COMP26120
Academic Session: 2022-23

Worksheet 2: Introducing Complexity Analysis
Model Solution

This worksheet is **not** assessed.

Learning Objectives

At the end of this worksheet, you should be able to:

1. Define asymptotic notation, functions, and running times.
2. Analyse the running time used by an algorithm via asymptotic analysis.
3. Examine and sketch examples of asymptotic analysis using fragments of C code.

Introduction

This worksheet is about analysing the complexity of algorithms using asymptotic notation. The initial questions are concerned with the understanding of the insertion sort algorithm. The remaining questions are concerned with the asymptotic analysis, including the O-, Θ-, and Ω-notations. We also provide questions to compare the running times of algorithms and to find the time complexity of code fragments.

Suggested Reading. This worksheet is based on Sections 2.1, 2.2, and 3.1 of *"Introduction to Algorithms by Cormen"*, by Leiserson, Rivest and Stein, ISBN 978-0-262-03384-8, MIT Press 2009. Section 2.1 describes the analysis of the insertion sort algorithm, while Section 2.2 introduces a generic uniprocessor, random-access machine (RAM) model of computation as our implementation technology to analyse algorithms. Section 3.1 discusses the order of growth of an algorithm running time. In particular, it looks at input sizes large enough to make only the order of growth of the running time relevant.

Exercises

Exercise 1 (Sorting). Consider the INSERTION-SORT algorithm explained during the lecture in week 2. Your task here is to illustrate the operation of the INSERTION-SORT algorithm on the array $A = \{31, 41, 59, 26, 41, 58\}$, where $n = 6$ (i.e., the number of elements). In your solution, you should output the content of array A produced on each iteration of the *for*-loop of the INSERTION-SORT algorithm as follows.

	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
1st iteration						
2nd iteration						
3rd iteration						
4th iteration						
5th iteration						
6th iteration						

Solution:

This exercise was about making sure you understood the insertion sort algorithm. If you implemented the algorithm to get the output then you've given yourself a headstart on the first Lab.

	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$
1st iteration	31	41	59	26	41	58
2nd iteration	31	41	59	26	41	58
3rd iteration	31	41	59	26	41	58
4th iteration	26	31	41	59	41	58
5th iteration	26	31	41	41	59	58
6th iteration	26	31	41	41	58	59

Exercise 2 (Sorting). Rewrite the INSERTION-SORT procedure to sort the array into non-increasing instead of non-decreasing order, as shown during the lecture in week 2. In particular, you need to modify the condition of the while-loop statement in the pseudocode to solve this question. Does the loop invariant still hold? Justify your answer.

Solution: Yes. The loop invariant $A[1..i-1]$ consists of the elements originally in $A[1..i-1]$, but in non-increasing order. The modified INSERTION-SORT procedure can be seen below.

```
procedure InsertionSort(A, n)
    for i = 2 to A.length do
        key = A[i];
        j = i - 1;
        while (j > 0) and (A[j] < key) do
            A[j+1] = A[j];
            j = j - 1
        end while
        A[j+1] = key;
    end for
end procedure
```

Exercise 3 (Asymptotic Analysis). Given $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, for each of the following statements use the O-notation to either show that the statement is valid or otherwise give a counterexample.

(i) $f_1(n).f_2(n) \in O(g_1(n).g_2(n))$

(ii) $f_1(n)^{f_2(n)} \in O(g_1(n)^{g_2(n)})$

Solution:
(i)The O-notation must be applied to $f_1(n)$ and $f_2(n)$:
$f_1(n) \leq c_1 \cdot g_1(n)$ for $n \geq n_1$ and $c_1, n, n_1 \geq 0$
$f_2(n) \leq c_2 \cdot g_2(n)$ for $n \geq n_2$ and $c_2, n, n_2 \geq 0$
It follows that $f_1(n) \cdot f_2(n) \leq c_1 \cdot c_2(g_1(n) \cdot g_2(n))$
If we replace each term of the above equation by: $f'(n) = f_1(n) \cdot f_2(n)$, $g'(n) = g_1(n) \cdot g_2(n)$, $c' = c_1 \cdot c_2$, then we get: $f'(n) \leq c' \cdot g'(n)$, which can be rewritten as $f'(n) \in O(g'(n))$. Thus, we can conclude that $f_1(n).f_2(n) \in O(g_1(n).g_2(n))$ (TRUE).

(ii) Counterexample:
$f_1(n) = n$, $g_1(n) = n$, $g_1(n) = O(n)$
$f_2(n) = 2$, $g_2(n) = 1$, $g_2(n) = O(1)$
Since we have $f_1(n)^{f_2(n)} \in O(g_1(n)^{g_2(n)})$, then, $n^2 \notin O(n^1)$ (FALSE).

Exercise 4 (Big-O notation). The statements below describe some features of the "Big-O" notation for the functions $f \equiv f(n)$ and $g \equiv g(n)$. For each statement below, *(1)* state with justification whether it is *true* or *false*; *(2)* if it is *false*, then provide the correct "Big-O" notation.

(i) $400n + 90n^2 + 1000n^3 \in O(n^4)$.

(ii) $250n + 700n^2 + n^3 \in O(n^2 logn)$.

Solution:
(i) **(TRUE). The dominant term is** n^3.

Proof. There exist positive constants c and n_0 such that $0 \leq 400n + 90n^2 + 1000n^3 \leq c \times n^4$ for all $n \geq n_0$. If we divide both terms by n^4, we obtain: $0 \leq 400/n^3 + 90/n^2 + 1000/n \leq c$. So, $c \geq 1490$ and $n_0 = 1$. □

(ii) **(FALSE).** $250n + 700n^2 + n^3 \in O(n^3)$.
We can use the formal definition of the Big-O notation to verify that $250n + 700n^2 + n^3 \neq O(n^2 logn)$. Suppose for the purpose of contradiction that c and n_0 exist such that $250n + 700n^2 + n^3 \leq c \times n^2 logn$ for all $n \geq n_0$. However, if we divide both terms by n^2 yields $250/n + 700 + n \leq c \times logn$, which cannot possibly hold for arbitrary large n since c is a constant. However, the statement $250n + 700n^2 + n^3 \in O(n^3)$ holds.

Proof. There exist positive constants c and n_0 such that $0 \leq 250n + 700n^2 + n^3 \leq c \times n^3$ for all $n \geq n_0$. If we divide both terms by n^3, we obtain: $0 \leq 250/n^2 + 700/n + 1 \leq c$. So, $c \geq 951$ and $n_0 = 1$. □

Exercise 5 (Θ-notation). Express the following functions in terms of Θ-notation:

(i) $n^3/1000 - 100n^2 - 100n + 3$
(ii) $2n^2 + 3n + 1$

Solution:
(i) $n^3/1000 - 100n^2 - 100n + 3 \in \Theta(n^3)$.

Proof. There exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 \times n^3 \leq n^3/1000 - 100n^2 - 100n + 3 \leq c_2 \times n^3$. If we divide this equation by n^3, we obtain: $0 \leq c_1 \leq 1/1000 - 100/n - 100/n^2 + 3/n^3 \leq c_2$. For sufficiently large n, the term $1/1000$ is kept. So, $c_2 \geq 1/1000$. $n = 100001$ is the smallest value for c_1 to be a positive constant. □

(ii) $2n^2 + 3n + 1 \in \Theta(n^2)$.

Proof. There exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 \times n^2 \leq 2n^2 + 3n + 1 \leq c_2 \times n^2$. If we divide this equation by n^2, we obtain: $0 \leq c_1 \leq 2 + 3/n + 1/n^2 \leq c_2$. For sufficiently large n, the term 2 is kept. So, $c_2 \geq 2$. $n = 0$ is the smallest value for c_1 to be a positive constant. □

Exercise 6 (Asymptotic notation). We can extend aysmptotic notation to the case of two parameters n and m that can go to infinity independently at different rates. For a given function $g(n, m)$, we denote by $O(g(n, m))$ the set of functions as follows:

$$O(g(n,m)) = \left\{ f(n,m) : \begin{array}{l} \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that} \\ 0 \leq f(n,m) \leq cg(n,m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0 \end{array} \right\}$$

Give corresponding definitions for $\Omega(g(n, m))$ and $\Theta(g(n, m))$.

Solution:

$$\Omega(g(n,m)) = \left\{ f(n,m) : \begin{array}{l} \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that} \\ f(n,m) \geq cg(n,m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0 \end{array} \right\}$$

$$\Theta(g(n,m)) = \left\{ f(n,m) : \begin{array}{l} \text{there exist positive constants } c_1, c_2, n_0, \text{ and } m_0 \text{ such that} \\ c_1 g(n,m) \leq f(n,m) \leq c_2 g(n,m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0 \end{array} \right\}$$

Exercise 7 (Asymptotic notation). Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

Solution:
$0 \leq 2^{n+1} \leq c \times 2^n$ for all $n \geq 0$, so $2^{n+1} = O(2^n)$.
However, 2^{2n} is not $O(2^n)$. If it were, there would exist n_0 and c such that $n \geq n_0$ implies $2^n \times 2^n = 2^{2n} \leq c^{2n}$, so $2^n \leq c$ for $n \geq n_0$, which is clearly impossible since c must be a constant.

Exercise 8 (Comparison of running time). For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds. For example, for $t = 1\,\text{s}$ and $f(n) = log\, n$, we obtain $log\, n = 1 \times 10^6$; therefore, $n = 2^{1\times 10^6}$. For $\sqrt{n}$, we obtain $\sqrt{n} = 1 \times 10^6$; therefore, $n = 2^{1\times 10^{12}}$. For n, we obtain $n = 1 \times 10^6$. For $n\, log\, n$, we obtain $n\, log\, n = 1 \times 10^6$; therefore, $n = 62746$. For n^2, we obtain $n^2 = 1 \times 10^6$; therefore, $n = 1000$. For n^3, we obtain $n^3 = 1 \times 10^6$; therefore, $n = 100$. For 2^n, we obtain $2^n = 1 \times 10^6$; therefore, if we apply log to both terms, we get $n = 6/log2 = 19$. Lastly, for $n!$, we obtain $n! = 1 \times 10^6$; therefore, $n = 9$.

t	$log\, n$	$\sqrt{n}$	n	$n\, log\, n$	n^2	n^3	2^n	$n!$
1 second	$2^{1\times 10^6}$	$2^{1\times 10^{12}}$	$2^{1\times 10^6}$	62746	1000	100	19	9
1 minute								
1 hour								

Solution:

	$log\, n$	$\sqrt{n}$	n	$n\, log\, n$	n^2	n^3	2^n	$n!$
1 second	$2^{1\times 10^6}$	1×10^{12}	1×10^6	62746	1000	100	19	9
1 minute	$2^{6\times 10^7}$	3.6×10^{15}	6×10^7	2801417	7745	391	25	11
1 hour	$2^{3.6\times 10^9}$	1.29×10^{19}	3.6×10^9	133378058	60000	1532	31	12

Exercise 9 (Comparison of running time). John is working as a software developer at Boost-Code UK Ltd. He needs to compare two implementations of sorting algorithms on the same machine, which will be later integrated into a software product. In particular, for inputs defined by size n, the first algorithm denoted by A runs in $4n^2$, while the second algorithm denoted by B runs in $128\, n\, lg_2\, n$. John needs to compute values of n where algorithm A beats algorithm B so that he can make the best choice of which one will be integrated into the product. For which values of n will algorithm A run faster than algorithm B?

Solution: We wish to determine for which values of n the inequality $4n^2 < 128\, n\, log_2\,(n)$ holds. This happens when $n < 32\, log_2\,(n)$, or when $n \leq 255$. In other words, algorithm A runs faster than algorithm B when we're sorting at most 255 items; otherwise algorithm B is faster once we have $n \geq 257$ since both algorithms have the same running time when $n = 256$.

Exercise 10 (Time complexity of code). What is the Big-O time complexity of the following code fragment?

```
for( int i = n; i > 0; i /= 2 ) {
  for( int j = 1; j < n; j *= 2 ) {
    for( int k = 0; k < n; k += 2 ) {
      ... // constant number of operations
    }
  }
}
```

Solution: The running time of the inner, middle, and outer loop can be represented by n, $log\, n$, and $log\, n$, respectively. As a result, the overall complexity can be denoted as $O(n(log\, n)^2)$ since all three loops of this C code fragment are nested.

Exercise 11 (Time complexity of code). Given an array a that contains n integer values, a method *randomInt* that takes a constant number c of computational steps to produce a random integer number, and a sort method that takes $n \log n$ computational steps to sort the array. What is the Big-O time complexity for the following code fragment?

```
for( int i = 0; i < n; i++ ) {
  for( int j = 0; j < n; j++ ) {
    a[j] = randomInt(i);
  }
  sort(a);
}
```

Solution: The complexity of this code fragment can be determined as: (1) the first and second successive innermost loops have $O(n)$ and $O(n \log n)$ complexity, respectively; (2) so the overall complexity is $O(n^2 \log n)$.