Analyzing Algorithms

CSE 326
Data Structures

Algorithm Analysis: Why?

- Correctness:
  - › Does the algorithm do what is intended?
- Performance:
  - › What is the running time of the algorithm?
  - › How much storage does it consume?
- Different algorithms may correctly solve a given task
  - › Which should I use?

Evaluating an algorithm

Mike: My algorithm can sort 10^6 numbers in 3 seconds.
Bill: My algorithm can sort 10^6 numbers in 5 seconds.

Mike: I've just tested it on my new Pentium IV processor.
Bill: I remember my result from my undergraduate studies (19xx).

Mike: My input is a random permutation of $1..10^6$.
Bill: My input is the sorted output, so I only need to verify that it is sorted.

Program Evaluation / Complexity

- Processing time is surely a bad measure!!!
- We need a 'stable' measure, independent of the implementation.

* A complexity function is a function T: N   N.
T(n) is the number of operations the algorithm does on an input of size n.
"input" generally refers to parameters or data
* We can try to calculate at least three different things.

- Worst-case complexity
- Best-case complexity
- Average-case complexity

The RAM Model of Computation

- Each simple operation takes 1 time step.
  - E.g. elementary arithmetic operations and assignments
- Loops and subroutines are not simple operations.
- Each memory access takes one time step, and there is no shortage of memory.

For a given problem instance:

- Running time of an algorithm = # RAM steps.
- Space used by an algorithm = # RAM memory cells

useful abstraction ⇒ allows us to analyze algorithms in a machine independent fashion.

Why the RAM Model is Justified

- Most CPUs have a similar basic instruction set
  - Similar operations take similar numbers of machine steps, to a constant factor
  - As technology improves, speed up is generally linear (a constant factor)

Big O Notation

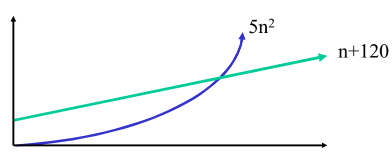
- Goal :
  - Be able to compare complexity function
  - A stable measurement independent of the machine.
- Way:
  - ignore constant factors.
- $f(n) = O(g(n))$ if $c \cdot g(n)$ is upper bound on $f(n)$

⇔ There exist c, N, s.t. for any $n \geq N$, $f(n) \leq c \cdot g(n)$

Consider large inputs (asymptotic behavior)

Ignore constants

Big O Notation



For all $n \geq 5$ $(N=5)$
$n+120 \leq 5n^2$

$\Rightarrow n+120 = O(n^2)$

Ω Notation

- $f(n) = \Omega(g(n))$ if $c \cdot g(n)$ is lower bound on $f(n)$

$\Leftrightarrow$ There exist c, N, s.t. for any $n \geq N$, $f(n) \geq c \cdot g(n)$

6/22/2004 9

Θ Notation

- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

$\Leftrightarrow$ There exist c_1, c_2, N, s.t. for $n \geq N$,

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

6/22/2004 10

o Notation ("little o")

- $f(n) = o(g(n))$ if $f(n) = O(g(n))$ but $g(n) \mathrel{!=} O(f(n))$

6/22/2004 11

Ω, Θ Examples

Examples:

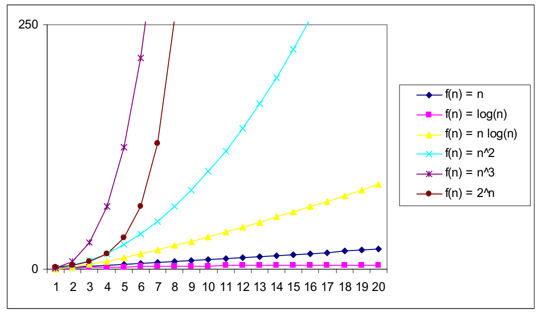
$4x^2+100 = O(x^2)$	$4x^2+100 \neq \Theta(x^3)$
$4x^2+100 = \Omega(x^2)$	$4x^2+100 = O(x^3)$
$4x^2+100 = \Theta(x^2)$	$4x^2+100 = \Omega(x)$
$4x^2 -100 = O(x^2)$	$4x^2 + x\log x = O(x^2)$
$123400 = O(1)$	

6/22/2004 12

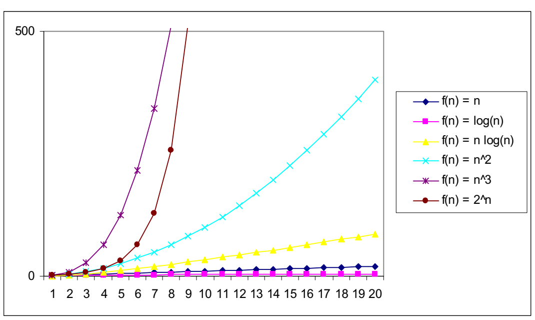
Growth Rates

- Even by ignoring constant factors, we can get an excellent idea of whether a given algorithm will be able to run in a reasonable amount of time on a problem of a given size.
- The “big O” notation and worst-case analysis are tools that greatly simplify our ability to compare the efficiency of algorithms.

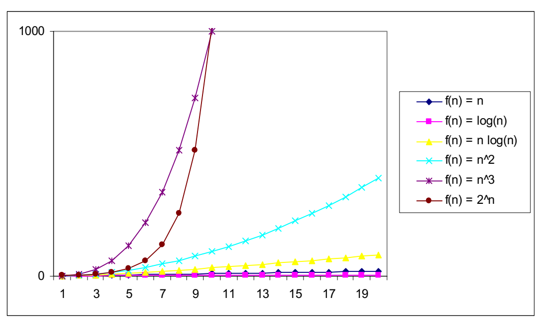
Practical Complexity



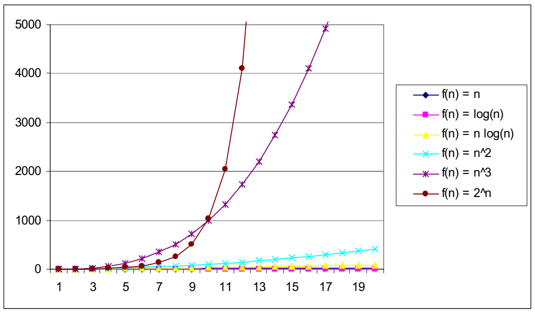
Practical Complexity



Practical Complexity



Practical Complexity



Big O Fact

- A polynomial of degree k is $O(n^k)$
- Proof:
  - › Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a = \max_i \{b_i\}$
  - › $f(n) \leq an^k + an^{k-1} + \ldots + an + a$

    $\leq kan^k \leq cn^k$ (for $c=ka$).

Iterative Algorithm for Sum

- Find the sum of the first **num** integers stored in an array **v**.

```
sum(v[ ]: integer array, num: integer): integer{
    temp_sum: integer ;
    temp_sum := 0;
    for i := 0 to num - 1 do
        temp_sum := v[i] + temp_sum;
    return temp_sum;
}
```

Note the use of pseudocode

Programming via Recursion

- Write a *recursive* function to find the sum of the first **num** integers stored in array **v**.

```
sum (v[ ]: integer array, num: integer): integer {
    if (num = 0) then
        return 0
    else
        return (v[num-1] + sum(v,num-1));
}
```

Pseudocode

- In the lectures algorithms will sometimes be presented in pseudocode.
  - This is very common in the computer science literature
  - Pseudocode is usually easily translated to real code.
  - This is programming language independent
- Pseudocode can also be used for pencil-and-paper homework

Review: Induction

- Suppose
  - S(k) is true for fixed constant k
    - Often k = 0
  - S(n) implies S(n+1) for all n >= k
- Then S(n) is true for all n >= k

Proof By Induction

- Claim:S(n) is true for all n >= k
- Base:
  - Show S(n) is true for n = k
- Inductive hypothesis:
  - Assume S(n) is true for an arbitrary n
- Step:
  - Show that S(n) is then true for n+1

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Basis: 1. show that $a^0 = (a^{0+1} - 1)/(a - 1)$: $a^0 = 1 = (a^1 - 1)/(a - 1)$. 2. Show true for n=2.
  - Inductive hypothesis:
    - Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show true for n+1): $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$ $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

Program Correctness by Induction

- **Basis Step:** sum(v,0) = 0.
- **Inductive Hypothesis (n=k):** Assume sum(v,k) correctly returns sum of first k elements of v, i.e. `v[0]+v[1]+…+v[k-1]`
- **Inductive Step (n=k+1):** sum(v,n) returns `v[k]+sum(v,k)` which is the sum of first k+1 elements of v.

Algorithms vs Programs

- Proving correctness of an algorithm is very important
  - › a well designed algorithm is guaranteed to work correctly and its performance can be estimated
- Proving correctness of a program (an implementation) is fraught with weird bugs
  - › Abstract Data Types are a way to bridge the gap between mathematical algorithms and programs

Moore's Law

- Moore's Law: Transistor density doubles roughly every 18 months
  - › Translates into a CPU speed-up of the same amount
  - › Has been true for 20 years
- Similar "laws" have been observed in some other technology areas
- Question for discussion: why doesn't Moore's law save us from worrying about efficiency?