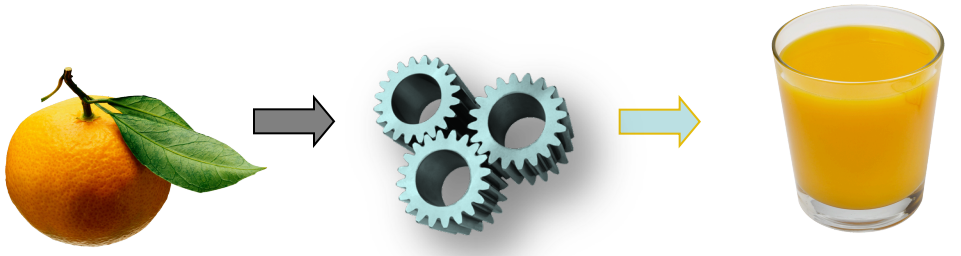
Analysis of Algorithms

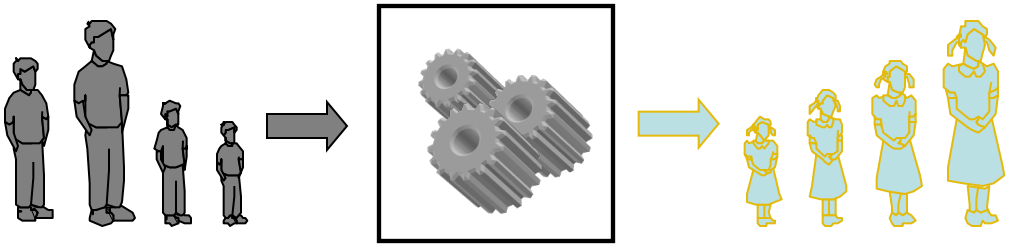


Slides adapted from Goodrich, Tamassia, 2005, Goodrich, Tamassia, Goldwasser, 2014 and CSI2110 Uottawa Team.

Analysis of algorithms

1. Intro to analysis of algorithms
2. Asymptotic analysis: Growth rate and Big-Oh notation
3. Comparing two algorithms for the same problem.
4. Asymptotic analysis: Big-Omega and Big-Theta; math review.

Intro to Analysis of Algorithms



Input **Algorithm** **Output**

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

Analyze an algorithm = determine its efficiency

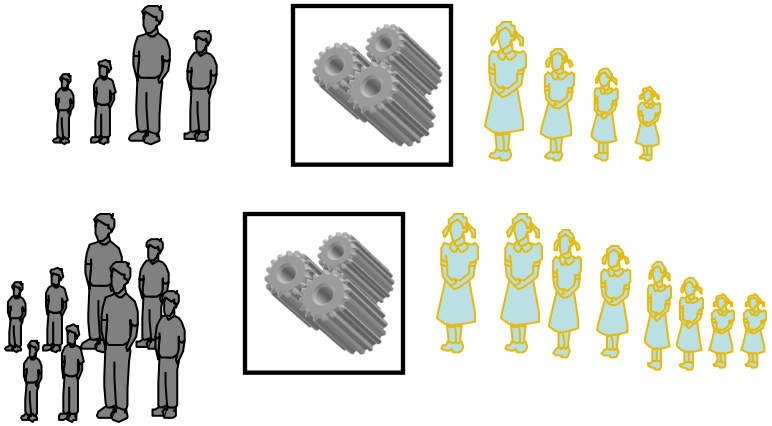
Efficiency ?

Running time

Memory

Running Time

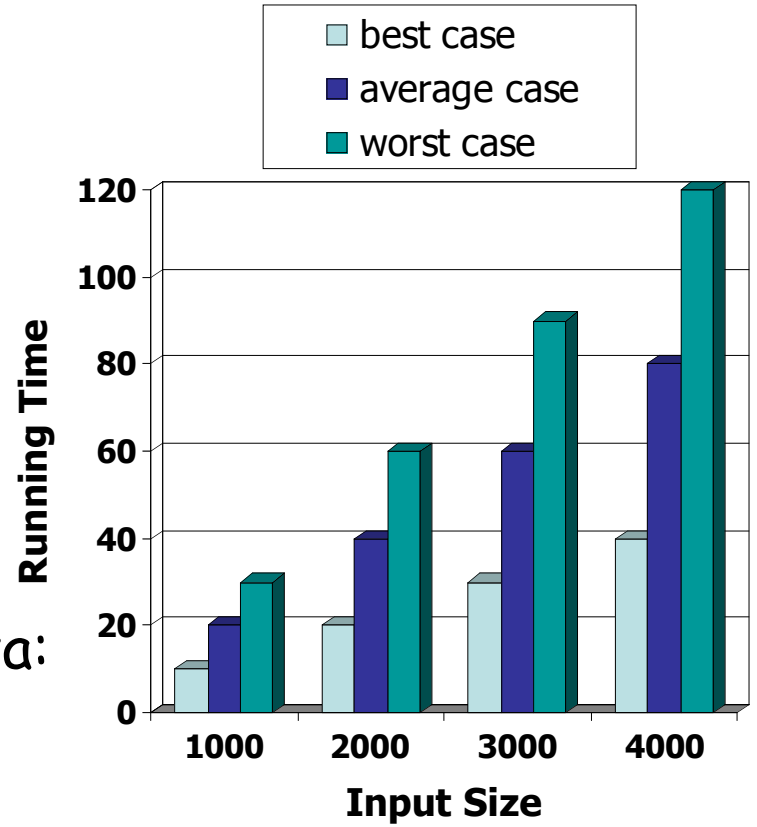
The running time of an algorithm typically grows with the input size.



But it also depends on the input data:
different inputs of the same size
can have different running times

Running Time

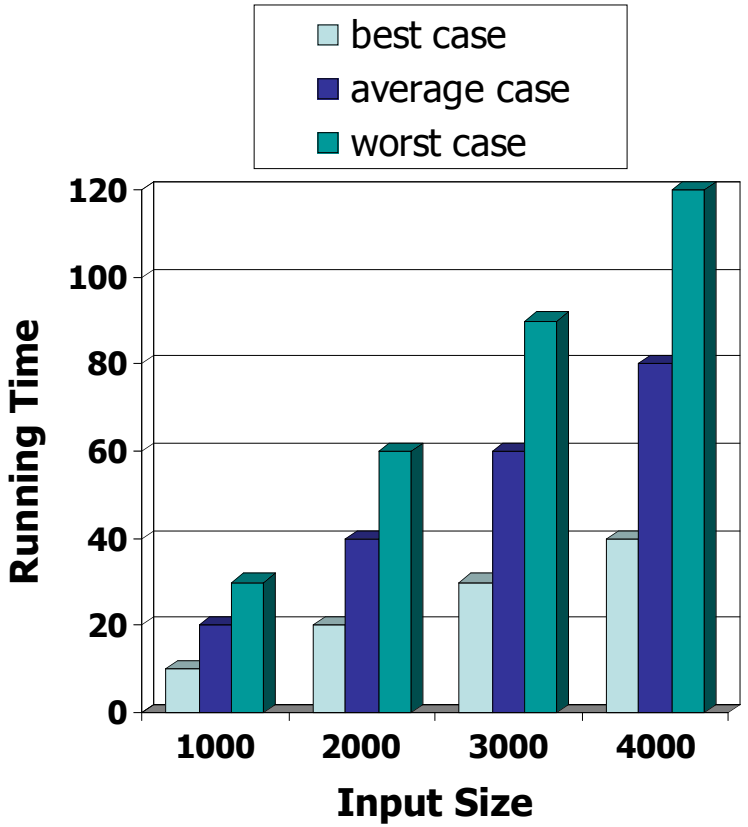
The running time of an algorithm typically grows with the input size.



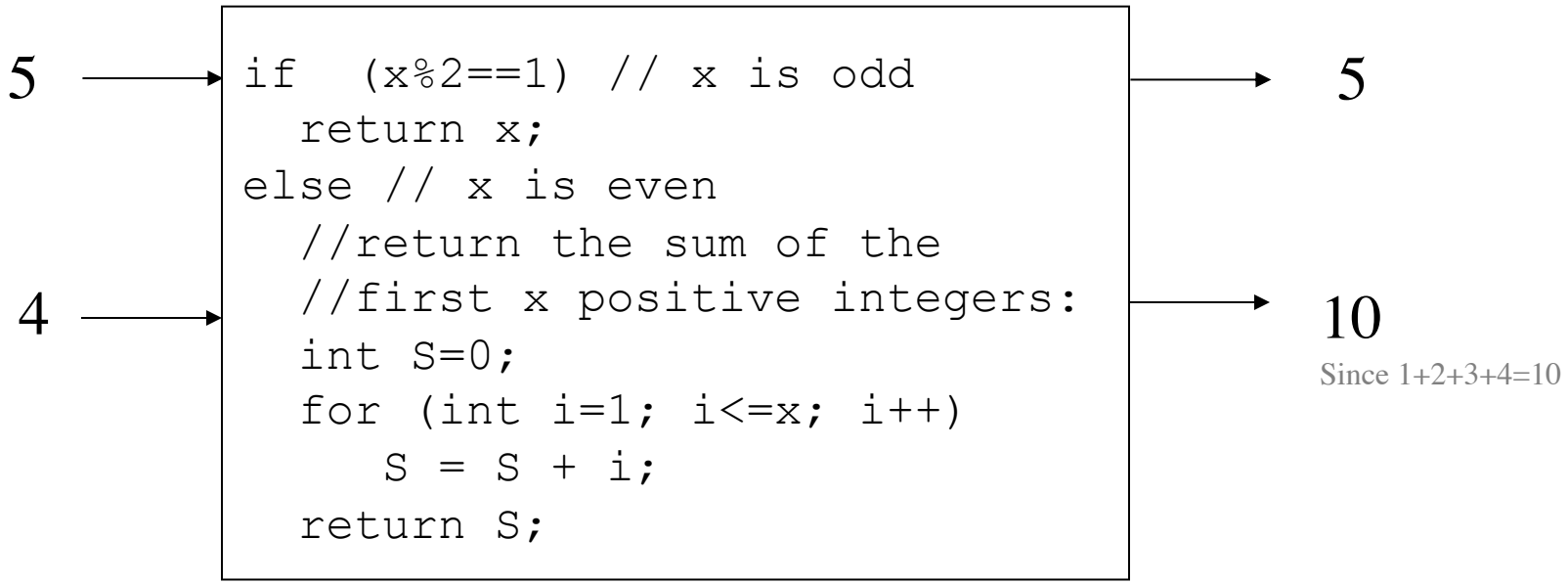
But it also depends on the input data: different inputs of the same size can have different running times

Running Time of an algorithm

- 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



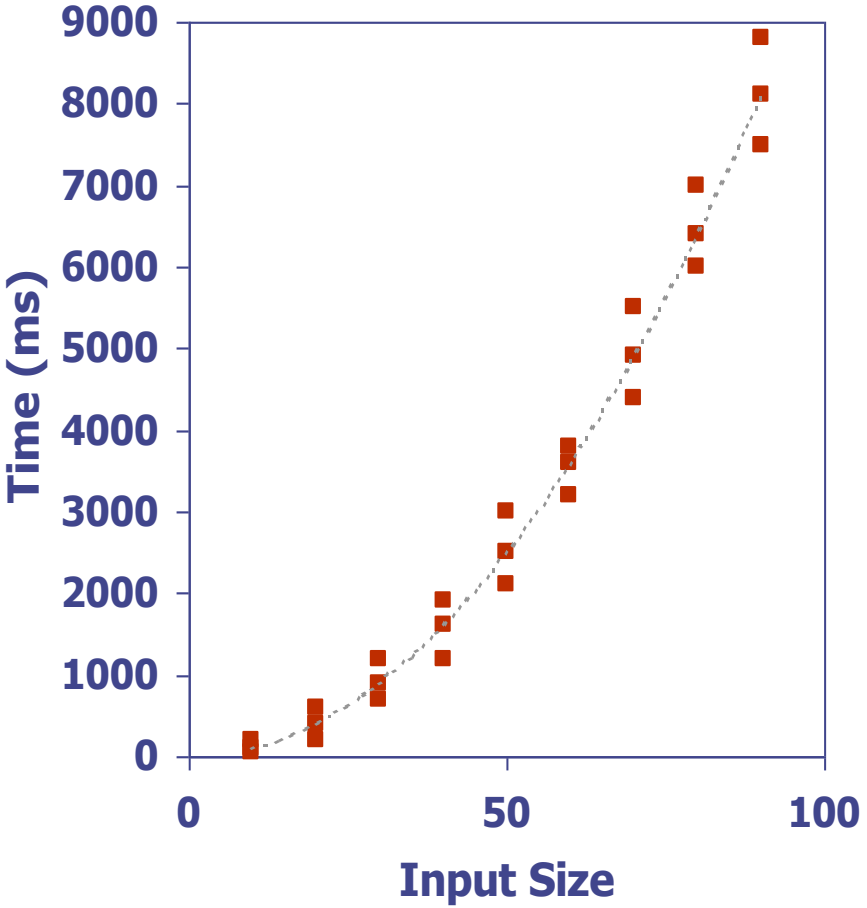
Example



If the input are different integers with the same size (number of bits), the running will be very different depend if the input is odd or even! This is because for odd x, we execute just a couple of operations, while for even x, we run a loop x times.

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
- Plot the results

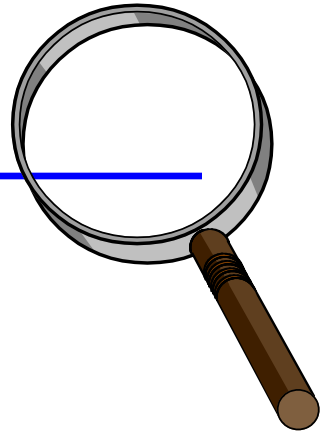



```
1  long startTime = System.currentTimeMillis( );        // record the starting time
2  /* (run the algorithm) */
3  long endTime = System.currentTimeMillis( );          // record the ending time
4  long elapsed = endTime − startTime;                  // compute the elapsed time
```

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- 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

Theoretical Analysis



It is a general methodology that:

- Uses a high-level description of the algorithm (pseudo code) independent of implementation.
- Characterizes running time as a function of the input size.
- Takes into account all possible inputs.
- Is independent of the hardware and software environment.

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

Pseudocode Details

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

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

  Input …

  Output …
- Method call

  method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions:

  ← Assignment

  = Equality testing

  n^2 Superscripts and other mathematical formatting allowed

From Code do Pseudocode

Java code:

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;
4    double currentMax = data[0];          // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)             // consider all other entries
6      if (data[j] > currentMax)           // if data[j] is biggest thus far...
7        currentMax = data[j];             // record it as the current max
8    return currentMax;
9  }
```

Pseudocode:

```
Algorithm arrayMax(data, n):
        Input: An array data storing n integers.
        Output: The maximum element in A.
currentMax ← data[0]
for i ← 1 to n -1 do
  if data[i] > currentMax then
      currentMax ← data[i]
return currentMax
```

From Code do Pseudocode

Python code:

```
def array_max(data)
        current_max= data[0]
        for val in data:
                if val > current_max
                        current_max = val
        return current_max
```

Pseudocode:

```
Algorithm arrayMax(data, n):
        Input: An array data storing n integers.
        Output: The maximum element in A.
currentMax ← data[0]
for i ← 1 to n -1 do
  if data[i] > currentMax then
      currentMax ← data[i]
return currentMax
```

From Code do Pseudocode

C code:

```
int arrayMax(int data[], int n) {
        int currentMax= data[0];
        for (int i=1; i<n; i++)
                if (data[i] > currentMax)
                        currentMax = data[i];
        return currentMax;
}
```

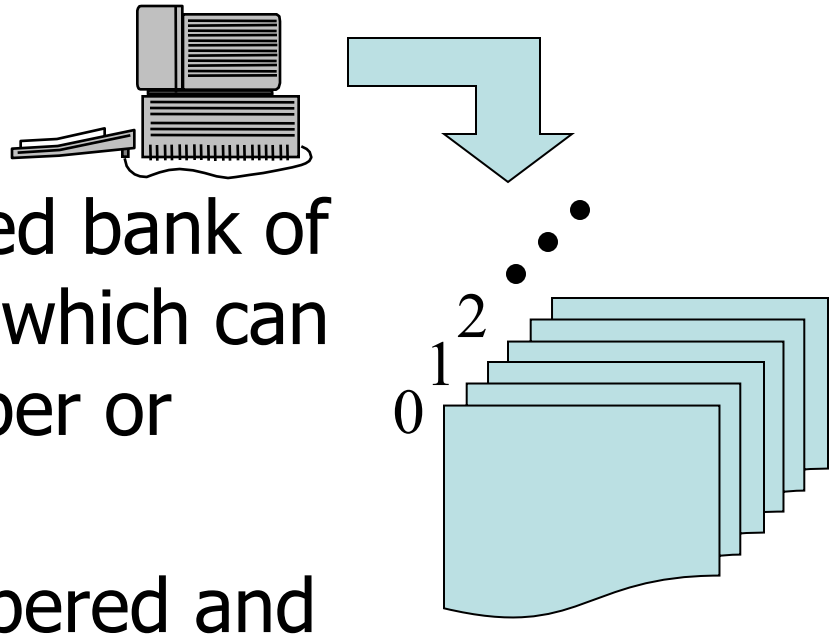
Pseudocode:

```
Algorithm arrayMax(data, n):
        Input: An array data storing n integers.
        Output: The maximum element in A.
currentMax ← data[0]
for i ← 1 to n -1 do
  if data[i] > currentMax then
      currentMax ← data[i]
return currentMax
```

The Random Access Machine (RAM) Model

A RAM consists of

- A CPU
- A potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time



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

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- 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 arrayMax(A, n):
        Input: An array A storing n integers.
        Output: The maximum element in A.
1   currentMax ← A[0]
2   for i ← 1 to n -1 do
3       if currentMax < A[i] then
4           currentMax ← A[i]
5    return currentMax
```

Step 1: 2 ops

Step 2: 2n ops

Step 3: 2(n-1) ops

Step 4: 0 to 2(n-1) ops

Step 5: 1 ops

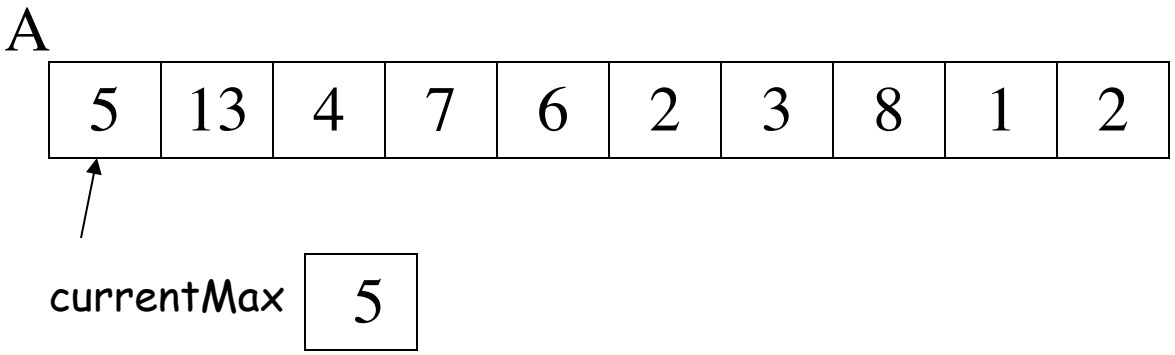
A

5	13	4	7	6	2	3	8	1	2

currentMax

Assignments? Comparisons?

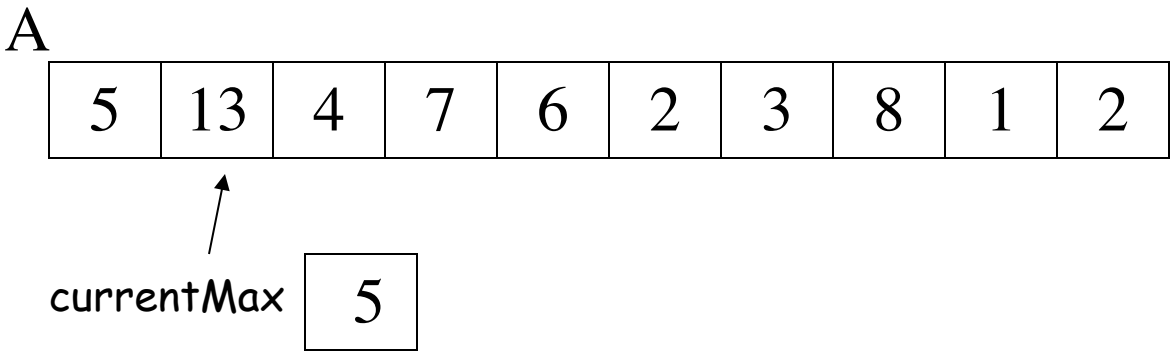
```
currentMax ← A[0]
for i ← 1 to n -1 do
  if currentMax < A[i] then
        currentMax ← A[i]

return currentMax
```



Assignments?

Comparisons?

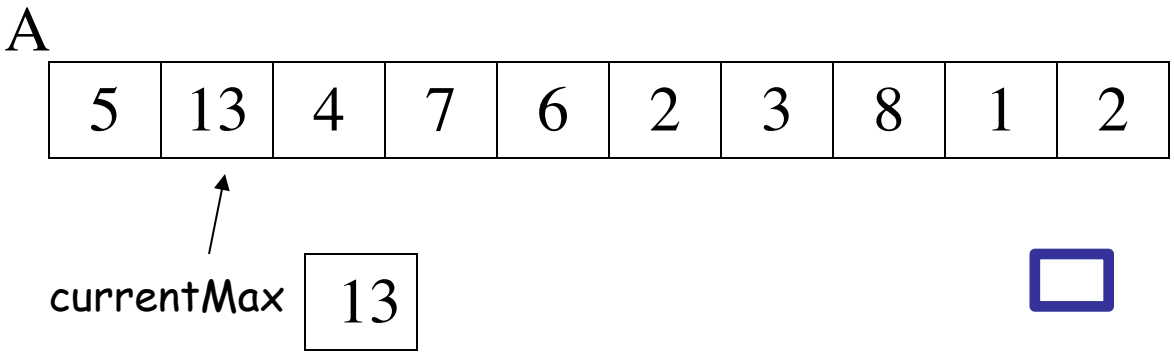
```
currentMax ← A[0]
for i ← 1 to n -1 do
    if currentMax < A[i] then
            currentMax ← A[i]

return currentMax
```



Assignments?

Comparisons?

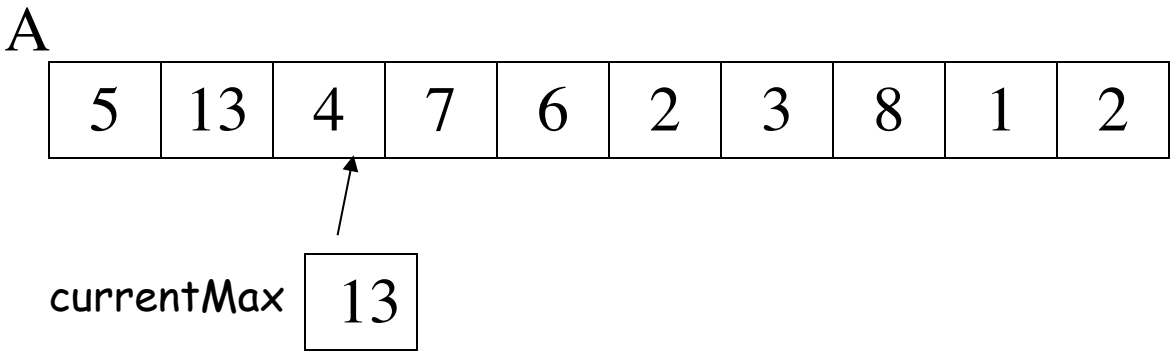
```
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax < A[i] then
            currentMax ← A[i]

return currentMax
```



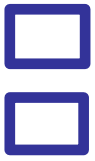
Assignments?

Comparisons?

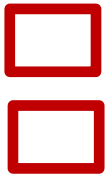
```
currentMax ← A[0]
for i ← 1 to n-1 do
  if currentMax < A[i] then
        currentMax ← A[i]

return currentMax
```



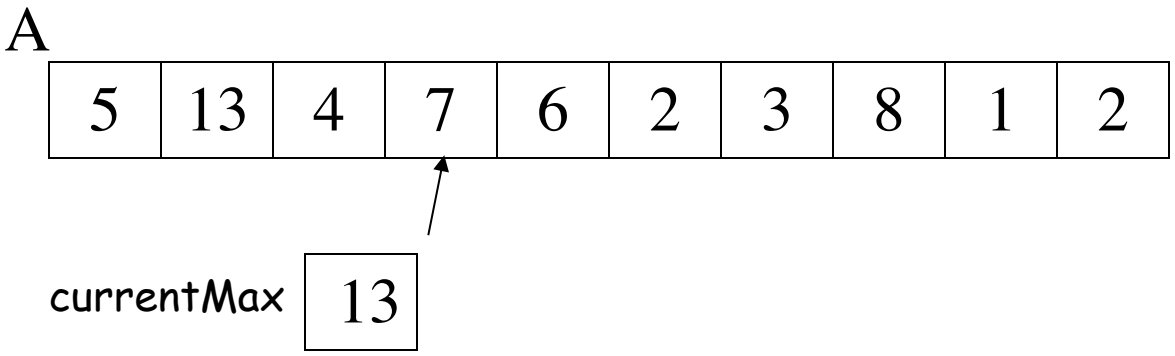
Assignments?



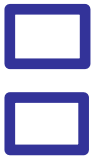
Comparisons?



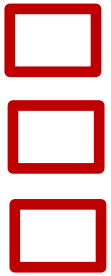
```
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax < A[i] then
            currentMax ← A[i]

return currentMax
```



Assignments?



Comparisons?



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


return currentMax
```

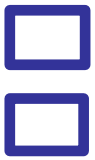
A

5	13	4	7	6	2	3	8	1	2

....

currentMax | 13

Assignments?



Comparisons?

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

return currentMax
```

A

5	13	4	7	6	2	3	8	1	2

....

currentMax | 13

Assignments? 1, 2, …, n

Comparisons? n-1

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

return currentMax
```

WORST CASE

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

return currentMax
```

1 assignment

n-1 comparisons
n-1 assignments

5	7	8	10	11	12	14	16	17	20

BEST CASE

15	1	12	3	9	7	6	4	2	2

```
currentMax ← A[0]                    ----------> 1 assignment
for i ← 1 to n-1 do
  if currentMax < A[i] then
        currentMax ← A[i]            ---> n-1 comparaisons
                                          0  assignments

return currentMax
```

Complexity - Summary

Best Case running time:
n-1 comparisons
1 assignment

Worst Case running time:
n-1 comparisons
n assignments

Counting Primitive Operations

- 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 arrayMax(A, n):
        Input: An array A storing n integers.
        Output: The maximum element in A.
1   currentMax ← A[0]
2   for i ← 1 to n -1 do
3       if currentMax < A[i] then
4           currentMax ← A[i]
5    return currentMax
```

Step 1: 2 ops

Step 2: 2n ops

Step 3: 2(n-1) ops

Step 4: 0 to 2(n-1) ops

Step 5: 1 ops

Complexity - Summary

Best Case running time:
n-1 comparisons
1 assignment

Total:
4n+1 primitive operations

Worst Case running time:
n-1 comparisons
n assignments

Total:
6n-1 primitive operations

Summary so Far

- We use theoretical analysis of algorithm
- We mostly focus on worst-case running time, but sometimes consider best-case or average-case running times.
- We sometimes refer to worst case running time as the complexity of an algorithm.
- The analysis is based on pseudocode.
- We use the Random Access Machine (RAM) model to measure primitive operations.

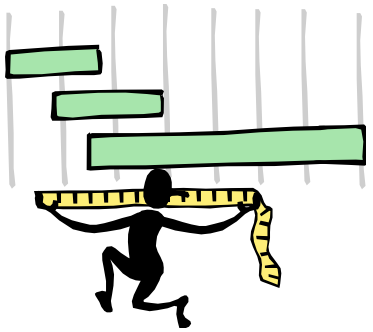
Still to come:

- We need less detail in determining primitive operations and we need an estimation that is independent of hardware (relative cost of different basic operations).
- For this we need to estimate the growth rate of the running time using big-Oh notation.

Analysis of algorithms

1. Intro to analysis of algorithms
2. Asymptotic analysis: Growth rate and Big-Oh notation
3. Comparing two algorithms for the same problem.
4. Asymptotic analysis: Big-Omega and Big-Theta; math review.

Estimating Running Time



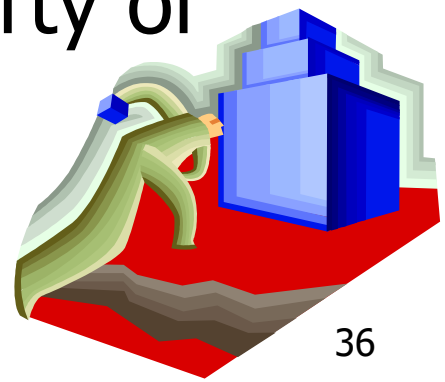
- Algorithm arrayMax executes $6n - 1$ primitive operations in the worst case, $4n + 1$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of arrayMax. Then

$$a(4n + 1) \leq T(n) \leq b(6n - 1)$$

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



Slide by Matt Stallmann
included with permission.

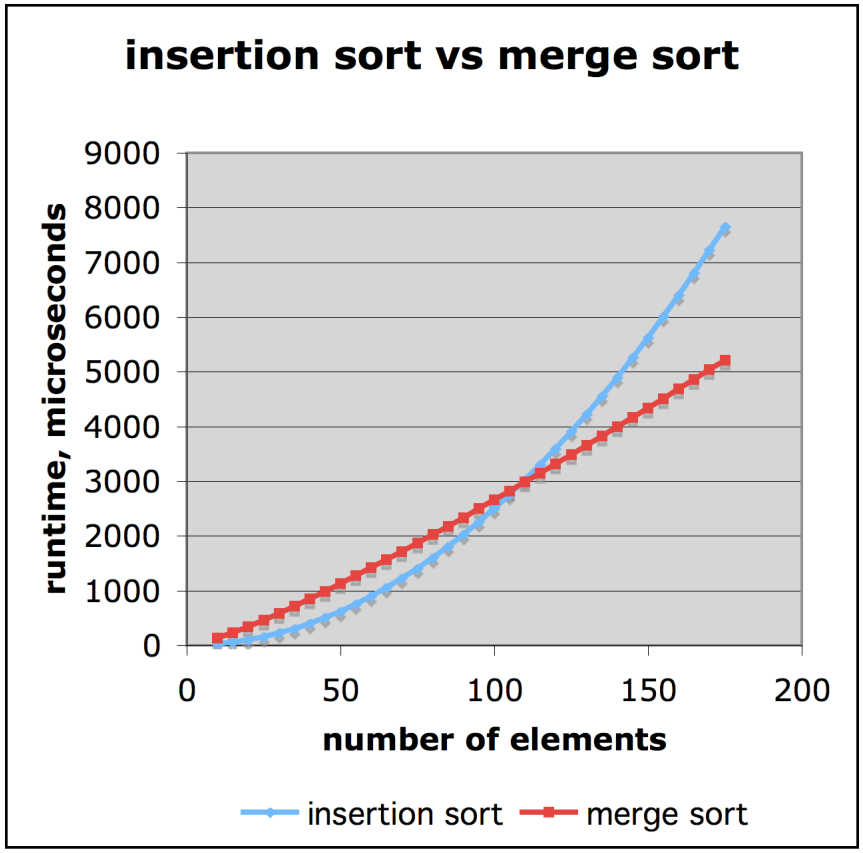
Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

runtime quadruples when problem size doubles

Slide by Matt Stallmann
included with permission.

Comparison of Two Algorithms



insertion sort is
$n^2 / 4$

merge sort is
$2 n \lg n$

sort a million items?

insertion sort takes
roughly 70 hours

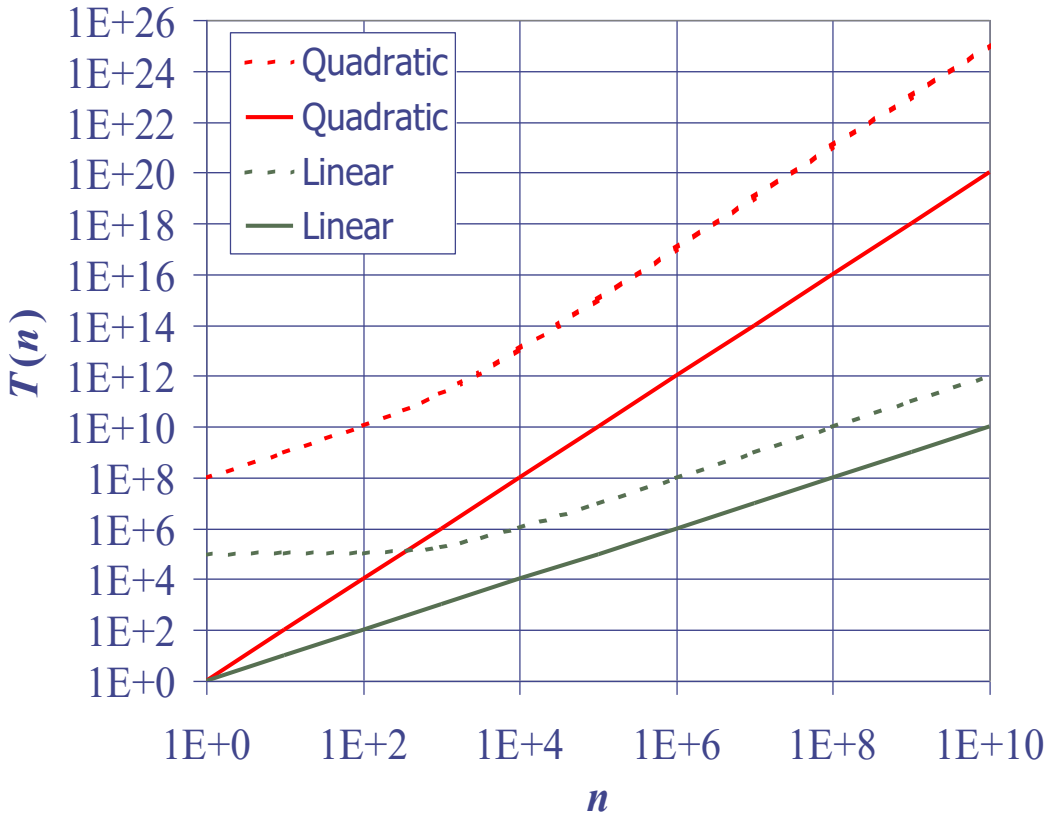
while

merge sort takes
roughly 40 seconds

This is a slow machine, but if
100 x as fast then it's 40 minutes
versus less than 0.5 seconds

Constant Factors

- 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



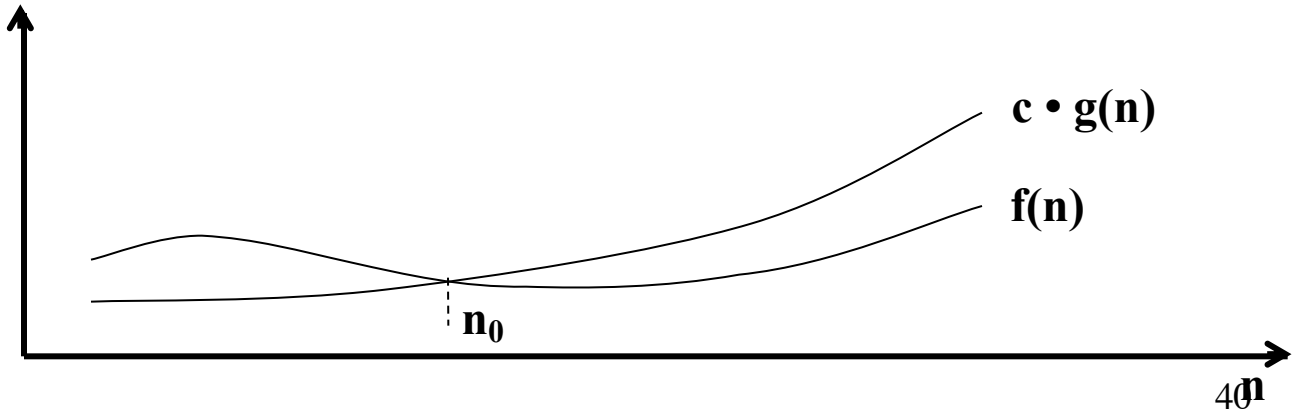
Big-Oh

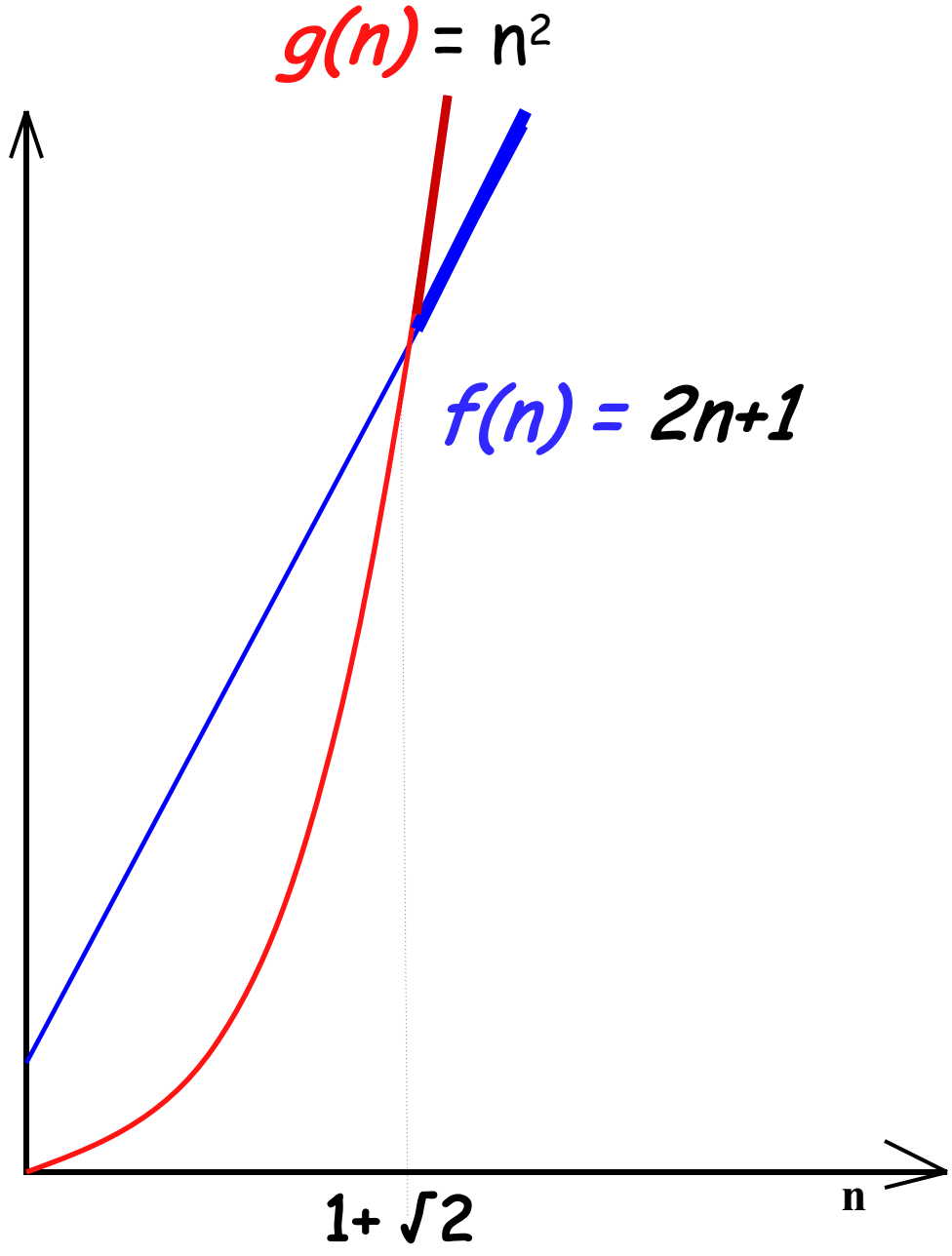
(upper bound)

- given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$

if and only if there are positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \quad \text{for all} \quad n \geq n_0$$





Graphical example …

$f(n)=2n+1$ is $O(n^2)$

Proof:

Solving $n^2 - 2n - 1 \geq 0$,

we conclude $n \geq 1+\sqrt{2}$.

So,

$2n+1 \leq n^2$, for all $n \geq 1+\sqrt{2}$.

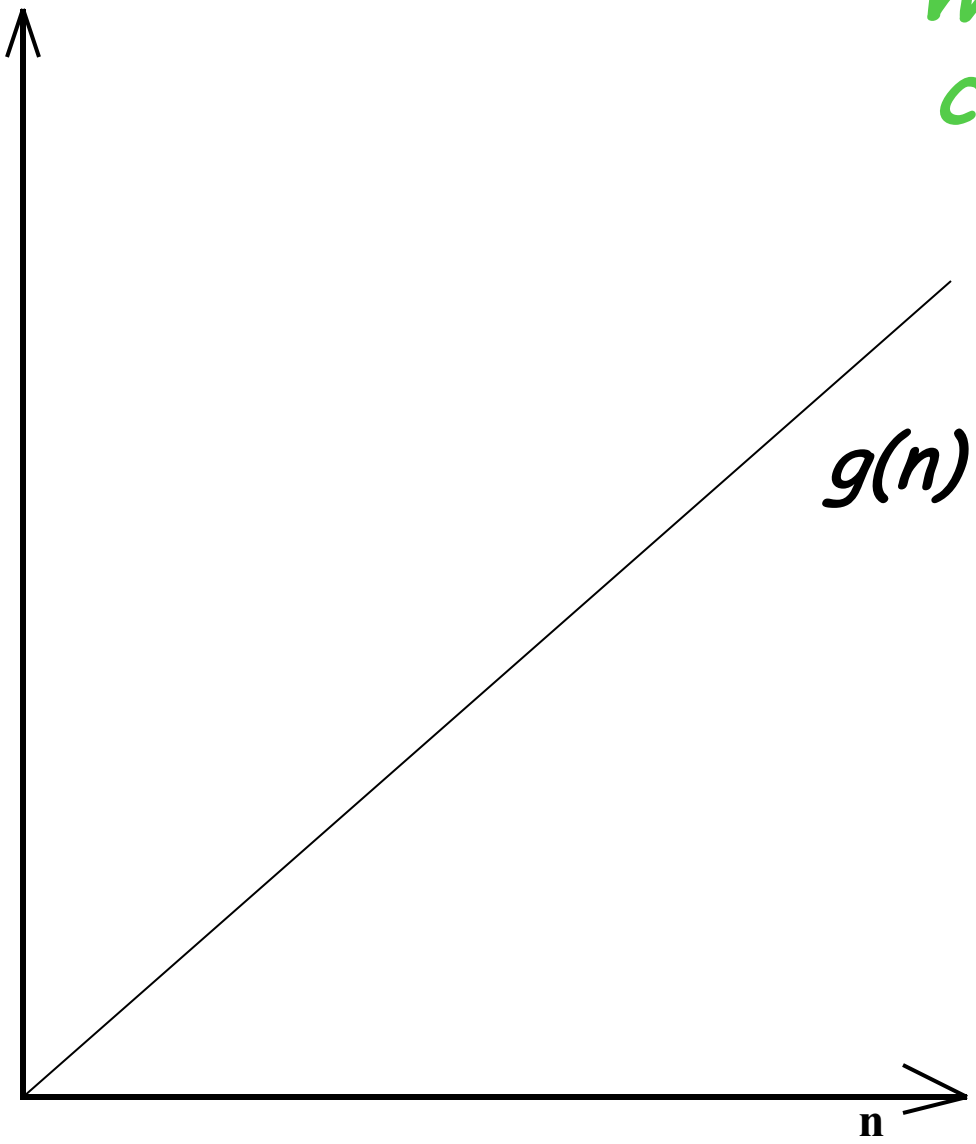
Take $c=1$ and $n_0=1+\sqrt{2}$,

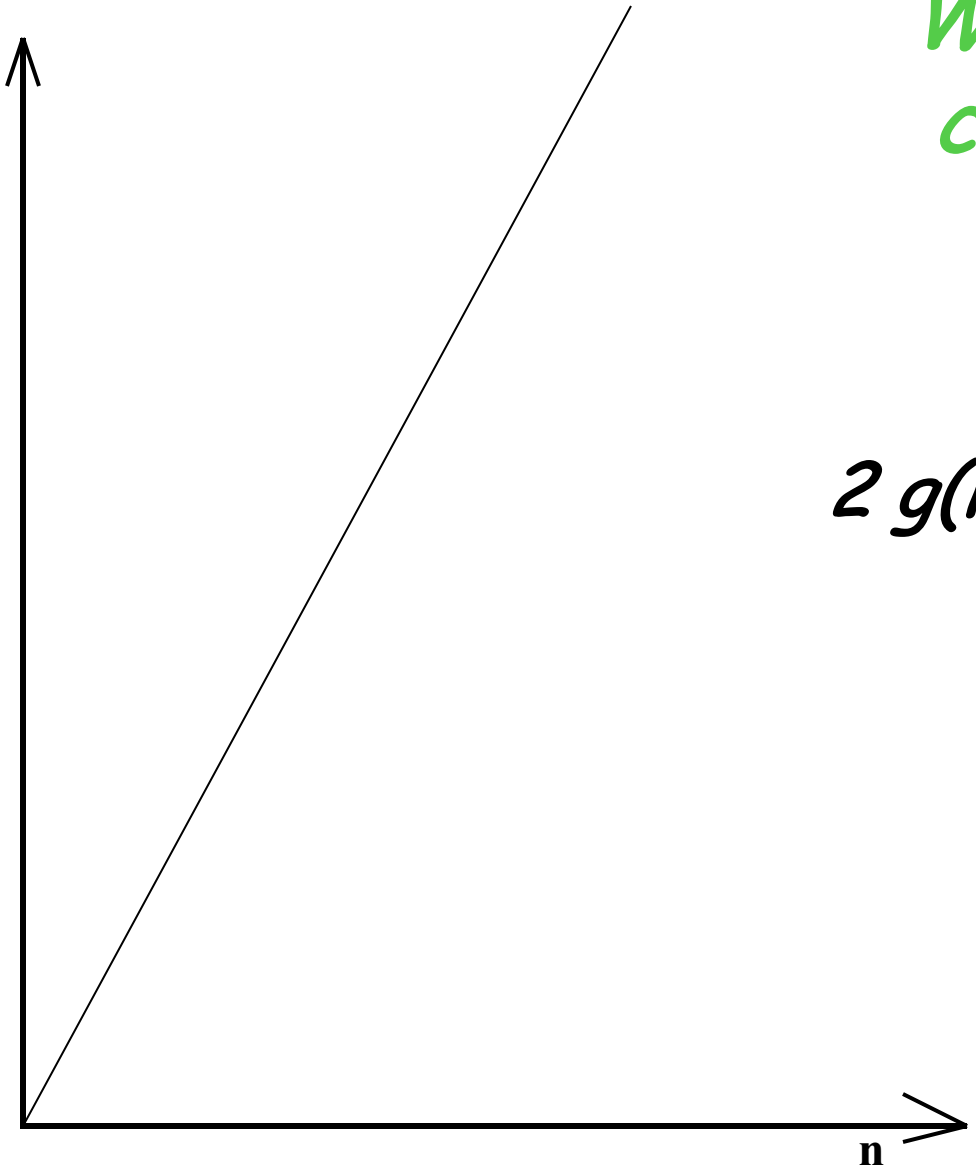
$2n+1 \leq c\, n^2$, for all $n \geq n_0$

Thus $2n+1$ *is* $O(n^2)$.

What does it mean c g(n) ?

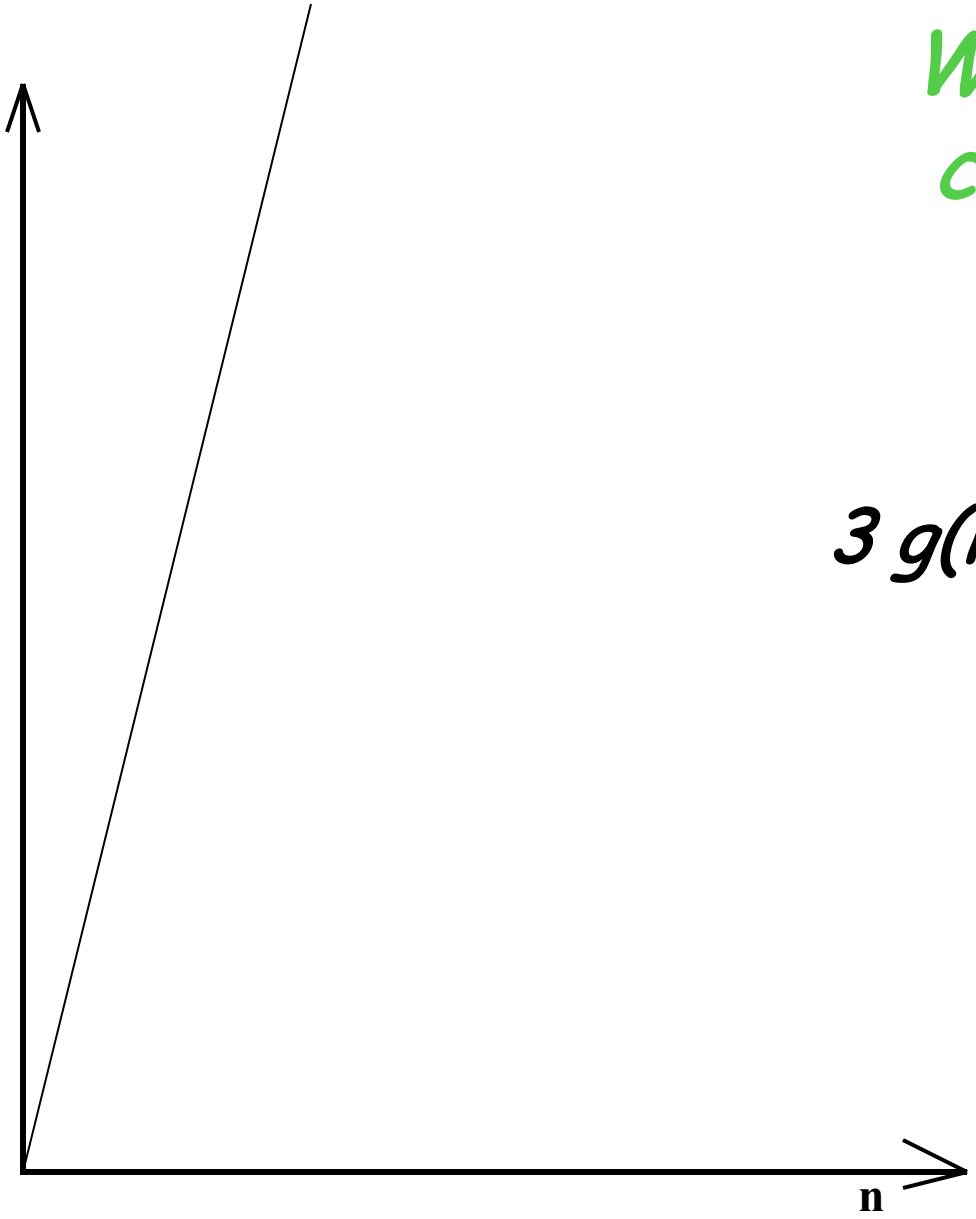
Example:





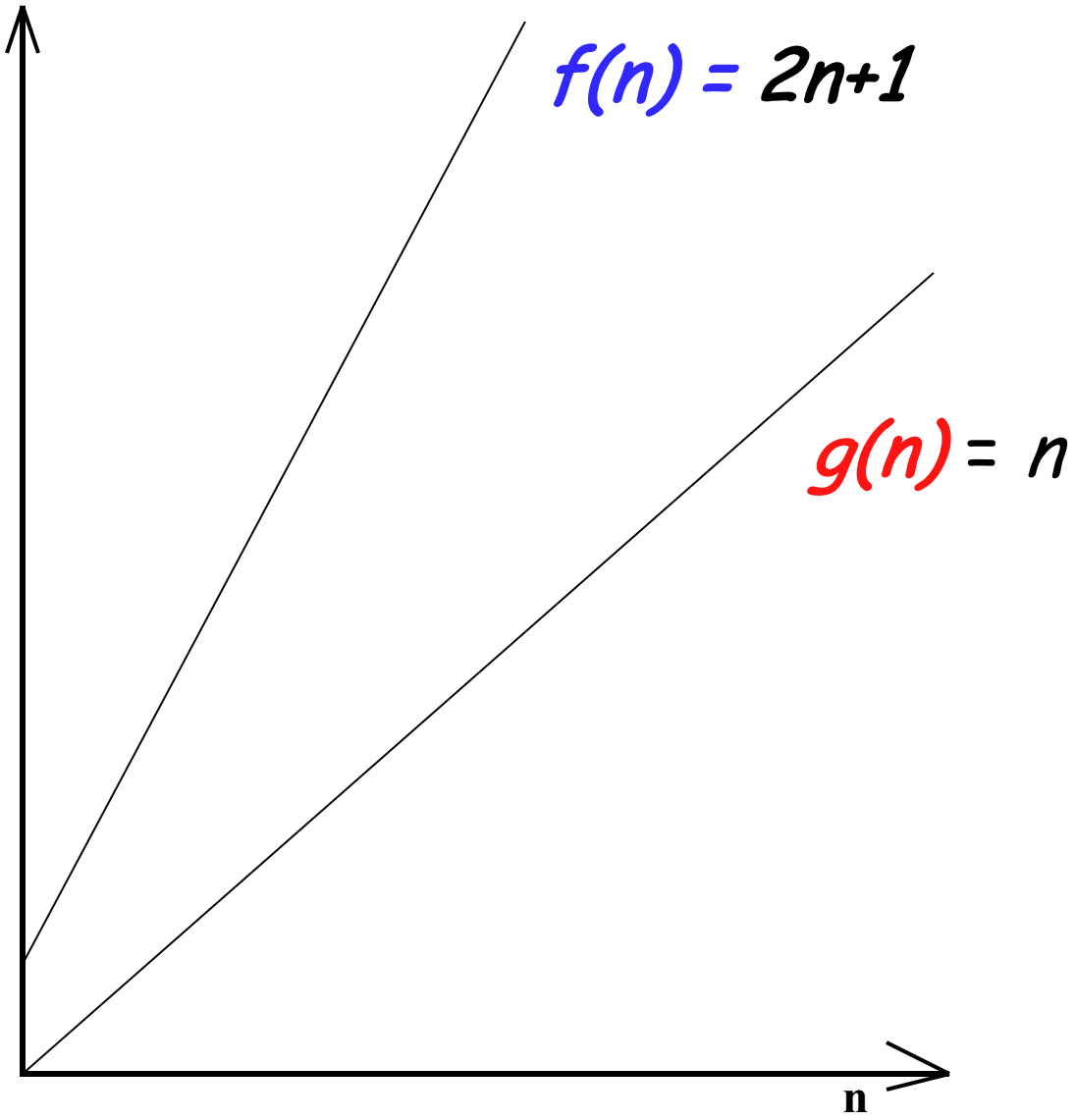
What does it mean c g(n) ?

$2\,g(n) = 2\,n$

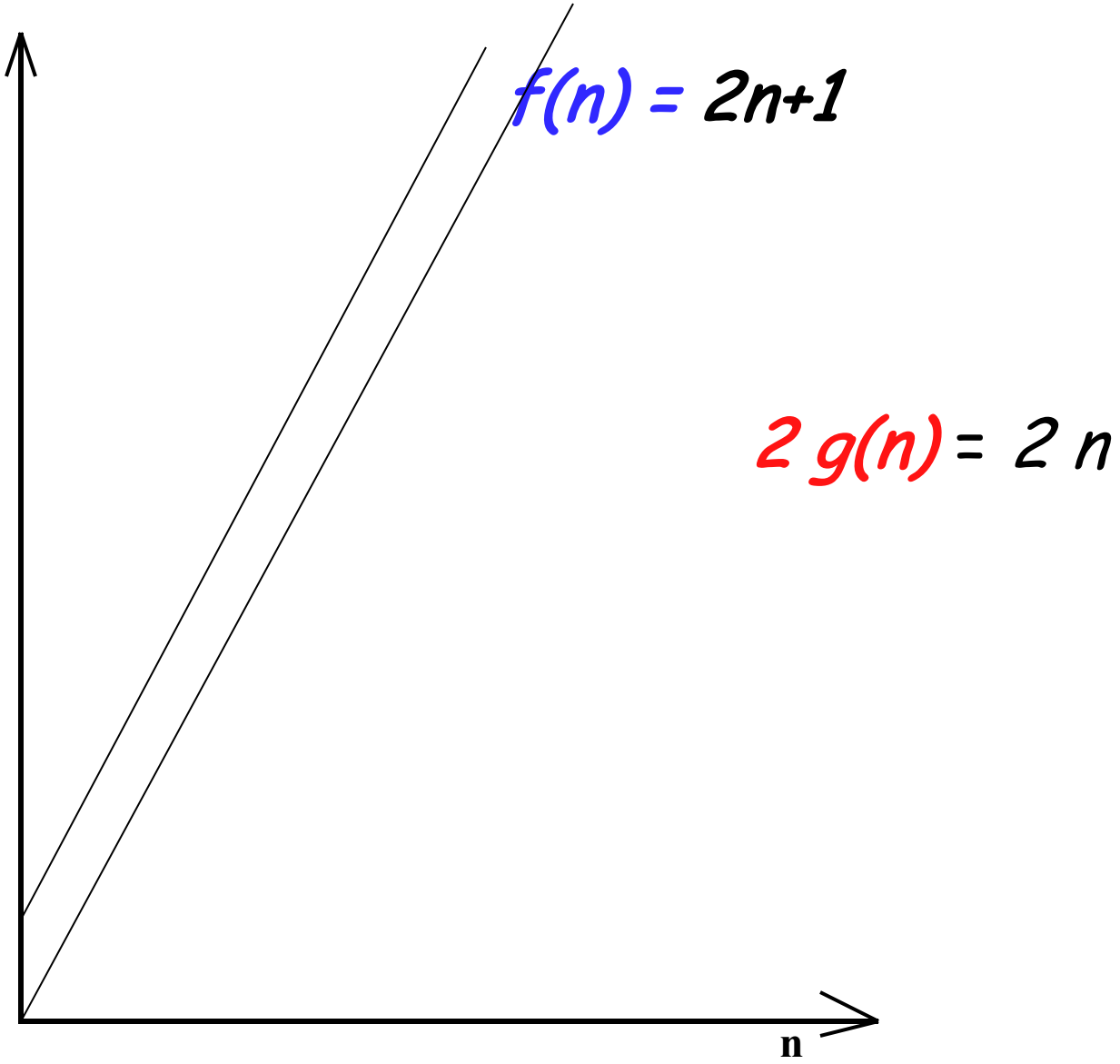


What does it mean c g(n) ?

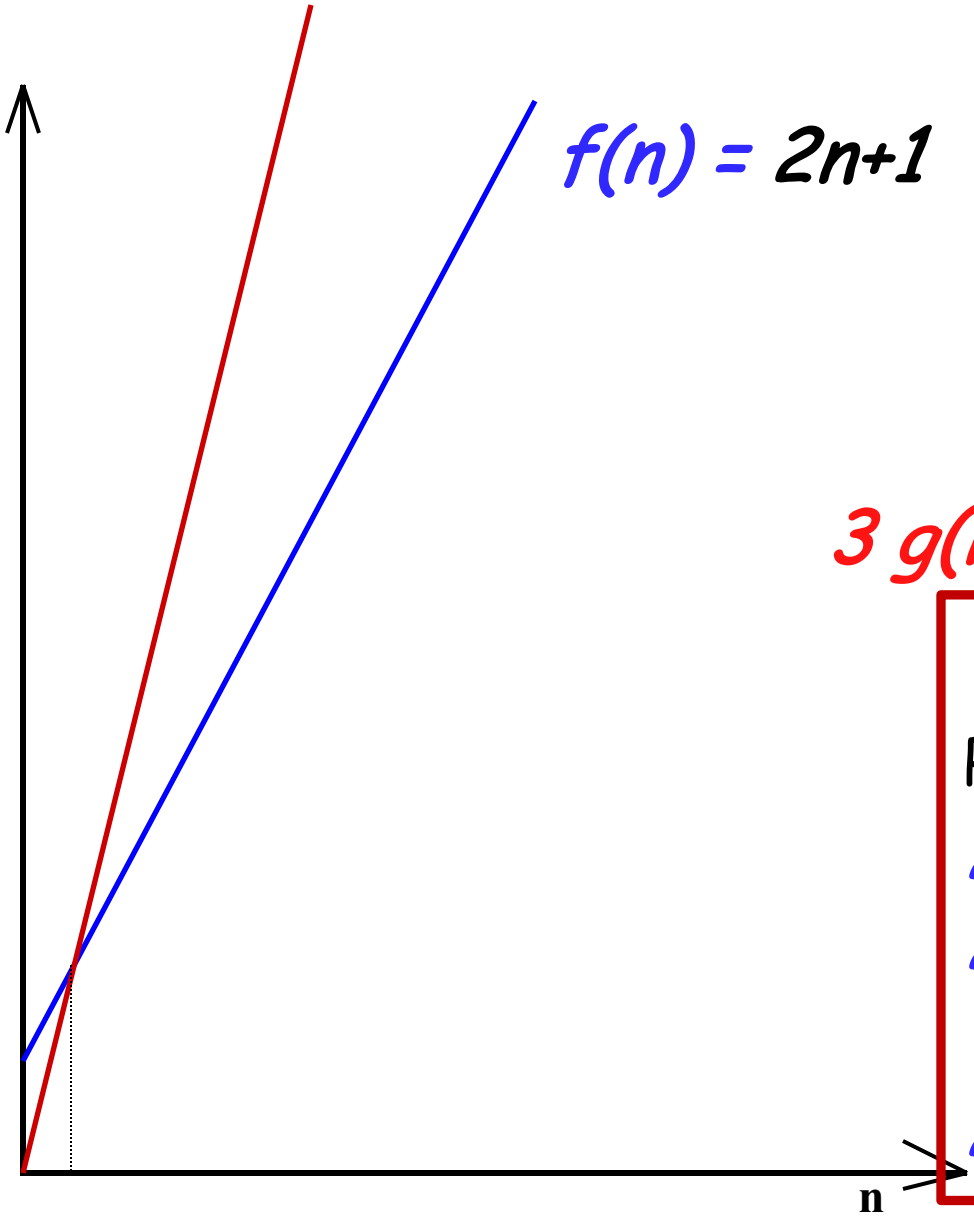
$3\,g(n) = 3\,n$



Need $f(n) \leq c\, g(n)$ for all $n \geq n_0$



Need $f(n) \leq c\, g(n)$ for all $n \geq n_0$



$f(n)=2n+1$ is $O(n)$.

Proof:

$2n+1 \leq 2n+n$, for all $n \geq 1$

$2n+1 \leq 3n$, for all $n \geq 1$

Take $c=3$ and $n_0=1$:

$2n+1 \leq c\,n$, for all $n \geq n_0$

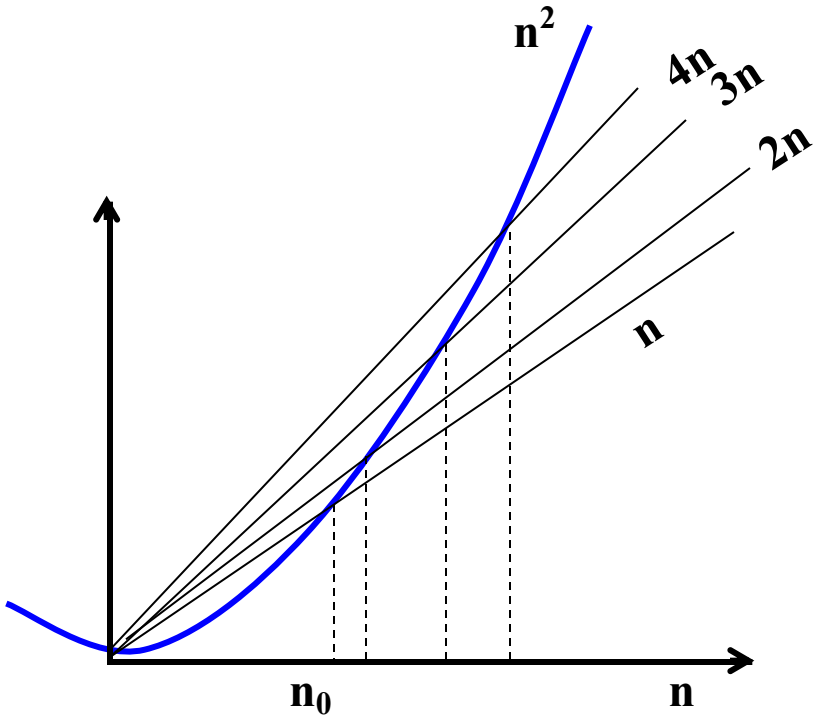
Need $f(n) \leq c\,g(n)$ for all $n \geq n_0$

On the other hand…

n^2 is not $O(n)$ because there is no c and n_0 such that:

$$n^2 \leq cn \text{ for } n \geq n_0$$

(no matter how large a c is chosen there is an n big enough (e.g. $n > c$) such that $n^2 > c\,n$)

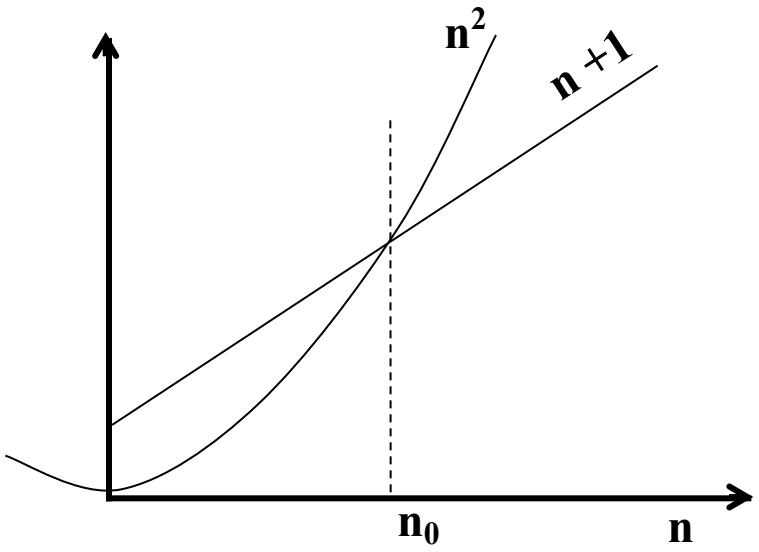


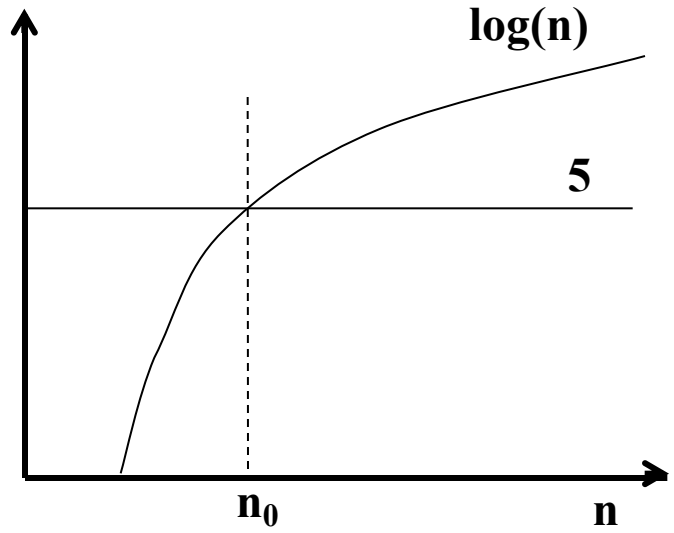
Big-Oh and Growth Rate

- 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

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) \ldots$





More Big-Oh Examples

- $7n - 2$

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

  need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c\,n$ 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 $3n^3 + 20n^2 + 5 \leq c\,n^3$ 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 $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

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

Big-Oh Conventions

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors

  Examples: $7n-3$ is $O(n)$; $8n^2\log n + 5n^2 + n$ is $O(n^2\log n)$;
  $12n^3 + 5000n^2 + 2n^4$ is $O(n^4)$
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

  (both are correct, but the first is a tighter bound)
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

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
- Example: Since arrayMax had worst case 6n-1 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

An Example

Prove that $f(n) = 60n^2 + 5n + 1$ is $O(n^2)$

We must find a constant c and a constant n_0 such that:

$$60n^2 + 5n + 1 \leq c\, n^2 \text{ for all } n \geq n_0$$

$5n \leq 5n^2$ for all $n \geq 1$

$1 \leq n^2$ for all $n \geq 1$

$60n^2 + 5n + 1 \leq 60n^2 + 5n^2 + n^2$, for all $n \geq 1$

$60n^2 + 5n + 1 \leq 66n^2$, for all $n \geq 1$

Taking $c = 66$ and $n_0 = 1$ => $f(n) = 60n^2 + 5n + 1$ is $O(n^2)$

Theorem:

If $g(n)$ is $O(f(n))$, then for any constant $c > 0$

$g(n)$ is also $O(c\, f(n))$

Theorem:

$O(f(n) + g(n)) = O(\max(f(n), g(n)))$

Ex 1:

$$2n^3 + 3n^2 = O(\max(2n^3, 3n^2))$$

$$= O(2n^3) = O(n^3)$$

Ex 2:

$$n^2 + 3 \log n - 7 = O(\max(n^2, 3 \log n - 7))$$

$$= O(n^2)$$

Theorem:

If $g(n)$ is $O(f(n))$, then for any constant $c > 0$

$g(n)$ is also $O(c\, f(n))$

Theorem:

$O(f(n) + g(n)) = O(\max(f(n), g(n)))$

Ex 1:

$$2n^3 + 3n^2 = O(\max(2n^3, 3n^2))$$

$$= O(2n^3) = O(n^3)$$

Ex 2:

$$n^2 + 3 \log n - 7 = O(\max(n^2, 3 \log n - 7))$$

$$= O(n^2)$$

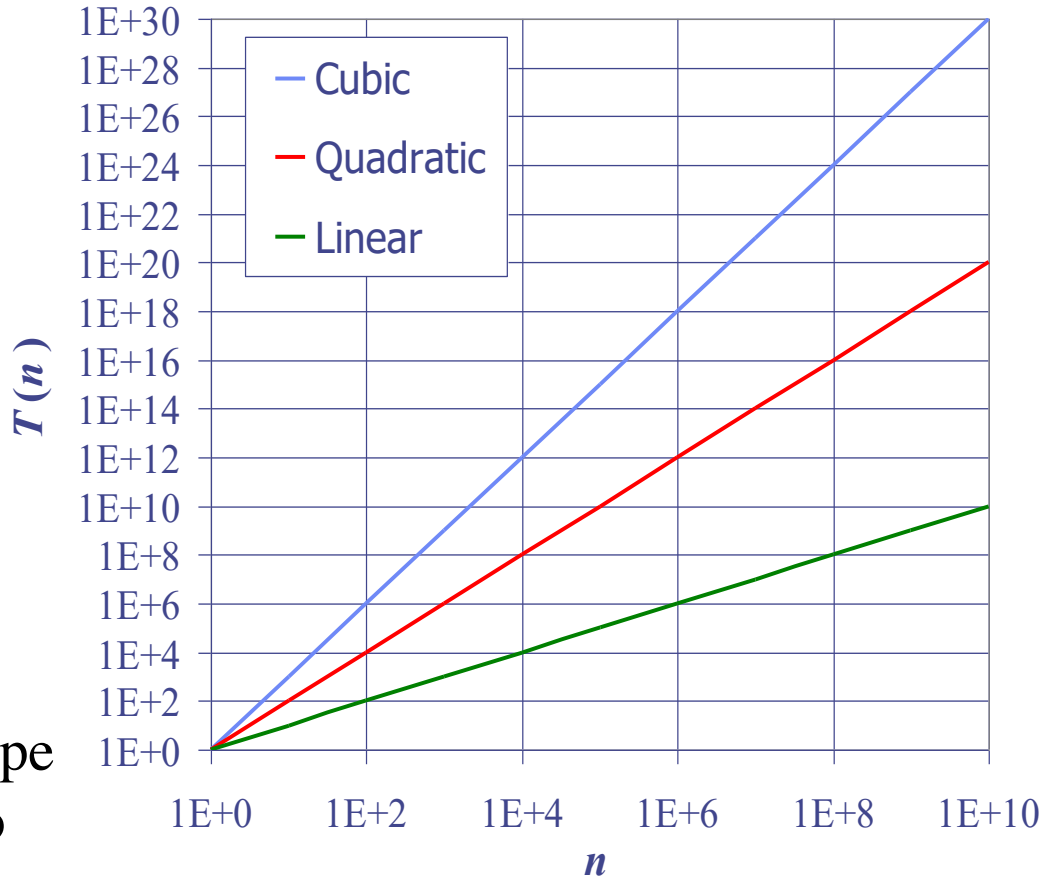
Asymptotic Notation *(terminology)*

- Special classes of algorithms:

constant:	$O(1)$
logarithmic:	$O(\log n)$
linear:	$O(n)$
quadratic:	$O(n^2)$
cubic:	$O(n^3)$
polynomial:	$O(n^k)$, for some fixed $k > 0$
exponential:	$O(a^n)$, for some fixed $a > 1$

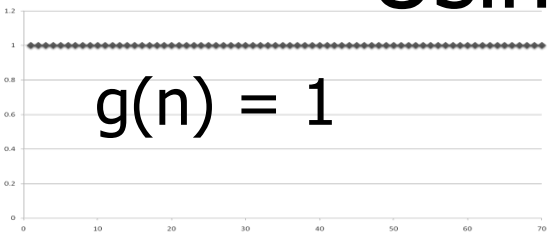
Seven Important Functions

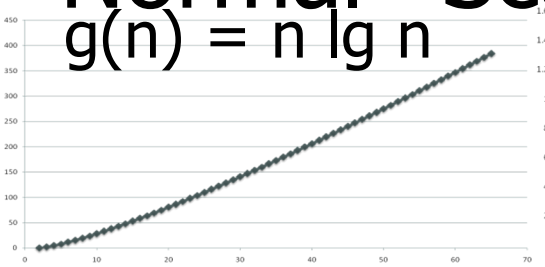
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ *1*
  - Logarithmic ≈ log *n*
  - Linear ≈ *n*
  - N-Log-N ≈ *n* log *n*
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate

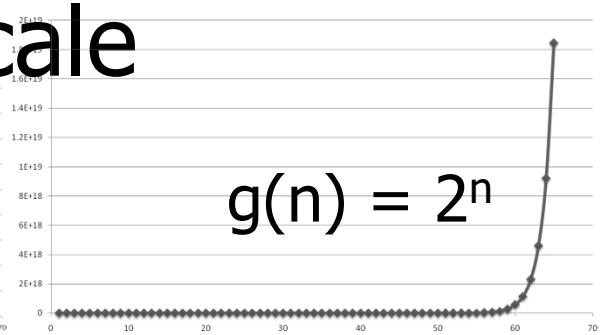


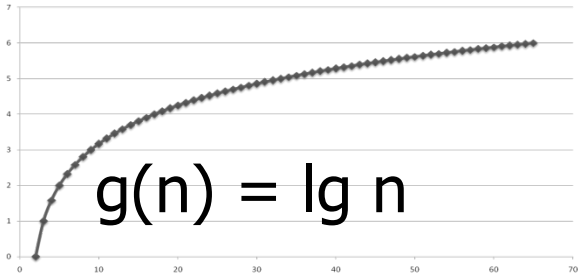
Slide by Matt Stallmann included with permission.

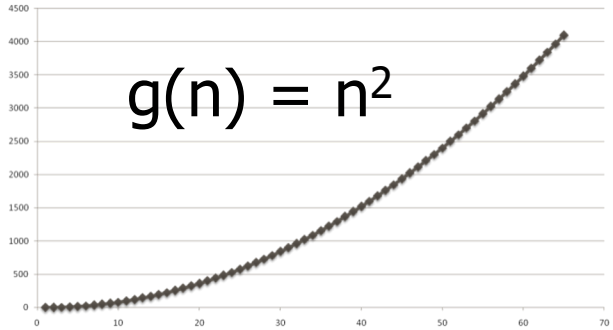
Functions Graphed Using "Normal" Scale

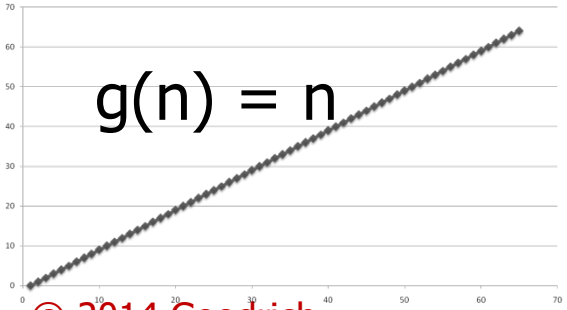


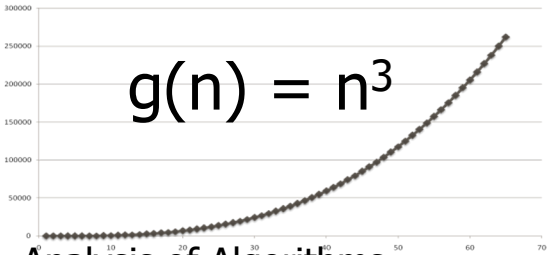












Analysis of algorithms

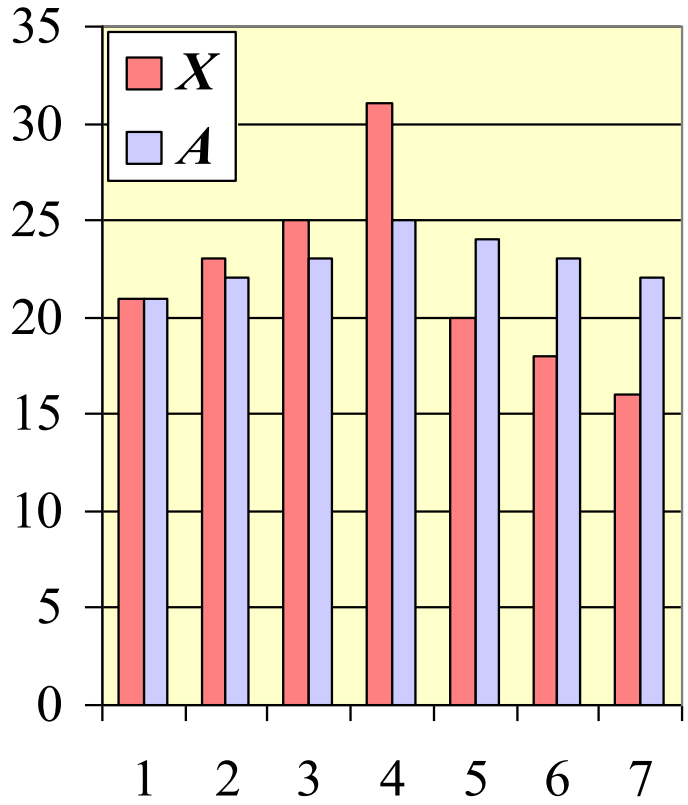
1. Intro to analysis of algorithms
2. Asymptotic analysis: Growth rate and Big-Oh notation
3. Comparing two algorithms for the same problem.
4. Asymptotic analysis: Big-Omega and Big-Theta; math review.

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:

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

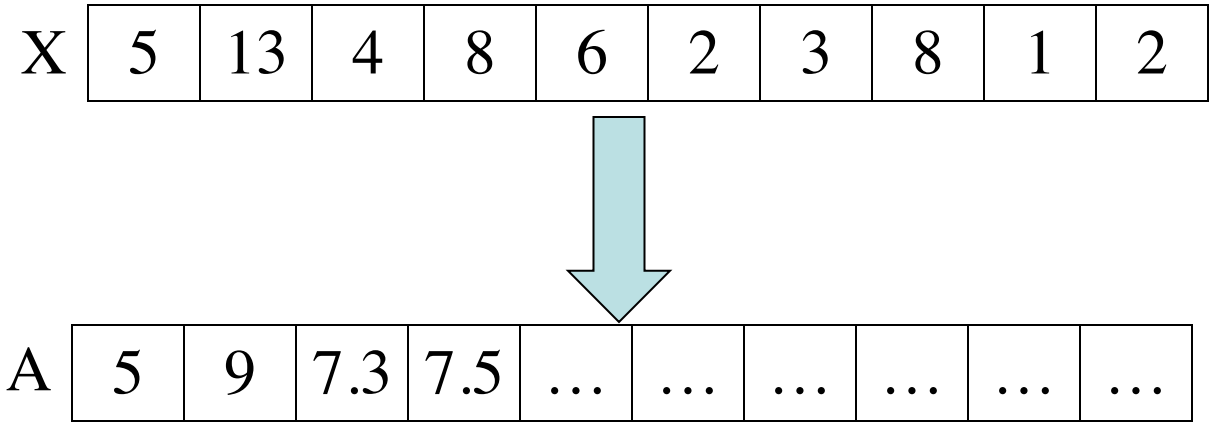
- Computing the array A of prefix averages of another array X has applications to financial analysis



Computing Prefix Averages

The i-th prefix average of an array X is average of the first $(i+1)$ elements of X

$$A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{(i+1)}$$



Prefix Averages 1

The following algorithm is a naïve algorithms that computes prefix averages by applying the definition

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[ ] prefixAverage1(double[ ] x) {
3    int n = x.length;
4    double[ ] a = new double[n];                    // filled with zeros by default
5    for (int j=0; j < n; j++) {
6      double total = 0;                             // begin computing x[0] + ... + x[j]
7      for (int i=0; i <= j; i++)
8        total += x[i];
9      a[j] = total / (j+1);                         // record the average
10   }
11   return a;
12 }
```

Prefix Averages 1

The following algorithm is a naïve algorithms that computes prefix averages by applying the definition.

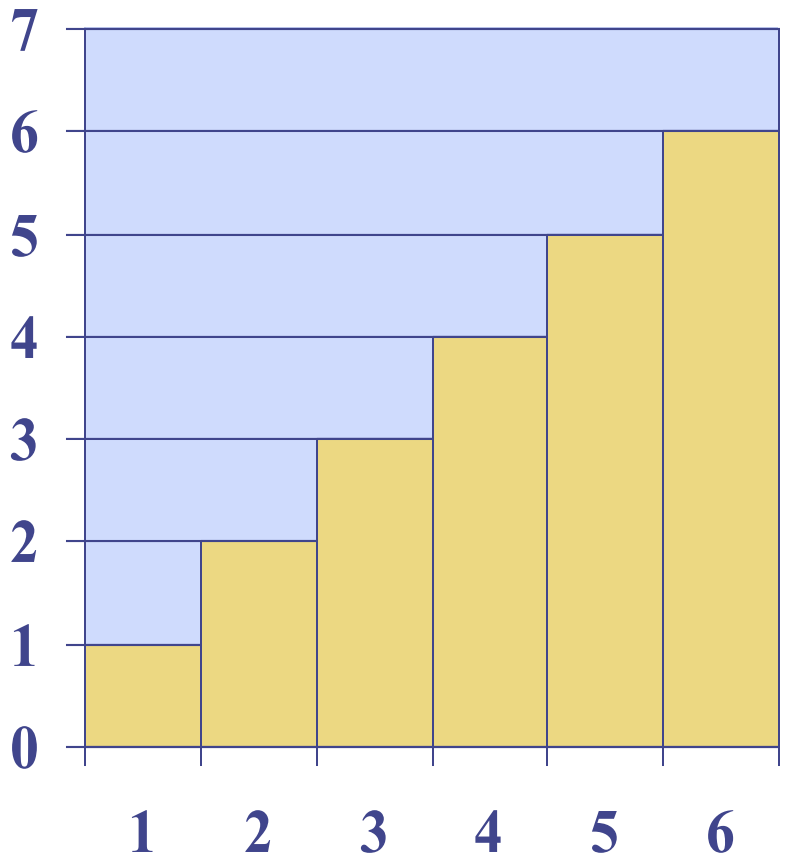
Algorithm ***prefixAverages1***(*X*, *n*)
Input array *X* of *n* integers
Output array *A* of prefix averages of *X*

1.	*A* ← new array of *n* integers	…….. *1*
2.	**for** *i* ← 0 **to** *n* – 1 **do**	…….. *n*
3.	*s* ← *X*[0]	…….. *n*
4.	**for** *j* ← 1 **to** *i* **do**	…….. *1+2+….+(n-1)+n*
5.	*s* ← *s* + *X*[*j*]	…….. *0+1+2+….+(n-1)*
6.	*A*[*i*] ← *s* / (*i* + 1)	…….. *n*
7.	**return** *A*	…….. *1*

Running time of prefixAverages1: $T_1(n)$ is O(1+2+…+n)
^^^^^^^^^(simplify!)

Arithmetic Progression

- The running time of prefixAverage1 is $T_1(n)$ is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm prefixAverage1 runs in $O(n^2)$ time (quadratic)



Prefix Averages 2

The following algorithm uses a running summation to improve the efficiency

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[ ] prefixAverage2(double[ ] x) {
3    int n = x.length;
4    double[ ] a = new double[n];          // filled with zeros by default
5    double total = 0;                     // compute prefix sum as x[0] + x[1] + ...
6    for (int j=0; j < n; j++) {
7      total += x[j];                      // update prefix sum to include x[j]
8      a[j] = total / (j+1);               // compute average based on current sum
9    }
10   return a;
11 }
```

Prefix Averages 2

The following algorithm uses a running summation to improve the efficiency.

Algorithm prefixAverages2(X,n):

Input: An *n*-element array X of n numbers.

Output: An *n* -element array A of prefix averages of X

```
1    A ← new array of n integers
2    s ← 0
3    for i ← 0 to n-1 do
4        s ← s + X[i]
5        A[i] ← s/(i+1)
6    return array A
```

```
s← 0
for i ← 0 to n-1 do
    s ← s + X[i]
    A[i] ← s/(i+1)
```

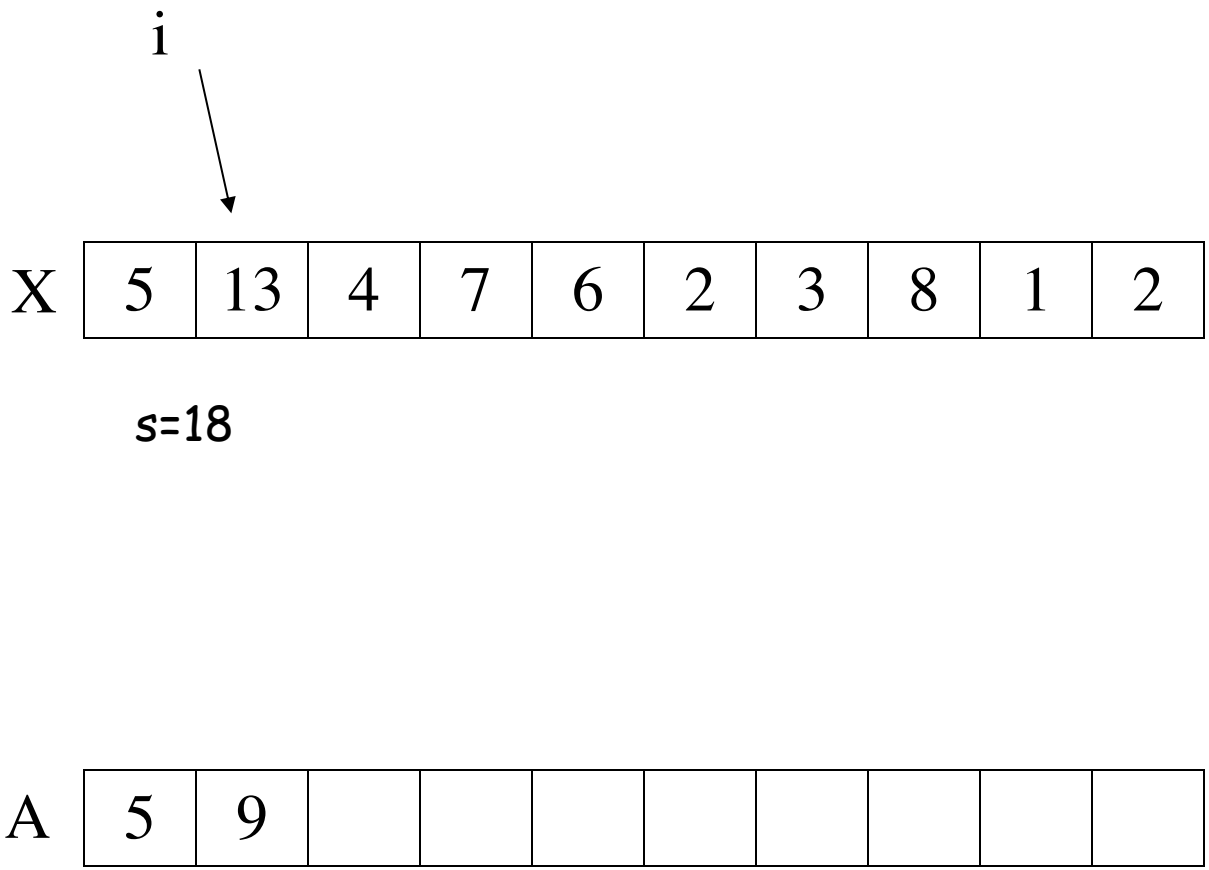
i

X	5	13	4	8	6	2	3	8	1	2

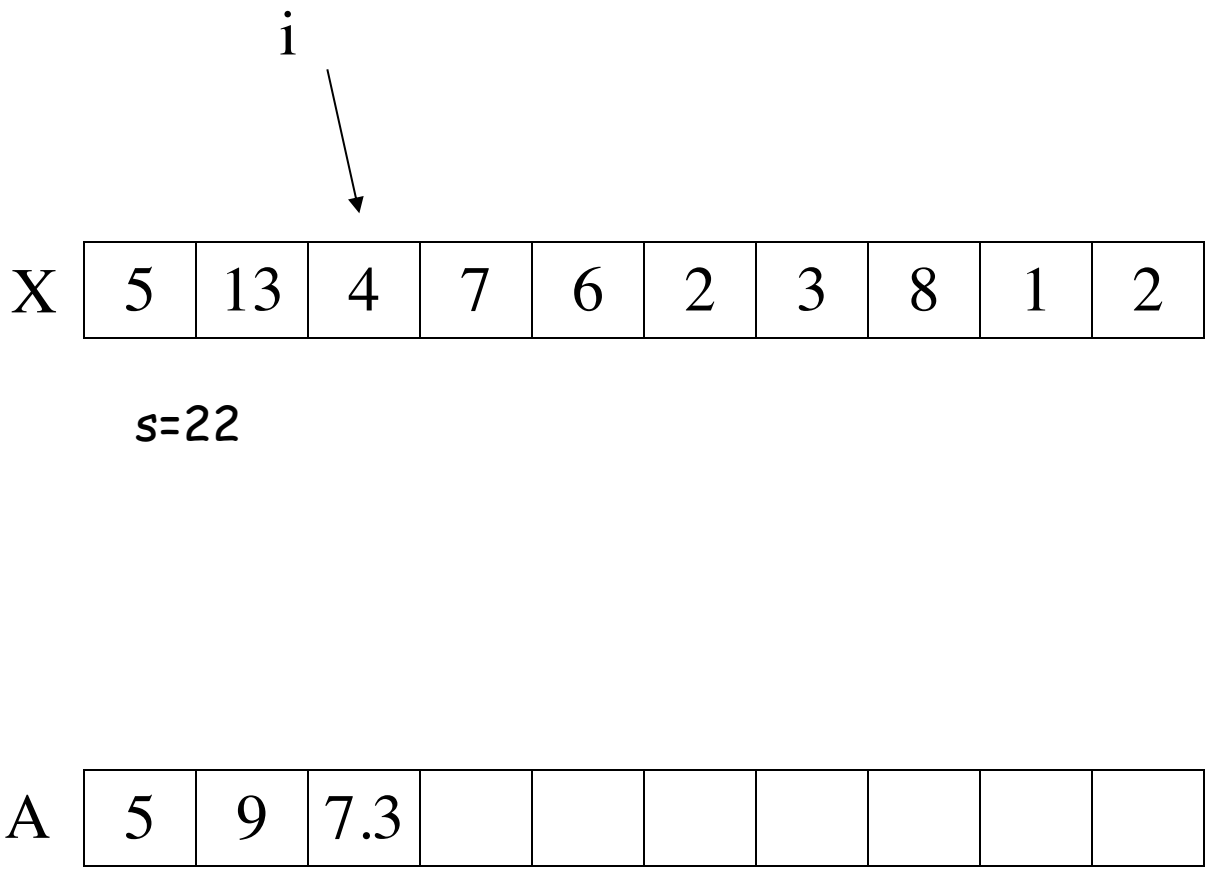
s=5

A	5									

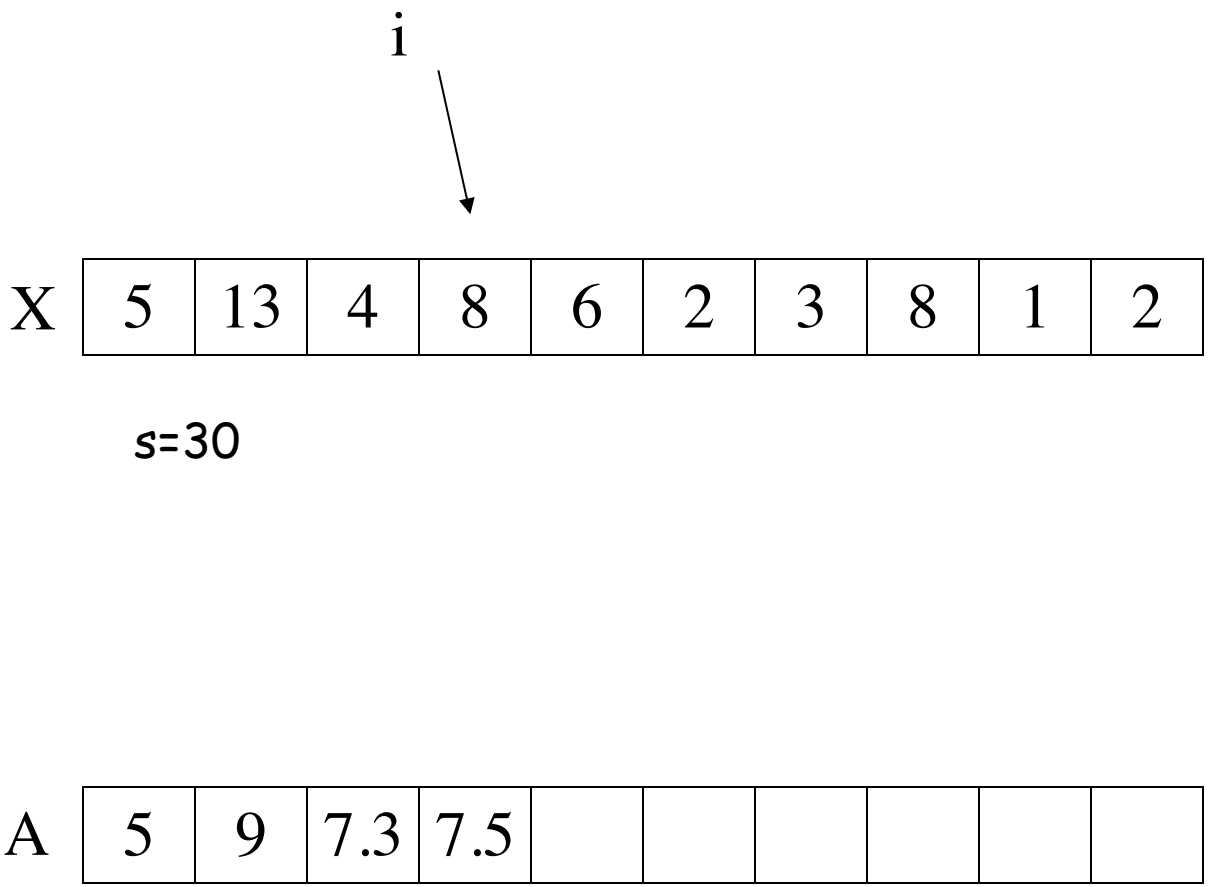
```
s← 0
for i ← 0 to n-1 do
        s ← s + X[i]
        A[i] ← s/(i+1)
```



```
s← 0
for i ← 0 to n-1 do
        s ← s + X[i]
        A[i] ← s/(i+1)
```



```
s← 0
for i ← 0 to n-1 do
        s ← s + X[i]
        A[i] ← s/(i+1)
```



Prefix Averages 2

The following algorithm uses a running summation to improve the efficiency.

Algorithm prefixAverages2(X,n):

Input: An *n*-element array X of n numbers.

Output: An *n* -element array A of prefix averages of X

```
1    A ← new array of n integers ..... 1
2    s ← 0 ..... 1
3    for i ← 0 to n-1 do ..... n+1
4        s ← s + X[i] ..... n
5        A[i] ← s/(i+1) ..... n
6    return array A ..... 1
```

Running time of prefixAverage2: $T_2(n)$ is $\boldsymbol{O(n)}$ time (linear).

Comparison

prefixAverages1	prefixAverages2
$O(n^2)$	$O(n)$

Analysis of algorithms

1. Intro to analysis of algorithms
2. Asymptotic analysis: Growth rate and Big-Oh notation
3. Comparing two algorithms for the same problem.
4. Asymptotic analysis: Big-Omega and Big-Theta; math review.

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

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

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

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

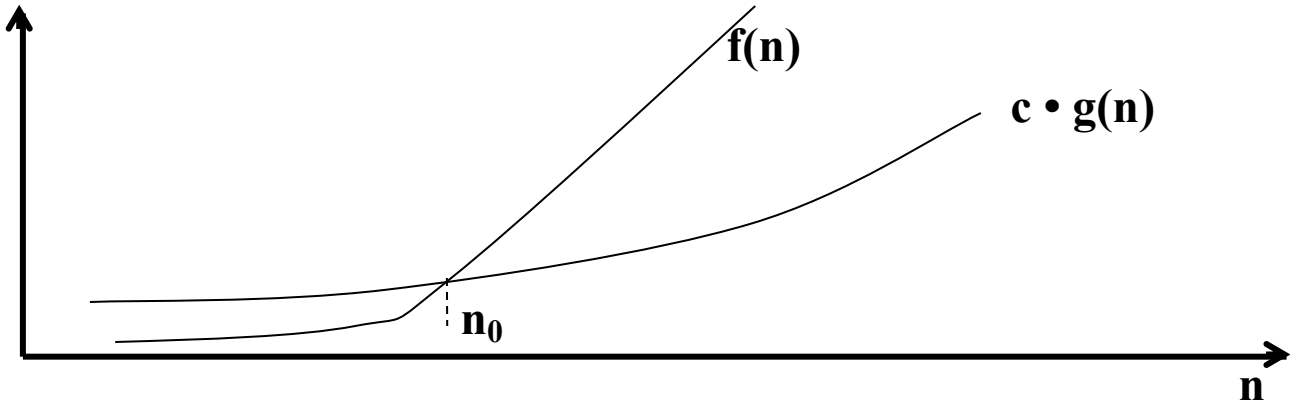
big-Omega

(lower bound)

Definition: f(n) is $\Omega(g(n))$ (f(n) is big-Omega of g(n))

if there exist $c > 0$ and $n_0 \geq 1$ such that

$f(n) \geq c \cdot g(n)$ for all $n \geq n_0$



Theorem: f(n) is $\Omega(g(n))$ iff g(n) is O(f(n))

big-Theta

Definition: f(n) is big-Theta of g(n)

- 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

$$c'g(n) \leq f(n) \leq c''g(n), \text{ for all } n \geq n_0$$

Theorem.

f(n) is $\Theta(g(n))$ <===> if $f(n) \in O(g(n))$ AND $g(n) \in O(f(n))$

Theorem.

f(n) is $\Theta(g(n))$ <===> if $f(n) \in O(g(n))$ AND $f(n) \in \Omega(g(n))$

An Example

We have seen that

$f(n) = 60n^2 + 5n + 1$ is $O(n^2)$

but $60n^2 + 5n + 1 \geq 60n^2$ for $n \geq 1$

So: with $c = 60$ and $n_0 = 1$

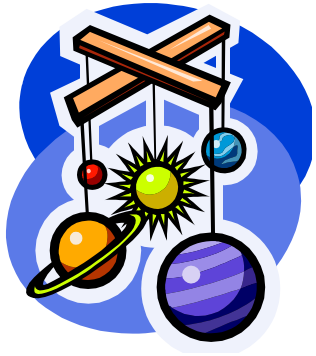
$f(n) \geq c \bullet n^2$ for all $n \geq 1$

$60n^2 + 5n + 1$ is $\Omega(n^2)$

$60n^2 + 5n + 1$ is $O(n^2)$
AND
$60n^2 + 5n + 1$ is $\Omega(n^2)$

$60n^2 + 5n + 1$ is $\Theta(n^2)$

Example Uses of the Relatives of Big-Oh



- **$5n^2$ 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 $f(n) \geq c\ g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ 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 $f(n) \geq c\ g(n)$ for $n \geq n_0$

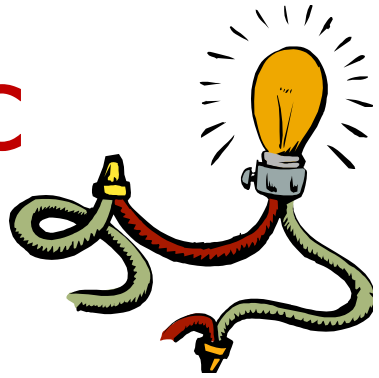
  let $c = 1$ and $n_0 = 1$

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

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c\ g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

Intuition for Asymptotic Notation



big-Oh

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

big-Omega

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

big-Theta

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

Math You Need to Review

- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

- Properties of logarithms:

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b x^a = a\log_b x$

  $\log_b a = \log_x a / \log_x b$

More Math to Review

- Floor: $\lfloor x \rfloor$ = the largest integer $\leq x$
- Ceiling: $\lceil x \rceil$ = the smallest integer $\geq x$
- Summations
- Aithmetic progression
- Geometric progression

More Math to Review

Arithmetic Progression

Calculating sum of first n terms of arithmetic progression:

$$S' = \sum_{i=0}^{n} i\,d = 0 + d + 2d + \ldots + nd$$

In prefixAverages1, we encountered sum with d=1:

$$\begin{aligned} S &= 0 + 1 + 2 + \ldots + (n-1) + n \\ S &= n + n-1 + n-2 + \ldots + 1 + 0 \\ 2S &= n + n + n + \ldots + n + n = (n+1)\,n \end{aligned}$$

$$S = \frac{n(n+1)}{2}$$

$$S' = \sum_{i=0}^{n} i\,d = d \sum_{i=0}^{n} i = d\,\frac{n(n+1)}{2}$$

More Math to Review
Geometric Progression

$$S = \sum_{i=0}^{n} r^i = 1 + r + r^2 + \ldots + r^n$$

$$rS = r + r^2 + \ldots + r^n + r^{n+1}$$

$$rS - S = r + r^2 + \ldots + r^n + r^{n+1} - r - r^2 - \ldots - r^n - 1$$

$$(r-1)S = r^{n+1} - 1$$

$$S = \frac{(rn^{+1-1})}{(r-1)}$$

For r=2,

$S = 1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1}-1$

To memorize, recall adding numbers in binary:

1+	1+
2+	10+
4+	100+
…+	…+
2^n	10..00
	11..11
$2^{n+1}-1$	= 100..00 -1