LESSON 7
Algorithms
§3.1, 3.2, 3.3 ← Some of each.

ALGORITHM a finite sequence of precise instructions for performing a computation or solving a problem.

Example 1 Find the largest value in a finite sequence of numbers.

Algorithm:

1. Set the temporary maximum M equal to the first number in the sequence.
2. Compare the next number with M, if larger change M to that number.
3. Repeat step 2 if there are more numbers in the sequence.
4. Stop when there are no more numbers left. M will be the largest number.

Example 2 Sort the numbers 3, 2, 4, 1, 5 into increasing order.

BUBBLE SORT ALGORITHM.

1st pass	Compare the first two numbers, reverse them if necessary	3 2 4 1 5 2, 3, 4, 1, 5
	Compare the next two numbers, reverse them if necessary	2 3 4 1 5
	Repeat until you reach the last two numbers	2 3 1 4 5 2 3 1 4 5 ↑ last number guaranteed correct
next pass	Repeat the first pass	2 3 1 4 5 2 1 3 4 5 2 1 3 4 5 2 1 3 4 5 ← last 2 numbers guaranteed correct
	Repeat until the numbers are in order.	1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 ← last 3 numbers guaranteed correct

$\boxed{\text{Example 3}}$ A "greedy" algorithm makes a "best" choice at each step. For example, Dijkstra's Algorithm for finding the shortest path in a graph, or Kruskal's and Prim's Algorithms for finding a minimal spanning tree in a graph.

$\boxed{\text{Example 4}}$

Evaluating a polynomial:

$P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ given values for $a_0, a_1, a_2, \ldots a_n$ and x and n is an integer.

<u>Algorithm #1.</u>

1) Set $S = a_0$, $k=1$
2) while $k \le n$
   a) replace S with $S + a_kx^k$
   b) replace k with $k+1$
   Endwhile
3) Print S.

a_0
$a_0 + a_1x$
$a_0 + a_1x + a_2x^2$
$\vdots$

<u>Algorithm #2</u> Horner's Algorithm

Uses the fact that
$$a_0 + \underbrace{a_1x + a_2x^2 + a_3x^3}$$
$$= a_0 + x(a_1 + a_2x + a_3x^2)$$
$$= a_0 + x(a_1 + x(\underbrace{a_2 + a_3x}_{\text{start here.}}))$$

1) Set $S = a_n$ and $k=1$.
2) While $k \le n$
   (a) Replace S with $a_{n-k} + Sx$
   (b) Replace k with $k+1$
   Endwhile
3) Print S

Complexity of an Algorithm

"size" of a problem: amount of information on which a solution is based.

e.g. n = number of numbers we are sorting in the bubble sort

n = degree of the polynomial

"size" of an algorithm: # of operations required to get the answer. (COMPLEXITY)

e.g. $+, -, \times, \div$, comparison of two numbers

examining a vertex to see if it is already in the spanning tree as it is being built

Example 1. Finding the largest value in a sequence of numbers. Let n be the number of numbers in the sequence. On the ith step we need to determine if we have reached the end of the sequence. 1 comparison of i with n. If we haven't, we compare the i^{th} number with the temporary maximum M.

So 2 comparisons at each step $= 2(n-1)$ comparisons total plus one more comparison at $i=n+1$ to leave the loop.

So the "size" of this algorithm is $2(n-1)+1 = 2n-1$.

Example 2 Bubble sort of n numbers.

On the i^{th} pass, the $i-1$ largest numbers are already in place. So $n-i$ comparisons are made.

Total $(n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2}$ comparisons.

Criteria for choosing between algorithms:

1) Smallest size

or 2) Complexity grows the slowest with n.

So we need a way of comparing growth of complexity. This is where Big O notation comes in.

Big O notation

If $f(x)$ and $g(x)$ are real-valued functions, we say $f(x)$ is $O(g(x))$ if there are positive constants C and k such that

$$|f(x)| \le C|g(x)| \quad \text{for all } x > k.$$

"$f(x)$ is big-oh of $g(x)$".

What does this mean?

$$\left|\frac{f(x)}{g(x)}\right| \le C \quad \text{for all } x > k.$$

As x gets larger and larger, $f(x)$ does not grow any faster than $g(x)$. As $x \to \infty$, this quotient has an upper bound, this quotient cannot approach ∞. It cannot keep growing without bound. $g(x)$ is putting a limit on how fast $f(x)$ can grow.

Example

$f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Why? What is the relationship between x^2+2x+1 and x^2:

At first it looks like $x^2 < x^2+2x+1$ for $x>1$, which means that x^2 is $O(x^2+2x+1)$.

But multiplying by a constant shows the reverse is true as well:

For $x > 1$, $x^2+2x+1 < x^2+2x^2+x^2 = 4x^2$.

So $x^2+2x+1 < \underset{\substack{\uparrow \\ C}}{4}x^2$ for all $x > \underset{\substack{\uparrow \\ k}}{1}$.

x^2+2x+1 does not grow faster than x^2. The growth of x^2 puts a bound on the growth of x^2+2x+1.

In this case, since x^2+2x+1 is $O(x^2)$ and vice versa, x^2 is $O(x^2+2x+1)$, we say these two functions are of the *same order*.

In general, any polynomial $a_0+a_1x+a_2x^2+\cdots+a_nx^n$ is $O(x^n)$

Why? for $x>\underbrace{1}_{k}$, $a_0+a_1x+a_2x^2+\cdots+a_nx^n \le a_0x^n+a_1x^n+\cdots+a_nx^n$

$$= \underbrace{|a_0+a_1+\cdots+a_n|}_{c}\,x^n$$

Some more examples:

- $7x^2$ is $O(x^3)$ why? For $x>\underbrace{1}_{k}$, $\frac{7x^2}{x^3}=\frac{7}{x}<\underbrace{7}_{c}$.

- n^2 is <u>not</u> $O(n)$ why? $\frac{n^2}{n}=n$ which grows without bound as n grows. There is no possible c bounding $\frac{n^2}{n}$.

- $n!$ is $O(n^n)$ Why? $n! = n(n-1)(n-2)\cdots 3\cdot 2\cdot 1$ eg $4! = 4\cdot 3\cdot 2\cdot 1$.
  $\le n(n)(n)\cdots(n)(n)(n)$
  $= n^n.$ for $n>1$

- $1+2+3+\cdots+n$ is $O(n^2)$ Why? $1+2+3+\cdots+n \le n+n+n+\cdots+n = n\cdot n = n^2.$ for $n>1$.

In applying this to the complexity of algorithms, we are usually looking for a function $g(x)$ that has the slowest growth.

In order of size:

$$1 \quad \log n \quad \underset{\substack{\uparrow \\ \text{this is considered good}}}{n} \quad n\log n \quad n^2 \quad 2^n \quad n! \quad n^n$$

$\boxed{\text{Example 1}}$ Finding the largest number in a list of n numbers.
Size is $2n-1$ which is $O(n)$.

$\boxed{\text{Example 2}}$ Bubble sort. Size is $\frac{n(n+1)}{2} = \underbrace{\frac{1}{2}n^2 + \frac{1}{2}n}_{\text{polynomial}}$ which is $O(n^2)$

$\boxed{\text{Example 4}}$ Polynomial evaluation

Algorithm #1. Step 2. For each k: is $k \le n$? 1 comparison

$$S \to \underbrace{\underbrace{S + \underbrace{a_k \underbrace{x^k}_{3k-2 \text{ operations}}}_{1 \text{ multiplication}}}_{1 \text{ addition}}}_{3k \text{ operations}}$$

$k \to k+1$ 1 addition

Total $3k+2$ operations

Total operations for all steps $k = 1, 2, 3, \cdots, n$

$$(3\cdot 1+2) + (3\cdot 2+2) + (3\cdot 3+2) + \cdots + (3\cdot n+2) + \underbrace{1}$$

for the last comparison when $k = n+1$ to stop the procedure.

$$= 3(1+2+3+\cdots+n) + n\cdot 2 + 1$$

$$= 3\left(\frac{n(n+1)}{2}\right) + 2n + 1$$

$$= \frac{3}{2}(n^2+n) + 2n + 1$$

$$= \frac{3}{2}n^2 + \frac{7}{2}n + 1 = f(n) = \text{complexity}$$

which is $O(n^2)$.

Algorithm #2. Horner's method has complexity only $5n+1$ which is $O(n)$.