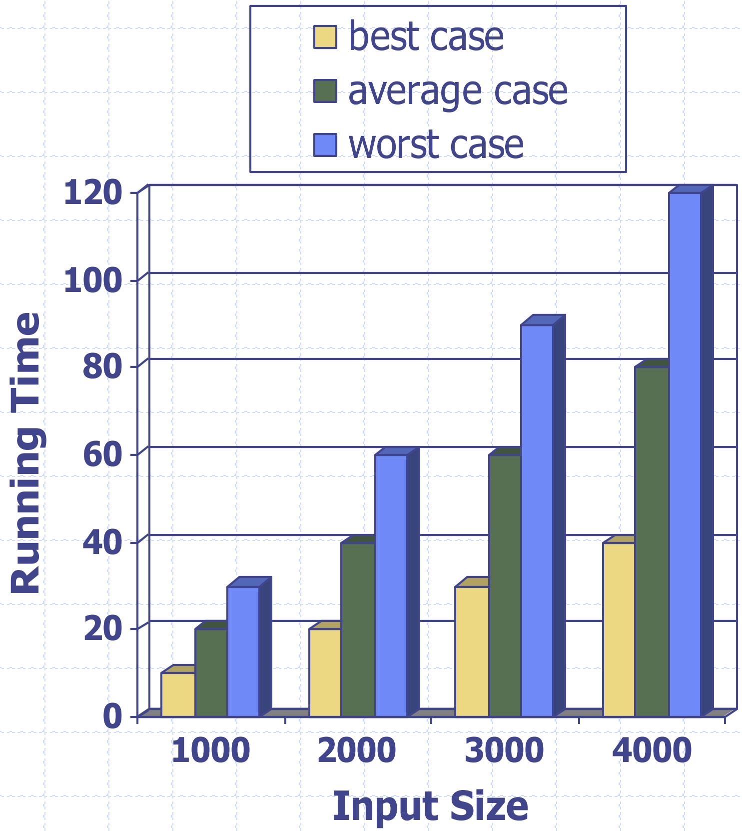
4CCS1DST – Data Structures

Lecture 3:

Analysis of Algorithms (Ch. 4)

Running Time

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



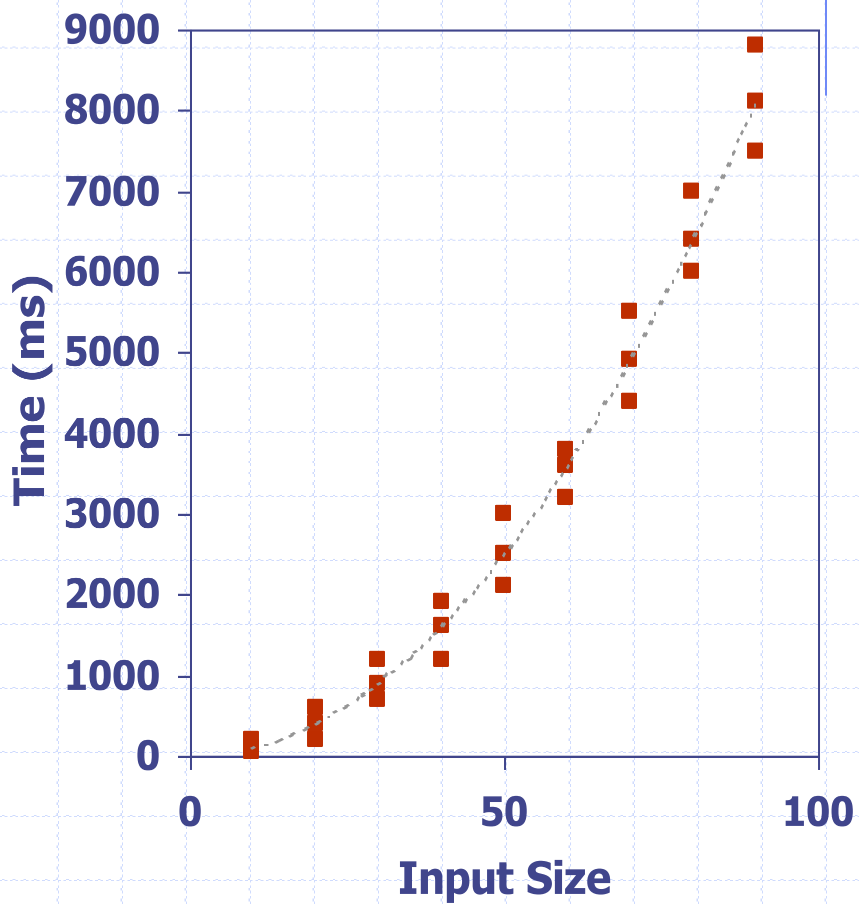
Experimental Studies

- Write a program implementing a given algorithm
- Run the program with inputs of varying sizes and composition
- Use a method like

  System.currentTimeMillis()

  to measure the actual running time

  (see Lecture 1, class FibonacciTest)
- Plot the results
- Do the same for other algorithm(s) and compare

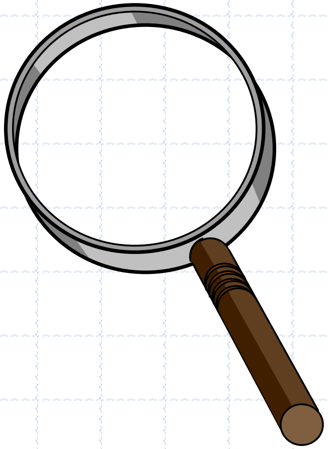


Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult (costly).
- We may have a number of potential algorithms for a given task, but we would like to implement only one – the best one.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

© 2010 Goodrich, Tamassia

Theoretical Analysis



- Uses a high-level description of the algorithm (pseudocode) instead of an implementation.
- Takes into account all possible inputs.
- Allows us to evaluate the speed of an algorithm independent of a hardware/software environment.
- Characterizes running time of an algorithm as a function of the input size, ***n*** :

> For a given algorithm, determine a function ***f*(*n*)** that characterizes the (worst-case) running time of this algorithm as a function of the input size ***n***.

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- 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

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - **Indentation replaces braces**
- Method declaration

  Algorithm *methodName* (*arg* [, *arg*…])

  Input …(explain the arguments)

  Output …(explained the return values)

- Method call

  methodN (*arg* [, *arg*…])

  var.methodN (*arg* [, *arg*…])

- Return value

  return *expression*

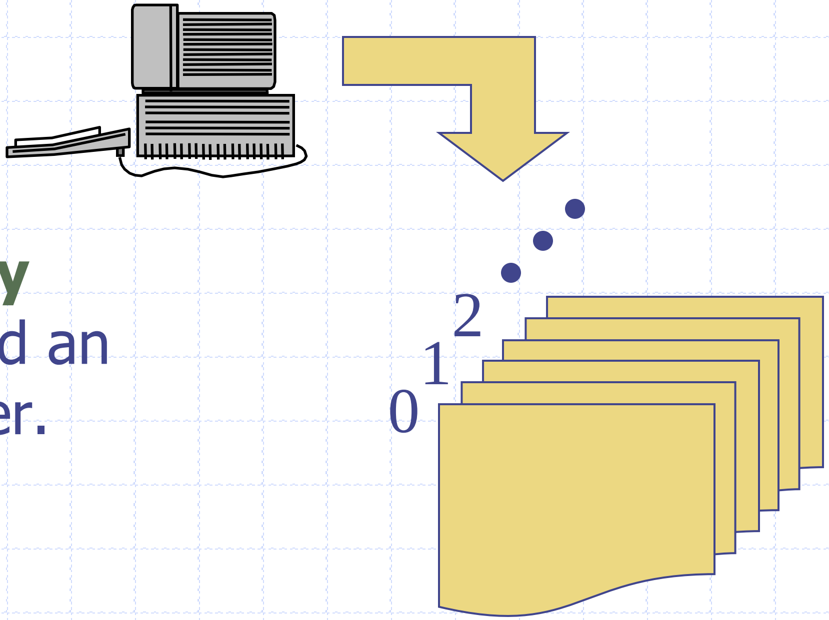
- Expressions

  $\leftarrow$ Assignment (like $=$ in Java)

  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model



- A **CPU**
- A **memory**: a potentially unbounded bank of **memory cells**, each of which can hold an arbitrary number or character.

  Memory cells are numbered.
- **Unit time**
  - Each CPU operation takes unit time.
  - Accessing any cell in memory takes unit time.
- Approximation of real computers but mostly sufficient for predicting real running times.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model (constant number of time units)

- Examples:
  - Assigning a value to a variable
  - Comparing two numbers
  - Evaluating an expression
  - Indexing into an array
  - Following an object reference
  - Calling a method
  - Returning from a method

Counting Primitive Operations

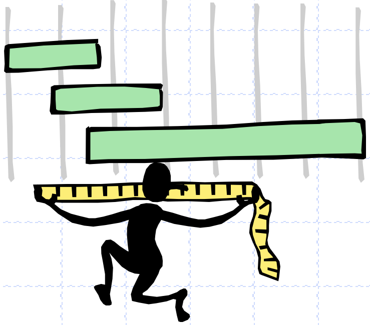
- The function $f(n)$, which characterizes the running time of a given algorithm: the maximum number of primitive operations executed by this algorithm, as a function of the input size.
- For a given n, $f(n)$ is the maximum number of primitive operations executed by this algorithm for any input od size n.
- We can determine this maximum number of primitive operations by inspecting the pseudocode of the algorithm.

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n + 1$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return $currentMax$	1
Total	$8n - 2$

array indexing + assignment

initialize i plus n x (subtract +compare)

Estimating Running Time



- Algorithm ***arrayMax*** executes $8\boldsymbol{n} - 2$ primitive operations in the worst case.
- Define:

  $\boldsymbol{a}$ = time taken by the fastest primitive operation

  $\boldsymbol{b}$ = time taken by the slowest primitive operation
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***arrayMax.*** Then
  $\underline{(8\boldsymbol{a})\,\boldsymbol{n} - 2\boldsymbol{a}} \;=\; \boldsymbol{a}\,(8\boldsymbol{n} - 2) \;\leq\; \boldsymbol{T}(\boldsymbol{n}) \;\leq\; \boldsymbol{b}\,(8\boldsymbol{n} - 2) \;=\; \underline{(8\boldsymbol{b})\,\boldsymbol{n} - 2\boldsymbol{b}}$
- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two <u>linear functions</u> of the argument $\boldsymbol{n}$.
- Input size $\boldsymbol{n} \to \boldsymbol{10\,n}$: running time $\boldsymbol{T}(\boldsymbol{10\,n}) \approx \boldsymbol{10} \cdot \boldsymbol{T}(\boldsymbol{n})$

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 $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax,***

  and is independent of hardware, implementation and computing environment.

Slide by Matt Stallmann included with permission.

Why Growth Rate Matters

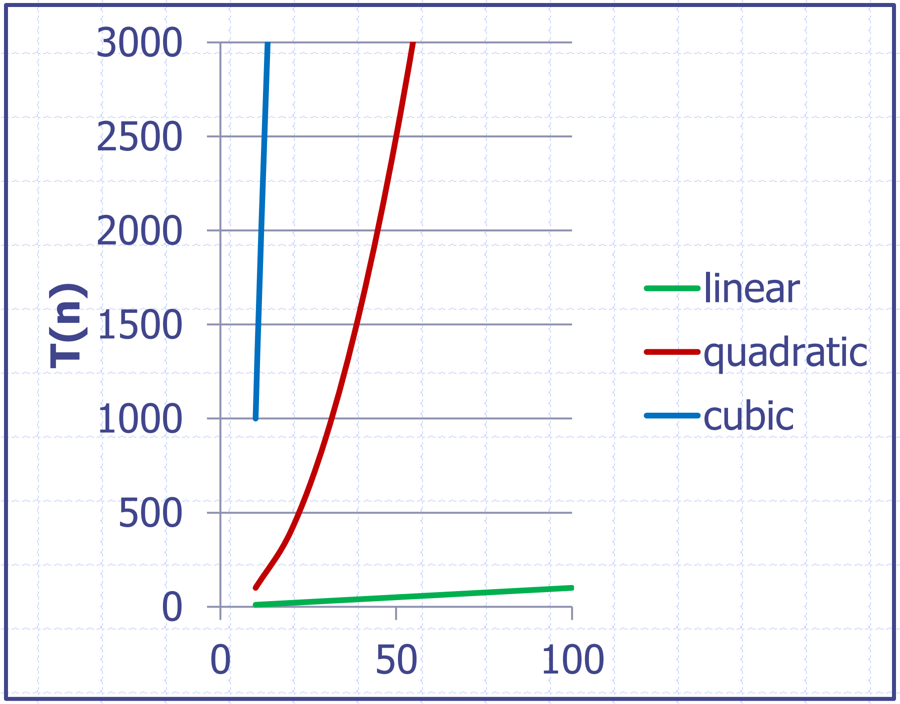
if runtime (time for n) is...	time for n + 1	time for 2 n	time for 4 n
$c \log_2 n$	$c \log_2 (n + 1)$	$(c \log_2 n) + c$	$(c \log_2 n) + 2c$
$c\ n$	$c(n+1) = c\ n + c$	**$2 * c\ n$**	$4\ c\ n$
$c\ n \log n$	$\sim c\ n \log n + c \log n$	$2c\ n \log n + 2cn$	$4c\ n \log n + 8cn$
$c\ n^2$	$\sim c\ n^2 + 2c\ n$	**$4 * c\ n^2$**	$16\ c\ n^2$
$c\ n^3$	$\sim c\ n^3 + 3c\ n^2$	$8\ c\ n^3$	$64\ c\ n^3$
$c\ 2^n$	$c\ 2^{n+1} = 2(c\ 2^n)$	$c2^{2n} = 2^n(c2^n)$	$c2^{4n} = 2^{3n}(c2^n)$

Linear runtime: doubles when problem size doubles

Quadratic runtime: quadruples when problem size doubles

Seven Important Functions

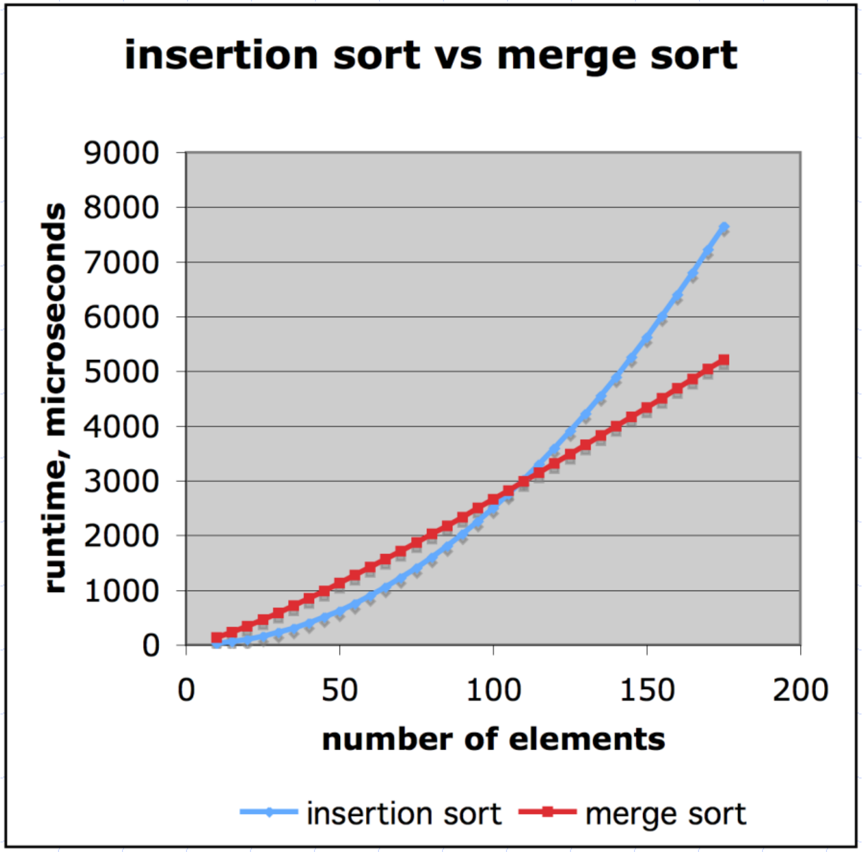
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic: $\approx \log n$
  - Linear: $\approx n$
  - N-Log-N: $\approx n \log n$
  - Quadratic: $\approx n^2$
  - Cubic $\approx n^3$
  - Exponential $\approx 2^n$
- Not easy to compare these functions using plots with "normal" (linear) scale.



n	linear	quadratic	cubic
1,000	1,000 µs's	1,000,000 ms's	1,000,000,000 sec's

Slide by Matt Stallmann included with permission.

Comparison of Two Algorithms



selection sort is similar

insertion sort is $n^2 / 4$

$f(2000) = 4 \times f(1000)$

merge sort is $5 \text{ n} \log \text{n}$

$f(2000) = 2.2 \times f(1000)$

sort one million items:

insertion sort takes roughly 70 hours

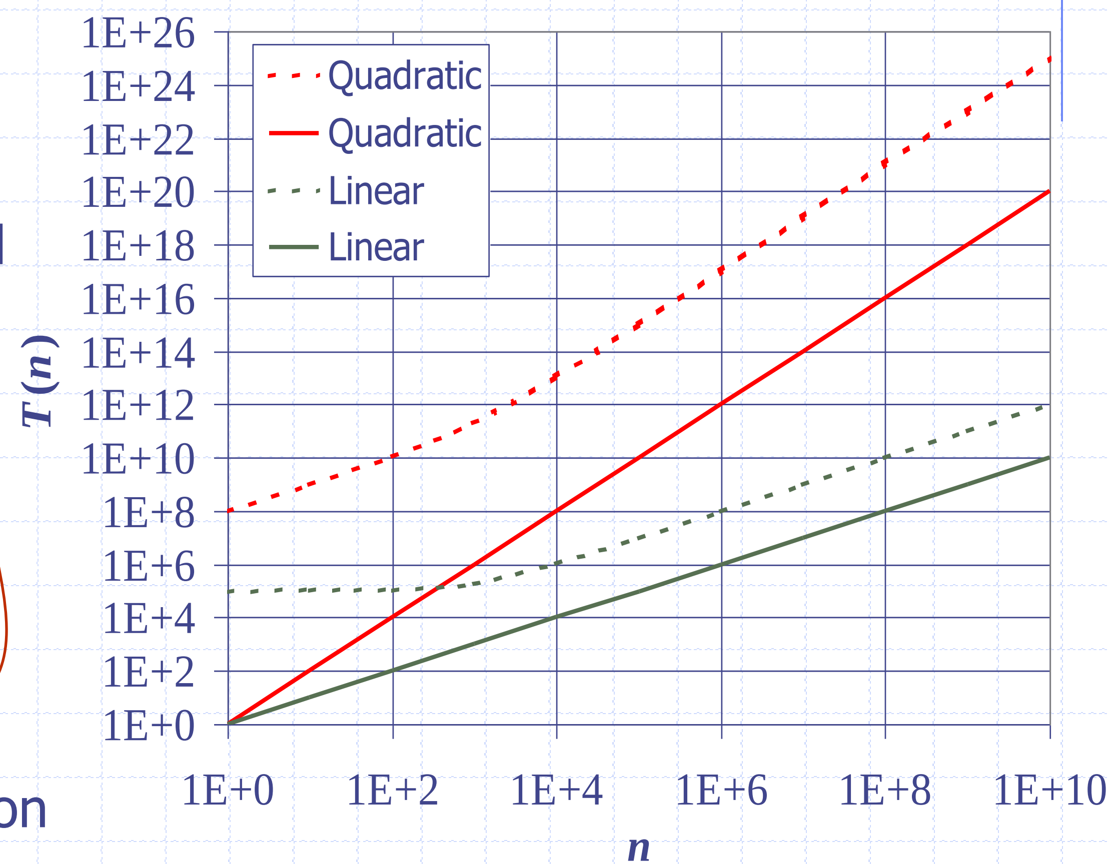
while

merge sort takes roughly 100 seconds

On a slow machine, but if 100 x faster machine, then it's 40 minutes versus 1 sec.

Constant Factors

- The **growth rate** is not affected by
  - constant factors and
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Big-Oh Notation

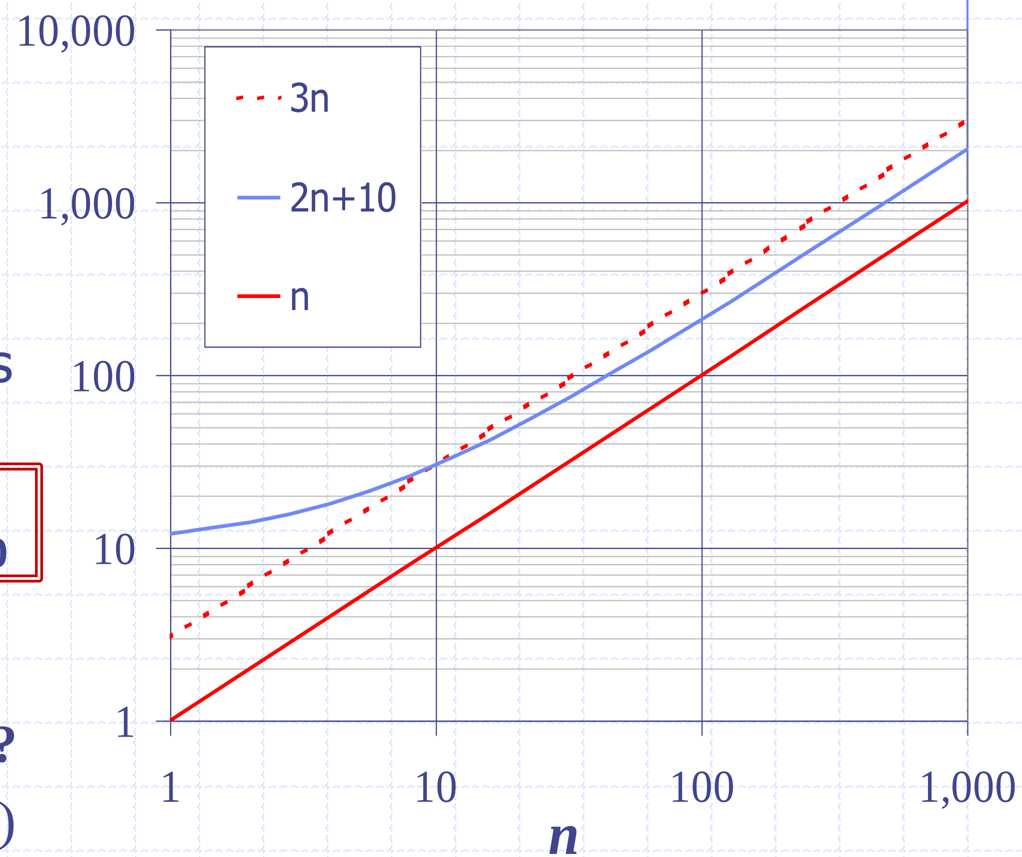
- Given functions $f(n)$ and $g(n)$, we say that

$$\boxed{f(n) \text{ is } O(g(n))}$$

  if there are positive constants c and n_0 such that

$$\boxed{f(n) \le cg(n) \text{ for } n \ge n_0}$$

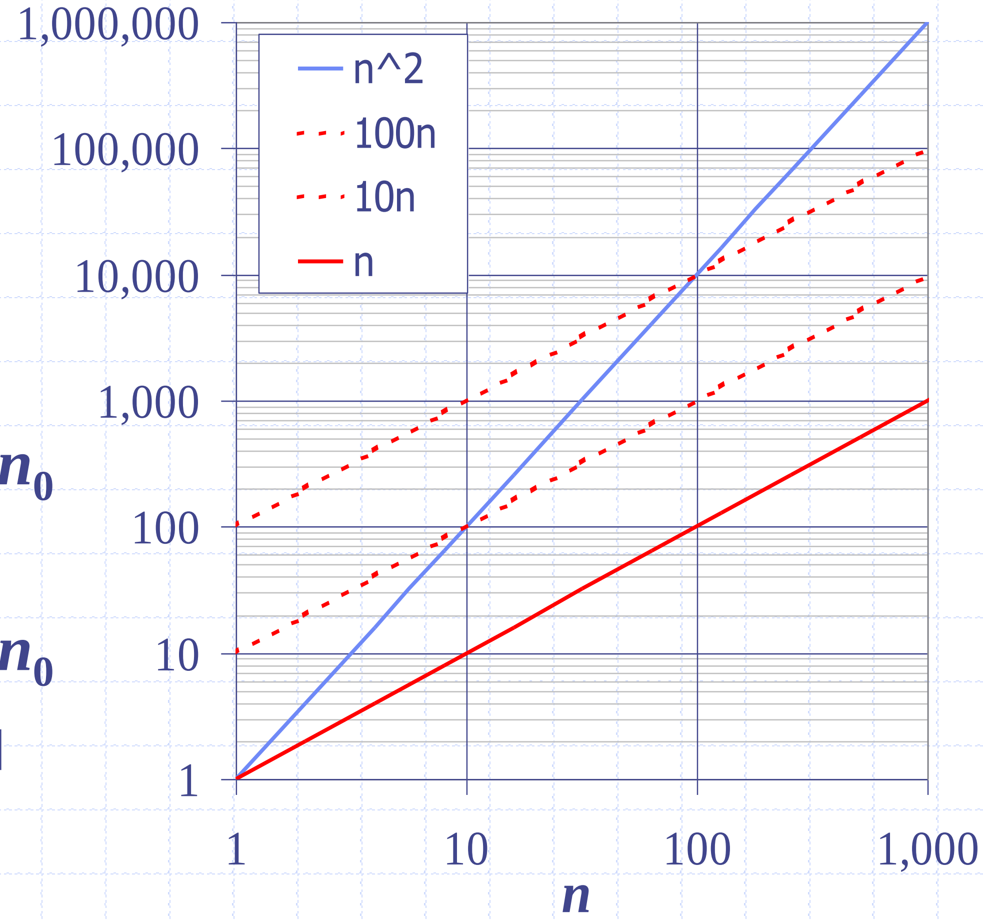
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$, for all $n \ge n_0$?
  - $(c - 2)\, n \ge 10$ (need $c > 2$)
  - $n \ge 10/(c - 2)$ for all $n \ge n_0$?
  - Yes, pick $c = 3$ and $n_0 = 10$:



$$\boxed{2n + 10 \le 3n, \text{ for all } n \ge 10}$$

Big-Oh Example

- Example:
  the function n^2
  is not $O(n)$
  - $n^2 \leq cn$?
    for some c and all $n \geq n_0$
  - $n \leq c$?
    for some c and all $n \geq n_0$
  - This 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 $7n-2 \leq cn$, 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 $3n^3 + 20n^2 + 5 \leq cn^3$, for $n \geq n_0$

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

- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$, for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

In all examples, other values of c and n_0 could also work!

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement $\boxed{f(n) \text{ is } O(g(n))}$ means that the growth rate of $f(n)$ is not greater 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.,
  - Drop lower-order terms
  - Drop the constant factor (change to 1) in the highest order term
  - Example: $5n^4 + 20n^3 - 7n + 13$ is $O(n^4)$
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"
- We also write, e.g., "$3n + 5 = O(n)$"

Asymptotic Algorithm Analysis

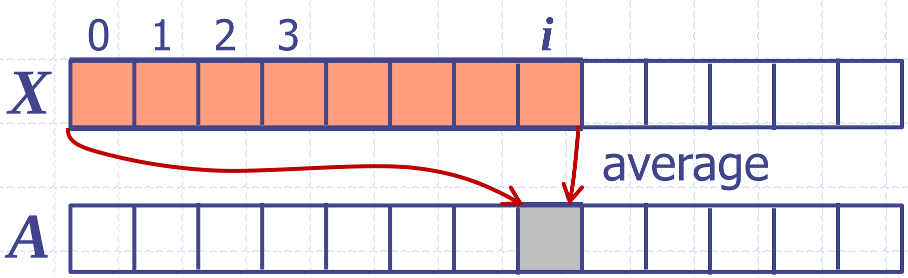
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation (or its "relatives").
- To perform the asymptotic analysis:
  - We find the worst-case number of primitive operations executed as a function of the input size.
  - We don't need this function exactly, we only want to express it using big-Oh notation.
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $8n - 2$ primitive operations.
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time."
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.

© 2010 Goodrich, Tamassia

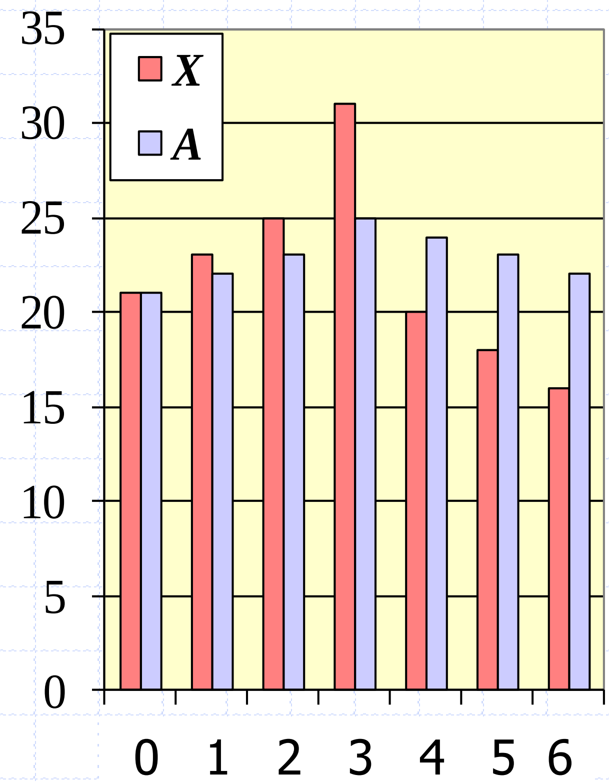
Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages.
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$



- Computing the array A of prefix averages of another array X has applications in financial analysis.



Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by directly applying the definition

	#operations
Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	$n + 1$
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (i+1) + \ldots + n$
$s \leftarrow s + X[j]$	$0 + 1 + \ldots + i + \ldots + (n-1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Arithmetic Progression

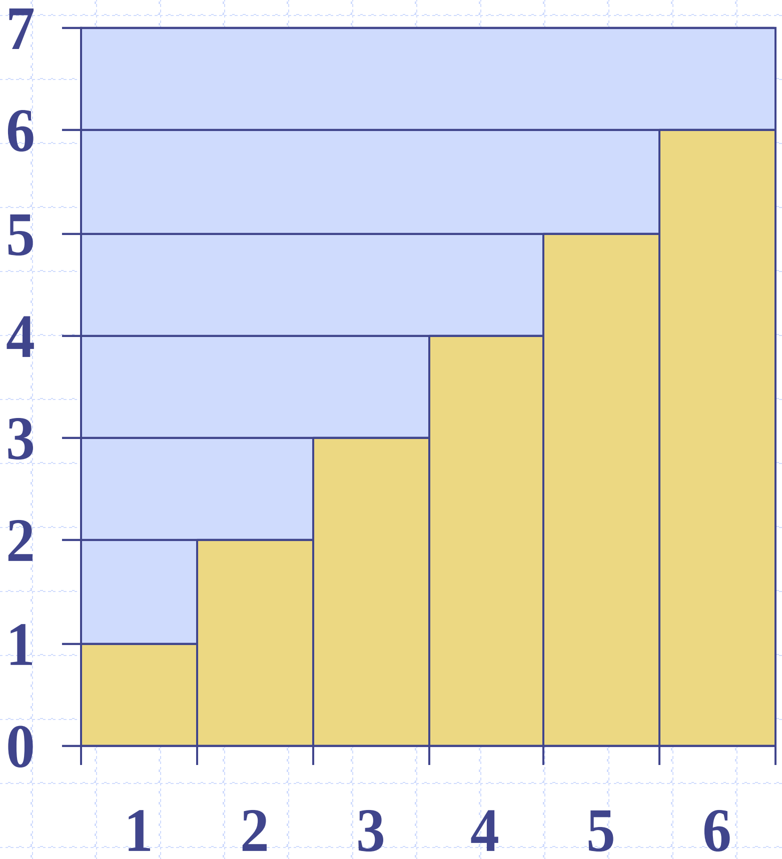
- The running time of ***prefixAverages1*** is

  $2\cdot(1 + 2 + \ldots + \boldsymbol{n}) + 3\boldsymbol{n} + 2$

  $= \boldsymbol{O}(1 + 2 + \ldots + \boldsymbol{n})$

- The sum of the first $\boldsymbol{n}$ integers is $\boldsymbol{n}(\boldsymbol{n}+1)/2$
  - There is a simple visual proof of this fact
- Thus algorithm ***prefixAverages1*** runs in

  $\boldsymbol{O}(\boldsymbol{n}(\boldsymbol{n}+1)/2) = \boxed{\boldsymbol{O}(\boldsymbol{n}^2)\ \text{time}}$



Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

Algorithm *prefixAverages2*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
$A \leftarrow$ new array of n integers	n
$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	$n + 1$
$s \leftarrow s + X[i]$	n
$A[i] \leftarrow s / (i + 1)$	n
return A	1

- Algorithm *prefixAverages2* runs in $O(n)$ time.

© 2010 Goodrich, Tamassia

Big-Oh and Relatives

Big-Oh

f(n) is O(g(n)) if, asymptotically,
f(n) is less than or equal to g(n)

Big-Omega

f(n) is Ω(g(n)) if, asymptotically,
f(n) is greater than or equal to g(n)

Big-Theta

f(n) is Θ(g(n)) if, asymptotically,
f(n) is equal to g(n)

f(n) is *O*(g(n))

and

f(n) is Ω(g(n))

Relatives of Big-Oh

- Big-Omega

$$\boxed{f(n) \text{ is } \Omega(g(n))}$$

if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$\boxed{f(n) \geq c \cdot g(n) \text{ for } n \geq n_0}$$

- Big-Theta

$$\boxed{f(n) \text{ is } \Theta(g(n))}$$

if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$\boxed{c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \text{ for } n \geq n_0} \Leftrightarrow \boxed{\begin{array}{c} f(n) \text{ is } O(g(n)) \\ \text{and} \\ f(n) \text{ is } \Omega(g(n)) \end{array}}$$

Example Uses of the Relatives of Big-Oh

- **$5n^2 + 10$ is $\Omega(n^2)$**

  $f(n)$ 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 all $n \geq n_0$.

  Let $c = 1$ and $n_0 = 1$

- **$5n^2 + 10$ is $\Omega(n)$**

  $f(n)$ 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 all $n \geq n_0$.

  Let $c = 1$ and $n_0 = 1$

- **$5n^2 + 10$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$, if it is $\Omega(g(n))$ and $O(g(n))$.

  We already know that $\mathbf{5n^2 + 10}$ is $\Omega(\mathbf{n^2})$. Show that $\mathbf{5n^2 + 10}$ is $O(\mathbf{n^2})$.

  $f(n)$ 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 all $n \geq n_0$.

  Let $c = 6$ and $n_0 = 4$

Exercises

Exercise 1: asymptotic notation

$10\ n$ is:	$O(n)$	$O(n^2)$	$O(\log n)$	$\Theta(n^2)$	$\Theta(n)$		
$n/2 + 5 \log n$ is:	$O(n)$	$O(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$		
$7n^3 - 9n^2$ is:	$O(n)$	$O(n^2)$	$\Omega(n^2)$	$O(\log n)$	$\Theta(n^2)$	$\Theta(n^3)$	

Circle all correct answers.

Exercise 2

The following Java method determines whether the elements in a given range of array arr are all unique.

```
public static boolean isUniqueLoop(int[] arr, int start, int end) {
    for ( int i = start; i < end; i++ )
        for ( int j = i+1; j <= end; j++ )
            if ( arr[i] == arr[j] )
                return false;     // the same  element at locations i and j
    // all elements are unique
    return true;
}
```

What is the worst-case running time of this method, in terms of the number n of elements under consideration (n = end − start + 1) ?

Is there a better (faster) way to find out if all elements are unique?

Exercise 3

The following Java method determines whether three sets of integers, given in arrays a, b and c, have a common element.

```
public static boolean haveSameElement(int[] a, int[] b, int[] c) {
    for ( int i=0; i < a.length; i++ )
        for ( int j=0; j < b.length; j++ )
            for ( int k=0; k < c.length; k++ )
                if ( (a[i] == b[j]) && (b[j] == c[k]) )
                    return true;      // a common element found
    // no common element
    return false;
}
```

What is the worst-case running time of this method, if each array is of size *n* ?

Is there a better (faster) way to find out if the arrays have a common element?

Exercise 4

Design the following algorithm and implement it as a Java method

Algorithm ***countOnes***(***A***, ***n***)

Input two-dimensional ***n*** x ***n*** binary array ***A*** (each entry is either 0 or 1)

Output two-dimensional ***n*** x ***n*** array ***S***, where ***S***[***i***][***j***] is the number of 1's in the "subarray" with the top-left corner at (0,0) and the bottom-right corner at (***i***, ***j***).

Example. Input:

	0			***j***	
0	1	0	1	1	0
	0	1	1	1	0
i	1	0	1	0	1
	1	1	0	0	1
	0	0	1	1	0

Output:

				j	
	1	1	2	3	3
	1	2	4	6	6
i	2	3	6	8	9
	3	5	8	10	12
	3	5	9	12	14

What is the running time of your algorithm in terms of n?
Try to obtain the running time as low as you can.
The target running time is $\Theta(n^2)$.