CM2035: Algorithms & Data Structures II

Summary

Arjun Muralidharan

1st July 2021

Contents

List of Figures

List of Tables

List of Listings

1 Analysis of Algorithms

Learning Outcomes

- ✓ Determine time and memory consumption of an algorithm described using pseudocode
- ✓ Determine the growth function of the running time or memory consumption of an algorithm
- ✓ Use Big-O, Ω and Θ notations to describe the running time or memory consumption of an algorithm

1.1 Introduction to Analysis of Algorithms

We generally assume a generic one-processor, ***random-access machine (RAM)***, where instructions are executed one after the other with no concurrent operations.

The RAM model contains instructions commonly found in real computers, including arithmetic, data movement and control, all of which take a constant amount of time. We also assume that that integers generally require around $c \log_2 n$ bits to store (e.g. the number 8 takes roughly $\log_2 8 = 3$ bits to store - it really needs 4 bits) with a constant word size $c \geq 1$ so it does not grow arbitrarily. For example, if the inputs are numbers with 5 digits each, we will require, roughly, at least $5 \log_2 n$ bits.

Other instructions, such as exponentiation, are not considered constant-time instructions, with the exception of multiplication with 2^k, which is essentially just shifting a number left by k digits.

Algorithms can be analyzed along the following aspects:

1. Correctness
2. Ease of understanding
3. Resource consumption

We analyze the resource consumption of algorithms in terms of **input size** and **running time**.

Input Size Generally the number of items in the input, for example the size of an array for sorting. In certain problems it's more appropriate the consider the total number of bits needed to represent the input in memory.

Running Time The number of primitive operations or "steps" executed. We assume that each line i of our pseudocode takes c_i time to execute. The total running time is the sum of the products of *cost per statement* c_i and *executions per statement* n_i.

$$T(n) = \sum_{i=1}^{n} c_i n_i$$

These aspects can be analyzed using either **empirical measurement** or **theoretical estimation** — the latter requires an assumption with regards to the machine model, e.g. the RAM model.

	Pro	**Con**
Empirical Measurement	Real result and no need for manual calculations	Machine-dependent results and higher implementation effort
Theoretical Estimation	Universal results and no implementation effort	Approximate results and higher manual calculation effort

Table 1. *Advantages and Drawbacks of Analysis Approaches*

When we analyse an algorithm, we can count up space and time units. In Algorithm 1, the time units taken for execution is 16, counting every memory access, memory write (assignment), logical comparison and control statement once. The space taken is 1, given that only one new variable is created in this algorithm (a).

Algorithm 1 Counting up time and space

```
function F1($a$,$b$,$c$)
    $max = a$                // 1 read, 1 write
    if $b > max$ then        // 2 read, 1 logical comparison, 1 control statement
        $max = b$            // 1 read, 1 write
    end if
    if $c > max$ then        // 2 read, 1 logical comparison, 1 control statement
        $max = c$            // 1 read, 1 write
    end if
    return $max$             // 1 read, 1 control statement
end function
```

An algorithm using non-simple operations such as loops is shown in Algorithm 2. The overall sum of the time units can be calculated as

$$1 + (N + 1) * 4 + 5N + 3N + 1 = 12N + 6$$

The algorithm takes only 1 space unit as we are only creating one new variable (i).

Algorithm 2 Counting up time and space

```
function F2($A, N, x$)
    for $0 \leq i \leq N$ do          // 1 write ($i = 0$), $(N + 1)*$(1 control, 1 logical, 2 read)
        if $A[i] == x$ then           // $N*$ (3 read, 1 logical, 1 control)
            return $i$                // does not execute in the worst case
        end if
    end for                           // $N*$ (1 read, 1 numerical op., 1 write)
    return -1                         // 1 control (no read as we return a fixed number)
end function
```

Note: Be mindful of the loop's comparison. If the loop executes until the counter is smaller than some number N, the loop will execute it's comparison $N + 1$ times and it's afterthought $(i + +)$ N times, as shown in Algorithm 2. However, if the loop executes until the counter is *equal* to some number $(i \leq N)$, then the loop comparison executes $N + 2$ times, and the loop afterthought executes $N + 1$ times.

When dealing with *while* loops, first determine how often the condition will be checked, and how often the loop will be executed. This will determine the time units used for running the loop, and then running the loop instructions.

1.2 Growth of Functions

Because the RAM model makes simplifying assumptions about a computer, we don't need the precision we get when counting time and space units. Instead, it is sufficient to observe how the running time grows as a function of the input size. Various functions are shown in increasing order of time complexity in Table 2. The graphs of various functions is shown in Figure 1.

Type	Notation
Constant	$\mathcal{O}(1)$
Logarithmic	$\mathcal{O}(\log(n)) = \mathcal{O}(\log(n^c))$
Polylogarithmic	$\mathcal{O}((\log(n))^c)$
Linear	$\mathcal{O}(n)$
Quadratic	$\mathcal{O}(n^2)$
Polynomial	$\mathcal{O}(n^c)$
Exponential	$\mathcal{O}(c^n)$

Table 2. *Growth of functions*

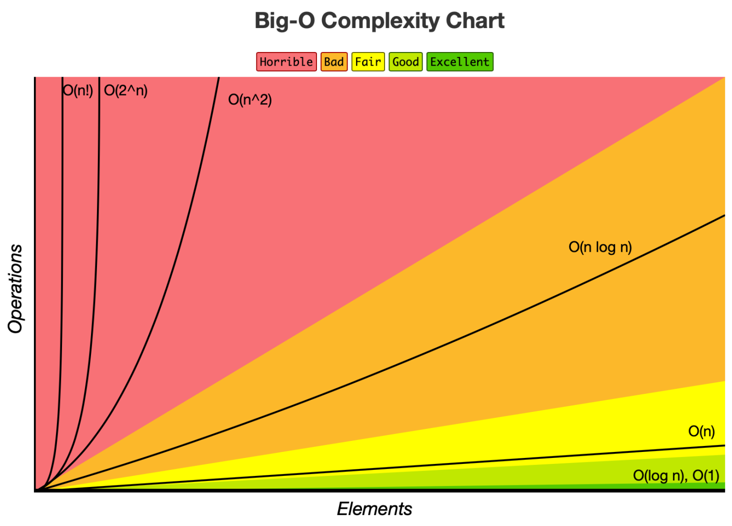

Figure 1. *Graphs of functions*

Investing time in faster algorithms is generally preferable to investing in faster computing infrastructure, because algorithms might run faster on faster algorithms, but they still grow at the same rate relative to the input size. Much greater gains can be made by moving to an algorithm of a different order of growth.

Some algorithms, like a procedure to find the largest element in an array, have identical *best* and *worst* case running times. In other cases, we are interested in the *worst-case* running time, as this gives us an upper bound of the time units taken to execute an algorithm. These cases arise then the *content* of the input data changes the growth function in addition to the input size.

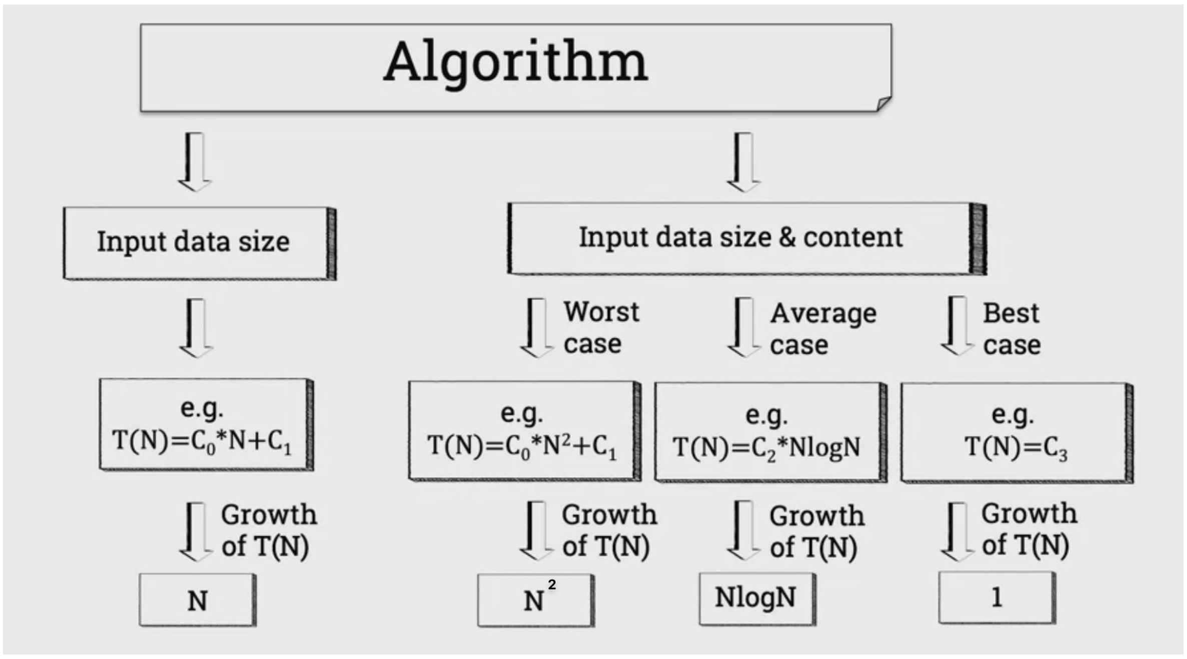

Figure 2. *Analyzing content in addition to input size*

1.3 Asymptotic Analysis

Let M be a RAM machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f : N \to N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n . If $f(n)$ is the running time of M , we say that M runs in time $f(n)$ and that M is an $f(n)$ time random access machine. Customarily we use n to represent the length of the input.

1.3.1 Big-O Notation

The **running time** of an algorithm on a particular input is the number of steps executed. Machine-independently, we define "time-steps". The total running time of an algorithm is defined as

$$\text{time cost per statement} \times \text{no. of executions per statement} = \text{total running time}$$

Function growth can be described and compared using **Big-O Notation**. In *asymptotic analysis*, we can consider only the highest order terms of a function and ignore the constants. For example, given the function

$$f(n) = 6n^3 + 2n^2 + 20n + 45$$

then the *asymptotic* or *Big-O* notation is given as

$$f(n) = \mathcal{O}(n^3)$$

A table of common functions and their respective Big-O notation is shown in Table 2 and sorted by growth rate from slowest to fastest.

Type	**Notation**
Constant	$\mathcal{O}(1)$
Logarithmic	$\mathcal{O}(\log(n)) = \mathcal{O}(\log(n^c))$
Polylogarithmic	$\mathcal{O}((\log(n))^c)$
Linear	$\mathcal{O}(n)$
Quadratic	$\mathcal{O}(n^2)$
Polynomial	$\mathcal{O}(n^c)$
Exponential	$\mathcal{O}(c^n)$

Table 3. *Notation in Pseudocode*

Formally, this means that any function is asymptotically *upper-bounded* by its respective Big-O class. Let f and g be functions. It holds that

$$\exists k \exists n_0 \forall n \{|f(n)| \leq k \cdot g(n) \mid k > 0, n > n_0\} \to f(n) \in \mathcal{O}(g(n))$$

stating that the size of the function $f(n)$ will eventually be overtaken by the Big-O class $g(n)$ multiplied by a constant k. It is an upper bound.

Because Big-O classes live inside each other, we always seek to ascertain the *smallest* Big-O class in which a function exists. It states the **worst-case running time**.

1.4 Omega Notation

Omega notation defines the lower bound of a function. It expresses when a function grows at least as fast as $\Omega(N)$ at large values of N.

Let f and g be functions. It holds that

$$\exists k \exists n_0 \forall n \{0 \leq k \cdot g(n) \leq |f(n)| \mid k > 0, n > n_0\} \rightarrow f(n) \in \Omega(g(n))$$

stating that the size of the function $f(n)$ will always be larger than $g(n)$ multiplied by a constant k. It is a lower bound, and states the **best-case running time**.

1.5 Theta Notation

Theta notation defines a function that is both an upper and a lower bound, depending on the constant the function $g(N)$ is multiplied by.

Let f and g be functions. It holds that

$$\exists k_1 \exists k_2 \exists n_0 \forall n \{0 \leq k_1 \cdot g(n) \leq |f(n)| \leq k_2 \cdot g(n) \mid k > 0, n > n_0\} \rightarrow f(n) \in \Theta(g(n))$$

stating that the size of the function $f(n)$ will always be exactly between two bounds. It is a statement of the **average case running time**.

2 Recursive Algorithms

Learning Outcomes

- ✓ Trace and write recursive algorithms
- ✓ Write the recursive version of an iterative algorithm using pseudocode
- ✓ Calculate the time complexity of recursive algorithms

2.1 Structure of Recursive Algorithms

A recursive algorithm is an algorithm that invokes itself.

Algorithm 3 Basic structure of a recursive algorithm

```
function HELLO
    print("hello")
    hello()
end function
```

Each call to a recursive function pauses the calling function until the called function completes. The algorithm in Algorithm 3 never ends, and is an example of *infinite recursion.*

Algorithm 4 Basic structure of a recursive algorithm

```
function HELLO
    if $n == 0$ then
        return
    end if
    print("hello")
    hello($n - 1$)
end function
```

A *finite recursive algorithm* needs to have at least one **base case**, and a recursive call with an input argument that gets closer to the base case, as shown in Algorithm 4.

Recursive solutions are more concise than iterative solutions, however, they are harder to understand and require more memory, as they occupy more space in the call stack.

2.2 Writing Recursive Algorithms

2.2.1 Transforming iterative algorithms to recursive algorithms

To transform an iterative algorithm into a recursive one, there are three steps.

1. **Initial Value:** Iterative algorithms set the initial value of the iterations by initializing the counter of a loop. Recursive alogrithms do this by setting the input argument to the function.
2. **Repetition:** The initial value is checked for a condition. Iterative: In the loop definition; Recursive: As a conditional in the function.
3. **Updating the control variable:** Iterative: In the loop afterthought; Recursive: In the recursive call.

2.3 Analysing Recursive Algorithms

2.3.1 Time Complexity of Recursive Algorithms

The time complexity of an algorithm is the *asymptotic* number of simple operations executed by an algorithm.

The running time $T(N)$ of a recursive algorithm can be expressed with a **recurrence equation**, expressed in terms of the running time of the *same algorithm* with *smaller inputs.* For example:

$$T(N) = C + T(N-1)$$

When solving a recurrence relation, there are two steps:

1. Find a value of N for which $T(N)$ is known.
2. Expand the right side of the recurrence relation.
3. Keep going until you identify the relationship / growth of all coefficients
4. Replace all coefficients with an expression in terms of k
5. Replace k with a value so that expressions of $T(N)$ can be replaced by known values (often C, 0 or 1).

Example where T(1) = C

$$\begin{aligned}
T(N) &= C_5 + T(N-1) \\
T(N) &= C_5 + (C_5 + T(N-1-1)) \\
T(N) &= C_5 + (C_5 + T(N-2)) \\
&\vdots \\
T(N) &= k * C_5 + T(N-k) \\
T(N) &= (N-1) * C_5 + T(N-(N-1)) \\
T(N) &= (N-1) * C_5 + T(1) \\
T(N) &= (N-1) * C_5 + C \\
T(N) &= C_5 N + C_6
\end{aligned}$$

2.4 Master Theorem

A faster way to determine the time complexity of recursive algorithm is to use the Master theorem.

If $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function, then then the time complexity of a recursive relation

$$T(n) = a * T(\frac{n}{b}) + f(n)$$

is given by

1. If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

3 Comparison Sorting Algorithms

Learning Outcomes

- ✓ Identify the different approaches of different comparison sorting algorithms
- ✓ Implement different comparison sorting algorithms
- ✓ Calculate the time complexity of different comparison sorting algorithms

3.1 Comparison and non-comparison sorting algorithms

The most known comparison sort algorithms are shown in Table 4.

Table 4. *Comparison Sort Algorithms*

	Worst Case	**Best Case**
Bubble	$\mathcal{O}(N^2)$	$\Omega(N)$
Insertion	$\mathcal{O}(N^2)$	$\Omega(N)$
Selection	$\mathcal{O}(N^2)$	$\Omega(N^2)$
Quicksort	$\mathcal{O}(N^2)$	$\Omega(N \log N)$
Mergesort	$\mathcal{O}(N \log N)$	$\Omega(N \log N)$

There are no faster algorithms among comparison sort algorithms than Mergesort. Because they examine every element in the input, they can't be faster than $\mathcal{O}(N \log N)$.

Therefore, we resort to non-comparison sorting algorithms.

3.2 Bubble Sort

Bubble sort is popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

It starts at the head of a vector, compares the element to the adjacent elements, and swaps the elements if the first element is larger than the second element.

Passes The Bubble sort algorithm may need multiple passes to sort the entire vector. The maximum number of passes is given by the most difficult vectors to solve - such that are completely in reverse order. In this case, the number of required passes is $n-1$ where n is the number of elements. Therefore, **the maximum number of passes required** is always $n-1$.

Algorithm 5 Bubble Sort — $\mathcal{O}(n^2)$

```
function BUBBLESORT($vector$)
    $n \leftarrow$ LENGTH[$vector$]
    for $1 \leq i \leq n-1$ do
        $count \leftarrow 0$
        for $1 \leq j \leq n-1$ do
            if $vector[j+1] < vector[j]$ then
                Swap($vector, j, j+1$)
                $count \leftarrow count + 1$
            end if
        end for
        if $count = 0$ then
            break
        end if
    end for
    return $vector$
end function
```

3.3 Insertion Sort

Insertion sort works from the left of a vector, starting with the second element. It compares the element to the element on the left. If it needs to move, the element is moved along to the left as far as needed.

As an algorithm, we can store the current element in a temporary variable, while we "move" elements to the right by overwriting the slots to the right and then restoring the original values from the temporary variable.

Algorithm 6 Insertion Sort — $\mathcal{O}(n^2)$

```
function InsertionSort($v$)
    for $2 \leq$ LENGTH[v] do
        $key \leftarrow v[j]$
        $i \leftarrow j - 1$
        while $i > 0 \wedge v[i] > key$ do
            $v[i+1] \leftarrow v[i]$
            $i \leftarrow i - 1$
        end while
        $v[i+1] \leftarrow key$
    end for
end function
```

4 Random-Access Machines, Growth of Functions and Time Complexity

Learning Outcomes

✓ Explain the model of random access machines.

✓ Explain asymptotic growth of functions and worst-case time complexity

✓ Describe worst-case time complexity of bubble and insertion sort

5 Non-comparison sorting algorithms

Learning Outcomes

✓ Identify the different approaches of different non-comparison sorting algorithms

✓ Implement different non-comparison sorting algorithms

✓ Calculate the time complexity of different non-comparison sorting algorithms

5.1

6 Hashing

Learning Outcomes

✓ Describe the different methods used to search for data

✓ Describe different collision resolution methods

✓ Implement a hash table with linear probing collision resolution

6.1

7 Linear Data Structures

Learning Outcomes

✓ Describe linear data structures and its operations using pseudocode

✓ Understand array and linked list based implementations of stacks and queues

✓ Implement a sorted linked list

7.1

8 Trees

Learning Outcomes

✓ Understand how to implement a tree

✓ Describe and trace different types of binary tree traversals using pseudocode

✓ Describe and trace binary search tree operations using pseudocode

8.1

9 Heaps

Learning Outcomes

✓ Check heap and shape properties

✓ Describe heap operations using pseudocode

✓ Implement heapsort using a heap

9.1

10 Graphs

Learning Outcomes

✓ Use different implementations of graphs (matrix adjacency, list adjacency. Edge list)

✓ Implement Prim's algorithm to find the minimum spanning tree

✓ Implement Dijkstra's algorithm to find the shortest path between two nodes

10.1