CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 2

January 17, 2022

Instructor: Sepehr Assadi

1 Runtime Efficiency and Asymptotic Analysis

As we stated before, in this course, we are primarily interested in the running time of the algorithms as their main measure of *efficiency*. However, computing the *exact* running time of the algorithms is quite hard (and is specific to the "computer" that runs the algorithm). Instead, we consider the following criteria for measuring the runtime of an algorithm:

1. Count the number of *basic* operations: simple arithmetic (e.g. adding or subtracting two numbers), conditional-statements (e.g., if- or while-statements), memory access (e.g., reading from or writing to an array cell), etc.

2. We count the number of operations *asymptotically*: this basically means that we *ignore* constant factors and assume the *input size* is very large (goes to infinity in limit).

   Example: How many basic operations are done in the following algorithm? Iterate over the elements of an integer array of size n: for each element, output that element multiplied by 10.

   At this stage, it is virtually impossible to *exactly* determine the number of operations done by the above algorithm: Is the algorithm maintaining a counter when iterating over the array? Do we need to also account for updates to that counter? Is outputting the element multiplied by 10 a single operation or multiple ones (e.g., computing the multiplication, storing it in a temporary variable, and then outputting it)? Is checking whether we reach the end of the for-loop yet another operation? If so, how many times we are running this operation?

   On the other hand, it is easy to see that this algorithm makes $\Theta(1)$[1] (namely some *constant* number of) operations for each element of the array, there are n elements in the array, and any extra book keeping, initialization etc. takes another $\Theta(1)$ operations. Thus, the runtime of the algorithm is $\Theta(n)$.

3. We analyze the *worst-case* runtime of the algorithm, i.e., the maximum time it may take on *any* (valid) input (unless specifically stated otherwise).

   Example: What is the runtime of the following algorithm? Iterate over the elements of an integer array A of size n: if for the current index i, $A[i] < A[i-1]$ terminate; otherwise continue.

   Determining the runtime of this algorithm (even asymptotically) crucially depends on the extra knowledge about the array A. For instance, if $A[1] > A[2]$, this algorithm only takes $\Theta(1)$ time. On the other hand, if all elements of the array are sorted in the increasing order, then this algorithm in fact takes $\Theta(n)$ time, and this is the maximum it may take on any input. As such, we say that the worst-case runtime of the algorithm is $\Theta(n)$.

So with these criteria, we can say that the runtime of our algorithm from previous lecture for finding maximum is $\Theta(n)$: the first and last step only take $\Theta(1)$ time and the second step involves a for-loop of length n which takes $\Theta(n)$ time. This concludes our entire design and analysis of an algorithm for the problem of maximum of an array[2].

[1] If you do not know what this notation means, see the next section first.

[2] Let us emphasize that proving correctness of the algorithm and analyzing its runtime is also a part of algorithm design

2 Review of Asymptotic Notation

Throughout the course, we will use $O(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$ notation when analyzing the algorithms in order to *hide* constants. This is called the *asymptotic notation*. Ideally, you should have seen this in the data structures course (or discrete math), but let us do a quick refresher for us to be all on the same page.

We start by reviewing the following two basic questions.

- **Why do we need asymptotic notation?** We primarily use asymptotic notation to denote how the *running time* of the algorithms *grow* as a function of the input size n. Asymptotic notation allows us to focus on the "key part" of the running time and make our life much easier when computing and comparing running time of different algorithms. It also gives us a very good idea of how the runtime of a single algorithm scales as a function of n: for an algorithm with runtime $\Theta(n)$, increasing the input size by a factor of 10 would increase the runtime also by a factor of 10, while for an algorithm with runtime $\Theta(n^2)$, the same increase in the input size makes the algorithm run 100 times slower.
- **Why is it okay to ignore constants?** Asymptotic notation does *not* target *small* inputs: it is entirely possible that an algorithm with worse asymptotic bounds behave better than an algorithm with better bounds on a *small enough* input (e.g. a linear search on an input of size 10 is going to be faster than binary search on the same input). However, we are not usually interested in designing efficient algorithms for small inputs. *At scale* it is not the constants that matter, it is the asymptotics. When considering typically large problems (or better yet "big data" problems), the constants are completely dominated by the asymptotics. In other words, on a *sufficiently large input*, an algorithm with better asymptotic bound runs *faster* also.

Now that we have seen why asymptotics matter, let us define them properly.

$\mathbf{O}(\cdot)$-notation: We are going to use O-notation to *upper bound* the running time of our algorithms (think of O as being the '$\leq$' relation in the limit). Considering what we just discussed, we want a way of comparing two running times for *large* inputs. In other words, we want to say that an algorithm with running time $f(n)$ is going to take time proportional to some simpler function $g(n)$ if we *ignore constants* and consider large enough n. More formally, we want to say that *in the limit (for n)*, $f(n)$ is at most $g(n)$ *times* some constant C. The formal mathematical way of stating this sentence is:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq C. \qquad \text{(for some constant C independent of } n\text{)}$$

Question? Why not instead write $\lim_{n\to\infty} f(n) \leq C \cdot \lim_{n\to\infty} g(n)$? This is simply because for any choices of $f(n)$ and $g(n)$ which are *not* constant, both left hand side and right hand side of this inequality is $+\infty$; but then we cannot compare these two different infinities with each other. To conclude, this is *not* the correct way for formalizing the sentence above.

Examples: Suppose we have two algorithms for the same problem, one with running time $1000 \cdot n$ and the other with running time n^2. Which of the two algorithms is faster for *large* n? The answer is $1000 \cdot n$. If we use the asymptotic notation also we have $1000 \cdot n = O(n^2)$; why? an easy check is the following:

$$\lim_{n\to\infty} \frac{1000 \cdot n}{n^2} = \lim_{n\to\infty} \frac{1000}{n} = 0. \qquad \text{(which is a constant independent of } n\text{)}$$

Throughout the course, it is worth remembering the following relations; you can easily prove all of them using the definition above but you do not need to do so in your homeworks or exams. For any $c > 0$:

$$n^c = O(n^{c+1}) \qquad (\log n)^c = O(n) \qquad n^c = O(2^n) \qquad c^n = O((c+1)^n). \tag{1}$$

(and a crucial part); thus, throughout this course, whenever we say design an algorithm for some problem, it is always assumed (unless specifically stated otherwise) that you will also prove its correctness and analyze its runtime.

Example. Let us see another example. A similar question like this appears in your homework – for that question, you can use any of the facts we use and prove here by simply mentioning them (without rewriting the proof explicitly).

Rank the following functions based on their asymptotic value in the increasing order, i.e., list them as functions f_1, f_2, f_3, and f_4 such that $f_1 = O(f_2)$, $f_2 = O(f_3)$, and $f_3 = O(f_4)$.

$$2^{\sqrt{\log n}} \qquad \sqrt{\log n} \qquad \log\log\log n \qquad n^{1/\log\log n}$$

The listing is as follows:

$$f_1 = \log\log\log n \qquad f_2 = \sqrt{\log n} \qquad f_3 = 2^{\sqrt{\log n}} \qquad f_4 = n^{1/\log\log n}.$$

We now prove each part:

- $\log\log\log n = O(\sqrt{\log n})$. Let us first use the transitivity of O-notation (similar to '$\leq$'): since $\log\log\log n = O(\log\log n)$, we can instead prove that $\log\log n = O(\sqrt{\log n})$. We do a change of variable: we write $m = \log n$ which means that we only need to prove $\log m = O(\sqrt{m})$. But this is equivalent to proving that $(\log m)^2 = O(m)$, and this is true by the second term in Eq (1).
- $\sqrt{\log n} = O(2^{\sqrt{\log n}})$. Let us do a change of variable: we define $m := \sqrt{\log n}$ so we only need to prove that $m = O(2^m)$ – this is true by the third term in Eq (1).
- $2^{\sqrt{\log n}} = O(n^{1/\log\log n})$. We do a change of variable first: define $m = \sqrt{\log n}$ so we only need to prove that $2^m = O(2^{m^2/2\log m})$. We prove this using the limit:

  $$\lim_{m\to\infty} \frac{2^m}{2^{m^2/2\log m}} = 0;$$

  We use the following standard rule for limits: if $\lim_{m\to\infty} f(m) - g(m) = -\infty$ then $\lim_{m\to\infty} \frac{2^{f(m)}}{2^{g(m)}} = 0$ (proof: for the second term to be true, we need to show that for every $\varepsilon > 0$, there exists some large enough m where $2^{f(m)}/2^{g(m)} < \varepsilon$. By taking a log from both sides we only need to prove that $f(m) < \log(\varepsilon) + g(m)$. But this is true by the first limit since it states that for large enough m, $f(m) - g(m)$ is smaller than any $\varepsilon' = \log(\varepsilon)$.)

  So it suffices to show that $\lim_{m\to\infty} m - m^2/2\log m = -\infty$. This is true because:

  $$\lim_{m\to\infty} m - m^2/2\log m = \lim_{m\to\infty} \frac{m^2}{2\log m} \cdot (\frac{2\log m}{m} - 1) = \lim_{m\to\infty} \frac{m^2}{2\log m} \cdot (-1) = -\infty.$$

  This concludes the proof.

$\Omega(\cdot)$-notation: We will use Ω-notation to *lower bound* the running time of our algorithms (think of Ω as being the '$\geq$' relation in the limit). In exactly the same spirit as O-notation, we want $f(n) = \Omega(g(n))$ to mean that *in the limit (for n)*, $f(n)$ is at least $g(n)$ *times* some constant C. The formal mathematical way of stating this sentence is:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq C. \qquad \text{(for some constant C independent of } n)$$

Note that by definition, $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$. This in particular means that we can use everything we know about the O-notation for proving new results for Ω-notation by simply replacing the two sides of the equation. For instance, all the following equations are simply restatements of Eq (1):

$$n^{c+1} = \Omega(n^c) \qquad n = \Omega((\log n)^c) \qquad 2^n = \Omega(n^c) \qquad (c+1)^n = \Omega(c^n).$$

$\Theta(\cdot)$-notation: Finally, the combination of the previous two notations is the Θ-notation (think of Θ as being the '=' relation in the limit). In other words, we write $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. When having two algorithms with running times that are Θ of each other, we know that both algorithms *asymptotically* have the same running time.

By the previous connection to O- and Ω-notation, we already know how to prove that $f(n) = \Theta(g(n))$; what if we want to instead prove that $f(n) \neq \Theta(g(n))$? Well then we need to either prove that $f(n) \neq O(g(n))$ or $f(n) \neq \Omega(g(n))$. But how do we prove that $f(n) \neq O(g(n))$? By definition, we should simply show that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \qquad \text{(because we want this to be larger than } \textit{every} \text{ constant } C)$$

(to show that $f(n) \neq \Omega(g(n))$ we can simply show $g(n) \neq O(f(n))$ using the method above).

Finally, let us simply define two new notation which correspond to *strict inequalities* in the limit:

- **$o(\cdot)$-notation:** We write $f(n) = o(g(n))$ if and only if $f(n) = O(g(n))$ but $f(n) \neq \Theta(g(n))$ – in other words, if $f(n)$ grows *strictly slower* than $g(n)$ in the limit (think of o as '$<$' relation in the limit).

  Equivalently, $f(n) = o(g(n))$ if and only if $f(n) \neq \Omega(g(n))$.

- **$\omega(\cdot)$-notation:** We write $f(n) = \omega(g(n))$ if and only if $f(n) = \Omega(g(n))$ but $f(n) \neq \Theta(g(n))$ (or equivalently, if $g(n) = o(f(n))$) – in other words, if $f(n)$ grows *strictly faster* than $g(n)$ in the limit (think of ω as '$>$' relation in the limit).

  Equivalently, $f(n) = \omega(g(n))$ if and only if $f(n) \neq O(g(n))$.

Remark. It is worth mentioning that all the O-notation in Eq (1) can also be switched to the o-notation.

Example. Let us see another example. Consider the following three different functions $f(n)$:

$$n! \qquad \log\log n \qquad n^5;$$

For each of these functions, determine which of the following statements is true and which one is false.

- $f(n) = \Theta(f(n-1))$;
- $f(n) = \Theta(f(4n))$;
- $f(n) = \Theta(f(\log\log n))$.

We consider the functions one by one:

1. For $n!$ all three equalities are false. Proof:
   - $n! \neq \Theta((n-1)!)$. To prove this, we show that $n! \neq O((n-1)!)$. This is because:
   $$\lim_{n\to\infty} \frac{n!}{(n-1)!} = \lim_{n\to\infty} n = +\infty.$$
   - $n! \neq \Theta((4n)!)$. We show that $n! \neq \Omega((4n)!)$ or equivalently $(4n)! \neq O(n!)$. This is because:
   $$\lim_{n\to\infty} \frac{(4n)!}{n!} = \lim_{n\to\infty} (4n)\cdot(4n-1)\cdots(n+1) = +\infty.$$
   - $n! \neq \Theta((\log\log n)!)$. We show that $n! \neq O((\log\log n)!)$. This is because $n > (\log\log n)$ and hence:
   $$\lim_{n\to\infty} \frac{n!}{(\log\log n)!} = \lim_{n\to\infty} n\cdot(n-1)\cdots(\log\log n + 1) = +\infty.$$

2. For $\log\log n$ the first two equalities are correct while the last one is false. Proof:

   - $\log\log n = \Theta(\log\log(n-1))$. For all n, we have $\log\log(n-1) \leq \log\log n$ and for $n \geq 16$, we have $(n-1) \geq n/2$ (because $16-1 \geq 16/2$), and $(\log n - 1) \geq \log n/2$ (because $4-1 \geq 4/2$) and $\log\log n - 1 \geq \log\log n/2$ (because $2-1 \geq 2/2$). Thus, we also have that

     $$\log\log(n-1) \geq \log\log(n/2) = \log\log n - 1 \geq \log(\log n/2) = \log\log n - 1 \geq \log\log n/2.$$

     So as n goes to infinity, $1/2 \cdot \log\log n \leq \log\log(n-1) \leq \log\log n$. As such, $\log\log n = O(\log\log(n-1))$ (by the first inequality) and $\log\log n = \Omega(\log\log(n-1))$ (by the second one).
   - $\log\log n = \Theta(\log\log(4n))$. For all $n > 1$, $\log\log 4n = \log 2 + \log n \leq 2\log\log n$. Moreover, $\log\log n \leq \log\log(4n)$ clearly. Thus again, $\log\log n = \Theta(\log\log(4n))$.
   - $\log\log n \neq \Theta(\log\log\log\log n)$. We show that $\log\log n \neq O(\log\log\log\log n)$. Let us first do a change of variable by writing $m = \log\log n$. So we prove that $m \neq O(\log\log m)$. To show this, we show that $\log\log m \neq \Omega(m)$ or equivalently $\log\log m = o(m)$. This follows from the remark that Eq (1) also holds in the o-notation and hence $\log\log m = o(\log m)$ and $\log m = o(m)$ and so by transitivity, $\log\log m = o(m)$.

3. For n^5 the first two equalities are correct while the last one is false. Proof:

   - $n^5 = \Theta((n-1)^5)$. Clearly $n^5 = \Omega((n-1)^5)$ so we only need to show that $n^5 = O((n-1)^5)$. However, by expanding, $(n-1)^5 = n^5 - 5n^4 + 10n^3 - 10n^2 + 5n - 1$ and the dominant term is n^5 and thus $n^5 = O((n-1)^5)$ as well.
   - $n^5 = \Theta((4n)^5)$. We simply have $(4n)^5 = 4^5 \cdot n^5$ and after ignoring the leading constant of 4^5 the two terms are equal, thus proving the claim.
   - $n^5 \neq \Theta((\log\log n)^5)$. We show that $n^5 \neq O((\log\log n)^5)$, by proving $(\log\log n)^5 = o(n^5)$. Firstly, since Eq (1) also holds for the o-notation, we have $(\log\log n)^5 = o(\log n)$ (you can see this either directly or by changing variable $m = \log n$) and $\log n = o(n)$ and finally $n = o(n^5)$ so by transitivity, $(\log\log n)^5 = o(n^5)$.