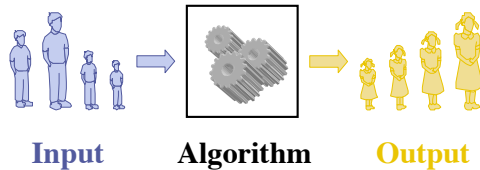


Analysis of Algorithms



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

Pseudocode (§3.1.2)

- ◆ High-level description of an algorithm
- ◆ More structured than English
- ◆ Less detailed than a program
- ◆ Preferred notation for describing algorithms
- ◆ Hides program design issues

Example: find max element of an array

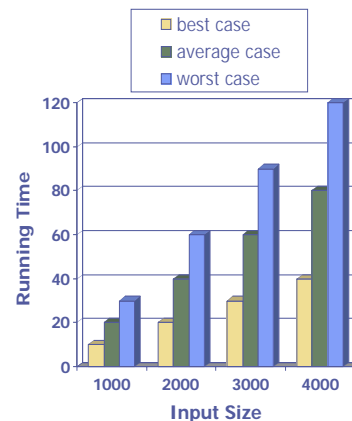
Algorithm *arrayMax*(*A*, *n*)
Input array *A* of *n* integers
Output maximum element of *A*

```

currentMax ← A[0]
for i ← 1 to n - 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
return currentMax
    
```

Running Time (§3.1)

- ◆ Most algorithms transform input objects into output objects.
- ◆ The running time of an algorithm typically grows with the input size.
- ◆ Average case time is often difficult to determine.
- ◆ We focus on the worst case running time.
 - Easier to analyze
 - Crucial to applications such as games, finance and robotics

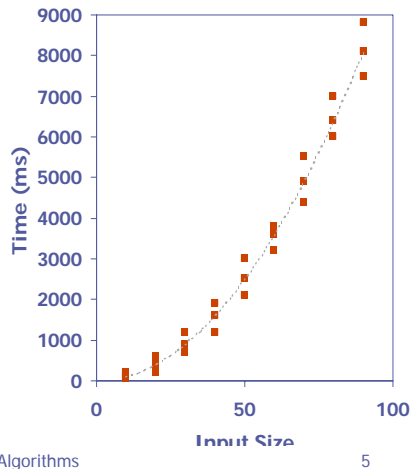


Running Time

- ◆ The running time of an algorithm depends on
 - Computer - quality and type
 - Programmer skill
 - Input - types and amounts
 - Algorithm
 - ◆ Different ways to solve the same problem - choose wisely

Experimental Studies (§ 3.1.1)

- ◆ Write a program implementing the algorithm
- ◆ Run the program with inputs of varying size and composition
- ◆ Measure the runtime using *time*
- ◆ Plot the results



10/29/04

Analysis of Algorithms

5

Limitations of Experiments

- ◆ It is necessary to implement the algorithm, which may be difficult
- ◆ Good range of inputs?
- ◆ In order to compare two algorithms, the same hardware and software environments must be used

10/29/04

Analysis of Algorithms

6

Theoretical Analysis

- ◆ Uses a high-level description of the algorithm instead of an implementation
- ◆ Characterizes running time as a function of the input size, n .
 - Look at large input sizes

10/29/04

Analysis of Algorithms

7

Theoretical Analysis

- ◆ Takes into account all possible inputs
- ◆ Evaluates algorithm independent of hardware, implementation, input set, etc.
- ◆ Count operations not actual clock time

10/29/04

Analysis of Algorithms

8

Primitive Operations

- ◆ Basic computations performed by an algorithm
 - Evaluating an expression
 - Assigning a value to a variable
 - Indexing into an array
 - Calling a method
 - Returning from a method
- ◆ Count primitive operations

Counting Primitive Operations (§3.4.1)

- ◆ By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *printArray(A, n)*

```
i ← 0           1 assignment
while i < n do  n + 1 comparisons
    cout << A[i] << endl  n outputs
    i ++           n increments
```

$1 + (n+1) + n + n = 3n + 2$ operations
Proportional to n , more items = more time

Counting Primitive Operations (§3.4.1)

- ◆ In class exercise

Algorithm *foo(n)*

```
x ← 0, y ← 0
while x < n do
    x ++
    while y < n do
        y ++
    y = 0
```

Counting Primitive Operations (§3.4.1)

- ◆ Do you think this is how designers calculate runtime?
 - Error prone
 - Takes too long

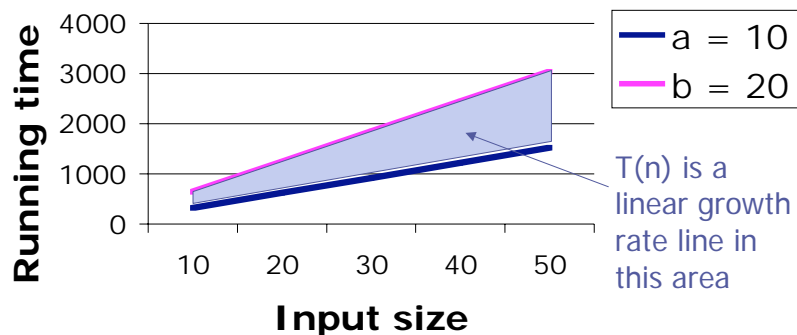
Estimating Running Time

- ◆ Algorithm *printArray* executes $3n + 2$ primitive operations in the worst case. Define:
 - a = Time taken by the fastest primitive operation
 - b = Time taken by the slowest primitive operation
- ◆ Let $T(n)$ be worst-case time of *printArray*. Then
$$a(3n + 2) \leq T(n) \leq b(3n + 2)$$
- ◆ Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

- ◆ Changing the hardware/ software environment
 - Affects $T(n)$ by a constant factor, but
 - Does not alter the growth rate of $T(n)$
- ◆ *printArray* has a linear growth rate

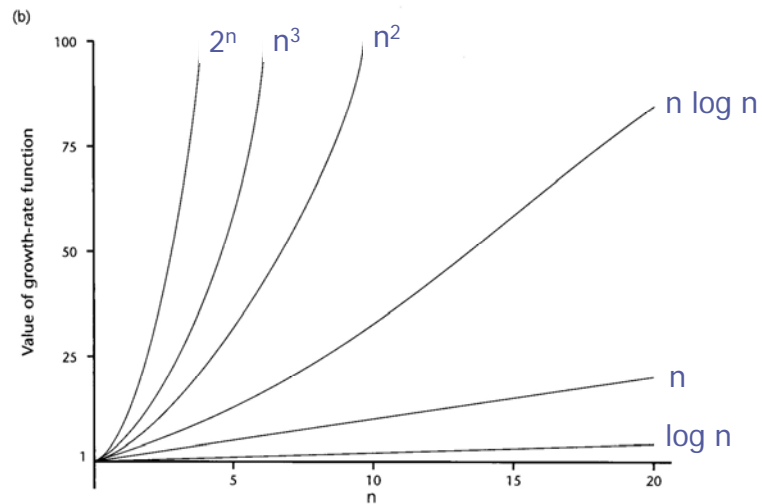
Growth Rate of Running Time



Growth Rates

- ◆ Growth rates of functions in order of increasing growth rate:
 - Constant ≈ 1
 - Logarithmic $\approx \log n$ (log base 2)
 - Linear $\approx n$
 - Logarithmic $\approx n \log n$ (log base 2)
 - Quadratic $\approx n^2$
 - Cubic $\approx n^3$
 - Exponential $\approx 2^n$

Growth Rates



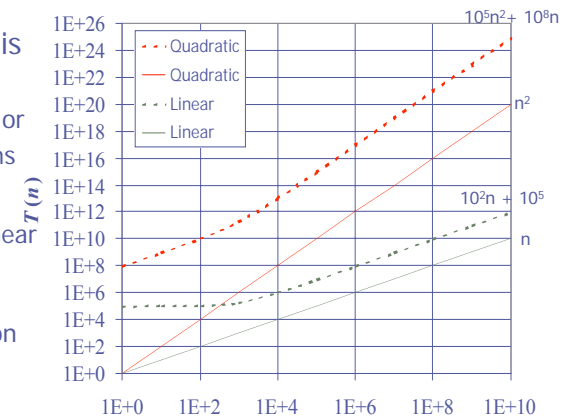
Constant Factors and Low-Order Terms

◆ The growth rate is not affected by

- constant factors or
- lower-order terms

◆ Examples

- ✓ $10^2n + 10^5$ is a linear function
- ✓ $10^5n^2 + 10^8n$ is a quadratic function



10/29/04

Analysis of Algorithms

18

Constant Factors and Low-Order Terms

◆ Examples

- $2n$ and $100n$ have the same relative growth rates
- $10n$ and $10n + 4$ have the same relative growth rates
- $3n^2 + 10n + 7$ and n^2 have the same relative growth rates
- $10000n + 1000$ and n have the same relative growth rates

10/29/04

Analysis of Algorithms

19

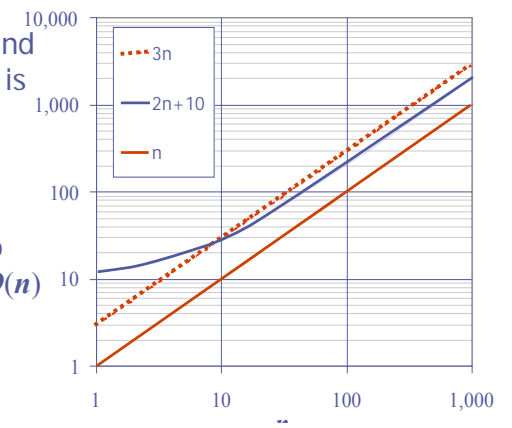
Big-Oh Notation (§3.5)

◆ Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

◆ Example: $2n + 10$ is $O(n)$

- ✓ $2n + 10 \leq cn$
- ✓ $(c - 2)n \geq 10$
- ✓ $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$



10/29/04

Analysis of Algorithms

20

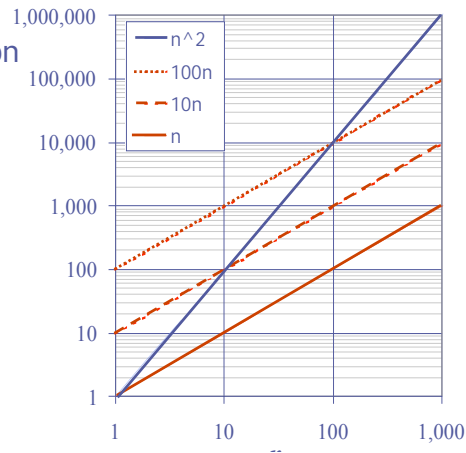
Big-Oh Example

◆ Example: the function n^2 is not $O(n)$

∨ $n^2 \leq cn$

∨ $n \leq c$

- The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

◆ $7n-2$

$7n-2$ is $O(n)$

need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$

this is true for $c = 7$ and $n_0 = 1$

■ $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$

this is true for $c = 4$ and $n_0 = 21$

■ $3 \log n + \log \log n$

$3 \log n + \log \log n$ is $O(\log n)$

need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \cdot \log n$ for $n \geq n_0$

this is true for $c = 4$ and $n_0 = 2$

Big-Oh and Growth Rate

- ◆ The big-Oh notation gives an upper bound on the growth rate of a function
- ◆ The statement " $f(n)$ is $O(g(n))$ " means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- ◆ We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

◆ If $f(n)$ is a polynomial of degree d , then $f(n)$ is $O(n^d)$, i.e.,

1. Drop lower-order terms
2. Drop constant factors

■ $f(n) = 4n^4 + n^3 \Rightarrow O(n^4)$

◆ Use the smallest possible class of functions

- Say " $2n$ is $O(n)$ " instead of " $2n$ is $O(n^2)$ "

Big-Oh Rules

◆ You can combine growth rates

- $O(f(n) + O(g(n)) = O(f(n) + g(n))$
- $O(n) + O(n^3 + 5) = O(n + n^3 + 5) = O(n^3)$

Rules for Analyzing Running Time

◆ Loops

- The running time of a loop is at most the running time of the statements inside the loop times the number of iterations

```
for ( x = 0; x < N; x ++ ) {  
    statement 1  
    statement 2  
    ...  
    statement c  
}
```

N iterations
c statements
 $O(cN) = O(N)$

Rules for Analyzing Running Time

◆ Nested Loops - analyze inside out

- The running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all loops

```
for ( x = 0; x < N; x ++ ) {  
    for ( y = 0; y < N; y ++ ) {  
        statement 1  
    }  
}
```

$N*N$ iterations
1 statements
 $O(N*N) = O(N^2)$

Rules for Analyzing Running Time

◆ Consecutive statements

- Just add them together - largest one counts

```
statement 1  
statement 2  
for ( x = 0; x < N; x ++ ) {  
    statement 3  
}
```

2 statements
N iterations
 $O(2+N) = O(N)$

Rules for Analyzing Running Time

◆ if/else statements

- The running time is never more than the running time of the test plus the larger of the running times of S1 and S2

```
if ( condition )
    S1          O(running time of condition) +
else          max ( O(running time of S1),
    S2          O(running time of S2) )
```

Asymptotic Algorithm Analysis

- ◆ The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- ◆ To perform the asymptotic analysis
 - We find the worst-case number of primitive operations executed as a function of the input size
 - We express this function with Big-Oh notation

Analyzing Running Time

- ◆ In class exercise - give the Big-Oh running time of the following code

```
for ( x = 0; x < N; x ++ )
    array[x] = x*N;

for ( x = 0; x < N; x ++ ) {
    if ( x < (N/2) )
        cout << array[x];
    else
        for ( y = 0; y < N; y ++ )
            cout << y*array[x];
}
```

Calculating Big-Oh

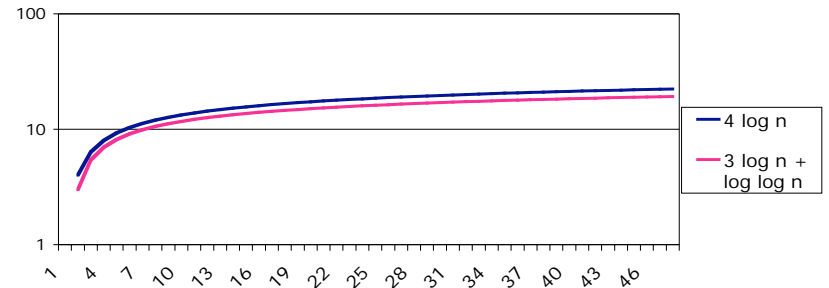
- ◆ In class exercise - Give the Big-Oh notation for the following functions:
 - $n + \log(n) =$
 - $8n \log(n) + n^2 =$
 - $6n^2 + 2^n + 300 =$
 - $n + n \log(n) + \log(n) =$
 - $40 + 8n + n^7 =$

Math you need to Review

- ◆ Summations (Sec. 1.3.1)
- ◆ Logarithms and Exponents (Sec. 1.3.2)
 - ◆ **properties of logarithms:**
 - $\log_b(xy) = \log_b x + \log_b y$
 - $\log_b(x/y) = \log_b x - \log_b y$
 - $\log_b x a = a \log_b x$
 - $\log_b a = \log_x a / \log_x b$
 - ◆ **properties of exponentials:**
 - $a^{(b+c)} = a^b a^c$
 - $a^{bc} = (a^b)^c$
 - $a^b / a^c = a^{(b-c)}$
 - $b = a^{\log_a b}$
 - $b^c = a^{c \cdot \log_a b}$
- ◆ Proof techniques (Sec. 1.3.3)
- ◆ Basic probability (Sec. 1.3.4)

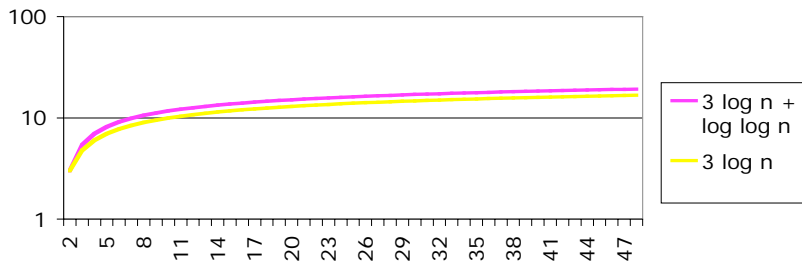
Big-Oh

- ◆ **big-oh**
 - $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
 - $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically **less than or equal** to $g(n)$
 - Example: $3 \log n + \log \log n = O(\log n)$ for $c = 4$ and $n \geq 2$



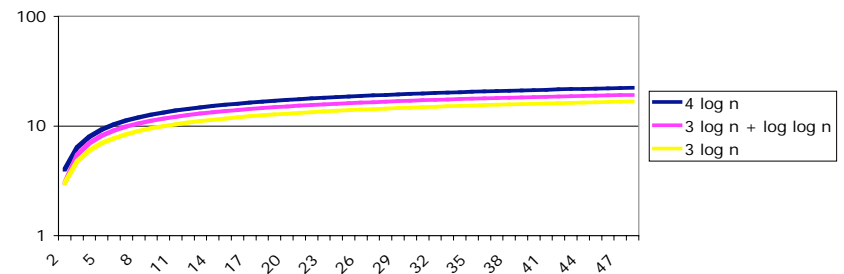
Relatives of Big-Oh

- ◆ **big-Omega**
 - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
 - $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal** to $g(n)$
 - Example: $3 \log n + \log \log n = \Omega(\log n)$ for $c = 3$ and $n \geq 2$



Relatives of Big-Oh

- ◆ **big-Theta**
 - $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$
 - $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal** to $g(n)$
 - Example: $3 \log n + \log \log n = \Theta(\log n)$ for $c' = 3$ and $c'' = 4$ and $n \geq 2$



Relatives of Big-Oh

◆ little-oh

- $f(n)$ is $o(g(n))$ if, **for any** constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) < c \cdot g(n)$ for $n \geq n_0$
- $f(n)$ is $o(g(n))$ if $f(n)$ is asymptotically **strictly less** than $g(n)$

◆ little-omega

- $f(n)$ is $\omega(g(n))$ if, **for any** constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) > c \cdot g(n)$ for $n \geq n_0$
- $f(n)$ is $\omega(g(n))$ if is asymptotically **strictly greater** than $g(n)$

Array Big Oh Running Times

- | | |
|---------------------------------|------------------------|
| ■ Unsorted insert | ■ Sorted Remove |
| ◆ $O(1)$ - add to end | ◆ $O(N)$ - shift items |
| ■ Sorted insert | ■ Unsorted Remove |
| ◆ $O(N)$ - shift items | ◆ $O(1)$ - move last |
| ■ Num items | ■ Linear search |
| ◆ $O(1)$ - have to keep counter | ◆ $O(N)$ |
| ■ Print | |
| ◆ $O(N)$ | |

Space Complexity

- ◆ Similar to determining Big-Oh
- ◆ Give upper bound on space required based on the input size
 - Constant factors and low-order terms are not significant