CS 483 - Data Structures and Algorithm Analysis

Lecture II: Chapter 2

R. Paul Wiegand

George Mason University, Department of Computer Science

February 1, 2006

Outline

Analyzing for Efficiency

- Though there are other factors, we concentrate our analysis on efficiency
  - Time efficiency — how fast an algorithm runs
  - Space efficiency — how much memory an algorithm
- Input size
  - Wall-clock time depends on the machine, but algorithms are machine independent
  - Typically, algorithms run longer as the size of its input increases
  - $\therefore$ We are interested in how efficiency scales wrt input size

Measuring input size

What do we measure to judge the size of the problem?

Measuring input size

What do we measure to judge the size of the problem?

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

$$p(x) = a_n x^n + \cdots + a_1 x + a_0$$

Measuring input size

What do we measure to judge the size of the problem?

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

- Number of list elements

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

$$p(x) = a_n x^n + \cdots + a_1 x + a_0$$

Measuring input size

What do we measure to judge the size of the problem?

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

- Number of list elements

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

- Order of matrices?
- Total number of elements?

$$p(x) = a_n x^n + \cdots + a_1 x + a_0$$

Measuring input size

What do we measure to judge the size of the problem?

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

- Number of list elements

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

- Order of matrices?
- Total number of elements?

$p(x) = a_n x^n + \cdots + a_1 x + a_0$

- Degree of polynomial?
- Total number of coefficients?

Measuring Running Time

What do we measure to judge the running time on an algorithm?

Measuring Running Time

What do we measure to judge the running time on an algorithm?

- Could count all operations executed ...
- But we typically concern ourselves with the *basic operation*, the one contributing the most to the total running time

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

Measuring Running Time

What do we measure to judge the running time on an algorithm?

- Could count all operations executed ...
- But we typically concern ourselves with the *basic operation*, the one contributing the most to the total running time

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

- Key comparisons

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

Measuring Running Time

What do we measure to judge the running time on an algorithm?

- Could count all operations executed ...
- But we typically concern ourselves with the *basic operation*, the one contributing the most to the total running time

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

- Key comparisons

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

- Additions?
- Multiplications?

Measuring Running Time

What do we measure to judge the running time on an algorithm?

- Could count all operations executed ...
- But we typically concern ourselves with the *basic operation*, the one contributing the most to the total running time

$\textsc{Sort}(a_1, a_2, \cdots, a_n)$

- Key comparisons

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{m1} \\ \vdots & \ddots & \vdots \\ b_{1k} & \cdots & b_{mk} \end{bmatrix} =$$

- Additions?
- Multiplications?

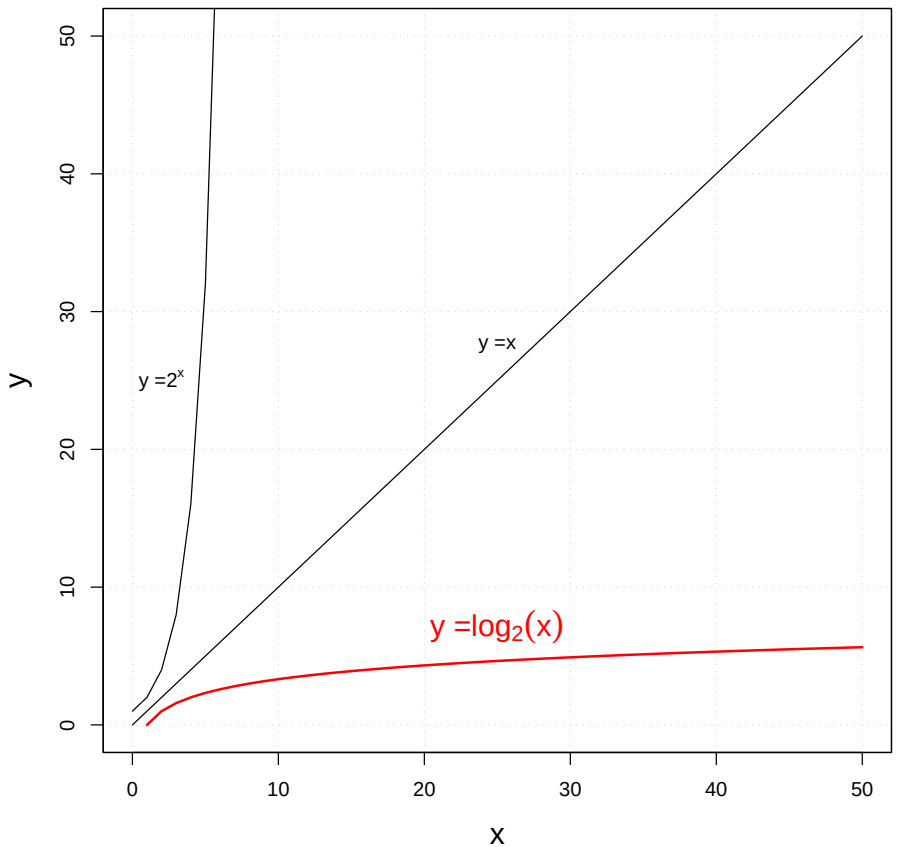
- In general, we can approximate running time as: $T(n) \approx c_{op} C(n)$
- Example: Given $C(n) := \frac{1}{2} n (n-1)$, what is the run-time effect of doubling the input size?

Sidebar: Logarithm Basics

- If $b^x = y$ then $\log_b y = x$
- We can change bases: $\log_a x = \frac{\log_b x}{\log_b a}$

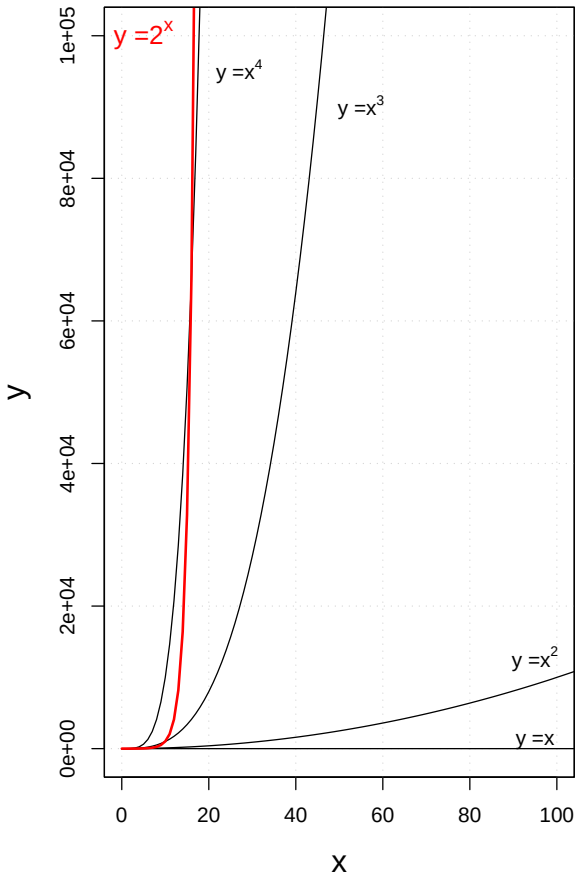
Sidebar: Logarithm Basics

- If $b^x = y$ then $\log_b y = x$
- We can change bases: $\log_a x = \frac{\log_b x}{\log_b a}$



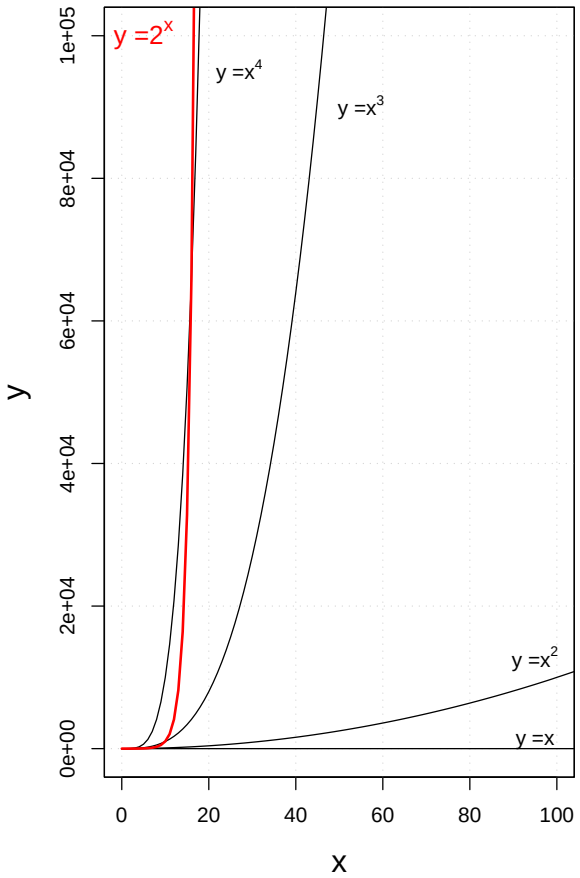
- $\log_b x^y = y \log_b x$
- $\log xy = \log x + \log y$
- $\log \frac{x}{y} = \log x - \log y$
- $a^{\log_b x} = x^{\log_b a}$

How Does Your Running Time Grow?



- We aren't so interested in running times on small inputs, but in how running time *scales* on very large inputs
- We are interested in an algorithm's *order of growth*, ignoring constant factors

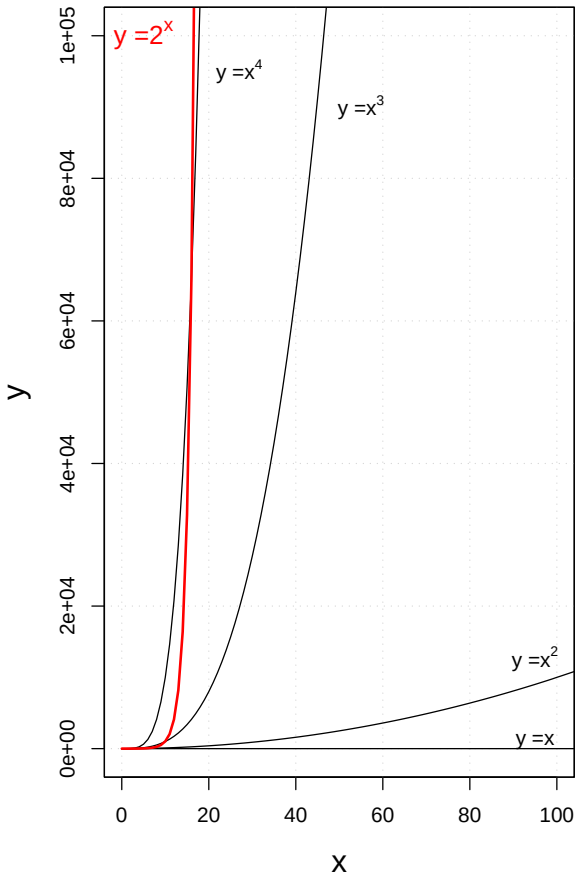
How Does Your Running Time Grow?



- We aren't so interested in running times on small inputs, but in how running time *scales* on very large inputs
- We are interested in an algorithm's *order of growth*, ignoring constant factors

1	*Constant*
$\log n$	*Logarithmic (sub-linear)*
n	*Linear*
$n \lg n$	*n-log-n*
n^k, for some constant k	*Polynomial*
k^n, for some constant k	*Exponential*
$n!$	*Factorial ("Exponential")*

How Does Your Running Time Grow?



- We aren't so interested in running times on small inputs, but in how running time *scales* on very large inputs
- We are interested in an algorithm's *order of growth*, ignoring constant factors

1	*Constant*
$\log n$	*Logarithmic (sub-linear)*
n	*Linear*
$n \lg n$	*n-log-n*
n^k, for some constant k	*Polynomial*
k^n, for some constant k	*Exponential*
$n!$	*Factorial ("Exponential")*

NOTE: Given a nonnegative integer d, we can write a polynomial in the form $p(n) = \sum_{i=0}^{d} a_i n^i$, where a_i are constants. We call d the *degree* of the polynomial.

Worst-Case Efficiency

SequentialSearch$(A[0 \ldots n-1], K)$

$i \leftarrow 0$
`while` $i < n$ `and` $A[i] \neq K$ `do`
    $i \leftarrow i + 1$
`if` $i < n$ `return` i
`else return` -1

Worst-Case Efficiency

SequentialSearch$(A[0 \ldots n-1], K)$

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Worst case is when K is not in $A[]$
- Search every element, requiring $C_{worst}(n) = n$ comparisons
- Worst case analysis provides an upper bound

Best-Case Efficiency

SequentialSearch$(A[0 \ldots n-1], K)$

$i \leftarrow 0$
while $i < n$ and $A[i] \neq K$ do
    $i \leftarrow i + 1$
if $i < n$ return i
else return -1

Best-Case Efficiency

SequentialSearch$(A[0 \ldots n-1], K)$

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Best case is when K is in $A[0]$
- Search first element, requiring $C_{best}(n) = 1$ comparisons
- Best case analysis provides a lower bound
- Generally, best-case efficiency is not as useful...

Average-Case Efficiency

SequentialSearch$(A[0 \ldots n-1], K)$

$i \leftarrow 0$
`while` $i < n$ `and` $A[i] \neq K$ `do`
    $i \leftarrow i + 1$
`if` $i < n$ `return` i
`else return` -1

Average-Case Efficiency

SequentialSearch$(A[0 \ldots n-1], K)$

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Average case asks a useful question: "What kind of running time to we *expect* to get when we don't know the data?"
- Let $p \in [0, 1]$ be probability that $K \in A[]$
- If successful, let the $Pr\{K = A[1]\} = Pr\{K = A[i]\}\ \forall i \in [0, n-1]$
- So the $Pr\{K = A[i]\} = \frac{p}{n}$
- $C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1-p) \rightsquigarrow p \cdot \frac{n+1}{2} + (1-p) \cdot n$
- If $p = 1$ (i.e., $K \in A[]$), running time is linear

O, Ω, Θ: Sets of functions

Informally, we can think of O, Ω, Θ as *sets* of functions

- $O(g(n))$: set of all functions with the same or smaller order of growth as $g(n)$
  - $2n^2 - 5n + 1$? $O(n^2)$
  - $2^n + n^{100} - 2$? $O(n!)$
  - $2n + 6$? $O(\log n)$
- $\Omega(g(n))$: set of all functions with the same or larger order of growth as $g(n)$
  - $2n^2 - 5n + 1$? $\Omega(n^2)$
  - $2^n + n^{100} - 2$? $\Omega(n!)$
  - $2n + 6$? $\Omega(\log n)$
- $\Theta(g(n))$: set of all func. with the same order of growth as $g(n)$
  - $2n^2 - 5n + 1$? $\Theta(n^2)$
  - $2^n + n^100 - 2$? $\Theta(n!)$
  - $2n + 6$? $\Theta(\log n)$

O, Ω, Θ: Sets of functions

Informally, we can think of O, Ω, Θ as *sets* of functions

- $O(g(n))$: set of all functions with the same or smaller order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in O(n^2)$
  - $2^n + n^{100} - 2 \in O(n!)$
  - $2n + 6 \notin O(\log n)$
- $\Omega(g(n))$: set of all functions with the same or larger order of growth as $g(n)$
  - $2n^2 - 5n + 1$? $\Omega(n^2)$
  - $2^n + n^{100} - 2$? $\Omega(n!)$
  - $2n + 6$? $\Omega(\log n)$
- $\Theta(g(n))$: set of all func. with the same order of growth as $g(n)$
  - $2n^2 - 5n + 1$? $\Theta(n^2)$
  - $2^n + n^100 - 2$? $\Theta(n!)$
  - $2n + 6$? $\Theta(\log n)$

O, Ω, Θ: Sets of functions

Informally, we can think of O, Ω, Θ as *sets* of functions

- $O(g(n))$: set of all functions with the same or smaller order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in O(n^2)$
  - $2^n + n^{100} - 2 \in O(n!)$
  - $2n + 6 \notin O(\log n)$
- $\Omega(g(n))$: set of all functions with the same or larger order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in \Omega(n^2)$
  - $2^n + n^{100} - 2 \notin \Omega(n!)$
  - $2n + 6 \in \Omega(\log n)$
- $\Theta(g(n))$: set of all func. with the same order of growth as $g(n)$
  - $2n^2 - 5n + 1$? $\Theta(n^2)$
  - $2^n + n^100 - 2$? $\Theta(n!)$
  - $2n + 6$? $\Theta(\log n)$

O, Ω, Θ: Sets of functions

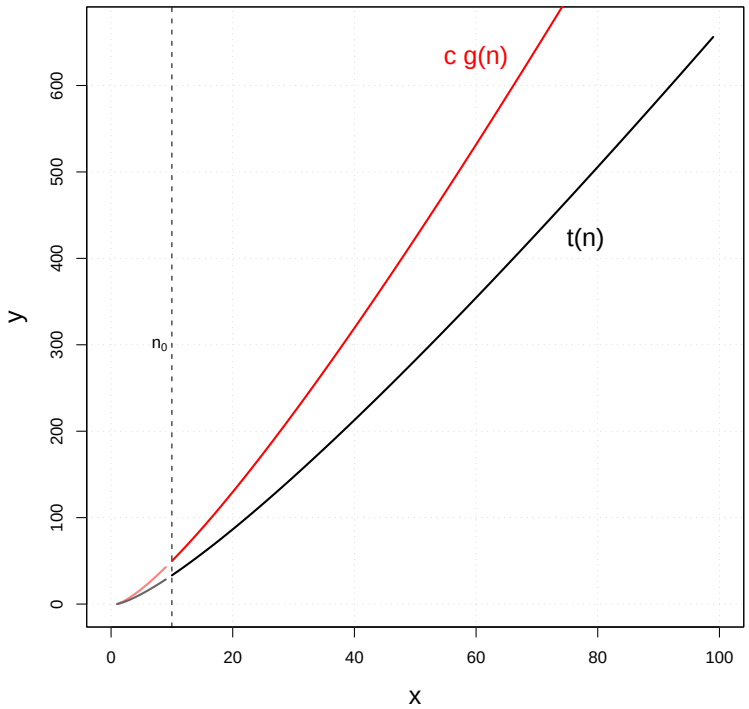
Informally, we can think of O, Ω, Θ as *sets* of functions

- $O(g(n))$: set of all functions with the same or smaller order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in O(n^2)$
  - $2^n + n^{100} - 2 \in O(n!)$
  - $2n + 6 \notin O(\log n)$
- $\Omega(g(n))$: set of all functions with the same or larger order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in \Omega(n^2)$
  - $2^n + n^{100} - 2 \notin \Omega(n!)$
  - $2n + 6 \in \Omega(\log n)$
- $\Theta(g(n))$: set of all func. with the same order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in \Theta(n^2)$
  - $2^n + n^100 - 2 \notin \Theta(n!)$
  - $2n + 6 \notin \Theta(\log n)$

O-Notation, more formally

Definition

$O(g(n)) = \{t(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq t(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$.

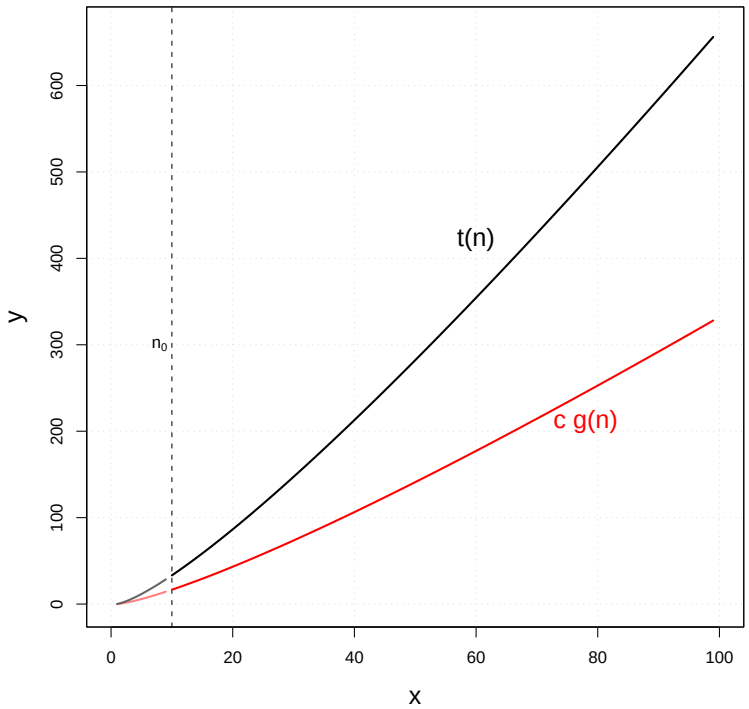


- I.e., $t(n)$ is in $O(g(n))$ if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all "large" n
- We call O an *asymptotic upper bound*

Ω-Notation, more formally

Definition

$\Omega(g(n)) = \{t(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq c \cdot g(n) \leq t(n)\ \forall n \geq n_0\}$.

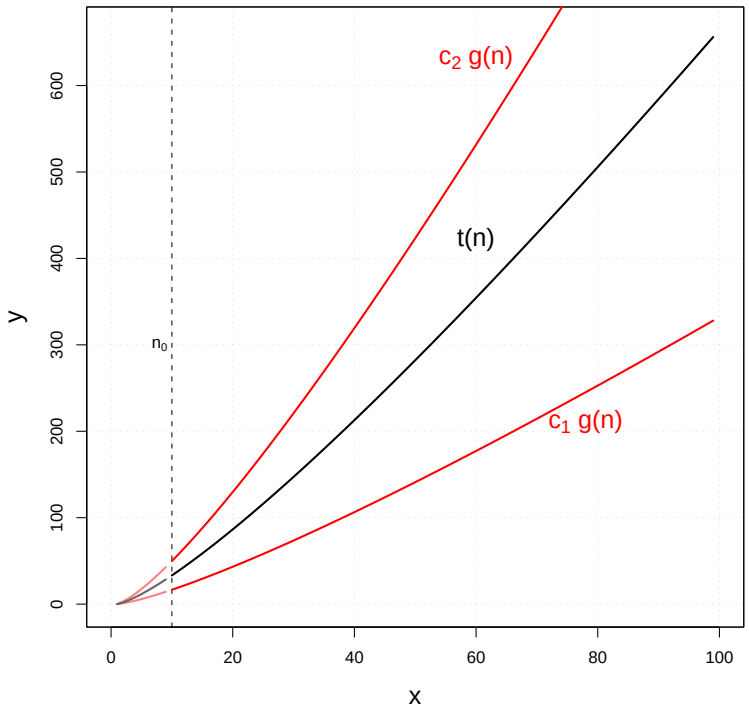


- I.e., $t(n)$ is in $O(g(n))$ if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all "large" n
- We call O an *asymptotic lower bound*

Θ-Notation, more formally

Definition

For any two functions $t(n)$ and $g(n)$ we have $t(n) \in \Theta(g(n))$ if and only if $t(n) \in O(g(n))$ and $t(n) \in \Omega(g(n))$.



- I.e., $t(n)$ is in $O(g(n))$ if $t(n)$ is bounded below by some constant multiple of $g(n)$, c_1, *and* bounded above by some constant multiple of $g(n)$, c_2, for all "large" n
- We sometimes call such a bound *tight*

A Useful Asymptotic Propery

Theorem

If $t_1(n) \in O(g_1(n))$ *and* $t_2(n) \in O(g_2(n))$ *then* $t_1(n) + t_2(n) \in O\left(\max\{g_1(n), g_2(n)\}\right)$.

- Review the proof in the book (p. 56)
- Advantage: We can restrict our analysis to one part of an algorithm if we know the other(s) to have a lower order of growth (e.g., CountingSort p. 24)
- Advantage: We can "front-load" an algorithm without increasing the time complexity as long as the preparation step has no greater an order of growth than main part (e.g., shuffling data before QuickSort)

Comparing Orders of Growth

One common way to compare two algorithms is by taking the limit of the ratios of their running times:

Comparing $t(n)$ and $f(n)$:

$$\lim_{n\to\infty} \frac{t(n)}{f(n)} = \begin{cases} 0 & \text{if } t(n) \text{ is of smaller order than } f(n) \\ c & \text{if } t(n) \text{ is of the same order as } f(n) \\ \infty & \text{if } t(n) \text{ is of greater order than } f(n) \end{cases}$$

Example: Compare the orders of growth of $\lg n$ and $\sqrt{n}$:

Comparing Orders of Growth

One common way to compare two algorithms is by taking the limit of the ratios of their running times:

Comparing $t(n)$ and $f(n)$:

$$\lim_{n\to\infty} \frac{t(n)}{f(n)} = \begin{cases} 0 & \text{if } t(n) \text{ is of smaller order than } f(n) \\ c & \text{if } t(n) \text{ is of the same order as } f(n) \\ \infty & \text{if } t(n) \text{ is of greater order than } f(n) \end{cases}$$

Example: Compare the orders of growth of $\lg n$ and $\sqrt{n}$:

$$\lim_{n\to\infty} \frac{\lg n}{\sqrt{n}} \qquad \text{(given)}$$

$$\lim_{n\to\infty} \frac{\frac{d}{dn}\lg n}{\frac{d}{dn}\sqrt{n}} \qquad \text{(L'Hopital's Rule)}$$

$$\lim_{n\to\infty} \frac{\lg e^{\frac{1}{n}}}{\frac{1}{2\sqrt{n}}} \qquad \text{(derivation)}$$

$$\lim_{n\to\infty} 2\lg e \frac{\frac{1}{n}}{\frac{1}{\sqrt{n}}} \qquad \text{(algebra)}$$

$$\lim_{n\to\infty} 2\lg e \frac{\sqrt{n}}{n} = 0$$

Comparing Orders of Growth

One common way to compare two algorithms is by taking the limit of the ratios of their running times:

Comparing $t(n)$ and $f(n)$:

$$\lim_{n\to\infty} \frac{t(n)}{f(n)} = \begin{cases} 0 & \text{if } t(n) \text{ is of smaller order than } f(n) \\ c & \text{if } t(n) \text{ is of the same order as } f(n) \\ \infty & \text{if } t(n) \text{ is of greater order than } f(n) \end{cases}$$

Example: Compare the orders of growth of $\lg n$ and $\sqrt{n}$:

$$\lim_{n\to\infty} \frac{\lg n}{\sqrt{n}} \quad \text{(given)}$$

$$\lim_{n\to\infty} \frac{\frac{d}{dn}\lg n}{\frac{d}{dn}\sqrt{n}} \quad \text{(L'Hopital's Rule)}$$

$$\lim_{n\to\infty} \frac{\lg e^{\frac{1}{n}}}{\frac{1}{2\sqrt{n}}} \quad \text{(derivation)}$$

$$\lim_{n\to\infty} 2\lg e \frac{\frac{1}{n}}{\frac{1}{\sqrt{n}}} \quad \text{(algebra)}$$

$$\lim_{n\to\infty} 2\lg e \frac{\sqrt{n}}{n} = 0$$

$\therefore \lg n$ is of smaller order than $\sqrt{n}$

General Analysis Plan

1. How will you measure input size? What algorithmic parameter is it?
2. What is the algorithm's basic operation? (Is it in the inner-most loop?)
3. Does the # of times the basic operation is executed depend on something other than the input size (e.g., ordering of the input)? If so, you have to consider worst, best, and average cases separately.
4. Specify a summation of the # times basic operation is executed
5. Find a closed-form solution if possible, determine the order of growth from the formula

Example: Matrix Multiplication

MatrixMultiply($A[0 \ldots n-1, 0 \ldots n-1], B[0$

```
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Example: Matrix Multiplication

MatrixMultiply$(A[0\ldots n-1, 0\ldots n-1], B[0$

```
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

1. **Input size measure:** n, the order of the matrices
2. **Basic operation:** Multiplication
3. **Dependency of op:** Depends only on input size
4. **Summation:** $M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$
5. **Find the order:** $M(n) \in O\left(n^3\right)$

Example: Binary Digits

Binary(n)

$count \leftarrow 1$
`while` $n > 1$ `do`
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor \frac{n}{2} \rfloor$
`return` $count$

Example: Binary Digits

BINARY(n)

```
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

1. **Input size measure:** n, the integer given
2. **Basic operation:** the $n > 1$ comparison
3. **Dependency of op:** Depends only on input size
4. **Summation:** Here it is actually a recurrence relation, but we know the answer: $B(n) = \lfloor \lg n \rfloor + 1$
5. **Find the order:** $B(n) \in O(\lg n)$

Example: Factorial

FACTORIAL(n)

```
if n = 0 return 1
else return Factorial((n − 1) · n)
```

Example: Factorial

FACTORIAL(n)

```
if n = 0 return 1
else return Factorial((n − 1) · n)
```

- n indicates input size, we count multiplications
- $F(n) = F(n-1) \cdot n$, for $n > 0$
- We call such equations *recurrence relations*
- This defines a sequence, but to make it *unique* we need an *initial condition* (i.e., `if n = 0 return 1`)
- Recurrence relations can be useful analyzing some nonrecursive algorithms, too (e.g., BINARY)

General Analysis Plan

1. How will you measure input size? What algorithmic parameter is it?
2. What is the algorithm's basic operation? (Is it in the inner-most loop?)
3. Does the # of times the basic operation is executed depend on something other than the input size (e.g., ordering of the input)? If so, you have to consider worst, best, and average cases separately.
4. Specify a recurrence relation for the # times basic operation is executed
5. Solve the recurrence relation, determine the order of growth of its solutions
   - Back substitution, induction
   - Recurrence trees
   - Masters Theorem

Solving Recurrences: Back Substitution & Induction

- Start with a large / difficult input, substitute it into the equation
- Continue for a few steps and note the pattern
- Use this intuition to posit a guess for the form of the solution
- Make use the of asymptotic definition to setup an inequality
- Use induction to show that solution works by showing that the inequality holds for all values of n.

Example: Binary (v2)

BINARY′ (n)

```
if n = 1 return 1
else return Binary′(⌊n/2⌋) + 1
```

Relation: $B(n) = B\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$

Example: Binary (v2)

BINARY′ (n)

```
if n = 1 return 1
else return Binary′(⌊n/2⌋) + 1
```

Relation: $B(n) = B\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$

$$
\begin{aligned}
B(2^k) &= B\left(2^{k-1}\right) + 1 \\
&= \left[B\left(2^{k-2}\right) + 1\right] + 1 = B(2^{k-2}) + 2 \\
&\cdots \\
&= B\left(2^{k-k}\right) + k = k
\end{aligned}
$$

Guess: $B(n) \in O(\lg n)$

Example: Binary (v2)

BINARY′ (n)

```
if n = 1 return 1
else return Binary′(⌊n/2⌋) + 1
```

Relation: $B(n) = B\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$

$$\begin{aligned} B(2^k) &= B\left(2^{k-1}\right) + 1 \\ &= \left[B\left(2^{k-2}\right) + 1\right] + 1 = B(2^{k-2}) + 2 \\ &\cdots \\ &= B\left(2^{k-k}\right) + k = k \end{aligned}$$

Guess: $B(n) \in O(\lg n)$

From the definition from O, we want to prove that $B(n) \leq c \lg n$

$$\begin{aligned} B(n) &\leq B(\lfloor n/2 \rfloor) + 1 \\ &\leq c \lg \lfloor n/2 \rfloor + 1 \\ &= c \lg n - c \lg 2 + 1 \\ &= c \lg n - K \leq c \lg n \end{aligned}$$

We also have to show that the base case holds.

General Analysis Plan

1. Choose an appropriate experimental design
2. Decide on the efficiency metric and the measurement unit (count vs. time)
3. How will input space be sampled? (range, size, etc.)
4. Implement the algorithm
5. Generate a sample of inputs
6. Run the algorithm on the sample inputs, record results
7. Analyze the results

What Are We Measuring?

- Counter

- Time

What Are We Measuring?

- Counter
  - Identify & count basic operations as program runs
  - Requires additions to the code
  - Sometimes you can use a *profiler* to help identify the basic operation
- Time

What Are We Measuring?

- Counter
  - Identify & count basic operations as program runs
  - Requires additions to the code
  - Sometimes you can use a *profiler* to help identify the basic operation
- Time
  - Measure the length of time it takes for the algorithm to complete the task
  - May be done in implementation or (sometimes) using OS commands
  - Times can be inaccurate
    - Small inputs may clock in below threshold of timer precision
    - On a multiprocessing OS, you will need to consider how much CPU time the user got, etc.
    - Times on different machines may differ

Obtaining Statistics

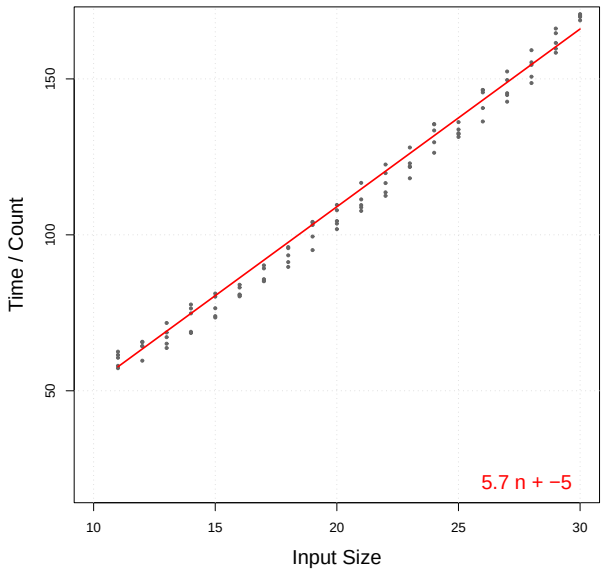
- Sample the input space
  - Counts & times can vary on different inputs of the same size
  - Times can vary in the *same* input
  - For some kinds of algorithms, counts can too (e.g., RandomizedQuickSort)
  - Choose an input range that makes sense for your purposes
- Report the results
  - Aggregate (average, median, etc.) results across runs for each input size (and possibly for the same input)
  - Tabulate or graph the data, typically as size versus time (p.89)
  - Fit a function to the data

Obtaining Statistics (2)

- Be careful what you conclude
  - Constants terms and factors can be misleading when input sizes are relatively small
  - When possible, use a regression method to be as statistically certain as possible
  - Remember: Empiricism is about *belief* (don't conclude too much)

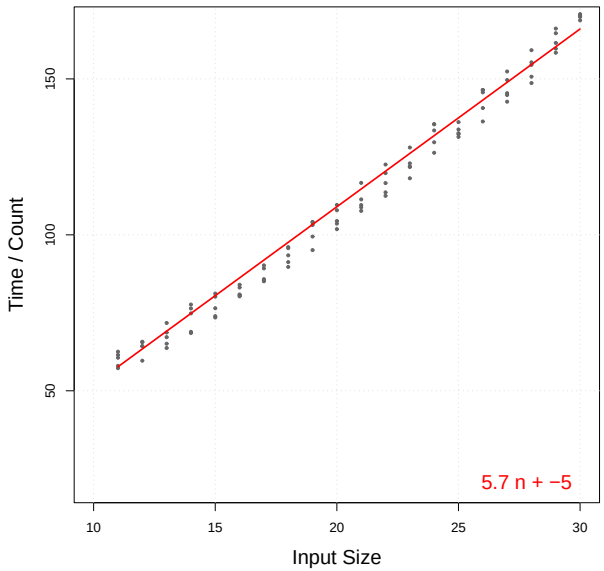
Obtaining Statistics (2)

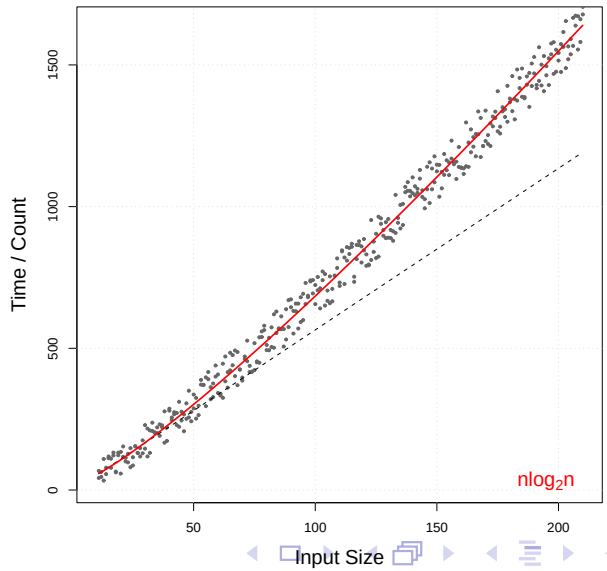
- Be careful what you conclude
  - Constants terms and factors can be misleading when input sizes are relatively small
  - When possible, use a regression method to be as statistically certain as possible
  - Remember: Empiricism is about *belief* (don't conclude too much)



Obtaining Statistics (2)

- Be careful what you conclude
  - Constants terms and factors can be misleading when input sizes are relatively small
  - When possible, use a regression method to be as statistically certain as possible
  - Remember: Empiricism is about *belief* (don't conclude too much)





Assignments

- This week's assignments:
  - Section 2.1: Problems 2, 6, and 9
  - Section 2.2: Problems 2, 5, and 6
  - Section 2.3: Problems 2 and 4
  - Section 2.4: Problems 1, 8, 9, and 10*
  - Section 2.5: Problem 6
  - Section 2.6: Problems 1, 6, and 8

*Challenge problem

Project I: Analysis of Simple Sorting

Write a paper comparing and contrasting BubbleSort, MergeSort, and ShellSort. Discuss the advantages and disadvantages of each in the context of efficiency of computational running-time with respect to input size. For BubbleSort, include a proof of correctness for the algorithm.

In addition, implement the three algorithms and conduct an empirical analysis. Describe how you chose to measure input size and running times, including your motivation for those choices. Be as clear and detailed as possible about all of these choices. For example, if you choose to count operations, explain which operation you will be counting and why. Additionally, the report should contain graphs and/or tables of input size versus running time and a discussion for how the data matches the formal running time.

Submit the project electronically by sending me an email message with attached gzipped tar file. The tarball should include:

1. The report in text, PS, PDF, HTML, or RTF format (do NOT send me a Word document);
2. Any graphs non-embedded graphics in non-proprietary formats such as PS, JPEG, PNG, etc. (do NOT send me an Excel spreadsheet);
3. Source code, but not binaries;
4. Notes for compiling the sources, if necessary;
5. Any relevant data files.

The project will be **due by midnight March 8**.