worst-case running time due to three reasons:

- The worst-case running time gives an upper bound on the running time for any input.
- From some algorithms the worst-case occurs fairly often.
- The average-case is often roughly as bad as the worst-case

1.3 Order of Growth

These derived equations are complex and give us more detail than we need. What we really care about when analysing algorithms is the rate / order of growth. We therefore consider only the leading term of a formula, an^2, since the lower-order terms are insignificant for large values of n. We then ignore the leading term's coefficient, since constant factors are less significant than the growth rate in determining the efficiency of large inputs.

So for the insertion-sort's worst case $T(n) = an^2 + b + c$, we take the leading term an^2, then isolate the factor n^2. The factor n^2 then becomes the order of growth of the worst-case running time of the insertion-sort.

The order of growth of running time is denoted by Θ, making the worst-case running time of the insertion-sort $\Theta\left(n^2\right)$, and the best-case running time is $\Theta(n)$

1.4 Designing Algorithms

1.5 O-notation, Ω-notation and Θ-notation

We determined the worst-case running time of the insertion-sort algorithm to be $\Theta\left(n^2\right)$. This notation is not the only notation used to describe the running time of an algorithm.

1.5.1 O-notation

Definition 1.5.1: O-notation

Characterizes an upper-bound on the growth rate of a function. I.e. it describes that the function grows no faster than a certain rate, based on the highest order term in the function.

For example the function

$$7n^3 + 100n^2 - 20n + 6$$

Its highest order term $7n^3$ and so the function's growth rate is n^3. Because this function grows no faster than n^3, we can say that the function is $O\left(n^3\right)$, $O\left(n^4\right)$, and therefore $O\left(n^c\right)$ where $c \geqslant 3$. This is because the function grows no faster than n^3 and therefore grows no faster than n^c where $c \geqslant 3$

1.5.1.1 Formal Definition

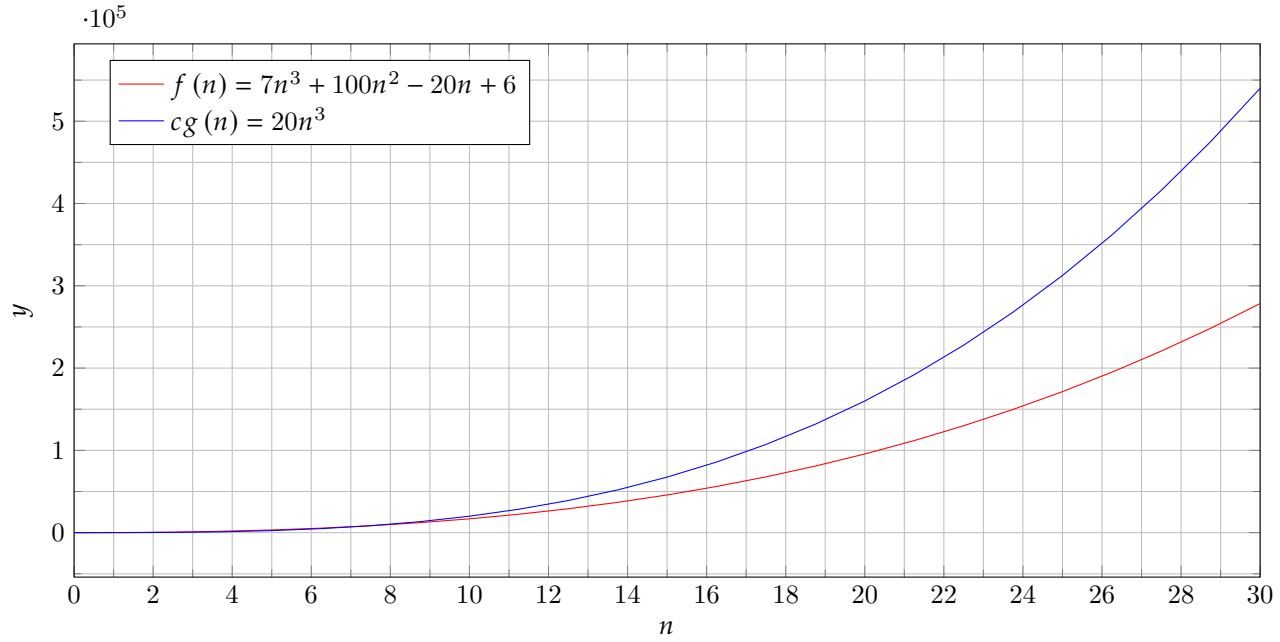
Definition 1.5.2: O-notation

For a given function $g(n)$, we denote by $O\left(g(n)\right)$ the set of functions:

$$O\left(g(n)\right) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leqslant f(n) \leqslant cg(n), \forall n, n \geqslant n_0\}$$

A function $f(n)$ belongs to the set $O\left(g(n)\right)$ if there exists a positive constant c such that $f(n) \leqslant cg(n)$ for sufficiently large n. This means that the value of function $f(n)$ will never exceed the value of $cg(n)$ for sufficiently large n, where $g(n)$ is the growth rate of the function.

For example for the function $f(n) = 7n^3 + 100n^2 - 20n + 6$, and the order of growth $g(n) = n^3$. We can say that $f(n)$ is $O\left(n^3\right)$ because when $n \geqslant 12$ and $c = 20$, $f(n) < 20n^3$



1.5.2 Ω-notation

Definition 1.5.3: Ω-notation

Characterizes a lower-bound on the growth rate of a function. I.e. it describes that the function grows no slower than a certain rate, based on the highest order term in the function.

This means for the same function

$$7n^3 + 100n^2 - 20n + 6$$

Its growth rate is n^3, and because the function grows no slower than n^3, we can say that the function is $\Omega\left(n^3\right)$, and therefore $\Omega\left(n^c\right)$, where $c \leqslant 3$

1.5.2.1 Formal Definition

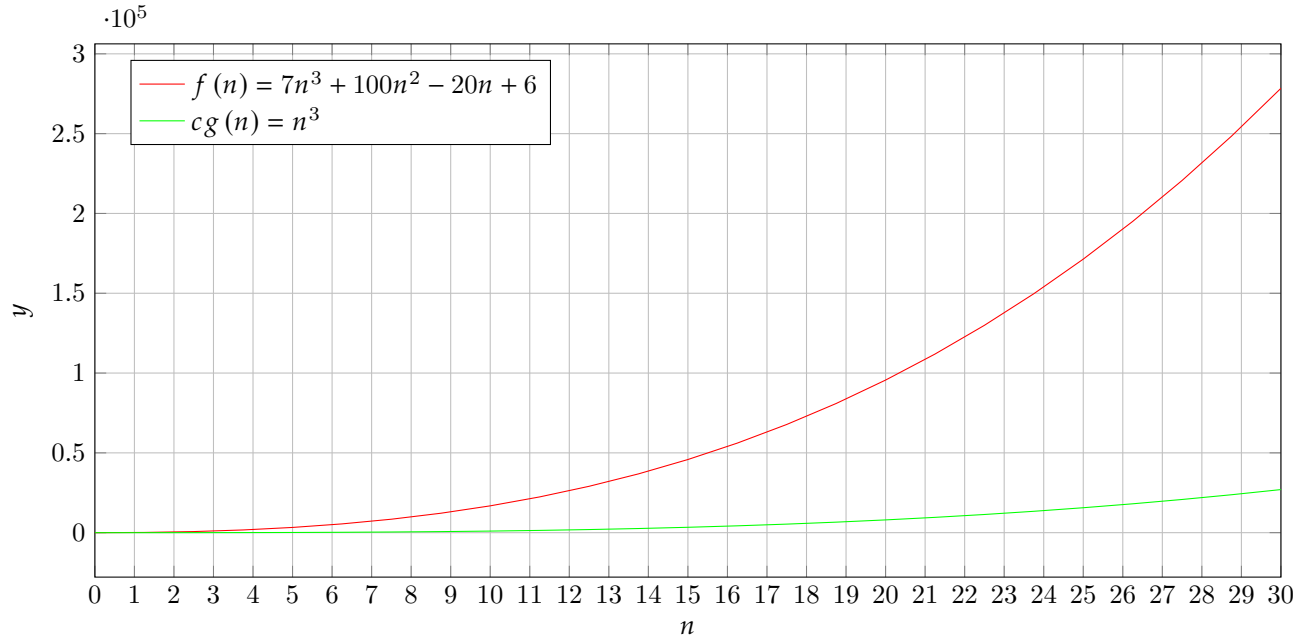
Definition 1.5.4: Ω-notation

For a given function $g(n)$, we denote by $\Omega\left(g(n)\right)$ the set of functions:

$$\Omega\left(g(n)\right) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leqslant cg(n) \leqslant f(n), \forall n, n \geqslant n_0\}$$

A function $f(n)$ belongs to the set $\Omega\left(g(n)\right)$ if there exists a positive constant c such that $cg(n) \leqslant f(n)$ for sufficiently large n. This means that the value of function $f(n)$ will never be less than the value of $cg(n)$ for sufficiently large n, where $g(n)$ is the growth rate of the function.

For example for the function $f(n) = 7n^3 + 100n^2 - 20n + 6$, and the order of growth $g(n) = n^3$. We can say that $f(n)$ is $\Omega\left(n^3\right)$ because when $n \geqslant 6$ and $c = 1$, $n^3 \leqslant f(n)$



1.5.3 Θ-notation

Definition 1.5.5: Θ-notation

Characterizes the tight bound on the growth rate of a function. I.e. it describes the precise rate at which the function grows based on the highest order term in the function.

This notation characterizes the growth rate of a function to within a constant factor from above and below. Therefore if you can show that a function is both $O(n^c)$ and $\Omega(n^c)$, then you have shown that the function is $\Theta(n^c)$. I.e for the same function

$$7n^3 + 100n^2 - 20n + 6$$

We found that the function is both $O(n^3)$ and $\Omega(n^3)$, it is also $\Theta(n^3)$

1.5.3.1 Formal Definition

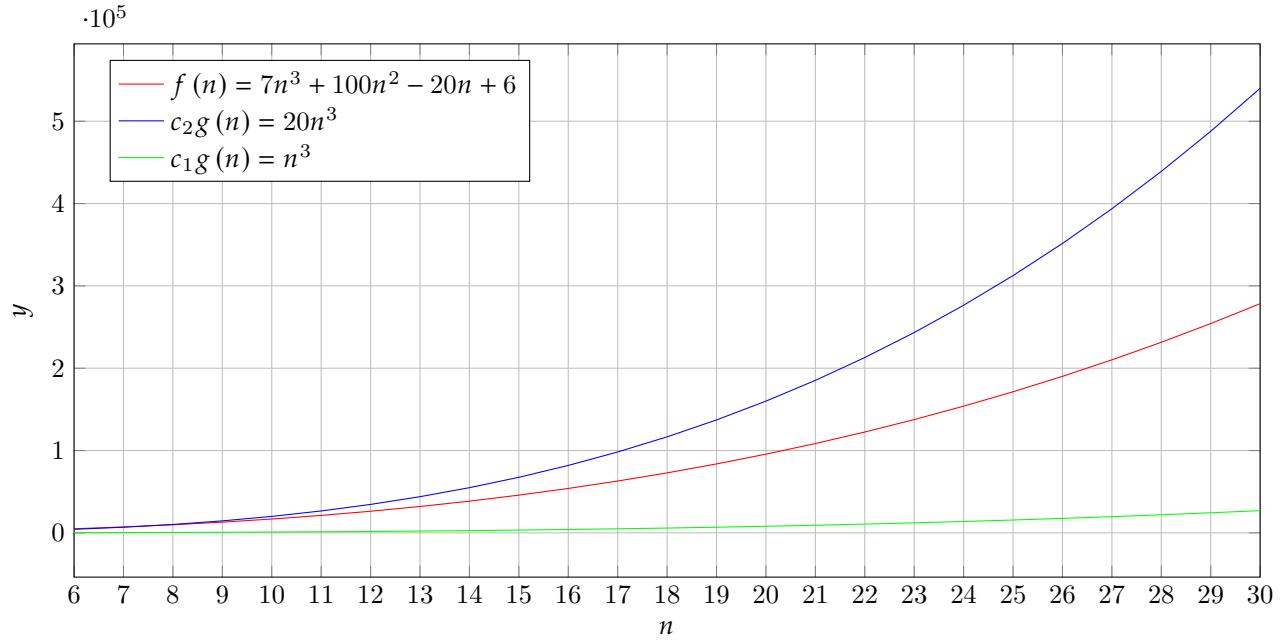
Definition 1.5.6: Θ-notation

For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions:

$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2 and n_0 such that $0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$, $\forall n, n \geqslant n_0\}$

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exists positive constants c_1 and c_2 such that $c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$ for sufficiently large n. This means that the value of function $f(n)$ will always be within a constant factor of the value of $g(n)$ for sufficiently large n, where $g(n)$ is the growth rate of the function.

For example the function $f(n) = 7n^3 + 100n^2 - 20n + 6$, and the order of growth $g(n) = n^3$. We can say that $f(n)$ is $\Theta(n^3)$ because when $n \geqslant 12$ and $c_1 = 1$ and $c_2 = 20$,



1.5.4 Determining The Running Time of an Algorithm

Going back to the insertion-sort algorithm we can better characterize its running time using asymptotic notation. Given that the procedure has nested loops, an outer for loop that always runs $n-1$ times and an inner while loop whose number of iterations depends on the data being sorted. The while loop variable j starts at $i-1$ and decreases by 1 each iteration until it reaches 0 or $A_j \leqslant$ key. This means the while loop could run 0 to $i-1$ times. This means that generally we can deduce an $O\left(n^2\right)$ running time for the insertion-sort algorithm.

In determining the worst case running time of the insertion sort we need to reiterate that the worst case occurs when the input is sorted in reverse order. For a value in a reverse sorted array to be inserted into the correct position in the sub array, it must have been moved in line 4 of the algorithm, and moved k positions. For an element to be moved k times line 4 needs to have been executed k times. In the worst case where elements need to be moved $i-1$ times where i is the current index of the element under consideration meaning at least the while loop must be executed $n-1$ times. This means that the worst case running time of the insertion sort is $\Omega\left(n^2\right)$, because for every input size n above a certain size n_0, there is at least one input of size n that takes at least cn^2 time where c is a positive constant.

Because we have shown that the insertion sort algorithm runs in $O\left(n^2\right)$ time in all cases, and that there is an input that takes $\Omega\left(n^2\right)$ time, we can conclude that the worst-case running time of the insertion-sort is $\Theta\left(n^2\right)$

1.6 Exercises

Question 1

Express the function

$$\frac{n^3}{1000} + 100n^2 - 100n + 3$$

in terms of Θ-notation

Solution: $\Theta\left(n^3\right)$

Question 2

Consider sorting n numbers stored in array $A[1:n]$ by finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the second smallest element of $A[2:n]$ and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known

as selection-sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection-sort in Θ-notation. Is the best-case running time any better?

Solution:

Algorithm 2 Selection-Sort (A, n)

```
1:  for i := 1 to n − 1 do
2:      min := A_i
3:      mI := i
4:      j := i
5:      while j < n − 1 do
6:          if A_j < min then
7:              min = A_j
8:              mI = j
9:          end if
10:         j = j +1
11:     end while
12:     Swap A_i and A_mI
13: end for
```

Chapter 2

Exercises

Question 3

Describe an algorithm that takes as input a list of n integers and finds the number of negative integers in the list

Solution:

Algorithm 3 Count-Negative (A, n)

```
1: count := 0
2: for i := 1 to n do
3:     if A[i] < 0 then
4:         count = count + 1
5:     end if
6: end for
7: return count
```

Question 4

Describe an algorithm that takes as in put a list of n integers and produces as outputs the largest difference obtained by subtracting an integer in the list from the one following it.

Solution:

Algorithm 4 Largest-Difference (A, n)

```
1: largest := A_2 − A_1
2: for i := 2 to n − 1 do
3:     diff := A_{i+1} − A_i
4:     if  diff > largest then
5:         largest = diff
6:     end if
7: end for
8: return largest
```

Question 5

Solution:

Algorithm 5 Repeated-Ints (A, n)

```
1:  j := 0
2:  i := 2
3:  while i ⩽ n do
4:      if A_i = A_{i−1} then
5:          j := j + 1
6:          B_j := A_i                          ▷ B is the list of repeated values
7:          while i ⩽ n and A_i = C_i do
8:              i := i + 1
9:          end while
10:     end if
11:     i := i + 1
12: end while
13: return B
```