Algorithm Analysis

Last updated January 19, 2013

Goal of software engineering - to develop high quality software at a reasonable cost - (adhering to a schedule and using resources wisely).

Objectives in software design:

1. design an algorithm that is easy to understand, code & debug
2. design an algorithm that makes efficient use of the computer's resources

What if you have to choose between an algorithm that satisfies objective (1) and another that satisfies objective (2)?

Measures of algorithm complexity:

- conceptual complexity - amount of human effort required to design, code, test, understand, maintain, and modify an algorithm.
- run-time complexity - amount of computer time a program/algorithm needs to run to completion.
- space complexity - amount of memory that a program/algorithm needs to run to completion.

Objective (1) is related to conceptual complexity. It is a software engineering issue.

Objective (2) is related to algorithm analysis and data structures.

We are concerned with both issues, but in this course, we are primarily with objective (2).

Running time of an algorithm (or run-time complexity) is a critical resource and a major concern.

Factors affecting the running time of a program:

- nature of the input to the program (size is usually most important)
- quality of the code generated by the compiler
- contention for resources with other users
- nature and speed of the CPU, busses, peripherals
- time complexity of the algorithm underlying the program

Which of these can we control?

So, how do we measure running time (or time complexity)?

1. empirical comparison – run experiment, observe results

   The process:
   - execute an algorithm many times, each on different data set size
   - record the time spent to process each data set
   - analyze the results

   Limitations of empirical comparison

- can only test a limited number of data sets
- must run all tests on same hardware in same software environment
- must fully implement an algorithm before it can be analyzed

2. algorithm analysis – a generalized way of analyzing the running times of algorithms
    - takes into account all possible inputs
    - allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of hardware and SW environment
    - can be performed without implementing algorithm – we can study the high-level description of an algorithm

The algorithm analysis technique:

We don't actually measure running time, since we don't run the program to determine it. And, the actual time could vary greatly from CPU to CPU.

We estimate running time by counting the number of basic operations required to process an input of a certain size.

What is a basic operation?
A low-level operation with an execution time that is constant.
- Example, adding two numbers: 2 + 5 or 2000 + 5 takes the same amount of time to execute

Running time is expressed as f(n) for some function f on input size n.

The function f(n) expresses a **growth rate** - the rate at which the cost (run time) of an algorithm grows as the size of the input grows.

Analyzing Non-recursive Algorithms
(These guidelines from *The Design and Analysis of Algorithms,* by Anany Levitin, Addison Wesley, 2nd edition.*)*

General plan:

1. Decide on a parameter (or parameters) that describe the input size.
2. Identify the algorithm's basic operation – it is located within the algorithm's inner-most loop.
3. Check if the number of times the basic operation is executed depends only on the size of the input.
    - If it also depends on some characteristic / property of the data, then worst-case, average-case and best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count, or, establish its order of growth

Consider three algorithms that accomplish the same task (squaring a number):

// Sample 1

```
sum = n²
```

// Sample 2

```
sum = 0
for i = 1 to n
    sum = sum + n
```

// Sample 3

```
sum = 0
for i = 1 to n
   for j = 1 to n
      sum = sum + 1
```

Assumptions:

- basic operation, assignment to variable sum - all assignments take same amount of run time
- other operations take no time

Measure run time by counting number of assignments made:

Sample 1 – for different values of n
- n = 5 ⇒ 1 assignment made
- n = 50 ⇒ 1 assignments made
- n = 500 ⇒ 1 assignments made

Sample 2 – for different values of n
- n = 5 ⇒ 6 assignments made
- n = 50 ⇒ 51 assignments made
- n = 500 ⇒ 501 assignments made

Sample 3 – for different values of n
- n = 5 ⇒ 26 assignments made
- n = 50 ⇒ 2501 assignments made
- n = 500 ⇒ 250,001 assignments made

Sample 1 always does the same amount of work, regardless of value of n

Sample 2 - amount of work grows as n grows

Sample 3 - amount of work grows as n^2 grows

So, what is the run time of each of the algorithms shown?

Assume that the amount of time required to make an assignment is some constant **c**, then the run times for the algorithms above are:

Sample 1: f(n) = 1 // constant growth rate (or running time)
Sample 2: f(n) = n + 1 // linear growth rate
Sample 3: f(n) = n^2 + 1 // quadratic growth rate

Common functions used to describe growth rate (in order of increasing growth rate):

Function	Name
c	Constant – time required does not change based on input size n; inputs of all sizes require same amount of work. Eg. pop a stack
log n	Logarithmic – work required grows more slowly than n; Eg. binary search, double size of list, adds only one more probe
n	Linear – work grows in proportion to n Eg. examine all elements in a list of size n
n log n	N log N – grows a little faster than linear and a lot slower than quadratic Eg. quicksort, mergesort, heapsort
n^2	Quadratic – grows as n^2 grows Eg. nested loops (one inside another) often exhibit this growth rate
n^3	Cubic – works grows as n^3 grows; not as common as those listed above Eg. three loops nested may exhibit this

All the functions above, fall into the **polynomial** category, and are considered reasonable/practical/feasible.

- An algorithm that does that much work is OK.
- If you can choose between two algorithms with differing polynomial times, will usually choose the most efficient (fastest) one

But there are many important algorithms that require much more work than what could be described by a polynomial…

One function that occurs in this category:

b^n	Exponential – b is a positive constant, often 2 Eg. recursive algorithm for computing *n*-th number in Fibonacci sequence

Values of some functions (orders of growth):

\ ***N*** **Function**	**10**	**50**	**100**	**300**	**1000**
1	1	1	1	1	1
n	10	50	100	300	1000
5n	50	250	500	1500	5000
$\log_2 n$	3.32	5.64	6.64	8.23	9.96
$n \log_2 n$	33.2	282	664	2469	9960
n^2	100	2500	10000	900000	1000000
n^3	1000	125000	1000000	27 million	1 billion
2^n	1024	16 digit #	31 digit #	91 digit #	302 digit #

Primitive (basic) operations – what do we count?

Examples from Goodrich and Tamassia (textbook sometimes used in this course)

- assigning a value to a variable
- calling a method
- comparing two values
- performing an arithmetic operation
- accessing an element in an array
- following an object reference
- returning from a method

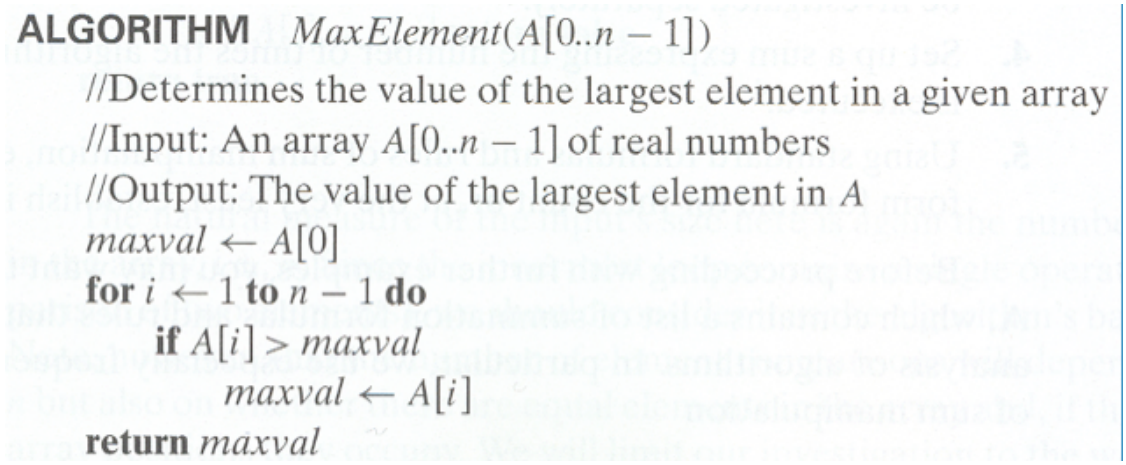
We count **how many times** a primitive operation is executed, and don't concern ourselves with how long the operation actually takes to execute.

Asymptotic analysis –

- content with rate of growth in work as input grows
- function in simplest form – focus on the function in terms of input size, not as concerned about constants: $2n^2$ and $4n^2$ not significantly different, n^2 is the important part

Some algorithms, like the sum algorithms, considered earlier, have an "every-case" efficiency measure – that means that the function describing efficiency is the same for all cases of input. However, the following algorithm does not exhibit "every-case" efficiency:

Consider the following (from the Levitin book):



```
ALGORITHM MaxElement(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

If we identify the basic operation as the assignment (←), then the function describing how many times assignment is done is not fixed, but varies on the data being processed. For example, if the data in array A are:

- ordered in descending order, the assignment is done only once (best case) - constant time
- ordered in ascending order, the assignment is done n times (worst case) - linear time
- unordered, we don't know how many times the assignment will be made, so we have to compute an average case complexity

If, on the other hand, we count comparison (>) as the basic operation, then we have an every-case complexity, since the comparison is made on EVERY iteration of the loop – done n times, linear time.

The **sequential search** algorithm is another example of an algorithm for which the amount of work done, depends on the order that the data appears in the list. See next:

Consider a sequential search of a list of size n for a value of K.
The running time depends on where K is located in the list.

How many list elements must be looked at (probed) in order to find K?

If K is first element - 1 element probed BEST CASE
If K is in element n/2 - n/2 elements probed AVERAGE CASE??
If K is in last element of list - n elements probed WORST CASE

The most useful analyses are the average and worst case analyses.

Average case analysis - compute the cost of a search for all possible element organizations, then take the average. See next page:

For the sequential search:

Position of K	Cost / probes
K in first position	1
K in second position	2
K in third position	3
…	
K in last (nth) position	n

The **sum** of probes: 1 + 2 +3 + …. + n is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

The **average** is the sum divided by the number of elements, n, or

$$\frac{n(n+1)}{2} \div n = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

So, on average the sequential search does about $\frac{1}{2}n$ probes – that is it looks at about half the elements

Consider a **binary search** of an ordered list of size *N* for a value of *x (see the iterative algorithm, page 46 in your Weiss text.*

Has you already know, unlike the sequential search, the binary search depends on the fact that the list being searched is ordered – at each iteration, it looks at (probes) the element in the "center" of the list being considered. The list is repeatedly "cut in half" until **x** is found, or it is determined that **x** is not in the list at all. It is not as clear, with this type of loop, how to determine how many times the loop iterates. Assume that the length of the list is a power of 2, say 64 with a indexed from 0 to 63. Assume that **x** is not in the list, and is smaller than all elements in the list, what is the maximum # of elements we look at?

1st call - Binary search called on original list of size n=64:
Check element at index midpoint (63-0)/2 = 31, since a[31] is not **x**, and **x** is smaller than a[31], high is reset to be 30 (the algorithm eliminated about half the list as only element at indices 0-30 are now of interest.

2nd call - Binary search called on list of size at most $\frac{n}{2}$:
Check element at midpoint (0+30)/2 = 15, **x** smaller than a[15]. Reset high to 14.
Check element at midpoint (0+14)/2 = 7, **x** smaller than a[7]. Reset high to 6.
Check element at midpoint (0+6)/2 = 3, **x** smaller than a[3]. Reset high to 2.
Check element at midpoint (0+2)/2 = 1, **x** smaller than a[1]. Reset high to 0.
Check element at midpoint (0+0)/2 = 0, **x** smaller than a[0]. Reset high to -1.
Loop terminated since **low <= high** is false.

So for a list of size 2^5, 6 values are probed. If the list size is double to $n = 2^6 = 128$, 7 (or $\log n - 1$) values will be probed, so the algorithm is O(log n). There is a formal way to calculate this, but for now, we'll stick with this informal technique, assuming it is understandable. On a successful search, fewer elements may be problems. So, in worst case Binary search is on the order of **log n**.

So, sequential search is, on average, O(n), while binary search is, in the worst case, $O(\log_2 n)$.

Which is faster, on average, for large values of n?

Asymptotic Analysis -

Ignore constants for large values of n.

- 10n and 5n not significantly different.
- n+5, n, n+ 50 are not significantly different
- n+5, n, n+ 50, 10n, 5n, and 5n+10 are not significantly different

Example: Given $f(n) = 5n^2 + 3n - 4$, the significant/dominant term is n^2.
Example: Given $f(n) = 1000n + 28$, the significant term is n.

Terms used to describe running times:

Upper bounds – a function that bounds the efficiency from above; the work done (efficiency) grows no faster than this.

Sometimes referred to as "big-Oh", denoted by O(f(N)).

Definition: $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(n) \leq c\ |g(n)|$ when $N > n_0$.

n_0 is the smallest value for which c(f(N)) is an upper bound.

Usage: The algorithm is in $O(n^2)$ in [best, average, worst] case.

Meaning: For all data sets big enough (i.e., $N > n_0$), the algorithm always executes in less than c(f(N)) steps in [best, average or worst case].

Example 1: $f(n) = 3n + 4$ is in O(n) because $3n+5 \leq 4n$ for all $n > 5$, where $c = 4$, $n_0 = 5$. For values of $n < 5$, $3n + 4$ is not <= 4n.

Example 2: $f(n) = 10n^2 + 4n + 2$ is in $O(n^2)$ since $10n^2+4n+2 \leq 11n^2$ for $n>4$.

Note, the function of Example 1 is also in $O(n^2)$ since $3n+4 \leq n2$ for all $n > 3$.

However, in order for big-Oh notation to be meaningful, we need the tightest (or smallest) upper bound. Thus, it is not accurate to say that 3n+4 is in $O(n^2)$.

See also, definitions of **Ω** and **Θ**, page 29. Omega (**Ω**) is a lower bound. When a function's upper and lower bounds are the same function, the term theta (**Θ**) is used

Sum and product rules (see General Rules, section 2.4.2):

Given two program segments P1 and P2, such that P1 is O(g(n)) and P2 and O(g(n)).
What is the run-time complexity if execution of P2 follows execution of P1?

```
        // segment P1 follows
…..
        // segment P2 follows
         .....
```

The sum rule is that the code above is O(max(g(n),f(n))).
(The segment that takes the most time dominates.)

What is the run-time complexity if execution of P1 includes execution of P2?

```
        // segment P1
        …. Call to method containing P2
```

The product rule is that the code above is O(g(n)f(n))).
(The product of the two functions is used.)