Asymptotic Notation

We will use aysmptotic notation frequently in this class in describing the performance characteristics (runtime, space consumption) of the algorithms we explore. The set definitions for this notation are given below

$$
\begin{aligned}
O(g(n)) = \quad & \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\
& \text{for all } n > n_0, 0 \leq f(n) \leq cg(n)\} \\[1em]
\Omega(g(n)) = \quad & \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\
& \text{for all } n > n_0, 0 \leq cg(n) \leq f(n)\} \\[1em]
\Theta(g(n)) = \quad & \{f(n) : \text{there exist positive constants } n_0, c_1, c_2 \text{ such that} \\
& \text{for all } n > n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} \\[1em]
o(g(n)) = \quad & \{f(n) : \text{for all } c > 0, \text{ there exists a positive } n_0 \text{ such that} \\
& \text{for all } n > n_0, 0 \leq f(n) \leq cg(n)\} \\[1em]
\omega(g(n)) = \quad & \{f(n) : \text{for all } c > 0, \text{ there exists a positive } n_0 \text{ such that} \\
& \text{for all } n > n_0, 0 \leq cg(n) \leq f(n)\}
\end{aligned}
$$

Note: These definitions require that every member of $f(n)$ is asymptotically nonnegative: $f(n) \geq 0$ whenever n is sufficiently large. Therefore $g(n)$ must also by asymptotically nonnegative, or else no $f(n)$ fulfills the definition. In general we will assume that any function used in this notation is asymptotically nonnegative.

Using these set definitions we will use the notation $f(n) = O(g(n))$ to indicate set membership: $f(n) \in O(g(n))$. We will also use asymptotic notation in equations; for example, you may see a recurrence equation such as $T(n) = T(n/2) + O(n^2)$. In this case $O(n^2)$ refers to some anonymous function in the set $O(n^2)$.

Recurrences

Recall the Merge Sort Algorithm: Given an array of n elements we wish to sort, we divide this list in half, recursively Merge Sort each half of the list, then merge the two sorted lists.

This algorithm is a recursive algorithm, and its runtime can be represented with the following recurrence formula:

$$T(n) = 2T(n/2) + n$$

We would like a closed form for the runtime of this algorithm, so we're going to have to solve this recurrence. You should already be familiar with two ways to do this: the substitution method, and the recursion tree method. Today I'm going to introduce a very useful method for solving recurrences, called the Master Method.

The Master Method

The Master Method is a convenient method for solving recurrences of the form $T(n) = aT(n/b) + f(n)$. Note that the form of the Master Method given below is more general than the form in CLRS; the additional $\lg n$ factor comes from the height of the recurrsion tree when both $n^{\log_b a}$ and $f(n)$ approximately match.

> Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where n/b corresponds to either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then
>
> 1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
> 2. If $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
> 3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Exercise: Use the master method to solve the recurrence formula for Merge Sort.

Problems

Evaluate the following recurrences.

1. $T(n) = 4T(n/2) + n^3$

   Solution: Case 3; $T(n) = \Theta(n^3)$

2. $T(n) = 9T(n/3) + n^2 \lg n$

   Solution: Case 2; $T(n) = \Theta(n^2 \lg^2 n)$

3. $T(n) = 5T(n/2) + n^2 \lg n$

   Solution: Case 1; $T(n) = \Theta(n^{log_2 5})$

4. $T(n) = 8T((n - \sqrt{n})/4) + n^2$

   Solution: Ignore low-order term to get $T(n) = 8T(n/4) + n^2$. Case 3; $T(n) = \Theta(n^2)$

5. $T(n) = 2T(\sqrt{n}) + \lg \lg n$

   Solution: Do a change of variables. Let $m = \lg n$. Now

   $T(2^m) = 2T(2^{m/2}) + \lg m$.

   Define $S(m) = T(2^m)$. Now

$S(m) = 2S(m/2) + \lg m$.

Applying the Master Method we get Case 1, so $S(m) = \Theta(m)$. Undoing our previous transformations we find that $T(2^m) = \Theta(m)$ and therefore $T(n) = \Theta(\lg n)$.