CS3230 Design & Analysis of Algorithms **August 22, 2023**

Lecture 2: Basics of Algorithm Analysis

Lecturer: Arnab Bhattacharyya

The focus of this course is to design algorithms that are efficient in terms of *worst-case running time*. To initiate this study, we first need to properly define how to measure running time. Here are a few requirements we would like our measure to satisfy:

1. Should qualitatively capture the running time of the algorithm on a 'standard computing device'
2. Should not depend strongly on the detailed specification of the computing device
3. Should allow us to make meaningful comparisons between different algorithms

One immediate option is of course to just run the algorithm in question and empirically measure the running time. However, there are two problems with this approach. First, such a study would give results depending on the details of the device running the experiment. Secondly, since our goal is to bound worst-case running time, we would need to either understand what the worst input is for the given algorithm or to run the algorithm on all possible inputs, both pretty challenging tasks.

Instead, the approach we will pursue is to define a *mathematical model of computation* that satisfies requirements (1) and (2) above. Then, to achieve (3), we describe the paradigm of *asymptotic analysis* to compare two algorithms in terms of how their running times qualitatively scale with the input size.

2.1 Model of Computation

The framework we mostly use subsequently is a conventional **word RAM** model, where the 'programming language' consists of sequential instructions and the running time is the total number of instructions executed. What constitutes an 'instruction'? We omit a formal specification, but it consists of the typical basic operations found in standard programming languages: arithmetic operations ($+, -, \times, /, \mod, <, >, \lfloor\cdot\rfloor, \dots$), memory access operations, and control flow operations (`if, return`, ...). Each instruction is allowed to operate on a *word* of data, and the word size is limited. (Clearly, it would be useless to have a model where each instruction could operate on inputs of unrestricted size in one time unit.) We typically assume that numbers (e.g., entries in an array) are each of word-size so that they can be operated on in unit time.

The motivation behind the word RAM model is that each instruction can be executed within a fixed constant amount of time on any computer, where the constant depends on the particular details of the device. Different computers have different CPU speeds, different speeds of loading memory to cache, and in other hardware details, but the expectation is that the actual running time will be be proportional to what we get by counting number of instructions in the word RAM model. This is not strictly true, as computers do take advantage of parallelism, frequency scaling, memory hierarchies, and other tricks, but if these issues become important, we would augment the word RAM model so as to explicitly model them. Furthermore, if one considers non-standard computers (e.g., quantum computers, DNA computers), then one needs completely different models of computation, but these issues are beyond the scope of this course.

Example 1 (Binary Search)**.** Consider the following sorted array A of 10 integers.

Value	3	9	28	35	57	63	73	82	94	98
Index	0	1	2	3	4	5	6	7	8	9

Suppose we want to find the position of value 63 on the array. There is an iterative algorithm and a recursive algorithm.

```
FindPositionIterative($x$)
1  for $i = 0$ to $A.length - 1$
2      if $A[i] == x$
3          return $i$
4  return None
```

Applied to the array A above, FindPositionIterative iterates through all position from 0 to 9, until it meet the positions with value equal to 63. The value returned is 5. The number of instructions executed is 19, because each of the first 5 iterations of the **for** loop makes 3 instructions each (one increment of i, one check that $i \leq A.length - 1$, and one **if** call), while the last iteration makes an extra **return** call.

```
FindPositionBinarySearch($x$)
1  $low = 0, high = A.length - 1$
2  while $low \leq high$
3      $mid = \lfloor (low + high)/2 \rfloor$
4      if $A[mid] == x$
5          return $mid$
6      elseif $A[mid] < x$
7          $low = high + 1$
8      else $high = mid - 1$
9  return None
```

Applied to the array A above, FindPositionIterative first checks the middle position of the array. This position is: $\lfloor (0+9)/2 \rfloor = 4$. Since $A[4] = 57$ is smaller than 63, we know that the correct position is larger than 4. We reduce the array size to $A[5\,..\,9]$. Apply the same logic, we find the new *mid*, $\lfloor (5+9)/2 \rfloor = 7$. $A[7] = 82$ is larger than 63. We reduce the array to $A[5\,..\,6]$. The next *mid* is $\lfloor (5+6)/2 \rfloor = 5$ and is the correct position. The number of instructions corresponds to 3 iterations of the **while** loop, with each loop having around 10 instructions.

Example 2. Let's work out on the recursive Fibonacci example from the slides

```
RecursiveFib($n, m$)
1  if $n == 0$
2      return 0
3  elseif $n == 1$
4      return 1
5  else return (RecursiveFib($n - 1, m$) + RecursiveFib($n - 2, m$)) mod $m$
```

Let $R(n)$ be the number of instructions executed by RecursiveFib(n, m). We have $R(0) = 2$, $R(1) = 3$, and $R(n) = R(n-1) + R(n-2) + 6$ for $n > 1$.

We claim that $R(n) \geq F(n)$. The proof is by *induction*. We have $R(0) \geq F(0)$, $R(1) \geq F(1)$. Assuming that the claim is correct for $n = n_0$, we want to prove that $R(n_0 + 1) \geq F(n_0 + 1)$. This immediate as $R(n_0+1) = 6 + R(n_0) + R(n_0 - 1) \geq F(n_0) + F(n_0 - 1) = F(n_0 + 1)$ by the inductive hypothesis. Therefore, $R(n) \geq F(n)$ for all $n \geq 0$.

Furthermore, $F(n) \geq 2^{(n-2)/2}$. The proof is again by induction. We have $F(2) = 1 = 2^{(2-2)/2} = 1$, $F(3) = 2 \geq 2^{(3-2)/2} \approx 1.4$. Assuming that the claim is correct for $n = n_0$, we want to prove that $F(n_0+1) \geq 2^{(n_0-1)/2}$. By the inductive hypothesis,

$$F(n_0+1) = F(n_0) + F(n_0-1) \geq 2^{(n_0-2)/2} + 2^{(n_0-3)/2} \geq 2^{(n_0-3)/2} * 2 = 2^{(n_0-1)/2}.$$

Therefore, $F(n) \geq 2^{(n-2)/2}$ for all $n \geq 2$.

Thus, the number of instructions executed by RECURSIVEFIB is $R(n) \geq F(n) \geq 2^{(n-2)/2}$. This is significantly larger than using the iterative algorithm, which only takes $5n$ instructions.

2.2 Asymptotic Analysis

Suppose you have two algorithms, $\mathcal{A}_1$ and $\mathcal{A}_2$, solving the same problem on inputs of size n. The algorithm $\mathcal{A}_1$ makes at most $5n^2$ instructions, while algorithm $\mathcal{A}_2$ makes at most $7n^2 + 100$ instructions. Should we say $\mathcal{A}_1$ is more efficient than $\mathcal{A}_2$?

The issue is complicated because the number of instructions in the word RAM model is *proportional* to the actual running time, where the constant of proportionality depends on the the details of the device. For instance, $\mathcal{A}_1$ may be performing $5n^2$ multiplications, while $\mathcal{A}_2$ doing $7n^2+100$ additions. For a typical processor, addition is faster than multiplication, and so $\mathcal{A}_2$ may well be running faster than $\mathcal{A}_1$. Thus, it makes no sense to exactly compare the running times of two algorithms as functions of n!

On the other hand, there *are* situations in which we can claim that one algorithm is faster than another, even using just the instruction count in the word RAM model. For example, suppose $\mathcal{A}_1$ makes at most $5n$ instructions, while $\mathcal{A}_2$ $10n^2$ instructions in the worst case. Now, suppose on an actual computing device, the running time of $\mathcal{A}_1$ is at most $5c_1n$ that for $\mathcal{A}_2$ is $10c_2n^2$. Now, note that if $n > \frac{c_1}{2c_2}$, then $5c_1n < 10c_2n^2$. So, if n is *large enough* (i.e., larger than $c_1/2c_2$ which is a **fixed constant** depending only on the device), $\mathcal{A}_1$ runs faster than $\mathcal{A}_2$. We focus on *asymptotic* runtime analysis, so as to understand the scaling of the runtime with the input size. In this setting, the case of n being smaller than a fixed constant is not relevant, and therefore, we can declare that $\mathcal{A}_1$ is asymptotically more efficient than $\mathcal{A}_2$.

Remark. The $o(\cdot)$ ("little-oh") and $\omega(\cdot)$ ("little-omega") notation were not introduced in the lecture this year, and so, you are not responsible for knowing their definitions. However, I have kept the discussion below for your general information.

The following definitions formally specify the notions of comparison in this asymptotic sense.

Definition 2.2.1 (Big Oh and Big Omega)**.**

- Big Oh (O): We write $f(n) = O(g(n))$, or $f(n) \leq O(g(n))$, if there exist constants $c > 0, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- Big Omega (Ω): We write $f(n) = \Omega(g(n))$, or $f(n) \geq \Omega(g(n))$, if there exist constants $c > 0, n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
- Theta (Θ): We write $f(n) = \Theta(g(n))$ if there exist constants $c_1, c_2, n_0 > 0$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$. Note that $f(n) = \Theta(g(n))$ iff $f(n) \leq O(g(n))$ and $f(n) \geq \Omega(g(n))$ simultaneously.

Definition 2.2.2 (Little Oh and Little Omega)**.**

- Little Oh (o): We write $f(n) = o(g(n))$, or $f(n) < o(g(n))$, if for any $c > 0$, there is an $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
- Little Omega (ω): We write $f(n) = \omega(g(n))$, or $f(n) > \omega(g(n))$, if for any $c > 0$, there is an $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

Let us give a few examples right away to illustrate the definitions:

Example 3.

- $2n^2 = O(n^3)$ ($c = 1$ and $n_0 = 2$), $3n^2 + 100n + 5 = O(n^3 + 10)$ ($c = 2$ and $n_0 = 10$), $n^2 - n = \Omega(n^2)$ ($c = 1/2, n_0 = 2$).
- $n = o(n^2)$ (for any $c > 0$, use $n_0 = 2/c$), $n^2 - n = \omega(n)$ (for any $c > 0$, use $n_0 = \frac{2}{c+1}$).

Informally, the idea is that O, Ω, Θ, o, and ω are qualitative substitutes for $\leq, \geq, =, <$, and $>$ respectively. For instance, if the instruction count for an algorithm $\mathcal{A}_1$ is $T_1(n)$ on inputs of size n and for $\mathcal{A}_2$ is $T_2(n)$, then if $T_1(n) = \Theta(T_2(n))$, this indicates that $\mathcal{A}_1$ and $\mathcal{A}_2$ have equal asymptotic time complexities. So, differences in performance between the algorithms must be due to particular characteristics of the computing device, and not due to the scaling of the algorithms' runtime with the input size. Similarly, $T_1(n) = o(T_2(n))$ means that for large enough[1] input sizes, $\mathcal{A}_1$ will display strictly smaller runtime than $\mathcal{A}_2$.

A few remarks need to be made at this point.

(1) For O and Ω, often the prefix "big" is omitted. So, for example, when we say "f is Oh of g", we mean $f(n) = O(g(n))$.

(2) To be more formal, asymptotic notation should really be defined in terms of sets of functions.

$$\begin{aligned}
O(g(n)) &= \{f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\} \\
\Omega(g(n)) &= \{f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0\} \\
\Theta(g(n)) &= \{f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\} \\
&= O(g(n)) \cap \Omega(g(n)) \\
o(g(n)) &= \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } 0 \leq f(n) < c \cdot g(n) \text{ for all } n \geq n_0\} \\
\omega(g(n)) &= \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } 0 \leq c \cdot g(n) < f(n) \text{ for all } n \geq n_0\}
\end{aligned}$$

Thus, if we say $n^3 + 5n^2 + 10n + 6 = n^3 + O(n^2)$, what this really means is that:

$$n^3 + 5n^2 + 10n + 6 = n^3 + f(n) \qquad \text{for some } f(n) \in O(n^2).$$

In this case, $f(n) = 5n^2 + 10n + 6$ which is in $O(n^2)$.

Similarly, in the next lecture, when we talk about recurrences, we will write equations like: $T(n) = 2T(n/2) + \Theta(n)$. This is shorthand for:

$$T(n) = 2T(n/2) + f(n) \qquad \text{for some } f(n) \in \Theta(n)$$

We will then show that $T(n) = \Theta(n \log n)$. More explicitly, this means that no matter what the choice of $f(n)$ is, $T(n)$ will be contained in the set $\Theta(n \log n)$.

Please also see the subsection in Chapter 3.1 of CLRS (pp. 49–50) titled ***Asymptotic notation in equalities and inequalities*** for a discussion of further subtleties in similar contexts.

(3) Another common reason for confusion is the use of functional notation. When we write $\Omega(n)$, here "n" does not denote a particular value but rather the[2] function $g(n) = n$. Similarly, if we write $o(1)$, the 1 refers to the constant function $g(n) = 1$, not the number 1. More explicitly, $f(n) = o(1)$ means:

$$\forall c > 0, \exists n_0 > 0 \text{ such that } 0 \leq f(n) < c \text{ for all } n \geq n_0.$$

So, $1/n = o(1)$ but $n^{.001} \neq o(1)$. (Make sure you understand why!)

[1]There is an issue here in that maybe n_0 is so large that inputs of size larger than n_0 never actually show up in practice. This may indeed be true, but in many cases of interest, n_0 is not too large and so asymptotic analysis is meaningful.

[2]g is an arbitrary name here. If you're familiar with λ-calculus terminology, it would be more appropriate to write this as an anonymous function: $\lambda n.n$.

The following result is often the most convenient way to measure asymptotic growth.

Lemma 2.2.3. *Assume $f(n), g(n) > 0$.*

$$\begin{aligned}
\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0 &\quad\Longrightarrow\quad f(n) = o(g(n))\\
\lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty &\quad\Longrightarrow\quad f(n) = O(g(n))\\
0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty &\quad\Longrightarrow\quad f(n) = \Theta(g(n))\\
\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0 &\quad\Longrightarrow\quad f(n) = \Omega(g(n))\\
\lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty &\quad\Longrightarrow\quad f(n) = \omega(g(n))
\end{aligned}$$

Two quick examples:

$$\lim_{n\to\infty}\frac{n^3+3n^2+4n+1}{n^2} = \lim_{n\to\infty} n+3+\frac{4}{n}+\frac{1}{n^2} = \infty \quad\Longrightarrow\quad n^3+3n^2+4n+1 = \omega(n^2)$$

$$\lim_{n\to\infty}\frac{3n\lg(n)+4\lg(n)+2}{n\lg(n)} = \lim_{n\to\infty} 3+\frac{4}{n}+\frac{2}{n\lg(n)} = 3 \Longrightarrow 3n\lg(n)+4\lg(n)+2 = \Theta(n\lg(n))$$

As you see above, knowing how to evaluate limits is key to figuring out asymptotic relationships. Recall L'Hôpital's rule (simplified for our context): If f and g are positive functions, differentiable for all large enough x, and $g'(x) \neq 0$ for all large enough x, then:

$$\lim_{x\to\infty} f(x) = \infty, \lim_{x\to\infty} g(x) = \infty \quad\Longrightarrow\quad \lim_{x\to\infty}\frac{f(x)}{g(x)} = \lim_{x\to\infty}\frac{f'(x)}{g'(x)}$$

as long as the limit on the right exists.

Everyone in computer science worth their salt knows two rules of thumb about asymptotic growth, and you should too! Logs grow slower than polynomials, and polynomials grow slower than exponentials. Here is a general statement to this effect.

Lemma 2.2.4.

(i) For any constants $k, d > 0$, $(\lg n)^k = o(n^d)$.

(ii) For any constants $d > 0, u > 1$, $n^d = o(u^n)$.

Proof. (i)

$$\lim_{n\to\infty}\frac{(\lg n)^k}{n^d} = \left(\lim_{n\to\infty}\frac{\lg n}{n^{d/k}}\right)^k \overset{\text{L'Hôpital}}{=} \left(\lim_{n\to\infty}\frac{(\lg e)/n}{(d/k)n^{d/k-1}}\right)^k = \left(\lim_{n\to\infty}\frac{(k\lg e)/d}{n^{d/k}}\right)^k = 0.$$

(ii)

$$\lim_{n\to\infty}\frac{n^d}{u^n} = \left(\lim_{n\to\infty}\frac{n}{u^{n/d}}\right)^d \overset{\text{L'Hôpital}}{=} \left(\lim_{n\to\infty}\frac{1}{\frac{\ln u}{d}u^{n/d}}\right)^d = 0.$$

□

We end these notes with a couple of pathological examples (though you should be aware that such situations are somewhat artificial and don't often arise in the analysis of algorithms).

Example 4. Let $f(n) = n$ and $g(n) = n^{1+\sin(n)}$. Then, because of the oscillating behavior of the sine function, there is no n_0 after which f dominates g or f is dominated by g. So, we cannot compare f and g using asymptotic notation.

Example 5. Let $f(n) = n$ and $g(n) = n(2 + \sin(n))$. Note that $\frac{1}{3}g(n) \leq f(n) \leq g(n)$ for all $n \geq 0$, and so, $f(n) = \Theta(g(n))$. On the other hand, we cannot apply Lemma 2.2.3 because $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{1}{2+\sin(n)}$ does not exist.