CSM 61B

Spring 2018

Asymptoptics Crib Sheet

March 10, 2018

1 Overview

Asymptotic analysis is the analysis of the runtime of a program with respect to the input size and as the input $\rightarrow \infty$. The runtime can be bound using the following notations:

- Big O, $f(n) = O(g(n))$: upper bound. $f(n)$ grows no faster than $g(n)$.
- Big Omega, $f(n) = \Omega(g(n))$: lower bound. $f(n)$ grows no slower than $g(n)$.
- Big Theta, $f(n) = \Theta(g(n))$: tight bound. $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. $f(n)$ grows as fast as $g(n)$.

2 Runtime Analysis

Sum up the work done in the program being analyzed and simplify the expression using the following guidelines and summations.

Guidelines:

- Ignore lower order terms (ex: $(N^2 + 2N + 1) = \Theta(N^2)$)
- Ignore constant scaling factors (ex: $6 \log N = O(\log N)$)
- Constant < logarithmic < linear < polynomial < exponential
- Any polynomial > any power of a log (ex.: $N > \log^k(N)$)
- All logarithms are proportional to each other by the Change of Base formula

Common summations:

- $1 + 2 + \cdots + N = \sum_{i=1}^{n} i = \Theta(N^2)$
- $1 + 2 + 4 + 8 + \cdots + N = \sum_{i=0}^{\log N} 2^i = \Theta(N)$
- $1 + 2 + 4 + 8 + \cdots + 2^N \sum_{i=0}^{N} 2^i = \Theta(2^N)$

3 Amortized Analysis

Amortized analysis considers the "average" runtime of a function over a series of calls to the function.

3.1 ArrayList insertions: Insertions are $\Theta(1)$ in the best case when no resize is needed and $\Theta(N)$ in the worst case when the ArrayList runs out of space and needs to resize, because it must copy all the values into a new array. This is equivalent to $\Omega(1)$ and $O(N)$ overall.

(a) Resize every c elements, where c is a constant: The amortized cost is $\Theta(N)$. If the scheme is to add 99 spots to the end of the ArrayList each time it resizes, the amortized cost is $\frac{\Theta(N)}{99}$ which reduces to $\Theta(N)$. This is because 99 doesn't scale with size (imagine resizing an array that has 999,999,999 elements every 99 times).

(b) Resize every $\frac{N}{2}$ elements: The amortized cost is $\Theta(1)$. If the scheme is to double the size of the ArrayList each time it resizes, the amortized cost is $\frac{\Theta(N)}{\frac{N}{2}}$. This is because the resizing scheme scales with size such that no matter how big the ArrayList becomes, for size N, resizing only occurs every $\frac{N}{2}$ times. As the size of the ArrayList increases, the less frequently you will need to resize.

4 Examples

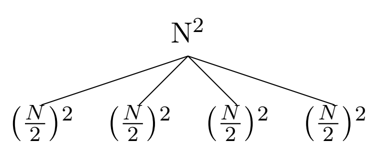
4.1 Write $f(n)$ in terms of $g(n)$ using O or Ω where $f(n) = n^{1.001}$ and $g(n) = 10^n$.

Solution: $f(n) = O(g(n))$ and $g(n) = \Omega(f(n))$

4.2 Give a tight asymptotic bound for `quad` as a function of N and draw a tree. If possible, give a $\Theta(\cdot)$ bound for the overall runtime. Otherwise, provide a $\Theta(\cdot)$ bound for both the best case and worst case runtime.

```java
public static void quad(int N) {
    if (N == 0) {
        return;
    }
    quad(N/2);
    quad(N/2);
    quad(N/2);
    quad(N/2);
    g(N); //this runs in O(Nˆ2) time
}
```

Solution: In this function, we can draw a tree with a branching factor of 4, with $O(\frac{N}{2^k})^2$ work being done at each node where k is the depth starting at 0. If we draw out the first two layers:



The first layer does $O(N^2)$ work. The next layer does $O(4 \cdot (\frac{N}{2})^2)$ work, also summing to $O(N^2)$ work. In total, there are $\log N$ layers since we continuously divide N by 2, and there are 4^k nodes per layer since each function call makes 4 more calls to `quad`. To find the runtime, we multiply the work per layer by the number of layers:
$\frac{work}{layer} \cdot \#\text{ layers} = N^2 \cdot \log N = O(N^2 \log N)$.