Algorithms

Matthew Cocci

1 Guiding Principles

We adopt the following guiding principles for analyzing algorithms:

1. *Worst case analysis*: We analyze the worst case running time of an algorithm given the most difficult input the algorithm will half to handle. It makes **no** assumptions about the input. This is in contrast to *average case analysis*, which considers the running time for the "average" case, taking account of the relative frequencies of certain inputs.

2. Don't pay attention to constant factors and lower order terms. Different programming languages and compiling schemes will lead varying constant factors by a small amount. It's just inappropriate to worry about it unless the code is crucial to your task.

3. Asymptotic Analysis: Focus on running time for large input sizes n.

Putting these three guiding principles together, we can define a "fast" algorithm as one whose worst-case running time grows slowly with the input size, where "slowly" in the best case scenario is typically *linear* growth in running time with increasing size, n. In effect, that's the Holy Grail, best-case running time that we'll shoot for.

2 Divide and Conquer Algorithms

Divide the problem into smaller sub-problems, solve recursively, where a *recursive algorithm* is an algorithm that calls itself.

2.1 Merge Sort

Merge sort solves the *sort problem*, where you want to order a unsorted list, by a divide-and-conquer, recursive approach.

2.1.1 Basic Structure

Merge Sort works as follows:

1. Split the list in half into two sub-lists.
2. Recursively sort the first half. Then, recursively sort the second half.
3. Merge/recombine the smaller sorted lists.
    a) You step through the output array, populating it with the minimum number from the two sorted sublists you're trying to recombine.
    b) Implicit in the process, the minimum element of each sublist is at the beginning of each sublist, so you don't have to waste time searching through them.

Why is this better? Well, it runs with an upper bound of

$$\text{operations} \leq 6n \log_2 n + 6n$$

while Insertion Sort, Bubble Sort, selection sort, and other's require n^2 operations. This running time is not quite linear since it's $6n \log_2 n$, although it's pretty good.

2.1.2 Runtime Analysis

Simplifying Assumptions: Suppose we have a list of length n where, for simplicity, we assume n is a power of 2 such that $j = \log_2 n$.

Recursive Structure Tree: We will make $\log_2 n = j$ divisions of the original list into sub-lists: dividing the original list in half, dividing the first level of two sublists in half again for a total of four sublists at the second level, etc. This will generate a tree with j levels.

Within our tree, at each specific level, k, we have 2^k sub-problems, with each problem of size $n/2^k$. There are 2^k sub-problems because we divide by two at each step down the tree. On top of that, the sub-problems are of size $n/2^k$ because with each division of a parent problem, we reduce the size of the problem by one half.

Merging Sublists: Now, when the Merge Sort has to merge two sub-lists to form a list of size m, it will use at most $6m$ steps (by a rough approximation that you can figure out by looking at the code or pseudocode for merging).

Operations at Level j: Combining previous results, we see that the number of steps at level j turns out to be:

$$\text{operations at level } j \leq \underbrace{2^j}_{\text{No. of level } j \text{ sub-probs}} \times \underbrace{6}_{\text{Merge steps}} \underbrace{\left(n/2^j\right)}_{\text{Sub-prob size at level } j} = 6n$$

This is awesome because the number of operations required at level j is *independent* of the level!

Total Runtime: This means that we take the work per level just computed $6n$, times the number of levels:

$$\text{total operations} \leq \underbrace{6n}_{\text{work per level}} \times \underbrace{(\log_2 n + 1)}_{\text{number of levels}}$$

3 Asymptotic Notation

3.1 Big-Oh Notation

This is a type of asymptotic notation concerning functions defined on the positive integers. In particular, we'll consider a function $T(n)$, $n = 1, 2, \ldots$.

Definition: A function $T(n) = O(f(n))$ if and only if

$$\exists\, c, n_0 > 0 \quad \text{s.t.} \quad T(n) \leq c\, f(n) \quad \forall n \geq n_0 \tag{1}$$

In words, $T(n)$ is $O(f(n))$ if $T(n)$ is bounded by a constant multiple of $f(n)$ for sufficiently large n.

3.1.1 Polynomial Functions

We want to prove that Big-Oh notation really does supress constant and lower order terms:

$$\begin{aligned} &\text{if} \quad T(n) = a_k n^k + \cdots + a_1 n + a_0 \\ &\Rightarrow \quad T(n) = O(n^k) \end{aligned} \tag{2}$$

Proof. To prove, we need to find a c and n_0 so that Equation 2 satisfies the definition. So let's go.

$$\begin{aligned} T(n) &= a_k n^k + \cdots + a_1 n + a_0 \\ &\leq |a_k| n^k + \cdots + |a_1| n + |a_0| \\ &\leq |a_k| n^k + \cdots + |a_1| n^k + |a_0| n^k \\ &\leq \sum_{i=0}^{k} |a_i| n^k \\ \Rightarrow \quad n_0 = 1 \qquad c &= \sum_{i=0}^{k} |a_i| \end{aligned}$$

This satisfies the definition for $T(n) = O(n^k)$. □

3.2 Omega Notation

Definition: We say that $T(n) = \Omega(f(n))$ if and only if there exist constants c and n_0 such that

$$T(n) \geq c \cdot f(n) \quad \forall n \geq n_0$$

3.3 Theta Notation

Definition: We say that $T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ AND $T(n) = \Omega(f(n))$. Visually, this means $T(n)$ is "sandwiched" between two different constant multiples of $f(n)$.

Equivalently, $T(n) = \Theta(f(n))$ if and only if

$$\exists c_1, c_2, n_0 \quad \text{s.t.} \quad c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \qquad \forall n \geq n_0$$

Although, we typically only care about Big-Oh notation, since we care abot upper bounds.

3.4 Little-Oh Notation

Similar to Big-Oh, but a little bit stronger. Formally, $T(n) = o(f(n))$ if and only if for *all* constants $c > 0$, there exists a constant n_0 such that

$$T(n) \leq c \cdot f(n) \qquad \forall n \geq n_0$$