

CS313E: Elements of Software Design

Efficiency of Algorithms

Dr. Bill Young
 Department of Computer Sciences
 University of Texas at Austin

Last updated: October 31, 2011 at 08:47

In looking at the Fibonacci sequence, we saw that the implementation really matters. The naive recursive implementation of `fib` was horribly inefficient (exponential). A better recursive implementation was much, much better (linear).

Notice they solved the same problem. So it's not *the problem* that is efficient or inefficient, but the solution. We're going to define a notion of the efficiency of an "algorithm" (a particular way of solving a problem).

However, there are problems that are believed to have no efficient solutions (NP problems).

Types of Analysis

Given an algorithm, you might ask how it behaves in the:

- Best case:** Find an input of size n that gives the smallest possible execution time.
- Worst case:** Find an input of size n that gives the largest possible execution time.
- Average or Expected case:** Determine the average execution time over all inputs of size n . This requires a probability distribution over the possible inputs.

Why Worst-Case Analysis?

We will focus primarily on worst-case analysis. Why?

- Gives an upper bound on the running time for any input (a guarantee).
- For many algorithms, the worst case occurs fairly often. E.g., when searching a list, the worst case occurs whenever the item is not present.
- The average case is often roughly as bad as the worst case. E.g., for bubble sort on a random array, the average case and the worst case both require the same number of comparisons.
- Often the worst and average cases are *asymptotically* the same.

This is the formal definition, but you don't need to worry about it, other than understanding the general idea.

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions:

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$.

If $f(n) \in O(g(n))$, then $f(n)$ is *asymptotically upper bounded* by $g(n)$. Think, $f(n)$ "is no larger than" $g(n)$. We usually say that f is "Big-O of g ."

Suppose you know that a particular algorithm takes $7n^2 + 4n$ steps to perform a computation on input of size n .

How would you show that the algorithm is $O(n^2)$?

Can you show that the algorithm is not $O(n)$?

Typically, the Big-O of a polynomial function is the highest power. E.g., $9n^4 - 3n^3 + 7n + 19$ is $O(n^4)$.

Interpreting the Analysis

- When we say "the execution time of Algorithm A is $O(f(n))$ " we mean that no matter what input of size n is chosen, the execution time on that input set is $O(f(n))$.
- For example, the execution time of insertion sort is $O(n^2)$. In addition, we can say that the worst-case execution time is $O(n^2)$, since there are inputs that cause the algorithm to take $O(n^2)$ time.

Why It Matters

Assuming that each input element can be processed in a microsecond.

	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
n	.00001 sec	.00002 sec	.00003 sec	.00004 sec	.00005 sec	.00006 sec
n^2	.0001 sec	.0004 sec	.0009 sec	.0016 sec	.0025 sec	.0036 sec
n^3	.001 sec	.008 sec	.027 sec	.064 sec	.125 sec	.216 sec
n^5	.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
2^n	.001 sec	1.0 sec	17.9 min	12.7 days	35.7 yr	366 cen
3^n	.059 sec	58 min	6.5 yr	3855 cen	2×10^8 cen	1.3×10^{13} cen

Now assume that we get a better computer, does it matter? The following are the size of the largest problem solvable in one hour.

	now	100× faster	1000× faster
n	N_1	$100 \times N_1$	$1000 \times N_1$
n^2	N_2	$10 \times N_2$	$31.6 \times N_2$
n^3	N_3	$4.64 \times N_3$	$10 \times N_3$
n^5	N_4	$2.5 \times N_4$	$3.98 \times N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Input: A sequence of n numbers a_0, a_1, \dots, a_{n-1} .

Output: A permutation (reordering) $a'_0, a'_1, \dots, a'_{n-1}$ of the input sequence such that $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$.

A specific sequence to be sorted is called an *instance* of the sorting problem. In general, an instance of a problem consists of all the inputs needed to compute a solution to the problem.

An algorithm is said to be *correct* if, for every input instance, it halts with the correct output. We say that a correct algorithm *solves* the given computational problem.

A Quirky Sorting Algorithm

Let's see what the Big-O complexity is of a novel sorting algorithm, sometimes called the *Las Vegas Card Sort*.

- 1 Shuffle a deck of cards
- 2 See if the result is sorted
- 3 If not, go back to step 1

What is the best, worst, and average case of this algorithm?

Efficiency of the Algorithm

The efficiency of the Las Vegas Card Sort is as follows, where N is the number of cards in the deck.

Worst Case: $O(\infty)$

Best Case: $O(1)$

Average Case: $O(N!)$

Can you explain each of these? BTW: $52! = 80658175170943878571660636856403766975289505440883277824000000000000$.

As we go through the discussion, think about when the worst case would arise.

Bubble Sort is probably the simplest sorting algorithm, one of the easiest to understand, and one of the slowest (on average). *Note that it sorts in place.*

```
def bubbleSort(lst):
    n = len(lst)
    for j in range(n):
        for i in range(1, n):
            if lst[i] < lst[i-1]:
                lst[i], lst[i-1] = lst[i-1], lst[i]
```

What is the complexity (Big-O) of this algorithm?

Complexity of a sorting algorithm is often described in terms of the number of *comparisons* or number of *swaps*.

A slightly better sorting algorithm is Insertion Sort.

```
def insertionSort(a):
    for i in range(1, len(a)):
        # Insert a[i] into the sorted sublist
        v = a[i]
        flag = False
        for j in range(i):
            if a[j] >= v:
                index = j
                flag = True
                break
        if flag:
            a.pop(i)
            a.insert(index, v)
    return a
```

What is the complexity (Big-O) of this algorithm?

Running Our Sort Algorithms

```
>>> A = [9, 4, 2, 10, 7, 3, 25]
>>> bubbleSort(A)
>>> A
[2, 3, 4, 7, 9, 10, 25]
>>> A = [9, 4, 2, 10, 7, 3, 25]
>>> insertionSort(A)
>>> A
[2, 3, 4, 7, 9, 10, 25]
>>> B = ["my", "dog", "has", "fleas", "do", "you"]
>>> insertionSort(B)
>>> B
['do', 'dog', 'fleas', 'has', 'my', 'you']
```

Insertion Sort Analysis

What is the Complexity of insertionSort?

```
def insertionSort(a):
    for i in range(1, len(a)):
        # Insert a[i] into the sorted sublist
        v = a[i]
        flag = False
        for j in range(i):
            if a[j] >= v:
                index = j
                flag = True
                break
        if flag:
            a.pop(i)
            a.insert(index, v)
    return a
```

The inner loop runs at most n times. The outer loop runs n times, where n is the length of A . *Would it help to replace the inner loop by a function that does the insertion?*

Linear:

$$f(n) = kn, f(2n) = (2kn)$$

Doubling the argument doubles the function.

Powers:

$$\bullet f(n) = n^2, f(2n) = (2n)^2 = 4f(n)$$

$$\bullet f(n) = n^3, f(2n) = (2n)^3 = 8f(n)$$

Doubling the argument increases the function by a multiplicative constant that depends on the constant power.

Exponentials:

$$f(n) = 2^n, f(2n) = 2^{2n} = (2^n)^2 = (f(n))^2$$

Doubling the argument increases the function by squaring it.

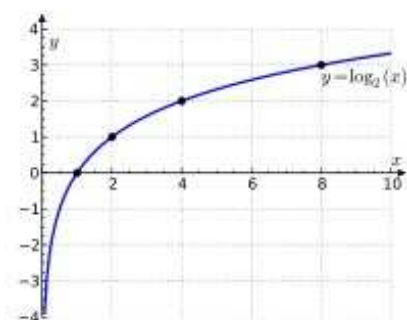
The logarithm is the “inverse” of the exponential function. You can think of it as, “to what power do I have to raise k to get n .”

For example, $\log_2(16) = 4$ because $2^4 = 16$; $\log_{10}(1000000) = 6$ because $10^6 = 1000000$.

Many algorithms are related to the $\log_k(n)$. For example, if I have a balanced binary tree with n leaves, the height of the tree is approximately $\log_2(n)$.

That means that searching in a binary tree is an $O(\log(n))$ operation, which is significantly better than linear.

Logarithmic Functions



$$f(n) = \log_2(n), f(2n) = \log_2(2n) = \log_2(n) + 1 = f(n) + 1$$

Doubling the argument increases the function by an additive constant.

Analyzing Code

If you have a loop in your code, the amount of work you have to do clearly depends on the number of times the loop is executed.

How many times is the loop executed in the following function?

```
def sum (n):
    partialSum = 0
    for i in range(n+1):
        partialSum += i * i * i
    return partialSum
```

The running time of a **for** loop is at most the running time of the statements inside the loop body (including tests) times the number of iterations.

Analyze nested loops inside out. The total running size of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

For consecutive statements, take the max running time of the statements.

What is the complexity of the following two code fragments?

```
for i in range(n):
    for j in range(n):
        a[i] += a[j] + j
```

```
for i in range(n):
    a[i] = 0
for i in range(n):
    for j in range(n):
        a[i] += a[j] + j
```

and this one?

```
def maxSubSum1(A):
    n = len(A)
    maxSum = 0
    for i in range(n):
        for j in range(n):
            thisSum = 0;
            for k in range(i, j+1):
                thisSum += a[k]
            if thisSum > maxSum:
                maxSum = thisSum
    return maxSum
```

and this one?

```
def maxSubSum4(A):
    n = len(A)
    maxSum = 0
    thisSum = 0
    for j in range(n)
        thisSum += a[ j ]
        if thisSum > maxSum
            maxSum = thisSum
        elif thisSum < 0
            thisSum = 0
    return maxSum
```

What is the efficiency of the following routine?

```
def binarySearch(a, x, lo=0, hi=None):
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        midval = a[mid]
        if midval < x:
            lo = mid+1
        elif midval > x:
            hi = mid
        else:
            return mid
    return -1
```

And this one?

```
def binarySearchR(a, x, lo, hi):
    if hi < lo:
        return -1
    else:
        mid = (lo+hi)//2
        midval = a[mid]
        if midval == x:
            return mid
        elif midval > x:
            return binarySearchR(a, x, lo, mid - 1)
        else:
            return binarySearchR(a, x, mid + 1, hi)
```

Analysis of Programs

Analysis of the complexity of an algorithm can be (and often is) done very carefully and precisely.

For most of purposes, a gross analysis will do. In general:

- an exponential algorithm is useless for anything but the smallest inputs;
- any exponential algorithm is worse than any polynomial algorithm;
- a polynomial algorithm of degree k ($> j$) is worse than any polynomial algorithm of degree j ;
- any polynomial algorithm is worse than any linear algorithm;
- any linear algorithm is worse than any logarithmic algorithm;
- any logarithmic algorithm is worse than any constant-time algorithm.