Section 2.5: Analysis of Algorithms

February 9, 2005

Abstract

By **analysis of algorithms** we mean the study of the efficiency of the algorithms. In this section we will measure the efficiency of an algorithm by counting operations (and of course we are generally shooting for a small number!).

1 Counting operations directly

In algorithm *SequentialSearch* (p. 148), we search for element x in a list of n items. *SequentialSearch* is a direct method, by comparison with algorithm *BinarySearch* (p. 130), which is recursive. Is one algorithm more efficient than the other?

In the *SequentialSearch*, there are three rather interesting cases:

- we find x on the very first try (total comparisons: 1!). This is called the "best-case" scenario.
- we find x on the last try (total comparisons: n). This is the "worst-case" scenario.
- On average, we require $n/2$ comparisons.

We will consider the worst-case scenario as the benchmark.

2 Counting Using Recurrence Relations

Algorithm *BinarySearch* is recursive: it calls itself. Starting from a list of length n it makes one comparison and then calls itself with a list of half its initial length. Hence the number of comparisons for the list of length n, $C(n)$, would be

$$C(n) = C(n/2) + 1$$

and $C(1) = 1$. Use the "expand, guess, and verify" approach: in the worst-case scenario, the algorithm will find the element (or not) on its last check (when it's down to a list of length 1).

$$C(n) = C(n/2) + 1 = (C(n/4) + 1) + 1 = ((C(n/8) + 1) + 1) + 1 = \ldots$$

Obviously this is only going to work (in the sense that $C(n/8)$, etc., make sense) if n is a power of 2. Assume that $n = 2^m$, for integer m.

Consider a change of variable: in

$$C(2^m) = C(2^{m-1}) + 1 \qquad C(1) = 1$$

we define $T(m) = C(2^m)$ so that

$$\boxed{T(m) = T(m-1) + 1} \qquad \boxed{n = 2^m}$$

solve for m

Note that $T(0) = C(1) = 1$. We can solve easily to get a closed-form solution of

$$\boxed{T(m) = m + 1} \qquad \boxed{\log_2 n = m}$$

Hence, $C(n) = C(2^m) = T(m) = m + 1 = log_2(n) + 1$. This compares quite favorably with the worst-case estimate from *SequentialSearch*, which would be n (linear in n).

If $n = 2^m$, then $\log_2 2^m = m$

(For those of you who've forgotten, the log function grows much more slowly than a linear function.)

Let's look at the general recurrence relation of the "divide and conquer" variety: given

$$S(1) = a$$
$$S(n) = cS(n/2) + g(n)$$

Assume $n = 2^m$ for some integer m. Then

$$S(2^0) = a$$
$$S(2^m) = cS(2^{m-1}) + g(2^m)$$

Now we perform the change of variables: let $T(m) = S(2^m)$, so that

$$T(0) = a$$
$$T(m) = cT(m-1) + g(2^m)$$

Using formula (8) of section 2.4, p. 134, we get

$$T(m) = c^{m-1}T(1) + \sum_{i=2}^{m} c^{m-i}g(2^i)$$

Then reindexing, since we start with 0 rather than 1, we get

$$T(m) = c^m T(0) + \sum_{i=1}^{m} c^{m-i}g(2^i)$$

Finally, substituting back in S and n, we get

$$S(2^m) = c^{\log_2 n}a + \sum_{i=1}^{\log_2 n} c^{\log_2 n - i}g(2^i)$$

Whew!

The *BinarySearch* algorithm starts with a sorted list, which is not a requirement for the *SequentialSearch* algorithm; so the comparison isn't really fair. What if we add a sort?

Example: Exercise 13, p. 154\ 6

13 a) 3

b) 4

c) 8

Example: Exercise 14, p. 156

14 $r+s-1$

(alternating lists)

Example: Exercise 15, p. 156

Example: Exercise 16, p. 156

#15 $C(1) = 0$

$C(n) = 2\,C(n/2) + 2(n/2) - 1$

$= 2\,C(n/2) + n - 1$

So we can carry out the *BinarySearch* algorithm following a *MergeSort* (see the exercises above for its definition), with

$$\overline{\underbrace{log_2(n)+1}_{\text{binary search}}+\underbrace{nlog_2(n)-n+1}_{\text{merge sort}}}$$

or

$$\boxed{(n+1)log_2(n)+2}$$

operations, compared with n operations for *SequentialSearch* - which wins in this case! $(n+1)log_2(n)$ is *super*linear - grows faster than the linear function n.

If we had started with a sorted list, however, it would make no sense to use *SequentialSearch*, since *BinarySearch* is so much more efficient.

3 Other criteria

An algorithm should not be analyzed quite so one-dimensionally as we've done here, of course: there may be other issues (such as how easily parallelized an algorithm is, for example) which are more important than simple operation counts.

As demonstrated in the case of the Euclidean Algorithm (or gcd) in this section, we may simply be shooting for an upper bound on the number of operations required (even worse than the worst case scenario!), when actual worst-case numbers are hard to come by.

$$T(0) = C(1) = 0$$

$$T(m) = C(2^m) = 2C(2^{m-1}) + 2^m - 1$$

$$\boxed{T(m) = 2T(m-1) + \underbrace{2^m - 1}_{g(m)}}$$

$$c=2$$

$$T(m) = 2^{m-1}T(1) + \sum_{i=2}^{m} 2^{m-i}(2^i - 1)$$

$$= 2^{m-1}T(1) + \sum_{i=2}^{m} 2^m - 2^{m-i}$$

$$= 2^{m-1}T(1) + \sum_{i=2}^{m} 2^m - \sum_{i=2}^{m} 2^{m-i}$$

$$= 2^{m-1}T(1) + (m-1)\,2^m - \underbrace{\left(2^{m-2} + 2^{m-3} + \dots + 2^0\right)}_{\substack{= 1 + 2 + 2^2 + \dots + 2^{m-2} \\ = 2^{m-1} - 1}}$$

$$= 2^{m-1}\overset{=1}{T(1)} + (m-1)\,2^m - 2^{m-1} + 1$$

$$= \cancel{2^{m-1}} + (m-1)\,2^m - \cancel{2^{m-1}} + 1$$

$$T(n) = (m-1)\,2^m + 1$$

$n = 2^m$

$m = \log_2 n$

$$C(n) = T(n) = (m-1)\,2^m + 1 =$$

$$= (\log_2 n - 1)\,n + 1$$

$$= n\log_2 n - n + 1$$