Time Complexity

Josh Wilcox (jw14g24@soton.ac.uk)

January 28, 2025

Contents

1 Travelling Salesman **2**
1.1 Brute force approach . . . 2

2 Lessons **2**

3 Time Complexity **2**
3.1 Input Size . . . 2
3.2 Running Time . . . 2
3.3 Defining Time Complexity . . . 3

4 Big-Theta Notation **3**
4.1 Estimating Run Time . . . 4
4.2 Disadvantages of Big-Theta . . . 4

5 Bounds of Rate of Growth **4**
5.1 Notations . . . 4
5.2 Use and Misuse of Notations . . . 5

1 Travelling Salesman

1.1 Brute force approach

- If a program to solve TSP enumerates all tours and selects the shortest
- If it takes just under half a second to solve problems with 10 cities
- *How long does it take to finish 100 cities*
- The total number of enumerations of nodes is $\frac{(n-1)!}{2}$
- This means it has a factorial order - which further means the brute force approach would be useless
- For 100 cities:
  - If we had 10^{87} cores - one for every particle in the universe
  - And could computer a tour in 3×10^{-34} seconds, the time it takes light to cross a proton
  - The brute force algorithm would still take 10^{39} times the age of the universe

2 Lessons

- There are right and wrong ways to solve **easy** problems
- Complexity is only really a problem when dealing with large inputs
- We should approach an algorithm with complexity in mind - we should be able to compare them *without running them*

3 Time Complexity

3.1 Input Size

- Running times of algorithms depend on the size of their input
- Input size can be seen to be problem-dependent
  - Number of elements to sort
  - Number of bits in arithmetic
  - Number of nodes/edges in a graph problem

3.2 Running Time

- Running time depends on implementation, compiler, and the hardware we run on
- We only want to consider the algorithm itself so we need to abstract away compilers and hardware
- Therefore, take the running time of an algorithm to be the number of **primitive operations** execute
  - **Primitive Operations** - Addition, Subtraction, Comparison, Assignment, ...
  - All primitive operations take $\approx$ the same amount of time
  - Therefore counting their total gives a good measure of the time taken by the algorithm
  - We can then assign a **cost** to each line of pseudocode by counting the primitive operations

```
MINIMUM(A)
  m = A[1]
  i = 2
  while (i<=A.length)
     m = min(m,A[i])
     i++
```

cost	times
c_1	1
c_2	1
c_3	n
c_4	$n-1$
c_5	$n-1$

- c_i counts the primitive operations in line i
- For each array of size n, the algorithm takes
  $$T(n) = c_1 + c_2 + c_3 * n + (c_4 + c_5) * (n - 1)$$
  primitive operations
- Thus, $T(n) = a * n + b$ with a, b constants
  - $T(n)$ only depends on the array size, and not on the specific input

Figure 1: Example: Minimum element in array

3.3 Defining Time Complexity

For $T : \mathbb{N} \to \mathbb{N}$:

$T(n)$ is the **maximum** number of primitive operations the algorithm uses on input size n

4 Big-Theta Notation

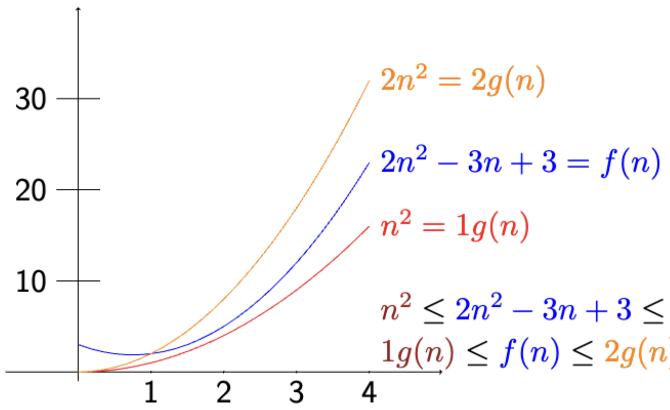
- Let $f, g : \mathbb{N} \to \mathbb{R}^+$
- The Θ notation captures the idea that the functions f and g have the *same rate of growth*
- $f(n)$ is $\Theta(g(n))$ if there exists constants $c > 0, d > 0$ and $N \in \mathbb{N}$ such that

$$cg(n) \leq f(n) \leq dg(n) \text{ for } n \geq N$$

- For large n values a $\Theta(n^2)$ will always be more efficient than a $\Theta(n^3)$ algorithm

Big-Theta Notation: Examples

- $2n^2 - 3n + 3$ is $\Theta(n^2)$:



$n^2 \leq 2n^2 - 3n + 3 \leq 2n^2$ for $n \geq 1$

$1g(n) \leq f(n) \leq 2g(n)$ for $n \geq 1$

- Any quadratic function is $\Theta(n^2)$

4.1 Estimating Run Time

- For an algorithm of $\Theta(n^2)$:
  - $T(n) \approx cn^2$ for $n \gg 1$
  - If it takes x seconds on average on input size 100
  - It will take about $\frac{x \times n^2}{100^2}$ seconds on avaregage on an input of size n

4.2 Disadvantages of Big-Theta

- Can't compare algorithms whose running times have the same rate of growth
- Can be misleading for small inputs - However we don't normally care about this until we typically run algorithms on small inputs

5 Bounds of Rate of Growth

- If statements can make finding the rate of growth of an algorithm really hard
- For example in the following:

```
// define stuff
for(int i=0; i<n; i++) {
    // do something
    if(/* condition */) {
        for(int j=0; j<n; j++) {
        // do other stuff
        }
    }
}
// clean uo
```

  - Rate of growth depends on the `if` statement
  - Therefore we know the true order will be between $\Theta(n)$ and $\Theta(n^2)$
- To avoid having to worry so much we can look for upper and lower bounds for the rate of growth

5.1 Notations

- **Big-O** notation is used to give the *upper bound* for rate of growth
  - For $O(n^2)$ algorithms, there is no case where it executes more than order n^2 operations
    - This is true for either worst, average, or best case
- **Big-Omega** gives the *lower bound* for rate of growth
  - For $\Omega(n^2)$ algorithms, there is no case where it executes less than order n^2 operations

Let $f, g : \mathbb{N} \to \mathbb{R}^+$.

- $f(n)$ is $O(g(n))$ if there exist constants $d > 0$ and $N \in \mathbb{N}$ such that
$$f(n) \leq d \cdot g(n) \quad \text{for } n \geq N$$
- $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $N \in \mathbb{N}$ such that
$$f(n) \geq c \cdot g(n) \quad \text{for } n \geq N$$
- And so $f(n)$ is $\Theta(g(n))$ if (and only if)
$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$
i.e. the lower bound is identical to the upper bound!

Any $O(n^2)$ algorithm is also an $O(n^3)$ algorithm, according to the definition of Big-O notation.

Big-O notation provides an upper bound on the growth rate of an algorithm. If an algorithm is $O(n^2)$, this means that its growth rate is at most proportional to n^2 for sufficiently large n.

Now, n^2 grows slower than n^3 as $n \to \infty$. Since $O(n^3)$ describes functions that grow as fast as or slower than n^3 , any $O(n^2)$ function will also satisfy the criteria for being $O(n^3)$.

5.2 Use and Misuse of Notations

- Big-O notation is most commonly used
- Often, people will say they have an $O(n^2)$ algorithm when they actually have a $\Theta(n^2)$
- Any $O(n^2)$ algorithm is also an $O(n^3)$ algorithm
- An $O(n^2)$ algorithm might not be faster than an $O(n^3)$ algorithm even for large n