COMP3506/7505 Week 2 Tutorial

Asymptotic Analysis & Algorithm Analysis I

Tutorial exercises are designed to be discussed collaboratively. You will have some time during tutorial sessions to attempt the questions, however it is **strongly** recommended you attempt them beforehand. Answers will be provided at the end of the week for most of the questions but are not guaranteed to cover the solution in a suitable level of detail. Tutors will try to provide much more detailed solutions during tutorials. Attendance at tutorials is not mandatory, however skipping tutorials may result in you being significantly disadvantaged for course assessment items.

1. Discuss the difference between *worst-case* and *best-case* behaviour, and describe this difference using one or more concrete examples.

2. Given two functions $f(n)$ and $g(n)$, discuss the difference between the following statements:

   - $f(n) \in \mathcal{O}(g(n))$;
   - $f(n) \in \Omega(g(n))$; and
   - $f(n) \in \Theta(g(n))$.

3. A polynomial of degree k is a function given by $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, where $a_0, ..., a_k$ are constants. Assume $a_k > 0$ for the following questions.

   (a) Give a tight Big-O bound on $f(n)$ using the simplification rules given in lectures.

   (b) Give a tight Big-Ω bound on $f(n)$ using the simplification rules given in lectures.

   (c) Does a Big-Θ bound exist for $f(n)$? If so, explain why it exists, and then provide it.

4. Recall that $f(n)$ is $O(g(n))$ (f is bounded from above by g) if there exist positive c and n_0 such that $f(n) \leq c\, g(n)$ for all $n \geq n_0$. Using this definition, find and prove big-O bounds for the following functions:

$$3 + 2\log_2 n, \qquad (n+1)(n-1), \qquad \sqrt{9n^5 - 5}.$$

5. Using previous question or otherwise, find big-O bounds for these functions (you are not required to prove these):

$$2^n + n^2, \qquad \log_2(2^n \cdot n^2).$$

6. In algorithm analysis, we often omit the base of logarithms for convenience. This question will prove this is a mathematically valid thing to do.

   Using the fact that $\log_a(x) = \log_b(x)/\log_b(a)$ for all $a, b > 1$, prove that $\log_a(n)$ is $O(\log_b(n))$.

7. Show that $f(n) = n$ is *not* $O(\log n)$ by proving no c and n_0 can satisfy the definition.

8. Recall that $f(n)$ is $\Omega(g(n))$ (f is bounded from below by g) if there exist c and n_0 such that $f(n) \geq c\, g(n)$ for $n \geq n_0$.

   Prove that any strictly positive and increasing function is $\Omega(1)$. Why is this not useful in algorithm analysis?

9. For simple mathematical functions (e.g. polynomial, exponential), it is easy to see that the big-O and big-Ω bounds coincide. However, algorithms often behave in more interesting ways.

   Give an example of a function $f(n)$ where the tightest big-O and big-Ω bounds are not the same. (Hint: consider piecewise functions.)

10. Let $f(n)$ and $g(n)$ be positive functions of n. Prove that if $f(n) \in O(g(n))$ then $g(n) \in \Omega(f(n))$.

11. Matt and Kenton are arguing about the performance of their sorting algorithms. Matt claims that his $\Theta(n \log n)$-time algorithm is always faster than Kenton's $\Theta(n^2)$ time algorithm. To settle the issue, they implement and run the two algorithms on many randomly generated data sets. To Matt's dismay, they find that if $n < 10000$, then Kenton's $\Theta(n^2)$-time algorithm actually runs faster, and only when $n \geq 10000$ is the $\Theta(n \log n)$-time algorithm better again. Explain why this scenario is possible.

12. Determine whether the following statements hold for *all* possible functions $f(n)$, $g(n)$, and $h(n)$. If true, explain why using the mathematical definitions of big-Ω and big-Θ. If false, provide a counterexample. (This question appeared in 2019's final exam.)

    (a) If $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n))$, then $f(n)$ is $\Omega(h(n))$.

    (b) If $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(f(n))$, then $f(n)$ is $\Theta(g(n))$.

13. For each of these questions, briefly explain your answer. (This question is from Skiena's *Algorithm Design Manual*.)

    (a) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on *some* inputs?

    (b) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on *all* inputs?

    (c) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on *some* inputs?

    (d) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on *all* inputs?

14. Suppose we have keys in the range $0, \ldots, m-1$. Each key has an associated element or null value. To store this, we will use an array with m cells and n non-null elements.

    For example, the following array has $m = 16$ and $n = 3$:

    $$[\mathbf{1}, \mathit{null}, \mathit{null}, \mathit{null}, \mathbf{2}, \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null}, \mathbf{3}].$$

    (a) What is the memory usage of this data structure (in big-O notation)?

    (b) Algorithm A executes an $O(\log n)$-time computation for each cell in this data structure. What is the worst-case running time (in big-O notation) of algorithm A?

    (c) What are some potential disadvantages of this data structure? Consider if we wanted to store elements bigger than integers.

    (d) Give an example of an alternative data structure for this purpose. When would this be better or worse than a sparse array?

15. Here is a function which searches an array for a particular element.

```
Algorithm arrayFind(x, A, n)
    Input: an element x and an n-element array, A
    Output: the first index i such that x = A[i] or −1 if x does not appear in A

    i ← 0
    while i < n do
        if x = A[i] then
            return i
        else
            i ← i + 1
    return −1
```

    This is used within another function, *matrixFind*, to find an element x within an $n \times n$ matrix B. *matrixFind* iterates over the rows of B, calling *arrayFind* (above) on each row until x is found or it has searched all rows.

    (a) How many primitive operations are required to compute *arrayFind*$(1, [10, 1], 2)$?

    (b) What are the best-case and worst-case running times of *arrayFind*? Give an example of A, n, and x for each.

    (c) What is the worst-case running time of *matrixFind* in terms of n?

(d) What is the worst-case running time of *matrixFind* in terms of N, where N is the total size of B?

(e) Considering (c) and (d), would it be correct to say that *matrixFind* is a linear-time algorithm? Briefly explain why or why not.

16. Consider the following algorithm, *arraySigma*, which performs some operation on an array of integers. Note that it returns a value and also modifies the array.

Algorithm *arraySigma*(A, n)
Input: an non-empty array A of length n
Output: returns an integer and modifies A

```
i ← n − 1
while i ≥ 0 do
    j ← 0
    x ← 0
    while j ≤ i do
        x ← x + A[j]
        j ← j + 1
    A[i] ← x
    i ← i − 1
return A[n − 1]
```

(a) The return value is some simple function of the array items. What is the returned value?

(b) What is the worst-case running time of *arraySigma* in big-O notation? Similarly, what is the best-case running time in big-Ω notation?

(c) Work through some examples by hand. What is contained in the array A after the function has executed?

(d) With (c) in mind, devise a more efficient algorithm which is functionally identical to *arraySigma* (i.e. returns the same value and modifies the array in the same way).