Algorithms Notes

Tiago Antunes

November 13, 2020

1 Complexity

1.1 Running time

Definition 1.1 ($T_A(n)$)**.** $T_A(n) =$ the computational cost for solving a problem instance of size n by algorithm A. Can be written as $T(n)$.

Definition 1.2 (Worst-case running time)**.** The **maximum** running time of an algorithm on any input of size n.

Definition 1.3 (Average-case running time)**.** The **expected** running time of an algorithm on any input of size n.

Definition 1.4 (Best-case running time)**.** The **minimum** running time of an algorithm on any input of size n.

Due to the complexity of a system it is common to use the RAM model. Each instruction takes a constant amount of time and there are no concurrent operations. It is **machine independent**, and compares the efficiency of the algorithms. Given the RAM model, the running time T and the number of basic operations L have the following relationship

$$T \propto L$$

1.2 Growth analysis

Definition 1.5 (Asymptotic analysis)**.** It is more interesting to analyse big problems, so when using the RAM model, it is required to analyse the growth of $T(n)$ as n grows bigger.

Example 1.1. Given two algorithms A and B, if $T_A(n) = n^2$ and $T_B(n) = n^3$, then we can state that A is more efficient than B due to having a lower asymptote than B.

Definition 1.6 (Θ-notation)**.** Encloses a function from both above and below and it is useful for measuring the average case. By definition, $f(n) = \Theta(g(n))$ if and only if there are constants c_1, c_2 and n_0 such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n : n \ge n_0$$

Example 1.2. Consider $f(n) = \frac{1}{2}n^2 - 3n$ and $g(n) = n^2$. We can show that $f(n) = \Theta(g(n))$:

$$\frac{1}{2}n^2 \leq c_2 n^2 \iff \frac{1}{2} - \frac{3}{n} \leq c_2$$

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \iff c_1 \leq \frac{1}{2} - \frac{3}{n}$$

Then it can be noted that $(c_1, c_2) = (\frac{1}{2}, \frac{1}{3})$ for all $n \geq n_0$, where $n \geq 18$ (to find n_0 just solve for x).

Definition 1.7 (Properties). There are a few properties for Θ-notation.

- Asymptotic notation in equations
$$T(n) = 2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$
- Transitivity
$$f(n) = \Theta(g(n)), g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
- Reflexivity
$$f(n) = \Theta(f(n))$$
- Symmetry
$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$$

Definition 1.8 (Big-O notation). By definition, $f(n) = O(g(n))$ if and only if exist positive constants c and n_0 such that

$$0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0$$

Remark. $O(g(n))$ can be read as "No matter what input is given to g, $T_g(n) \leq O(g(n))$

Definition 1.9 (Little-o notation). Also known as the asymptotic strict upper bound. By definition, $f(n) = o(g(n))$ if and only if for any positive constant $c_0 > 0$ exists $n_0 > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0$$

Example 1.3. Little-o has to do with strict limits. Then the following holds valid:

- $2n = o(n^2)$
- $2n^2 \neq o(n^2)$.

Remark. For any given function f, $o(f(n)) \in O(f(n))$

Definition 1.10 (Big-Ω). By definition, $f(n) = \Omega(g(n))$ if and only if there exists positive constants c and n_0 such that

$$0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0$$

Remark. $\Omega(g(n))$ can be read as "No matter what input is given to g, $T_g(n) \geq \Omega(g(n))$

1.3 Speed of growth

The speed of growth has to do with the different functions that describe the complexity of an algorithm.

Definition 1.11 (Function relations)**.** Consider $0 < k < 1$ and $p > 1$.

$$ln(n) < n^k < n < n \cdot ln(n) < n^p < 2^p$$

1.4 Problem complexity

Given a problem and a function $f(n)$, if $f(n)$ is the lower bound of any algorithm that solves the problem in its worst case (Big-O), $f(n)$ is called the lower bound of the problem.

Definition 1.12 (Reduction)**.** Given two problems, A and B, A can be reduced to B by:

1. Convert the input of A into a suitable input of B
2. Solve the problem B given that input
3. Convert the output of B into a correct solution of A

Then, if steps 1 and 3 can be performed in $\Theta(g(n))$ then we can say

$$A \propto_{g(n)} B$$

2 Incremental Approach & Loop Invariant

Definition 2.1 (Incremental Approach)**.** Given an input sequence of n numbers $< a_1, a_2, ..., a_n >$, an incremental approach consists of slowly building the solution on each main iteration of the algorithm (also known as main loop).

Example 2.1. Consider the case of **Insertion Sort**. The idea is to slowly build a sorted subarray $A[1..j-1]$ and to insert a single element, $A[j]$ into its proper place, thus creating the sorted subarray $A[1..j]$.

Definition 2.2 (Loop Invariant)**.** A property of an algorithm that allow us to prove its correctness. A loop invariant must contain three things:

1. **Initialization** - It is true before the first iteration of the loop.
2. **Maintenance** - If it is true before an iteration (any iteration) of the main loop, then it must remain true before the next iteration.
3. **Termination** - When the loop terminates, it gives a useful property that helps show the algorithm is correct.

The loop invariant can contain the loop variable and any other useful variables.

Remark. One can also think of the Loop Invariant as the intuition behind the algorithm. It is similar to mathematical induction with the exception of the termination step.

Example 2.2. For the case of the insertion sort, the loop invariant is the following:

> At the start of each iteration of the **for** loop of lines 1-8, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

Let us now prove its correctness.

1. **Initialization** - The loop invariant must hold before the first iteration where $j = 2$. The subarray is then $A[1..1]$ which is the element $A[1]$. Then, it is in sorted order, since an array of size 1 is naturally sorted.
2. **Maintenance** - A formal proof would require the loop invariant of the **while** loop. Informally, the while loop places the value initially in $A[j]$ in the correct place of the subarray $A[1..j]$, thus rendering a sorted subarray $A[1..j]$. In the next iteration, that is, $j+1$, the subarray $A[1..(j+1)-1]$ is sorted and only consists of the original elements in the subarray $A[1..(j+1)-1]$, so the loop invariant holds true.
3. **Termination** - When the loop terminates, the condition of the **for** loops has to have ended, so $j > A.length = n$. Each iteration increases j by 1, so we have $j = n + 1$. Substituting $n + 1$ for j in the loop invariant, we then have a subarray $A[1..n]$ in sorted order that consists of the original elements in $A[1..n]$. Since the subarray $A[1..n]$ is actually the original array, the entire array is sorted, and the algorithm is correct.

Example 2.3. Consider the 2-Color Dutch National Flag problem. Given an array $A[1..n]$ which contains red elements and blue elements, it is required to rearrange the array so that the red elements are to the left of the blue elements.

There are two possible loop invariants:

- Consider the case where we incrementally build the solution from the left. The loop invariant is:

  > Consider the current position as j and the split position as k. $A[1..k-1]$ is red, $A[k..j-1]$ is blue.

- Consider the case where we build the solution incrementally from each end. The loop invariant is:

  > Consider i_1 and i_2 to be left counter and right counter, respectively. The left loop invariant is for $A[1..i_1-1]$ to be red, and the right loop invariant is $A[i_2+1..n]$, and both are correct. Then, for the loop invariant, we have that while $i_1 < i_2$ then $A[1..i_1-1]$ is red and $A[i_2+1..n]$ is blue. If $i_1 > i_2$ then $A[1..i_2]$ is red, $A[i_1..n]$ is blue.