Section 3.3: Analysis of Algorithms

February 28, 2022

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, in our endless pursuit of optimization).

1 Counting operations directly

In algorithm *SequentialSearch* (p. 204), we search for element x in a list of n items. *SequentialSearch* is a direct method, by comparison with algorithm *BinarySearch* (p. 169), which is recursive. Is one algorithm more efficient than the other?

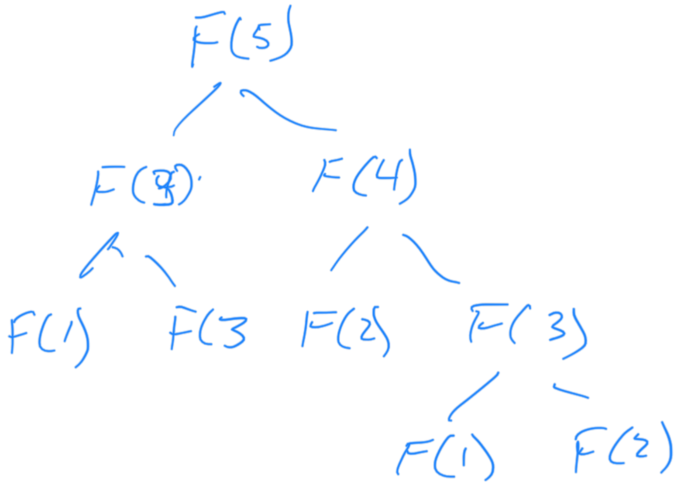
In the *SequentialSearch*, there are three rather interesting cases:

- we find x on the very first try (total comparisons: 1). This is called the "best-case" scenario.
- we find (or even don't find – it doesn't matter in terms of operations) x on the last try (total comparisons: n). This is the "worst-case" scenario.
- If x is in the list, then on average we require $(n+1)/2$ comparisons (remembering Gauss): we sum up all the cases from 1 to n, and divide by n:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

We will consider the worst-case scenario as the benchmark: best behavior is great, and average behavior is okay; but if we throw our user into infinite loops (for example) in the worst case, we're really doing a disservice....

Sometimes we can compute **exactly** how many operations will be required – e.g. our recursive calculation of $Fibonacci(n)$ using the recursive definition. Let's count how many additions $A(n)$ are required for the calculation of $Fib(n)$ by the recursive algorithm. (The result is somewhat amusing!)



The recurrence relation for $A(n)$ is non-homogeneous, but can be converted to a homogeneous linear recurrence, where its form is well known. This is a standard trick in mathematics: convert a problem into another that you know how to do, and then convert back.

It turns out that the closed form solution for the nth Fibonacci is

$$F_n = \frac{\gamma^n - (-\gamma)^{-n}}{\sqrt{5}}$$

where $\gamma = \frac{1+\sqrt{5}}{2}$ is known as "the golden mean" (or "golden ratio"). It turns out, however, that $F_n \approx \frac{\gamma^n}{\sqrt{5}}$, and that every Fibonacci F_n can be obtained by simply rounding this expression!

$$F_n = Round\left(\frac{\gamma^n}{\sqrt{5}}\right)$$

for all $n \in \mathbb{N}$. That's amazing, and demonstrates that Fibonacci numbers grow exponentially (within a unit of the exponential function $\frac{\gamma^n}{\sqrt{5}}$). More on Fibonacci numbers in a bit.

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?

Exercises 19–24 relate to a recursive sorting algorithm called *MergeSort*, which is described as follows: A one-element list is already sorted; no further work is required. Otherwise, split the list in half, sort each half using *MergeSort* (this is the recursive part), and then merge the two halves back into one sorted list.

19. The merge part of algorithm *MergeSort* requires comparing elements from each of two sorted lists to see which goes next into the combined, sorted list. When one list runs out of elements, the remaining elements from the other list can be added without further comparisons. Given the following pairs of lists, perform a merge and count the number of comparisons to merge the two lists into one.
    a. 6, 8, 9 and 1, 4, 5 — 6:1, 6:4, 6:5 — 3 comp.
    b. 1, 5, 8 and 2, 3, 4 — 1:2, 5:2, 5:3, 5:4 — 4 comp
    c. 0, 2, 3, 4, 7, 10 and 1, 8, 9 — 0:1, 2:1, 2:8, 3:8, 4:8, 7:8, 10:8, 10:9 — 8 comp
20. Under what circumstances will the maximum number of comparisons take place while merging two sorted lists? If the lengths of the lists are r and s, what is the maximum number of comparisons?
21. Write a recurrence relation for the number of comparisons between list elements done by algorithm *MergeSort* in the worst case. Assume that $n = 2^m$.
22. Solve the recurrence relation of Exercise 21.

$m = \log_2 n$

Example: #19, p. 215

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

$C(1) = 0$, $g(2^i) = 2^i - 1$

$c = 2 \quad C(1) = 0$

$$C(n) = 0 + \sum_{i=1}^{m} 2^{m-i}(2^i - 1)$$

Example: #20, p. 215

$r + s - 1$

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

$$= \sum_{i=1}^{m} 2^m - \sum_{i=1}^{m} 2^{m-i}$$

Example: #21, p. 215

$$C(n) = 2C(\tfrac{n}{2}) + \tfrac{n}{2} + \tfrac{n}{2} - 1$$

$$= m \cdot 2^m - (1 + 2 + \dots + 2^{m-1})$$

$C(1) = 0$

$g(n) = n - 1$

$C(n) = 2C(\tfrac{n}{2}) + n - 1$

$$= m 2^m - \frac{1 - 2^m}{1 - 2}$$

Example: #22, p. 215

$$= m 2^m + 1 - 2^m$$

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

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

$$n(log_2(n) - 1) + 1$$

operations (in the worst case), compared with n operations for *SequentialSearch* - which wins in this case! $nlog_2(n)$ 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 in that case.

Also, if we are doing multiple searches with the same lists, then the costs of **not sorting** begin to add up. The sorting is an initial (or fixed) cost; then there is a benefit each time one sorts thereafter for the merge-sort. It would be wise to compute the cut-off value of the number of searches for a given list size which would justify sorting and then using the binary search algorithm.

Doing s searches:

$n(\log_2 n - 1) + 1 + (\log_2 n + 1)s$

$= ns$

Solve for s as a function of n

2 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. In the case of searching, are we going to be reusing the list and doing the searches over and over? The sort is a one-time cost, while the searching is not.

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. This is demonstrated in the case of the Euclidean Algorithm (for computing the greatest common divisor, or gcd) in this section (the Euclidean Algorithm is first introduced on p. 133).

Here is a recursive definition for the Euclidean Algorithm, in lisp:

```
(defun
  ourgcd(a b &key initial-a initial-b )
  (if (not (and (integerp a) (integerp b)))
      (error "Sorry: only integer arguments allowed.")
    )
  (let* ((a (abs a))
         (b (abs b))
         (m (min a b))
         (n (max a b))
         (r (mod n m))
         (q (floor (/ n m))) ;; only needed to print out the equation used
         ;; These just store the original values of a and b for printing in the end:
         (initial-a (if initial-a initial-a n))
         (initial-b (if initial-b initial-b m))
         )
    ;; print out the current equation from the division algorithm:
    (format t "~%~d=~d*~d+~d" n q m r)
    (if (= r 0)
        ;; we have no remainder, so we're finished: print out the result (the gcd)
        (format t "~%gcd(~d,~d)=~d" initial-a initial-b m)
      ;; otherwise, iterate:
      (ourgcd m r :initial-a initial-a :initial-b initial-b)
      )
    )
  )
```

Actually, in this case, worst-case numbers are easy to get: the worst case for the Euclidean algorithm is a pair of consecutive Fibonacci numbers (there they are again, those rascals!). This is investigated in problems #37-40, p. 217. An example pair of consecutive Fibonacci numbers would be 5 and 3, or 89 and 55.

Our author shows (p. 210) that the number of divisions required satisfies the (rather poor) upper bound

$$E(n) \leq 2\log_2(n)$$

where n is the larger integer in the pair $gcd(n, m)$.

Let's see where this comes from: given the need to compute the greatest common divisor of a and b, $a \geq b$, we compute $gcd(a, b)$ starting with

$$a = q_1 b + r_1$$

and then "do it again" (recurse) – we compute

$$b = q_2 r_1 + r_2$$

and do so until $r_m = 0$ (at which point r_{m-1} would be the gcd). In this case, the index on r is a count of the number of divisions required (starting from the first of b into a, which resulted in r_1).

Now r_1, the remainder of dividing a by b, satisfies $r_1 < b$ (or we'd take out another factor of b).

gcd(a,b)

gcd(21,12):

21 = 1·12 + 9

gcd(12,9):

12 = 1·9 + 3

gcd(9,3)

9 = 3·3 + 0

gcd(a,b)
gcd(b,r1)
gcd(r1,r2)

Consider all possible cases (we show that $r_1 < \dfrac{a}{2}$):

- If $b < \dfrac{a}{2}$, then $r_1 < b < \dfrac{a}{2}$, so $r_1 < \dfrac{a}{2}$.
- If $b > \dfrac{a}{2}$, then $q_1 = 1$, and $r_1 = a - b < \dfrac{a}{2}$.
- If $b = \dfrac{a}{2}$, then $r_1 = 0 < \dfrac{a}{2}$ (and we're done, by the way! That's not worst-case analysis....).

Now if we successfully compute the two steps of the Euclidean algorithm, then we will have reduced our problem to the calculation of $gcd(r_1, r_2)$.

Since $r_1 < \dfrac{a}{2}$, we've "halved" the problem. So here's the resultant worst-case scenario recurrence relation (as usual for divide and conquer, let's suppose that $n = 2^k$ for some $k \in \mathbb{N}$):

$$E(1) = 0$$
$$E(n) = E(n/2) + 2, \text{ for } n \geq 1$$

Plugging $E(1) = 0$, $c = 1$, and $g(n) = 2$ into this formula from section 2.5,

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

we get the desired result:

$$E(n) \leq 1^{\log_2 n} 0 + \sum_{i=1}^{\log_2 n} 1^{\log_2 n - i} 2$$

or

$$E(n) \leq 2 \log_2(n).$$