Mat 2345

Week 6

Fall 2013

Student Responsibilities — Week 6

- **Reading**: Textbook, Section 3.1–3.2
- **Assignments**:
  1. for sections 3.1 and 3.2
  2. Worksheet #4 on Execution Times
  3. Worksheet #5 on Growth Rates
- **Attendance**: Strongly Encouraged

Week 8 Overview

- 3.1 Algorithms
- 3.2 Growth of Functions

3.1 Algorithms

- **Algorithm**: a finite set of unambiguous instructions for performing a computation or for solving a problem.
- Examples:
  - Shampoo Instructions: Lather, Rinse, Repeat
  - Recipe for making Italian Beef:
    - Place beef roast
    - 1 pkg Au Jus dry gravy mix
    - 1 pkg dry Italian dressing mix and
    - 1 C water in slow cooker
    - cook all day or over night
    - shred beef and serve with French Bread or rolls
  - The instructions that come with a sewing pattern
  - The instructions for a model airplane or rocket kit
  - INSERT DISK AND PRESS ANY KEY TO CONTINUE

Properties of Algorithms

- **Input** – an algorithm usually has input from a specified set
- **Output** – the solution to the problem, also from a specified set
- **Definiteness** – steps of an algorithm must be defined precisely
- **Correctness** – an algorithm must produce the correct values for each of the input values

Properties of Algorithms — Continued

- **Finiteness** – an algorithm must produce the desired output after a finite (but perhaps large) number of steps for any input in the set
- **Effectiveness** – it must be possible to perform each step of an algorithm exactly and in a finite amount of time
- **Generality** – an algorithm should be applicable for all problems of the desired form, not just for a particular set of input values

Finding the Maximum Value in a Finite Sequence — Pseudocode

```
integer max(a1, a2, ..., an :  integers)

currmax := a1

for i := 2 to n

     if currmax < ai then currmax := ai

{currmax is the largest element}
return currmax;
```

C++ Implementation of Max

```
template <class T>
T Max (const vector<T> & L){

// PRE: L not empty, type T is comparable
// POST: returns the largest value in vector L

T mymax = L[0];

for (int i = 1; i < L.size; i++)
      if (mymax < L[i])
         mymax = L[i];

return mymax;
}
```

Search Algorithms

- The problem: locate a particular element (**target**) in a list
- Distinguish between **unordered** and **ordered** (**sorted**) lists
- Two primary algorithms:
  - **Linear** or **Sequential Search** — look at each item in the list, first to last, comparing them to target until target is found or we reach end of list
  - **Binary Search** — (**only** used on **ordered** lists) — compare target to middle element; discard low or high half of list and repeat on remaining half of list until found or list is empty

The Linear Search Algorithm

```
integer LinearSearch
      ( x:  integer,
         a1, a2, ..., an:  distinct integers)

i := 1

while (i <= n and x != ai)
      i := i + 1

if i <= n
      then return i
      else return 0

{ the value returned is the subscript of term that
   equals x, or is 0 if x is not found }
```

Note: it doesn't matter if the list is ordered or not, this algorithm will still work

The Binary Search Algorithm

```
integer BinarySearch( x:  integer,
                 a1, a2, ..., an:  increasing integers)

i := 1     {left endpoint of search interval}
j := n     {right endpoint of search interval}
while i < j
begin
      m := floor[(i + j) / 2]
      if x > am
         then i := m + 1
         else j := m
end

if x = ai
      then return i
      else return 0
```

Notes: the value returned is the subscript of term that equals x, or is 0 if x is not found; can only be used on sorted list

Bubble Sort Algorithm

```
procedure Bubblesort (a1, ..., an:
                 real numbers with n >= 2)
for i := 1 to n-1
     for j := 1 to n - i
         if a(j) > a(j+1) then swap a(j) and a(j+1)

{ a1, ..., an is in increasing order }
```

Other Sorts:
Selection Sort
Insertion Sort
Merge Sort
Quick Sort
Bucket Sort
Radix Sort

3.2 Growth of Functions

- **Time Complexity** is a measure of the computational "steps" of an algorithm relative to the size of input
- Algorithms are analyzed to see how the number of computational steps grows in relation to the size of input, **n**
- Once we have a function to compute the **time complexity** of algorithms which solve the same problem, we can compare them to determine which is more efficient
- For example, if the time it takes one sorting algorithm to sort **n** values is

$$T_1(n) \;=\; \tfrac{3}{2}n^2 + 3n$$

and another takes time

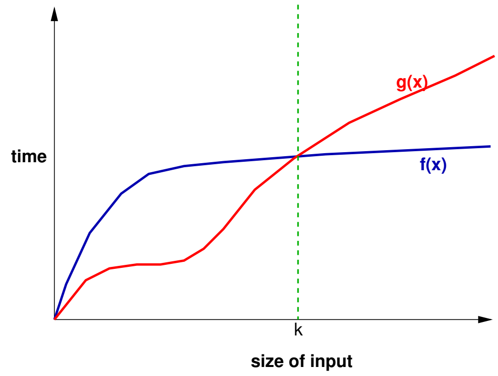
$$T_2(n) \;=\; 5n\log n + 29$$

which algorithm should we implement?

Comparing Function Growth

- Quantify the concept: g **grows at least as fast as** f
- What really matters when comparing the complexity of algorithms?
  - We mostly care about the behavior for **large** problems (i.e., what happens for "sufficiently large" input sizes)
  - Even bad algorithms can be used to solve "sufficiently small" problems
  - We can ignore some implementation details such as loop counter incrementation — we can straight–line any loop, etc.
- Remember, the functions we're discussing represent the **time complexities** of algorithms.

$f \in O(g)$



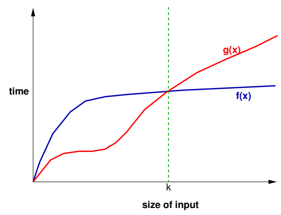
Big–Oh Notation

- g **asymptotically dominates** f:
  Let f and g be functions from $\mathbb{N}$ to $\mathbb{R}$.
  Then $f \in O(g)$ — f is Big–Oh of g or f is order g — IFF
  $\exists k\ \exists C\ \forall n[n > k \rightarrow |f(n)| \leq C|g(n)|,\quad k, C > 0]$
- In English: for **sufficiently** large n, if the function f is bounded from above by a positive, constant multiple of the function g, then we say f is "Big–Oh" of g.

$$\text{If } lim_{n\rightarrow\infty}\frac{f(n)}{g(n)} = 0 \text{ then } f \in o(g)$$

(f is Little–Oh of g, or f is strictly bounded by g)

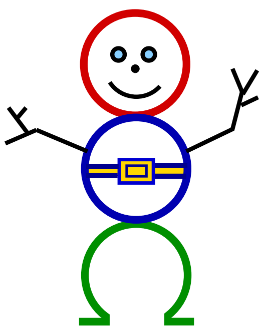
Proving Asymptotic Domination



To prove $f \in O(g)$, given f and g:

- Determine / choose some positive k
- Determine / choose a positive C (which may depend upon choice of k)
- Once k and C are chosen, the implication must be proven true

Three Important Complexity Classes



Bounds from above
Big – Oh

Growth rates equivalent
Theta

Bounds from below
Big – Omega

Omega Man

Complexity Classes

- The sets $O(g)$, $o(g)$, $\Omega(g)$, $\omega(g)$, and $\Theta(g)$ are called **complexity classes**.
- $O(g)$ is a **set** which contains all the functions which g **dominates**.

  f is $O(g)$ means $f \in O(g)$
- We say $f \in \Omega(g)$ if there are positive constants k and C such that $f(n) \geq Cg(n)$ whenever $n > k$
- If $f \in O(g)$ and $f \in \Omega(g)$, then $f \in \Theta(g)$
- We use "little-oh" and "little–omega" when we have **strict** inequality

Example

Let $f(n) = 4n + 5$ and $g(n) = n^2$.

We wish to show that $f \in O(g)$

We need to find constants k and C, then show the implication

$$\forall n > k, \quad f(n) \leq Cg(n)$$

is true for the values we chose.

To find k, we can set the functions equal, and solve for n:

$$\begin{aligned} 4n + 5 &= n^2 \\ 0 &= n^2 - 4n - 5 \\ 0 &= (n-5)(n+1) \end{aligned}$$

So, $n = 5$ or $n = -1$, *but* n is the size of input and therefore cannot be negative. Thus, k must be at least 5.
If we choose $k = 6$, then C can be any positive number greater than or equal to 1.

All that is left is the proof that $\forall n > k, \quad f(n) \leq Cg(n)$, which we shall revisit when we discuss induction proofs.

Big–Oh Properties

- f is $O(g)$ IFF $O(f) \subseteq O(g)$
- If $f \in O(g)$ and $g \in O(f)$, then $O(f) = O(g)$
- The set $O(g)$ is **closed under addition**:
  If $f \in O(g)$ and $h \in O(g)$, then $f + h \in O(g)$
- $O(g)$ is **closed under multiplication by a scalar** $a \in \mathbb{R}$:
  If $f \in O(g)$ then $af \in O(g)$
  I.e., $O(g)$ is a **vector space**
- Also, as you would expect,
  If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$

In particular,

$$O(f) \subseteq O(g) \subseteq O(h)$$

Theorem

If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then:

1. $f_1 f_2 \in O(g_1 g_2)$
2. $f_1 + f_2 \in O(max\{g_1, g_2\})$

Functional Values for Small n

$\log_2 n$	$\sqrt{n}$	n	n^2	2^n	$n!$	n^n
0	1.0000	**1**	1	2	1	1
1.0000	1.4142	**2**	4	4	2	4
1.5850	1.7321	**3**	9	8	6	27
2.0000	2.0000	**4**	16	16	24	256
2.3219	2.2361	**5**	25	32	120	3125
2.5850	2.4495	**6**	36	64	720	46,656
2.8074	2.6458	**7**	49	128	5040	823,543
3.0000	2.8284	**8**	64	256	40,320	1.67×10^7
3.1699	3.0000	**9**	81	512	362,880	3.87×10^8
3.3219	3.1623	**10**	100	1024	3.6×10^6	10^{10}

Approximate Functional Values for Powers of n

$\log_2 n$	$\sqrt{n}$	n	n^2	2^n	$n!$	n^n
3.32	3.16	10^1	10^2	1024	$3.63(10^6)$	10^{10}
6.64	10	10^2	10^4	$1.27(10^{30})$	$9.3(10^{157})$	10^{200}
9.97	31.62	10^3	10^6	$1.07(10^{301})$	$4(10^{2567})$	10^{3000}
13.29	100	10^4	10^8	$2(10^{3010})$	$2.9(10^{35,659})$	$10^{40,000}$
16.61	316.2	10^5	10^{10}	$10^{30,103}$	$2.9(10^{456,573})$	$10^{500,000}$
19.93	1000	10^6	10^{12}	$10^{301,030}$	$8.3(10^{5,565,708})$	$10^{60,000,000}$
39.86	10^6	10^{12}	10^{24}	BIG	LARGE	HUGE

Important Complexity Classes

Theorem. The hierarchy of several familiar sequences in the sense that each sequence is Big–Oh of any sequence to its right:

$$1,\ \log_2 n,\ \ldots,\ \sqrt[4]{n},\ \sqrt[3]{n},\ \sqrt{n},\ n,\ n\log_2 n,\ n\sqrt{n},\ n^2,\ n^3,\ n^4,\ \ldots,\ 2^n,\ n!,\ n^n$$

Similarly, stated in set notation:

$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^j) \subseteq O(c^n) \subseteq O(n!)$$

where $j > 2$ and $c > 1$

Time Equivalences

Second	1	10^0
Milliseconds	1,000	10^3
Microseconds	1,000,000	10^6
Nanoseconds	1,000,000,000	10^9

Largest Problem Sizes

Let $f(n)$ be the time complexity of an algorithm in MICROSECONDS.

The **largest** problem of size n that can be solved in:

1 second	is	$f(n)/10^6$
1 minute	is	$f(n)/(60 * 10^6)$
1 hour	is	$f(n)/(60 * 60 * 10^6)$
1 day	is	$f(n)/(24 * 60 * 60 * 10^6)$
1 month	is	$f(n)/(\mathbf{30} * 24 * 60 * 60 * 10^6)$
1 year	is	$f(n)/(12 * \mathbf{30} * 24 * 60 * 60 * 10^6)$
1 century	is	$f(n)/(100 * 12 * \mathbf{30} * 24 * 60 * 60 * 10^6)$

Largest Problems "Do–able" in 1 Second

1. Let $f(n) = n$. Then the largest problem for which we can compute an answer in one second is:
$$n/10^6 = 1$$
$$n = 10^6$$

2. Let $f(n) = n^2$. Then the largest problem for which we can compute an answer in one second is:
$$n^2/10^6 = 1$$
$$n = \sqrt{10^6} = 10^3$$

3. Let $f(n) = 2^n$. Then the largest problem for which we can compute an answer in one second is:
$$2^n/10^6 = 1$$
$$2^n = 10^6 \approx 2^{19}$$
$$n \approx 19$$

Let $f(n) = n!$. Then the largest problem for which we can compute an answer in one second is:

$$n!/10^6 = 1$$

$$n! = 10^6$$

Here, it helps to use a calculator. . . , and we find

$9! = 362,880$ — too small

$10! = 3,628,880$ — too large

$$n \approx 9$$

(Recall that n is input size)

Review — Exponents

$$x^a x^b = x^{a+b} \qquad 2^5 2^7 = 2^{12}$$
$$\frac{x^a}{x^b} = x^{a-b} \qquad \frac{3^8}{3^2} = 3^6$$
$$(x^a)^b = x^{ab} \qquad (5^2)^3 = 5^6$$

Notes

$x^n + x^n = 2x^n$ ...... **not** x^{2n} $\qquad 3^2 + 3^2 = 2(3^2)$

$5^3 + 5^3 =$ $\qquad 2^7 + 2^7 =$

$2^n + 2^n =$ $\qquad 2^{100} + 2^{100} =$

$3^4 + 5^4 =$ $\qquad 2^n + 3^n =$

Review — Logarithms

Logarithm: $x^a = b$ IFF $\log_x b = a$

$2^3 = 8$ IFF $\log_2 8 = 3$

$5^4 = 625$ IFF log ________________

__________ = __________ IFF $\log_3 81 = 4$

Theorem. $\log ab = \log a + \log b$

$$\begin{aligned} \log 32 &= \log(2^5) = 5 \\ &= \log(8 * 4) \\ &= \log 8 + \log 4 \\ &= \log 2^3 + \log 2^2 \\ &= 3 + 2 = 5 \ \checkmark \end{aligned}$$

Other Formulae You Should Know

- $\log \frac{a}{b} = \log a - \log b$
  - $\log \frac{32}{4} = \log \frac{2^5}{2^2} = \log 2^{5-2} = \log 2^3 = 3$ — and —
  - $\log \frac{32}{4} = \log 32 - \log 4 = \log 2^5 - \log 2^2 = 5 - 2 = 3$
  - Determine $\log \frac{1024}{64}$ both ways shown above
- $\log a^b = b \log a$
  - $\log 16^3 = \log(2^4)^3 = \log 2^{12} = 12$ — and —
  - $\log 16^3 = 3 \log 16 = 3 \log 2^4 = 3 * 4 = 12$
  - Determine $\log 128^5$ both ways shown above

Other General Knowledge

- $\forall\, x > 0,\ \log x < x \qquad \forall\, n > 0,\ \log n < n$
- $\log 1 = 0,\ \log 2 = 1,\ \log 1024 = 10,\ \log 1{,}048{,}576 = 20$
- $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

  Thus, if $n = 3$:

  $\sum_{i=0}^{3} 2^i = 2^0 + 2^1 + 2^2 + 2^3 = 1+2+4+8 = 15$

  — and —

  $2^{n+1} - 1 = 2^{3+1} - 1 = 2^4 - 1 = 16 - 1 = 15$

- In general:

  $\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$

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

  Thus, $\sum_{i=1}^{5} i = 1+2+3+4+5 = 15$ — and —

  $\sum_{i=1}^{5} i = \frac{5(5+1)}{2} = \frac{5*6}{2} = \frac{30}{2} = 15$