CS 240 — Lecture 3

Bartosz Antczak *Instructor: Eric Schost* *January 10, 2017*

3.1 Growth Rates

As the amount of data input gets bigger, how much more resource will an algorithm require? That is what a **growth rate** determines. More formally, a growth rate determines how quickly a function $f(n)$ grows as n increases.
Some common growth rates include (we use theta-notation because they tell us more about the algorithm than big-O notation does, which can be useful)

- $\theta(1)$: constant time
- $\theta(\log n)$: logarithmic complexity
- $\theta(n^2)$: quadratic complexity

Say we doubled the size of the problem instance ($n \to 2n$), how will the running time of certain algorithms be affected? It depends on their growth rate:

- Linear complexity ($T(n) = cn$): $T(2n) = 2T(n)$ (doubles the run time)
- Cubic complexity ($T(n) = cn^3$): $T(2n) = 8T(n)$ (increases run time by a factor of eight)

The above shows how different growth rates affect an algorithm. The larger the growth rate, the smaller the problem instance needed to reach a certain runtime.

3.1.1 Complexity vs. Growth Rate

Just because an algorithm has a simpler complexity than another algorithm, that doesn't mean that the simpler algorithm will *always* be faster than the second algorithm.

Example 3.1.1. *Consider $T_1(n) = 75n + 100$ and $T_2(n) = 5n^2$. When $n < 20$, $T_2(n)$ is faster (even though it has a larger growth rate).*

3.1.2 Better understanding Order Notation

Suppose that we have two runtimes, $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Consider

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Based on the value of L, we can sufficiently determine the runtime of $f(n)$:

- If $L = 0$, then $f(n) \in o(g(n))$
- If $0 < L < \infty$, then $f(n) \in \theta(g(n))$
- If $L = \infty$, then $f(n) \in \omega(g(n))$

This method can be used as an alternative to proving various runtimes in order notation (we've covered some proofs of order notation in lecture 2). However, not all order notation problems can be proven this way, such as proving that $n(\sin(\frac{n\pi}{2})) \in \theta(n)$.

Example 3.1.2. *Compare the growth rates of $f(n) = \log n$ and $g(n) = n^i$ (where $i > 0$).*

Here, we see that $\lim_{n\to\infty} \log n = \lim_{n\to\infty} n^i = \infty$, so L'Hopital's rule applies:

$$\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \frac{1/n}{in^{i-1}} = \frac{1}{in^i} \tag{3.1}$$

$$= 0 \tag{3.2}$$

This means that our L that we calculated is equal to zero, thus $\log n \in o(n^i)$.

3.1.3 Some Useful Relationships between Order Notations

- $f(n) \in \theta(g(n)) \iff g(n) \in \theta(f(n))$
- $f(n) \in \theta(g(n)) \iff f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in o(g(n)) \implies f(n)O(g(n))$

3.1.4 Algebra of Order Notations

We can manipulate order notation using certain algebra rules to yield some useful results:

- "Maximum" rule: $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$ (this rule can also be applied to θ-notation and Ω-notation)
- Transitivity (covered in previous lecture)

3.2 Algorithm Analysis Techniques

Mainly, use θ-bounds throughout your analysis. As previously mentioned, a θ-bound describes the most amount of information about an algorithm. To get a θ-bound, prove an O-bound and a matching Ω-bound separately.

3.2.1 Loop Analysis

Example 3.2.1. *An algorithm that calculates a specific sum.*

```
sum = 0
for (i = 1; i < n)
  for (j = i; j < n)
    sum = sum + (i - j)^2
    sum = sum^2
return sum
```

The number of operations that happen in this code is modelled by:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}(n-i+1) \tag{3.3}$$

$$= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 \tag{3.4}$$

$$= n^2 - \frac{n(n+1)}{2} + n = \frac{1}{2}n^2 + \frac{1}{2}n \tag{3.5}$$

$$\tag{3.6}$$

As expected, this algorithm has a runtime of $\theta(n^2)$.

Example 3.2.2. *An algorithm that returns the largest number in an array.*

```
max = 0
for (i = 1; i < n)
  for (j = i; j < n)
    sum = 0
    for (k = i; k < j)
      sum = A[k]
      if (sum > max) {
        max = sum
      }
return max
```

We calculate the number of operations in this code as:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 = \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) \tag{3.7}$$

Note that $\sum_{j=i}^{n}(j-i+1) = 1+2+3+\cdots+(n-i+1)$, thus we have

$$\sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2} \tag{3.8}$$

Now let $\ell = n-i+1$. We see that ℓ decreases at the same rate as i increases:

- When $i = 1$, $\ell = n$
- When $i = n$, $\ell = 1$

So the sum can be rewritten as

$$\sum_{\ell=1}^{n}\frac{\ell(\ell+1)}{2}$$

Which means that

$$\left(\sum_{\ell=1}^{n}\frac{\ell}{2}\ell^2\right) \in \theta(n^3) \leq \sum_{\ell=1}^{n}\frac{\ell(\ell+1)}{2} \leq \left(\sum_{\ell=1}^{n}\ell^2\right) \in \theta(n^3)$$

Thus, this algorithm has a runtime of $\theta(n^3)$.