CS 161 Fall 2017: Section 1 Solutions

Asymptotic Analysis

For each of the following functions, prove whether $f = O(g)$, $f = \Omega(g)$, or both ($f = \Theta(g)$). (For example, by specifying some explicit constants $n_0, c > 0$ (or n_0, c_1, c_2 in the case that $f = \Theta(g)$) such that the definition of Big-Oh, Big-Omega, or Big-Theta is satisfied.)

(a) $f(n) = n \log\left(n^3\right)$ $\quad g(n) = n \log n$

(b) $f(n) = 2^{2n}$ $\quad g(n) = 3^n$

(c) $f(n) = \sum_{i=1}^{n} \log i$ $\quad g(n) = n \log n$

(a) $f(n) \in \Theta(g(n))$. Since $f(n) = n \log\left(n^3\right) = 3n \log n$, choosing $c_1 = 2$ and $c_2 = 4$ bounds the function for all $n \geq 1$.

(b) $f(n) \in \Omega(g(n))$. To see why, $f(n) = 2^{2n} = 4^n$. Choosing $c = 1$ lower bounds the function for all n that satisfy $(4/3)^n \geq 1$ or $n \geq 1$.

(c) $f(n) \in \Theta(g(n))$. Inspect the terms of the summation:

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \ldots + \log(n/2) + \ldots + \log n$$

To see that the summation is upper bounded by $n \log n$, notice the expansion consists of n terms of at most $\log n$, so $\sum_{i=1}^{n} \log i \leq c_2 n \log n$ for $c_2 = 1$ and $n \geq 2$.

To see that the summation is lower bounded by $n \log n$, notice the expansion also consists of $n/2$ terms of at least $\log(n/2)$:

$$\sum_{i=1}^{n} \log i \geq (n/2)\log(n/2) = (n/2)(\log n - \log 2) \geq c_1 n \log n$$

Rearranging terms of the second inequality shows that $(n/2)(\log n - \log 2) \geq c_1 n \log n$ holds as long as $(1/2 - c_1)\log n \geq (1/2 \log 2)$; let's choose $c_1 = 1/3$ and $n \geq 8$.

Recurrence Relations

Recall the Master theorem from lecture:

Theorem 0.1. *Given a recurrence $T(n) = aT(\frac{n}{b}) + O(n^d)$ with $a \geq 1$, and $b > 1$, and $T(1) = \Theta(1)$, then*

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

What is the Big-Oh runtime for algorithms with the following recurrence relations?

(a) $T(n) = 3T(\frac{n}{2}) + \Theta(n^2)$

(b) $T(n) = 4T(\frac{n}{2}) + \Theta(n)$

(c) $T(n) = 2T(\sqrt{n}) + O(\log n)$

(a) Using the Master Theorem, $a = 3$, $b = 2$, and $d = 2$. Since $a = 3 < b^d = 4$, we fall into the second case. So, the runtime is $O(n^d) = O(n^2)$.

(b) Using the Master Theorem, $a = 4$, $b = 2$, and $d = 1$. Since $a = 4 > b^d = 2$, we fall into the third case. So, the runtime is $O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$.

(c) This problem also does not fit directly into the formula Master Theorem. However, we can massage this equation into a form that the theorem can work with. Define $k = \log n$, meaning that $n = 2^k$, and $\sqrt{n} = 2^{k/2}$. In terms of k, the recurrence formula is now:

$$T(2^k) = 2T(2^{k/2}) + O(k)$$

Next, define a function $S(k) = T(2^k)$. Now, rewrite the recurrence as:

$$S(k) = 2S(\frac{k}{2}) + O(k)$$

This expression matches the recurrence relation we've seen with MergeSort, so

$$S(k) = O(k^d \log k) = O(k \log k)$$

To get the bound in terms of n, we replace k with $\log n$, to get the bound:

$$T(n) = O(\log n \log(\log n))$$

Divide and Conquer: Majority Element

Suppose we are given an array, A, of length n, with the promise that there exists some number, x, that occurs at least $n/2 + 1$ times in the array. Additionally, we are only allowed to check whether two elements are equal (no comparisons).

(a) Complete the following pseudo-code for a divide-and-conquer algorithm that returns the majority element of A. Feel free to assume that the n is a power of 2.

```
MajorityElement(Input: array A of length n)
If n = 1, return A[1]
Else
   Let m1 = MajorityElement(A[1:n/2])
   Let m2 = MajorityElement(A[n/2+1:n])

   Let count = 0
   Foreach x in A
      If m1 = x
         count ++
   If count > n / 2
      Return m1
   Else Return m2
```

(b) Give a brief but formal proof of the correctness of your algorithm. Again, feel free to assume $n = 2^s$ for some integer s. [Hint: induction on s!!]

We proceed by induction on s, where $n = 2^s$. The base case, where $s = 0$, is trivially satisfied by the algorithm. Assuming the algorithm is correct for inputs of length $n/2 = 2^{s-1}$, consider an input of length $n = 2^s$. The majority element of the entire array must be the majority element of at least one of $A[1 : n/2]$ or $A[n/2 + 1 : n]$ since otherwise it would occur at most $n/2$ times. Hence, by our inductive hypothesis, either $m1$ or $m2$ (or both) must be the majority element. The remainder of the code checks if $m1$ is the majority element, and if it is not, then $m2$ must be the majority element, and the code outputs $m2$. This establishes the correctness for arrays of size $n = 2^s$, and by induction, the algorithm is correct for any input size that is a power of 2.

(c) Express the runtime of your algorithm via a recurrence relation, and solve the relation to give the asymptotic (Big-Oh) runtime of your algorithm.

$T(n) = 2T(n/2) + \Theta(n)$. Using Master theorem, this is $\Theta(n \log n)$.