Algorithms (3.1-3.3)

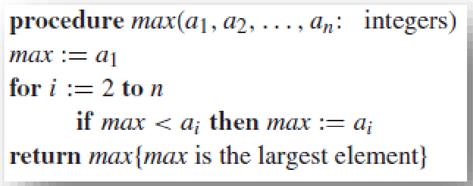
Algorithms are a huge topic. In CSE we have 2 theory classes purely dedicated to algorithms (EECS 477 and EECS 586) and a number of classes that are in effect applied algorithms (281 and 492 in particular come to mind). We're going to cover the topic in one day. Our goals are:

- To define what it means to be an algorithm
- To get a sense of the "runtime complexity" of an algorithm. In particular
  - What runtime complexity means
  - How and why we measure runtime as a function of input size
- To understand the notation associated with runtime complexity (big Oh, big Theta mostly)

What is an algorithm?

DEFINITION 1 An *algorithm* is a finite sequence of precise instructions for performing a computation or for solving a problem.

The above definition is taken from our text. It is (perhaps ironically) fairly vague. But it should include the steps needed to do a task. As always context is important. What one person views as a "precise" instruction another might view as impossibly vague. Consider the pseudo-code to the right (also from the text). What does that code do?



```
procedure max($a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ to $n$
    if $max < a_i$ then $max := a_i$
return $max${$max$ is the largest element}
```

Let's write a few simple algorithms ourselves. Let's start with an algorithm to search an unsorted list of integers a_1 through a_n, for the value x, returning the index where the value is found.

Question: in what way(s) is this problem statement unclear? What can we do to fix it?

Last time we set up a recurrence relation for a binary search. Let's be a lot more formal and look at the complete algorithm. Let's make sure this algorithm makes sense.

```
**procedure** *binary search* ($x$: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {$i$ is left endpoint of search interval}
$j := n$ {$j$ is right endpoint of search interval}
**while** $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    **if** $x > a_m$ **then** $i := m + 1$
    **else** $j := m$
**if** $x = a_i$ **then** *location* $:= i$
**else** *location* $:= 0$
**return** *location*{*location* is the subscript $i$ of the term $a_i$ equal to $x$, or 0 if $x$ is not found}
```

Question: in terms of the input (the n elements to be searched in this case), how many steps will each of these algorithms (linear and binary search) take in the worst case? Note: this is a non-trivial question. Think about it and justify your answer.

Other stuff from 3.2

Our text addresses a number of other important algorithms including sorting and defines what it means for an algorithm to be "greedy". You are responsible for that material though we won't be covering it in lecture. The text also proves the halting problem. You are *not* responsible for that material at this time (with luck we'll hit that at the end of the semester, but I don't know that we'll have time). You do need to be aware of the *result* of the proof.

To summarize, the basic statement is that if we model a computer as a device that can do "reasonable" computation (which is pretty much anything!) *and* we assume that said compute has infinite memory (a less likely thing, but large memory at least gets us close), then there exist important problems that said computer cannot solve. Ever. Full stop. One of those problems is figuring out if another computer program will ever halt… Thus the problem is unsolvable.

Growth of Function (3.2)

As we saw above, In order to discuss the runtime complexity in a reasonable way, we need some way to cleanly focus on the fundamentals of an algorithm, not the specifics of an implementation. What the CS community has largely settled on is a notation that shows us how an algorithm's runtime relates to the size of the input. And in order to do that, we're going to need some more notation and terminology!

DEFINITION 1 Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

Or put a bit more formally, $\exists k, C\ \forall(x > k)$ it is the case that $|f(x) \leq C|g(x)|$.

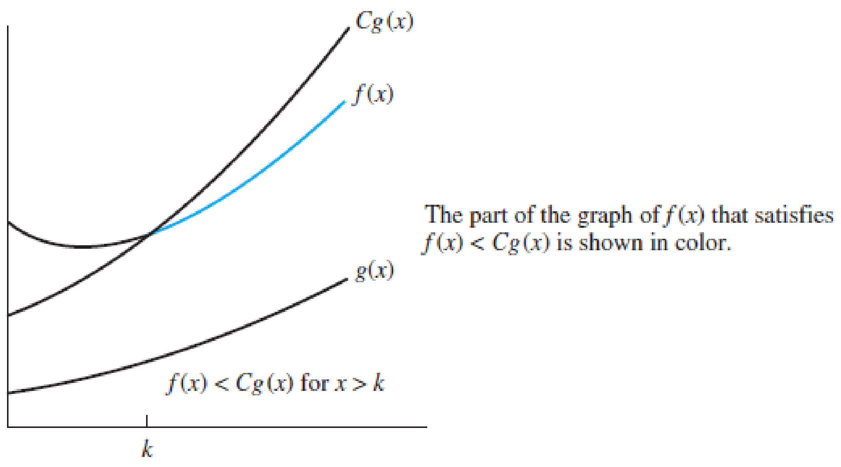
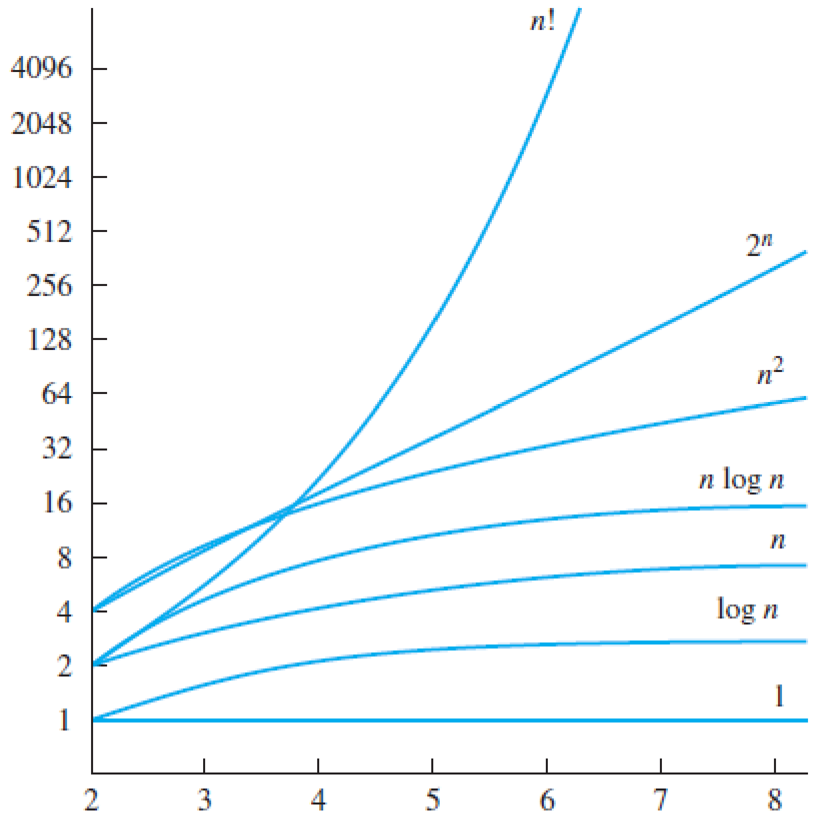


FIGURE 2 The Function $f(x)$ is $O(g(x))$.

1. For what values of C and k (if any) is it the case that $x^2=O(x^2+4)$?
2. For what values of C and k (if any) is it the case that $x^2=O(100x+4)$?
3. For what values of C and k (if any) is it the case that $x^2=O(x^3+4)$?
4. Is $\log(x)=O(x)$?
5. Is $2x^2+x=O(x^2)$?
6. Is $x^2=O(2x^2+x)$?

7. *Quickly* show that the sum of the integers from 1 to n is O(n^2). (Do this without any previous results about that sum…)



One cool result (and part of why big-O is so darn useful) is the following

THEOREM 2 Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|, |g_2(x)|))$.

1. Give a big-O estimate of $f(x) = (x + 1)\log(x) + x^2$
2. Give a big-O estimate of $f(n) = (2^n) + 100n^4 + \log(n)$

Big Theta and Big Omega

DEFINITION 2 Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

DEFINITION 3 Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. When $f(x)$ is $\Theta(g(x))$ we say that f is big-Theta of $g(x)$, that $f(x)$ is of *order* $g(x)$, and that $f(x)$ and $g(x)$ are of the *same order*.

Basically if f(x)=O(g(x)) then g(x)=Ω(f(x)). And if f(x) is both big-O of g(x) ***and*** big-Omega of g(x) it is also "of the order of" g(x) and f(x)=Θ(g(x))

Algorithm complexity (3.3)

We will generally use the big-O notation to describe the runtimes of algorithms in terms of their inputs.

Provide a reasonable big-O estimate for the run time of the following algorithms:

```
$t := 0$
for $i := 1$ to $n$
    for $j := 1$ to $n$
        $t := t + i + j$
```

```
$m := 0$
for $i := 1$ to $n$
    for $j := i + 1$ to $n$
        $m := \max(a_i a_j, m)$
```

```
$i := 1$
$t := 0$
while $i \leq n$
    $t := t + i$
    $i := 2i$
```

```
procedure closest-pair($(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$: pairs of real numbers)
$min = \infty$
for $i := 2$ to $n$
    for $j := 1$ to $i - 1$
        if $(x_j - x_i)^2 + (y_j - y_i)^2 < min$ then
            $min := (x_j - x_i)^2 + (y_j - y_i)^2$
            closest pair $:= ((x_i, y_i), (x_j, y_j))$
return closest pair
```

There are a few other important things from section 3.3.

Terminology

It turns out people rarely say things like Θ(n). Instead they will say "its runtime is linear" or some such. Below is a list of terms you might hear when discussing complexity.

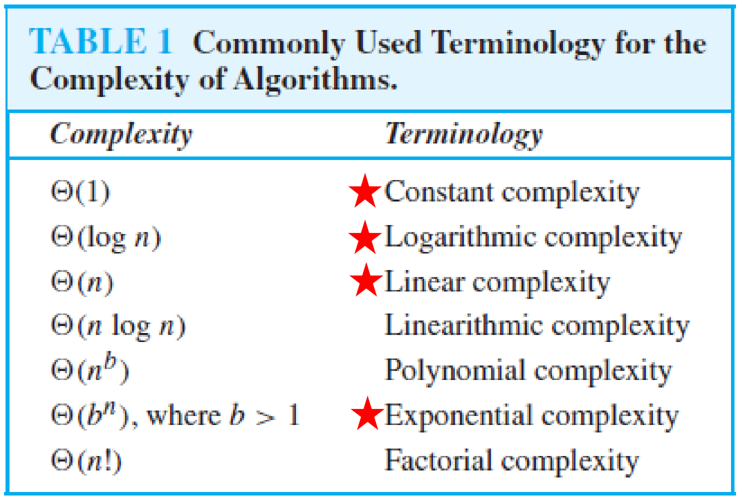


TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	★Constant complexity
$\Theta(\log n)$	★Logarithmic complexity
$\Theta(n)$	★Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	★Exponential complexity
$\Theta(n!)$	Factorial complexity

The ones with ★ next to them are terms you should know. In addition you should be aware that Θ(n^2) is often just referred to as "quadratic complexity".

Tractability

In general, we want to run algorithms that can actually solve problems. In section 3.2 the text showed that there exist problems which cannot be solved. But we also may encounter problems that are solvable, but aren't able to be solved in a reasonable period of time over a reasonable input set (yes, that was really vague on purpose).

As a pretty arbitrary line, CS folks have chosen to call anything that can't be solved in polynomial time or less "intractable" meaning that it may be solvable but we can't really solve it in the general case. Again, this is pretty arbitrary. While 2^n certainly grows faster than n^{100}, it is actually smaller for quite a long time (~n=1000) and well past when we'd have a chance of doing the computation (at n=1000 both are around 10E300).

That said, we rarely encounter polynomial complexity algorithms where the exponent is much greater than 4 or 5. While we do hit exponential algorithms on a regular basis.

The "P=NP" question asks if a certain class of important algorithms (including things like minesweeper!) have a polynomial complexity or not. That one is still out…