Algorithms and Data Structures

Jonah Benedicto

September 2025

Introduction

Contents

Chapter 1

Introduction

1.1 Algorithm

An algorithm is a step-by-step set of instructions used to solve a problem or perform a computation.

1.2 Data Structure

A data structure is a format for organising, storing and accessing data to be used for efficient processing, retrieval, and manipulation.

1.3 Abstract Data Type

An abstract data type is a model that defines the way data is processed, retrieved and manipulated without specifying the way it is organised, stored and accessed.

Chapter 2

Algorithms and Analysis

2.1 Experimental (Empirical) Analysis

Experimental (empirical) analysis is the process of evaluating an algorithm's efficiency by designing, implementing, and testing it in practice. This approach empirically measures time complexity (execution time) and space complexity (memory usage) in relation to the size of its inputs.

2.2 Theoretical Analysis

Theoretical analysis is the process of evaluating an algorithm's efficiency by mathematically determining its time complexity and space complexity in relation to input size, without implementation.

2.3 Theoretical Analysis Steps

1. Write the algorithm in pseudocode.
2. Determine the number of primitive operations executed.
3. Express the algorithm as a function of the input size. f(n)
4. Express the function in asymptotic notation through asymptotic analysis.

2.4 Asymptotic Analysis

Asymptotic analysis is the process of evaluating an algorithm's efficiency by classifying the asymptotic bound of its time complexity and space complexity functions.

2.5 Asymptotic Notation

There are three primary types of asymptotic notation:

- Big-O Notation - Describes an asymptotic upper bound on a function. Given functions $f(n)$ and $g(n)$. A function $f(n)$ is in $O(g(n))$ if there exists a positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- Big-Omega Notation - Describes an asymptotic lower bound on a function. Given functions $f(n)$ and $g(n)$. A function $f(n)$ is in $\Omega(g(n))$ if there exists a positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
- Big-Theta Notation - Describes an asymptotic tight bound on a function. Given functions $f(n)$ and $g(n)$. A function $f(n)$ is in $\Theta(g(n))$ if there exists a positive constant c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

2.6 Fundamental Functions

There are eight fundamental functions in asymptotic analysis:

- Constant: 1
- Logarithmic: $\log_2 n$
- Linear: n
- N-Log-N: $n \log_2 n$
- Quadratic: n^2
- Cubic: n^3
- Exponential: 2^n
- Factorial: $n!$

2.7 Best-Case, Worst-Case, Average-Case

There are three types of cases:

- Worst-case - The minimum time complexity or space complexity an algorithm takes to complete.
- Average-case - The average time complexity or space complexity an algorithm takes to complete.
- Best-case - The maximum time complexity or space complexity an algorithm takes to complete.

2.8 Problems

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:

   - $f(n) \in O(g(n))$;
   - $f(n) \in \Omega(g(n))$;
   - $f(n) \in \Theta(g(n))$.

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 $a_0, \cdots, a_k$ 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 the lectures.

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

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

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

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

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

$$2^n + n^2, \quad \log_2 2^n \cdot n^2$$

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 $O(\log_2(n))$ 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 $f(n) \geq cg(n)$ for $n \geq n_0$. Prove that for 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-Omega 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-Omega 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 $\Theta(n \log_2(n))$-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 $\Theta(n \log_2(n))$-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-Omega and big-Theta. If false, provide a counterexample.

    (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.

    (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 $0, \cdots, m-1$. 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 $n = 3$:

    $$[1, null, null, null, 2, null, null, null, null, null, null, null, null, null, null, 3]$$

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

    (b) Algorithm A executes an $O(\log_2(n))$-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 1 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

```
1: i ← 0
2: while i < n do
3:    if x = A[i] then
4:       return i
5:    else
6:       i ← i + 1
7:    end if
8: end while
9: 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, 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

(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-Omega notation?

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

Algorithm 2 arraySigma(A, n)

Input: A non-empty array A of length n
Output: Returns an integer and modifies A

```
1: i ← n − 1
2: while i ≥ 0 do
3:     j ← 0
4:     x ← 0
5:     while j ≤ i do
6:         x ← x + A[j]
7:         j ← j + 1
8:     end while
9:     A[i] ← x
10:    i ← i − 1
11: end while
12: return A[n − 1]
```

(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).

2.9 Solutions

1. The worst-case is the maximum time complexity or space complexity an algorithm takes to complete and the best-case is the minimum time complexity or space complexity an algorithm takes to complete. For example, consider a linear search algorithm through an unordered list. The worst-case would be the case that the target does not exist and the best-case would be the case that the target is the first element.

2.
   - The statement $f(n) \in O(g(n))$ means that the function $f(n)$ has an asymptotic upper bound $g(n)$.
   - The statement $f(n) \in \Omega(g(n))$ means that the function $f(n)$ has an asymptotic lower bound $g(n)$.
   - The statement $f(n) \in \Theta(g(n))$ means that the function $f(n)$ has an asymptotic tight bound $g(n)$.

3. (a) $O(n^k)$
   (b) $\Omega(n^k)$
   (c) Yes, the Big-Theta bound for f(n) exists since the Big-O bound for f(n) and the Big-Omega bound for f(n) are the same. $\Theta(n^k)$

4. $f(n) = 3 + 2\log_2 n$ is $O(\log_2 n)$.

$$\begin{aligned} f(n) &<= cg(n) \\ 3 + 2\log_2 n &<= 3\log_2 n + 2\log_2 n \\ 3 + 2\log_2 n &<= 5\log_2 n \\ c = 5, \quad n_0 &= 2 \quad \square \end{aligned}$$

   $f(n) = (n+1)(n-1)$ is $O(n^2)$

$$\begin{aligned} f(n) &<= cg(n) \\ (n+1)(n-1) &<= n^2 \\ n^2 - 1 &<= n^2 \\ c = 1, \quad n_0 &= 1 \quad \square \end{aligned}$$

   $f(n) = \sqrt{9n^5 - 5}$ is $O(\sqrt{n^5})$

$$\begin{aligned} f(n) &<= cg(n) \\ \sqrt{9n^5 - 5} &<= \sqrt{9n^5} \\ \sqrt{9n^5 - 5} &<= 3\sqrt{n^5} \\ c = 3, \quad n_0 &= 1 \quad \square \end{aligned}$$

5. $2^n + n^2$ is $O(2^n)$

$$\begin{aligned} log_2(2^n \cdot n^2) &= \log_2 2^n + \log_2 n^2 \\ &= n + 2 \end{aligned}$$

$\log_2(2^n \cdot n^2)$ is $O(n)$

6. $f(n) = log_a(n)$ is $O(log_b(n))$

$$\begin{aligned} f(n) &<= cg(n) \\ log_a n = \log_b(n)/\log_b(a) &<= \log_b(n) \\ c = 1, \quad n_0 = b & \quad \square \end{aligned}$$

7. $f(n) = n$ is not in $O(\log_2(n))$

$$\begin{aligned} f(n) &<= cg(n) \\ n &<= c\log_2(n) \\ \frac{n}{\log_2 n} &<= c \end{aligned}$$

There does not exist a c and n_0.

8.

$$\begin{aligned} f(n) &\geq cg(n) \\ f(n) &\geq f(1) \\ c = 1, \quad n_0 = 1 & \quad \square \end{aligned}$$

It is not useful because stating that an algorithm is $\Omega(1)$ is essentially equivalent to saying that any algorithm requires at least one operation to run. While this is true, it provides no meaningful information about the algorithm's time complexity or space complexity. In other words, the bound $\Omega(n)$ is trivial, since it holds for virtually all algorithms, and therefore does not help us distinguish between different time complexities.

Chapter 3

Recursive and Sorting Algorithms

3.1 Recursion

Recursion is when a function calls itself.

3.1.1 Linear Recursion

Linear recursion is when a function calls a single recursive call.

3.1.2 Tail Recursion

Tail recursion is when a function calls a recursive call as the last step.

3.1.3 Binary Recursion

Binary recursion is when a function calls two recursive calls.

3.1.4 Multiple Recursion

Multiple recursion is when a function calls multiple recursive calls.

3.1.5 Base Case

The condition where the function stops calling itself.

3.1.6 Recursive Case

The condition where the function calls itself.

3.2 Divide and Conquer Algorithms

Divide and conquer algorithms is an algorithm that divide (break the problem into subproblems), conquer (solves the subproblems) and combine (merge the solutions to the subproblems to solve the problem). These algorithms include: binary search, merge sort and quick sort.

3.3 Search Algorithms

A search algorithm is an algorithm used to find an item from a collection items.

3.3.1 Linear Search

A linear search is a search algorithm used to find an item by checking a collection of items one by one.
Time Complexity: $O(n)$

3.3.2 Binary Search

A binary search is a search algorithm used to find an item by checking halves of a sorted collection of items.
Time Complexity: $O(\log_2(n))$

3.4 Sorting Algorithms

A sorting algorithm is an algorithm used to sort a collection of items.

3.4.1 Selection Sort

Selection sort is a sorting algorithm that repeatedly finds the smallest (or greatest) item from the unsorted part of the collection and swaps it with the first item in the unsorted part of the collection until the collection is sorted.
Time Complexity:

- Best-case: $O(n^2)$
- Average-case: $O(n^2)$
- Worst-case: $O(n^2)$

Space-Complexity:

- In-place: $O(1)$

3.4.2 Insertion Sort

Insertion sort is a sorting algorithm that repeatedly gets the first item in the unsorted collection and finds and inserts it into the correct position in the sorted part of the collection until the collection is sorted.
Time Complexity:

- Best-case: $O(n)$
- Average-case: $O(n^2)$
- Worst-case: $O(n^2)$

Space Complexity:

- In-place: $O(1)$
- Not in-place: $O(n)$

3.4.3 Merge Sort

Merge sort is a sorting algorithm that recursively splits a collection into halves, sorts each half, and merges them back until the collection is sorted.
Time Complexity:

- Best-case: $O(n \log_2 n)$
- Average-case: $O(n \log_2 n)$
- Worst-case: $O(n \log_2 n)$

Space Complexity:

- Not in-place: $O(n)$

3.4.4 Quick Sort

Quick sort is a sorting algorithm that recursively partitions a collection around a random pivot where left side is less than pivot and right side is greater than pivot until the collection is sorted.
Time Complexity:

- Best-case: $O(n \log_2(n))$
- Average-case: $O(n \log_2(n))$
- Worst-case: $O(n^2)$

Space Complexity:

- In-place: $O(\log_2(n))$
- Not in-place: $O(n)$

3.5 Problems

3.6 Solutions

Chapter 4

Linear Data Structures and Amortisation

4.1 Array

An array is a data structure that stores a collection of elements in contiguous memory locations.
Time Complexity:

- Access at index: $O(1)$
- Insert at the end: $O(1)$
- Insert at the front: $O(n)$
- Insert in the middle: $O(n)$
- Delete at the end: $O(1)$
- Delete at the front: $O(n)$
- Delete at the middle: $O(n)$
- Modify at index: $O(1)$

Space Complexity:

- $O(n)$

4.2 Singly Linked List

A singly linked list is a data structure consisting of a sequence of nodes, where each node contains a value, and a pointer to the next node in the sequence. The structure allows for traversal forward but not backward. The first node is called the head and does not have a previous node and thus has a pointer to null.
Time Complexity:

- Access head: $O(1)$
- Remove head: $O(1)$
- Insert head: $O(1)$
- Modify head: $O(1)$
- Access index: $O(n)$
- Remove index: $O(n)$
- Insert index: $O(n)$
- Modify index: $O(n)$

Space Complexity:

- $O(n)$

4.3 Doubly Linked List

A singly linked list is a data structure consisting of a sequences of nodes, where each node contains a value, and two pointers: one to the next node and one to the previous node. The structure allows for traversal forward and backward. The first node is called the head and does not have a previous node and thus has a pointer to null. The last node is called the tail and does not have a next node and thus has a pointer to null.
Time Complexity:

- Access head: $O(1)$
- Remove head: $O(1)$
- Insert head: $O(1)$
- Modify head: $O(1)$
- Access tail: $O(1)$
- Remove tail: $O(1)$
- Insert tail: $O(1)$
- Modify tail: $O(1)$
- Access index: $O(n)$
- Remove index: $O(n)$
- Insert index: $O(n)$
- Modify index: $O(n)$

Space Complexity:

- $O(n)$

4.4 Circularly Linked List

A circularly linked list is a data structure consisting of a sequence of nodes, where each node contains a value, and two pointers: one to the next node and one to the previous node. Similar to a Doubly Linked List, but the last node points back to the first node and the first node points back to the last node.
Time Complexity:

- Access head: $O(1)$
- Remove head: $O(1)$
- Insert head: $O(1)$
- Modify head: $O(1)$
- Access tail: $O(1)$
- Remove tail: $O(1)$
- Insert tail: $O(1)$
- Modify tail: $O(1)$
- Access index: $O(n)$
- Remove index: $O(n)$
- Insert index: $O(n)$
- Modify index: $O(n)$

Space Complexity:

- $O(n)$

4.5 Amortisation

Amortisation is a method for evaluating the efficiency of an algorithm by averaging the cost of a sequence of operations over time, rather than focusing on the worst-case cost of operations.

4.6 Problems

4.7 Solutions