An Introduction to Asymptotic Analysis

Thomas Rockwell Tucker

Introduction

In computer science, an algorithm is a finite list of well-defined instructions for calculating a function relating to a defined set of problems. The given algorithm will always provide an answer which is correct, complete its execution in a finite number of steps, and will work for all instances of that class.[**?**]

When considering the algorithm for a given problem, it is important to observe two different aspects: is the algorithm an effective solution to it's given problem in all cases, and how does it perform in terms of amount of space and number of steps?

In this paper we will take the former question for granted and focus on the latter, with most of our concern centering around the amount of steps an algorithm would take given the specific task it was given to solve (i.e. a sorting algorithm being tasked to sort an array with n elements), and even more concerning the rate at which the number of steps taken by the algorithm grows when given a larger and larger input.

Excellent intro

1 Steps of an Algorithm and the $T(n)$ Equation

Definition 1.1. In computer science, we say that a *step* in an algorithm is a bounded-size set of instructions which executes in a consistent, constant time.[**?**]

Remark 1.1.1. While it is common convention for a single line of code which meets the definition above to be considered one step, we can also consider a group of lines which all execute in constant time to be a single step.

good remark

Definition 1.2. The function $T : \mathbb{N} \to \mathbb{N}$ denotes the worst-case number of steps taken to execute a given algorithm given an input which is conventionally written as n.

Remark 1.2.1. Often an algorithm's speed is only dependent on one input n, but it is not uncommon for an algorithm's execution to have more than one dependency, i.e. how the copying of a matrix has a runtime dependent on both the number of rows m and the number of columns n.

Example 1.2.2. Consider the block of python code below, which finds the factorial of a given natural number n.

```
1 def factorial(int n):
2   let fac = 1
3   let i = 1
4   while (i <= n):
5     fac = fac * i
6     i = i + 1
7   return fac
```

Lines 1 and 2 both run in constant time, and therefore take 1 step in total, as per remark 1.1.1. Line 4 runs a total of $n + 1$ times, since it checks each time we increment i. Lines 5 and 6 are both constant and therefore 1 step, but they will be run n times due to being inside the while loops above, meaning they add $n \cdot 1$ to the total. The last line runs only once and is consistent, so it takes 1 step.

In total we have a runtime of $T(n) = 1 + (n + 1) + (n \cdot 1) + 1 = 2n + 2$.

2 Big Θ Notation

Definition 2.1. Given a function $f : \mathbb{N} \to \mathbb{N}$, and an algorithm with a runtime of $T : \mathbb{N} \to \mathbb{N}$, we say that

$$T \in O(f) \iff \exists a, n_0 > 0 : \forall n > n_0 : T(n) \leq af(n)$$
$$T \in \Omega(f) \iff \exists b, n_0 > 0 : \forall n > n_0 : T(n) \geq bf(n)$$
$$T \in \Theta(f) \iff \exists a, b, n_0 > 0 : \forall n > n_0 : af(n) < T(n) < b(f(n)).$$

[?]

Remark 2.1.1. Notice that $\Theta(f) = \Omega(f) \cap O(f)$.

Remark 2.1.2. While uncommon, there are two other types of classification of an algorithm's runtime, where

$$T(n) \in o(f(n)) \iff T \in O(f(n)) \wedge \lim_{n\to\infty} \frac{T(n)}{f(n)} \to 0$$
$$T(n) \in \omega(f(n)) \iff T \in \Omega(f(n)) \wedge \lim_{n\to\infty} \frac{T(n)}{f(n)} \to \infty.$$

These are to represent when an algorithm's runtime is significantly less than f and significantly more than f, respectively. [?, ?]

Remark 2.1.3. It is common convention to write $T(n) = \Theta(n)$ rather than $T(n) \in \Theta(n)$, due to ease of reading when working with arithmetic.

Remark 2.1.4. When $T = \Theta(f)$, we say that the given algorithm with the runtime of T is in the *runtime class* of f.

Definition 2.2. Given a function f and an algorithm with a runtime T such that $T = \Theta(f)$, we say that that algorithm is in the *runtime class* of f. The common runtime classes in this regard are, in order of their magnitude:

Name	**Runtime class**	**Example**
Constant	$\Theta(1)$	Accessing an element of an array
Logarithmic	$\Theta(\lg n)$	Binary Search
Linear	$\Theta(n)$	Linear Search
Log-Linear	$\Theta(n \lg n)$	Mergesort
Quadratic	$\Theta(n^2)$	Initializing a square matrix
Cubic	$\Theta(n^3)$	Naïvely multiplying matrices
Polynomial	$\Theta(n^k)$	
Exponential	$\Theta(2^n)$	Listing all subsets of a set
Factorial	$\Theta(n!)$	Listing all orderings of an array

[?]

Remark 2.2.1. Computer scientists commonly consider anything better than log-linear time to be practical. Anything which performs at least as well as polynomial time is said to be *tractable*, or that it can successfully complete any size task given enough computing resources. The runtime classes beyond are considered *intractable*, meaning that it will rapidly reach an estimated runtime which exceeds the lifetime of the universe. [?]

Remark 2.2.2. Note that just because an algorithm has a nice runtime class, it does not mean that it runs in a reasonable amount of time. Even constant time algorithms can have an enormous constant.

Example 2.2.3. Observe the algorithm from example 1.2.2, where $T(n) = 2n + 2$. Suppose we want to prove that our function has a runtime class of $\Theta(n)$. In this instance, we would give the following proof:

Proof. Let $k = 1, a = 4, b = 2$. We have $an \geq 2n + 2 \geq bn$ for all $n > k$. □

3 Arithmetic with Runtime Classes

Theorem 3.1 (Rule of Sums [?]). *Given functions f_1, f_2, g_1, g_2,*

$$g_1 = \Theta(f_1) \wedge g_2 = \Theta(f_2) \implies g_1 + g_2 = \Theta(max\{f_1, f_2\}).$$

Proof. From definition 2.1 it is known that $a_1 f_1(n) \geq g_1(n) \geq b_1 f_1(n)$ for all $n > k_1$, and $a_2 f_2(n) \geq g_2(n) \geq b_2 f_2(n)$ for all $n > k_2$. By adding the inequalities we find that

$$\begin{aligned} g_1(n) + g_2(n) &\leq a_1 f_1(n) + a_2 f_2(n) \\ &\leq \max\{a_1, a_2\}(f_1(n) + f_2(n)) \\ &\leq 2 \cdot \max\{a_1, a_2\} \cdot \max\{f_1(n), f_2(n)\} \end{aligned}$$

for all $n > \max\{k_1, k_2\}$.

$$\begin{aligned} g_1(n) + g_2(n) &\geq b_1 f_1(n) + b_2 f_2(n) \\ &\geq \min\{b_1, b_2\}(f_1(n) + f_2(n)) \\ &\geq \frac{1}{2} \cdot \min\{b_1, b_2\} \cdot \max\{f_1(n), f_2(n)\} \end{aligned}$$

for all $n > \max\{k_1, k_2\}$. □

Theorem 3.2 (Rule of Products [?]). *Given functions f_1, f_2, g_1, g_2,*

$$g_1 = \Theta(f_1) \wedge g_2 = \Theta(f_2) \implies g_1 g_2 = \Theta(f_1 f_2)$$

Proof. From definition 2.1 we have $a_1 f_1(n) \geq g_1(n) \geq b_1 f_1(n)$ for all $n > k_1$, and $a_2 f_2(n) \geq g_2(n) \geq b_2 f_2(n)$ for all $n > k_2$. By multiplying the inequalities, we have

$$a_1 a_2 f_1(n) f_2(n) \geq g_1(n) g_2(n) \geq b_1 b_2 f_1(n) f_2(n).$$

for all $n \geq \max\{k_1, k_2\}$. □

Example 3.0.1. Suppose we have an algorithm with a runtime of $T(n) = 2n^3 + 5n$, and we wanted to find the algorithm's runtime class. Rather than making an educated guess and attempting to prove it, we can instead use theorem 3.1 to break up the function into the smaller sub-functions $f_1(n) = 2n^3$ and $f_2(n) = 5n$. From here, we can just see which of the two functions has a greater runtime class.

Using theorem 3.2 we can see that f_1 is made up of two sub-functions $f_1 1(n) = 2, f_1 2(n) = n^3$, and that f_2 is similarly made up of the sub-functions $f_2 1(n) = 5, f_2 2(n) = n$, and that the multiplied funtions have the respective runtime classes of $\Theta(n^3)$ and $\Theta(n)$. From here, it is obvious that $n^3 \geq n$ for all $n \geq 1$, and therefore we have

$$T(n) = 2n^3 + 5n = \Theta(n^3).$$

4 Recurrence Relations and the Master Theorem

Definition 4.1. A *divide & conquer algorithm* is an algorithm which breaks up a problem into smaller versions of the same problem and solves them recursively, that is, solves the sub-problems using another instance of itself.[?]

Example 4.1.1. The following is a classic example of a divide & conquer algorithm called mergesort which sorts a list of n comparable items:

```
def mergesort(array):
  let n == length(array)
  if n<2 then done.
  let a1 = array[0 to n/2 - 1]
  let a2 = array[n/2 to n-1]
  mergesort(a1)
  mergesort(a2)
  let i = j = 0
  for k from 0 to n-1 do:
    if j == length(a2) or (i<length(a1) and a1[i] < a2[j]) then:
    array[k] = a1[i]
      increment i
    else:
      array[k] = a2[j]
      increment j
```

[?]

Definition 4.2. A *recurrence relation* is a description of the running time of a recursive algorithm with an input of size n as a function of n.[?]

Example 4.2.1. The recurrence relation for the algorithm in example 4.1.1 can be found by observing that, on any given call, the sub-problem is reduced to two subproblems with the size $\frac{n}{2}$, with each instance requiring combing through every element in the list (an operation with a runtime class of $\Theta(n)$). This results in a recurrence relation of

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Theorem 4.1 (Master Theorem [?]). *Given an algorithm with the recurrence relation*

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k$$
$$T(1) = c$$

Where a, b, c, k are positive constants, it is generally true that

i. $a < b^k \implies T(n) = \Theta(n^k)$

ii. $a = b^k \implies T(n) = \Theta(n^k \lg n)$

iii. $a > b^k \implies T(n) = \Theta(n^{\log_b a})$

Proof. The total amount of recurrence is the sum of work at each level. At the top level, we do cn^k, at the first we do $ca\frac{n}{b^k}$, the next $ca^2\frac{n}{b^k}^k$, and so on. The amount of levels is equal to $\log_b(n)$, so we get the summation of

$$cn^k \sum_{i=0}^{log_b n} \left(\frac{a}{b^k}\right)^i$$

(i) Suppose $a < b^k$. It follows that $\frac{a}{b^k} < 1$, causing our summation to be a convergent series. If we have a maximum of infinite levels, we have a total of $\frac{cn^k}{1-a/b^k}$, and since a, b, c, k are constants, the total has a runtime class of $\Theta(n^k)$.

(ii) Suppose $a = b^k$. In this case, all terms of the summation are equal to 1, and since there are $\log_b n$ terms in the summation, we have a total recurrence of $cn^k(\log_b n + 1) = \Theta(n^k \lg n)$.

(iii) Suppose $a > b^k$. Observe that the last term of the summation dominates the entire equation, with a final summation of

$$cn^k \left(\frac{a}{b^k}\right)^{\log_b n} \left[\left(\frac{b^k}{a}\right)^{\log_b n} + \cdots + \frac{b^k}{a} + 1\right].$$

Since $\frac{b^k}{a} < 1$, we can use similar reasoning to our first case: leaving us with a maximum summation of

$$\begin{aligned}\frac{cn^k\left(\frac{a}{b^k}\right)^{\log_b n}}{1-\frac{b^k}{a}} &= \Theta\left(n^k\left(\frac{a}{b^k}\right)^{\log_b n}\right)\\ &= \Theta\left(a^{\log_b n}\right)\\ &= \Theta\left(n^{\log_b a}\right).\end{aligned}$$

$\square$

Example 4.2.2. Take our recurrence relation from example 4.2.1, where

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Using theorem 4.1, we can find the runtime class of mergesort. We have $a = 2, b = 2, c = 1, k = 1$, and it is evident that $a = b^k$. It follows that we have a runtime class of

$$T(n) = \Theta(n^k \lg n) = \Theta(n \lg n).$$

good work and really interesting.