INT202
Complexity of Algorithms
Introduction

XJTLU/SAT/INT

SEM2 AY2020-2021

Review

Running time

- Depends on input
    - E.g.: 1st is the maximum Vs. last is the maximum
- Upper bound on running time.
    - Guarantee to the user

Recursive Algorithms

- Recursion involves a procedure calling itself to solve *sub-problems* of a smaller size. These smaller subproblems can then be combined in some way to get a solution to a larger problem.
- Recursive procedures require a *base case* that can be solved directly without using recursion.

Recurrence Relations

- *Recurrence relations* sometimes allow us to define the running-time of an algorithm in the form of an equation.

Suppose that $T(n)$ denotes the running time of algorithm on input of size n. Then we might be able to characterize $T(n)$ in terms of, say, $T(n - 1)$. For example, we might be able to show that

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{for } n \geq 2 \end{cases}$$

Ideally, given such a relationship we would then want to express this recurrence relation in a *closed form*.

In the example, we can show that

$$T(n) = 7(n - 1) + 3 = 7n - 4.$$

3 10 17 24 …

Recurrence Relations (cont.)

- Recurrence relations may appear in many forms. Some examples include:

1. $C(n) = 3 \cdot C(n-1) + 2 \cdot C(n-2) + C(n-3)$ where

   $C(1) = 1, C(2) = 3, C(3) = 5$

2. The Fibonacci numbers

Recursion example: Fibonacci Numbers

The Fibonacci numbers are defined as the sequence

$f_1 = f_2 = 1$, and $f_n = f_{n-1} + f_{n-2}$, for $n \geq 3$. They can be found using the following pseudo-code that computes them recursively.

Problem: Write a piece of pseudocode to compute

Fibonacci numbers, n=50.

The terms of the Fibonacci sequence are: 1,1,2,3,5,8,13,21,34.

Problem

Algorithm: fibonacci numbers
Input: upper limit n
Output: The n-th term of Fibonacci

```
int fib (int n){
if n <=2
    return n;
else
    return fib(n-1)+fib(n-2);
}
```

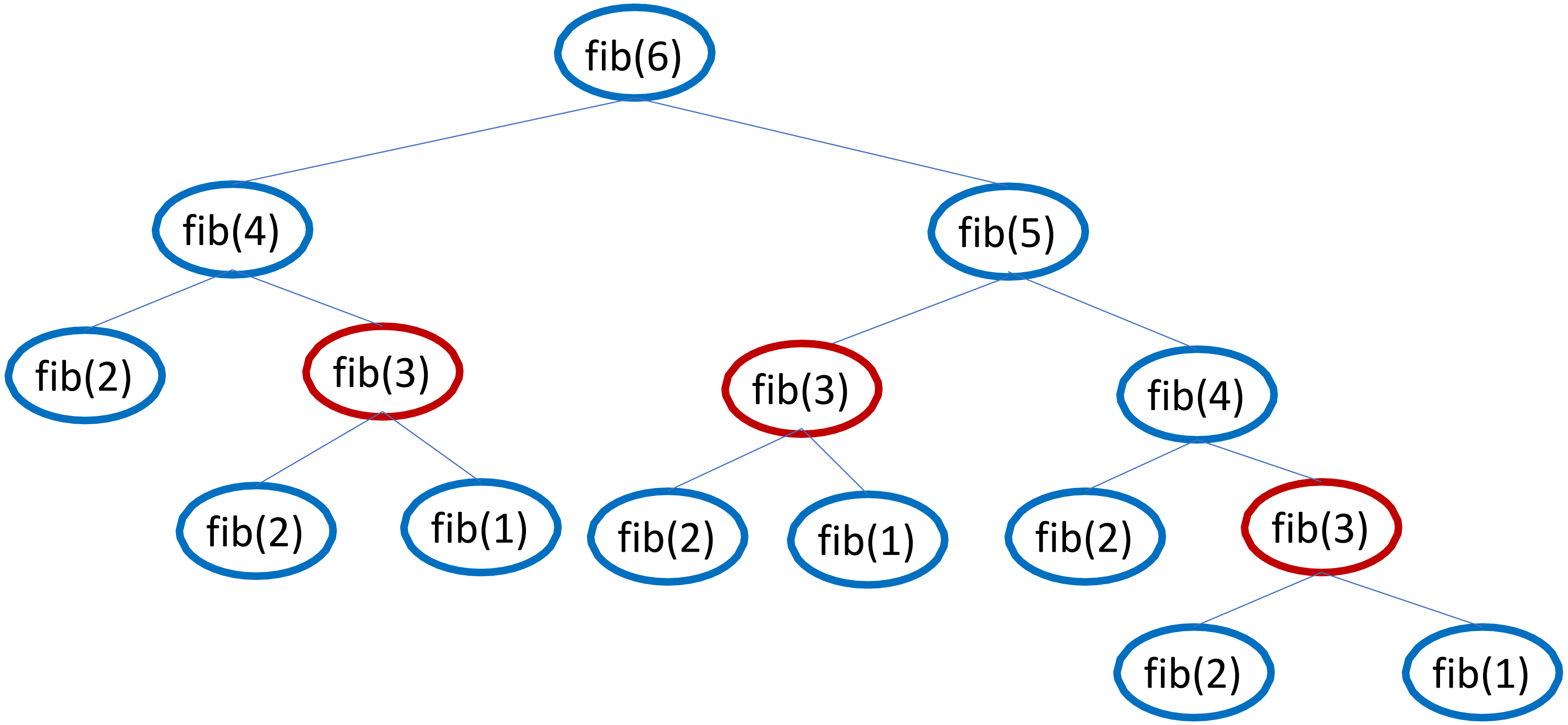
Recursive Algorithms: A Word of Caution...

While recursive algorithms are often “simpler” to write than a non-recursive version, there are often reasons to avoid them.

- In many situations, the smaller subproblems might be solved *repeatedly* during execution of the recursive algorithm.

Recursive Algorithms: A Word of Caution...

To compute Fibonacci(n), we must compute Fibonacci(n – 1) and Fibonacci(n – 2). **Both** of these function calls must then compute Fibonacci(n – 3) and Fibonacci(n – 4), etc. This repetition of work can massively increase the overall running time of the algorithm.



Recursive Algorithms: A Word of Caution...

To compute Fibonacci(n), we must compute Fibonacci($n - 1$) and Fibonacci($n - 2$). **Both** of these function calls must then compute Fibonacci($n - 3$) and Fibonacci($n - 4$), etc. This repetition of work can massively increase the overall running time of the algorithm.

- Because of the above phenomena, a recursive algorithm could also be impossible to perform on a computer (for large input values) because the repeated function calls might exhaust the memory of the machine.

Recursive Algorithms

- Recursion involves a procedure calling itself to solve *sub-problems* of a smaller size. These smaller subproblems can then be combined in some way to get a solution to a larger problem.
- Recursive procedures require a *base case* that can be solved directly without using recursion.

Exercise

- Write pseudo-code for a non-recursive method to compute *Fibonacci*(n), assuming that n is a positive integer.

Hint: Avoid the repeated computations mentioned above, by starting from the beginning of the sequence and "working up" to get the term you want.

Exercise

```
Algorithm: fibonacci numbers
Input: upper limit Nmax
Int f(int Nmax)
{
f1 ← 1;
f2 ←1;
 for n←3:(Nmax){
     fn← f2 + f1;
         f1 ← f2;
         f2 ← fn;
  return fn;
}}
```

Asymptotic notation

- *Asymptotic notation* allows characterization of the *main factors* affecting running time.

Used in a *simplified analysis* that estimates the number of primitive operations executed *up to a constant factor*.

Such notation lets us compare the running times of two algorithms.

Importance of asymptotics

Maximum size allowed for an input instance for various running times to be solved in 1 second, 1 minute and 1 hour, assuming a 1MHz machine:

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n \log n$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

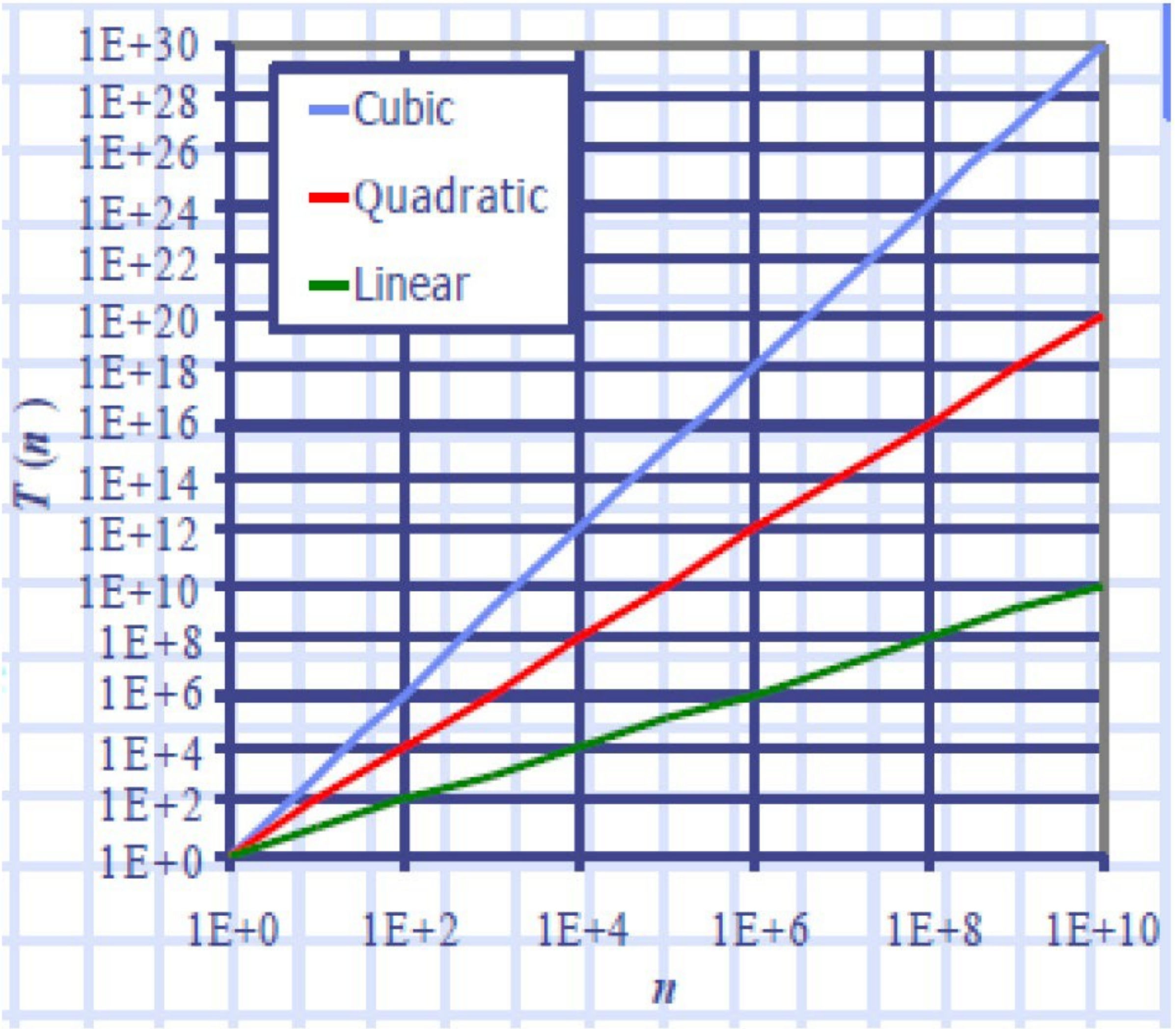
An algorithm with an asymptotically slow running time is beaten in the long run by an algorithm with an asymptotically faster running time.

Growth rate

Growth rates of functions:

- Linear ≈n
- Quadratic $\approx n^2$
- Cubic $\approx n^3$

In a log-log chart, the slope of the line corresponds to the growth rate of the function



Growth rate

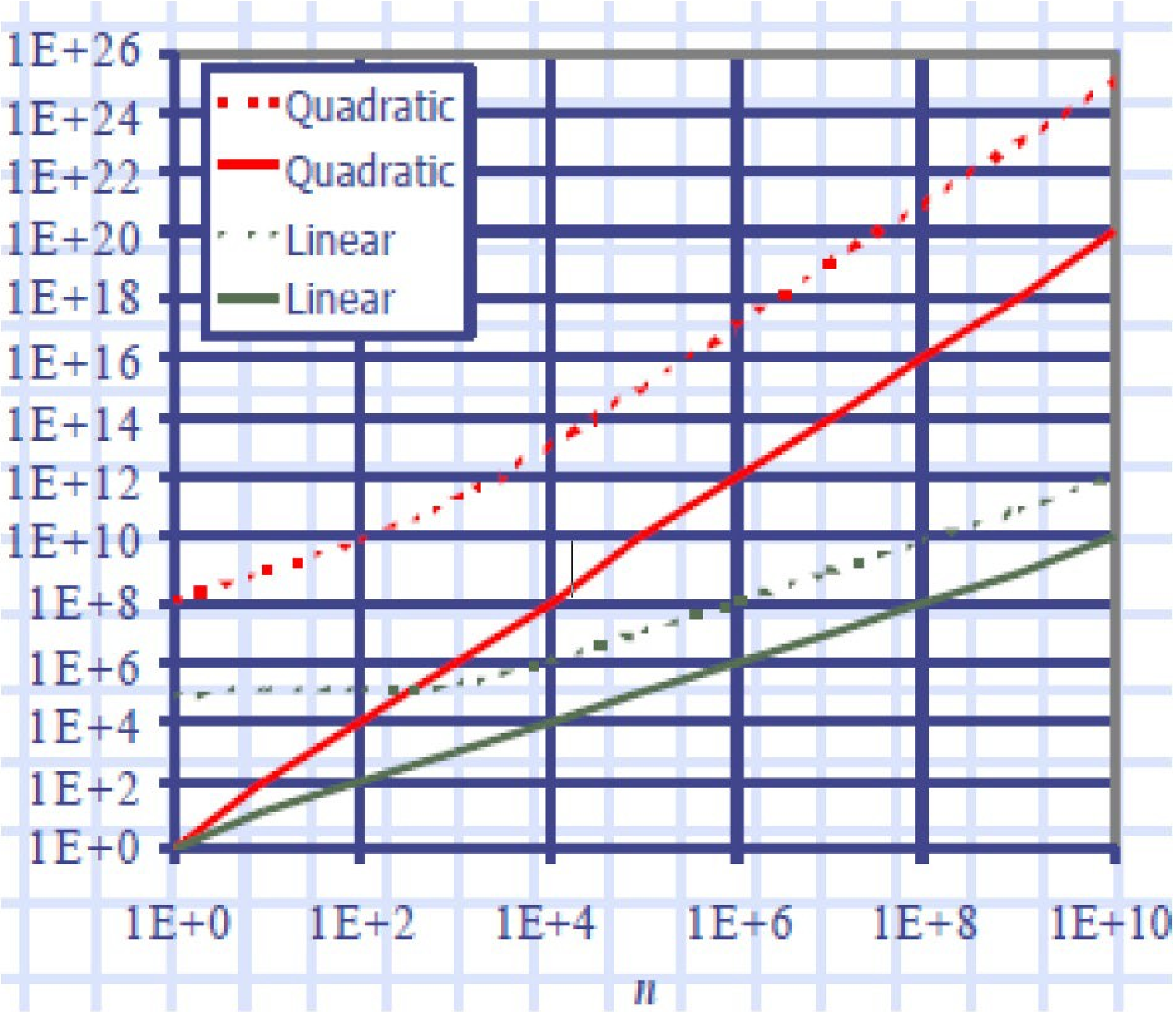
The growth rate is not affected by

•constant factors or

•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

“Big-Oh” notation is probably the most commonly used form of asymptotic notation.

- Given two positive functions $f(n)$ and $g(n)$ (defined on the nonnegative integers), we say f (n) is O(g(n)), written f (n) ∈O(g(n)), if there are constants c and n_0 such that:

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

“Big-Oh” Notation

- Given two positive functions $f(n)$ and $g(n)$ (defined on the nonnegative integers), we say f (n) is O(g(n)), written f (n) ∈O(g(n)), if there are constants c and n_0 such that:

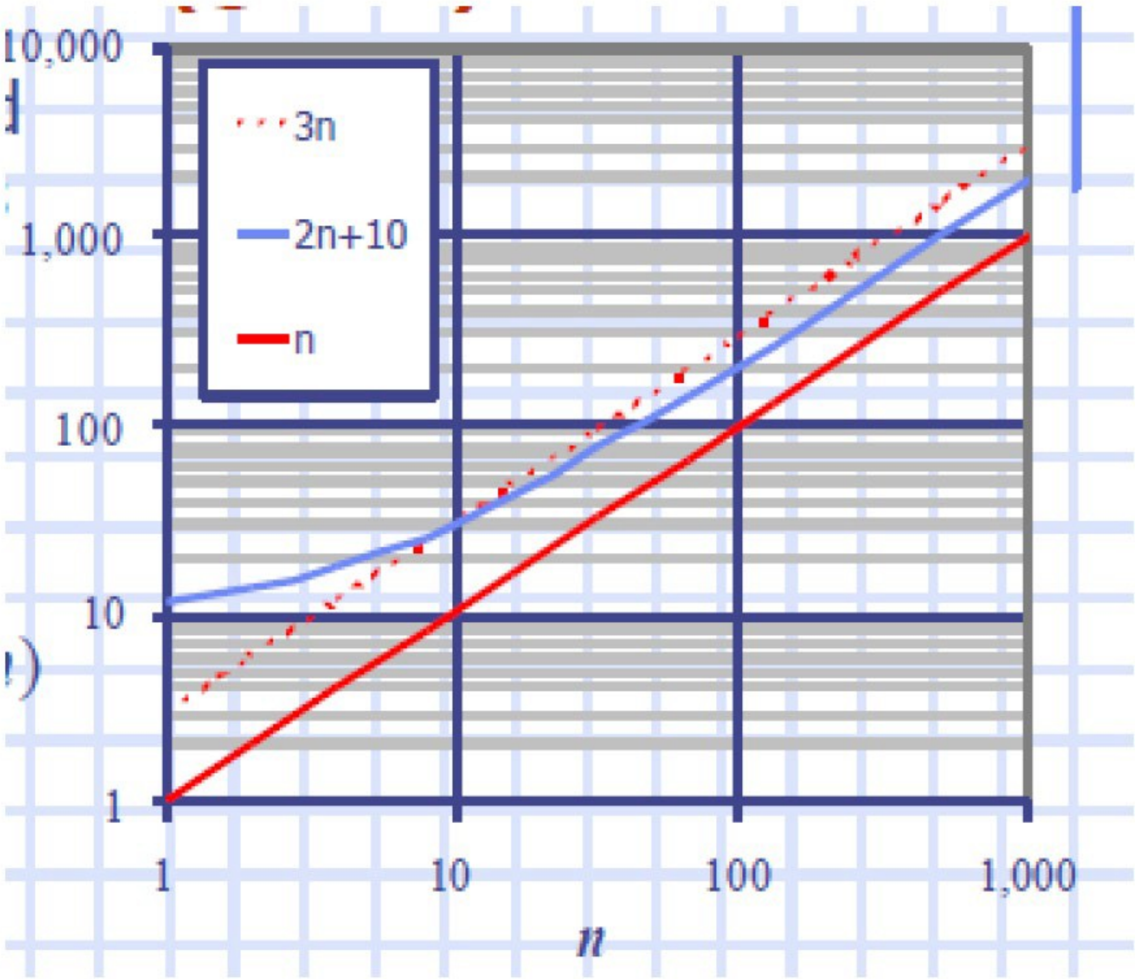
$$f(n) \le c \cdot g(n) \text{ for all } n \ge n_0.$$

Example: $2\boldsymbol{n} + 10$ is $\boldsymbol{O}(\boldsymbol{n})$

"Big-Oh" Notation

Example: 2$\boldsymbol{n}$ + 10 is $\boldsymbol{O}$($\boldsymbol{n}$)

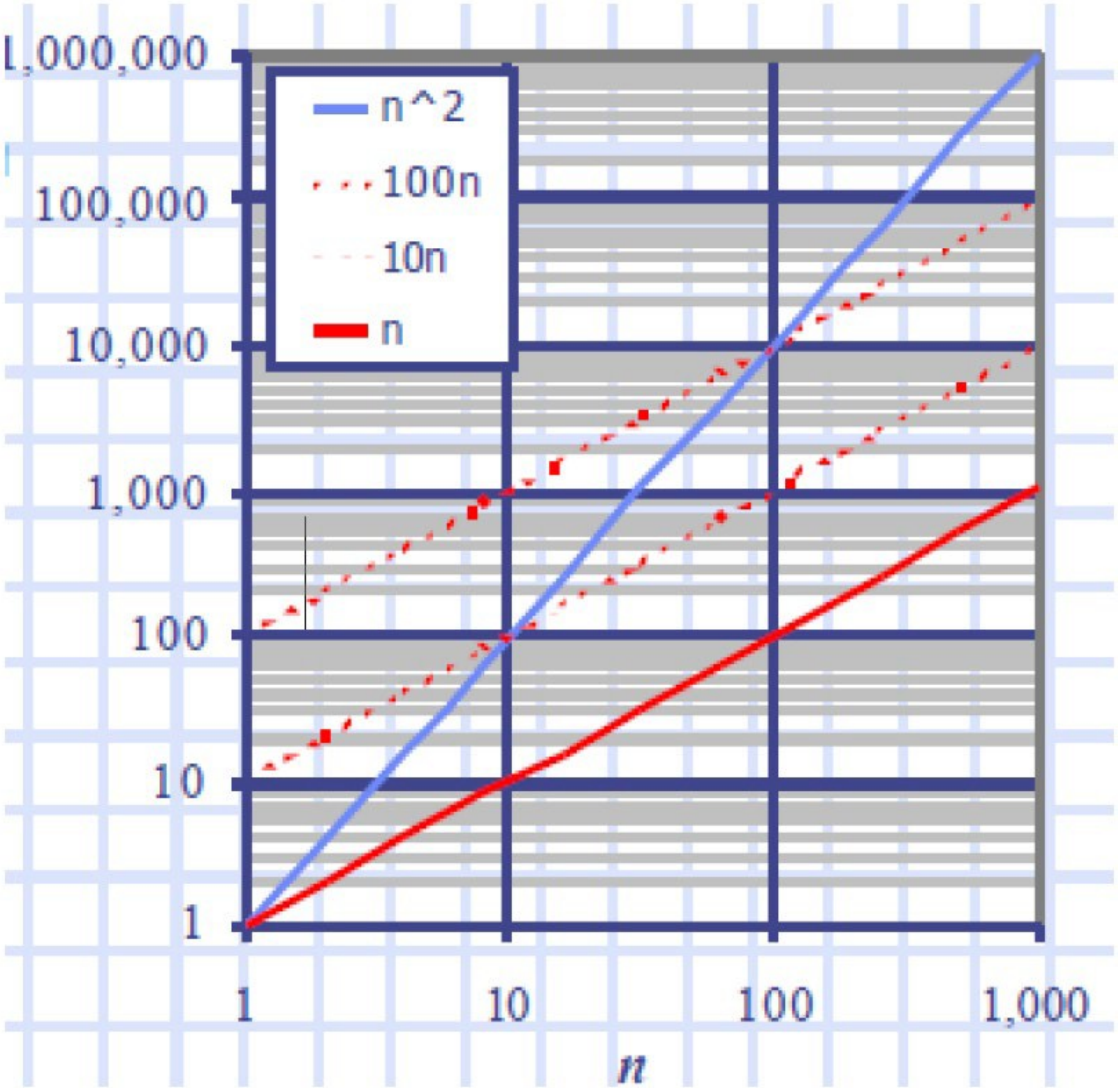
- 2$\boldsymbol{n}$ + 10 $\leq$ $\boldsymbol{cn}$
- ($\boldsymbol{c}$ - 2) $\boldsymbol{n}$ $\geq$ 10
- $\boldsymbol{n}$ $\geq$ 10/($\boldsymbol{c}$ - 2)
- Pick $\boldsymbol{c}$ = 3 and $\boldsymbol{n_0}$= 10



“Big-Oh” Notation

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



Growth rates (running time)

Functions ordered by growth rate:

$\log_2 n$	$N^{1/2}$	n	$n \log_2 n$	n^2	n^3	2^n
1	1.4	2	2	4	8	4
2	2	4	8	16	64	16
3	2.8	8	24	64	512	256
4	4	16	64	256	4096	65536
5	5.7	32	160	1024	32768	4294967296
6	8	64	384	4096	262144	1.84^{19}
7	11	128	896	16384	2097152	2.40×10^{38}
8	16	256	2048	65536	16777216	1.15^{77}
9	23	512	4608	262144	134217728	1.34×10^{154}
10	32	1024	10240	1048576	1073741824	1.79^{308}

More "Big-Oh" Examples

- 7n-2

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

  need **c** > 0 and $\boldsymbol{n_0} \geq 1$ such that $7\boldsymbol{n}$-2 $\leq$ $\mathbf{c}\boldsymbol{n}$ for $\boldsymbol{n} \geq \boldsymbol{n_0}$

  this is true for **c** = 7 and $\boldsymbol{n_0} = 1$

- $3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5$

  $3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5$ is $\mathbf{O}(\boldsymbol{n}^3)$

  need **c** > 0 and $\boldsymbol{n_0} \geq 1$ such that $3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5 \leq \mathbf{c}\boldsymbol{n}^3$ for $\boldsymbol{n} \geq \boldsymbol{n_0}$

  this is true for **c** = 4 and $\boldsymbol{n_0} = 21$

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

Common functions

Here is a list of classes of functions that are commonly encountered when analyzing algorithm

- *Constant O(1)*
- *Logarithmic* $O(\log n)$
- *Linear* $O(n)$
- *Log-linear* $O(n\log n)$
- *Quadratic* $O(n^2)$
- Cubic $O(n^3)$
- *Polynomial* $O(n^k)$
- *Exponential* $O(a^n), a > 1$
- Factorial O(n!)

Big-Oh Rules

2***n*** is ***O***(n^2) ?

- If ***f***(***n***) is a polynomial of degree ***d***, then ***f***(***n***) is ***O***(n^d), i.e.,
  - Drop lower-order terms
  - Drop constant factors
- Use the smallest possible class of functions
  - Say "2***n*** is ***O***(***n***)" instead of "2***n*** is ***O***(n^2)"
- Use the simplest expression of the class
  - Say "3***n***+ 5 is ***O***(***n***)" instead of "3***n*** + 5 is ***O***(3***n***)"

Further examples of Big-Oh

1. 13 n^3 + 7*n* log *n* + 3 is *O*(_ n^3_).

   Proof: 13 n^3 + 7*n* log *n* + 3 ≤ 16 n^3, for n ≥1

2. 3 log *n* + log log *n* is *O*(_log n__).

   Proof: 3 log *n* + log log *n* ≤ 4 log n, for n ≥ 2

3. 2^{70} is *O*(_ 1_).

   Proof: $2^{70} \leq 2^{70}$ *1, for n ≥1.

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:

- We determine that the algorithm "Maximum-Element(A) " executes at most $7n - 2$ primitive operations
- We say that algorithm "runs in $O(n)$ time"

Asymptotic Algorithm Analysis: Example

	#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
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Algorithm prefixAverage1 runs in O(?)

Asymptotic Algorithm Analysis: Example

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Asymptotic Algorithm Analysis: Example

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
$s \leftarrow s + X[i]$	n
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Algorithm prefixAverage2 O(n)

Asymptotic Algorithm Analysis: Exercises

1 Give a **big-Oh** characterization, in terms of n, of the running time of the method Loop1.

2 Perform a similar analysis for method Loop2.

3 Perform a similar analysis for method Loop3.

4 Perform a similar analysis for method Loop4.

Algorithm Loop1(n):

$s \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $s \leftarrow s + i$

Algorithm Loop2(n):

$p \leftarrow 1$
for $i \leftarrow 1$ **to** $2n$ **do**
    $p \leftarrow p \cdot i$

Algorithm Loop3(n):

$p \leftarrow 1$
for $i \leftarrow 1$ **to** n^2 **do**
    $p \leftarrow p \cdot i$

Algorithm Loop4(n):

$s \leftarrow 0$
for $i \leftarrow 1$ **to** $2n$ **do**
    for $j \leftarrow 1$ **to** i **do**
        $s \leftarrow s + i$

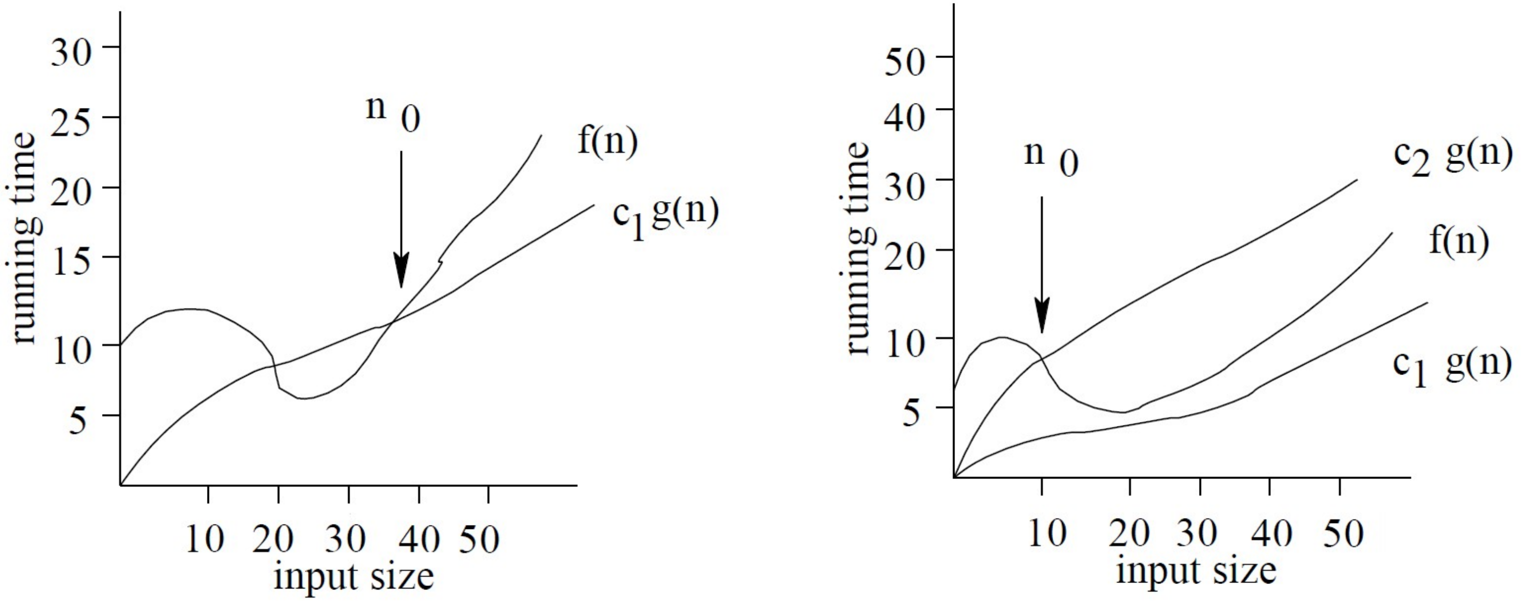
$\Omega(n)$ and $\Theta(n)$ notation

- We say that $f(n)$ is $\Omega(g(n))$ (*big-Omega*) if there are real constants c and n_0 such that:

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

- We say that $f(n)$ is $\Theta(g(n))$ (*Theta*) if $f(n)$ is $\Omega(g(n))$ and

  $f(n)$ is also $O(g(n))$.



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)

Examples

1. $3\log n + \log\log n$ is $\Omega($_ log n___).

Proof: $3\log n + \log\log n \geq 3\log n$, for $n \geq 2$.

2. $3\log n + \log\log n$ is $\Theta($_log n ___).

Space Complexity

- Space complexity is a measure of the amount of working storage an algorithm needs. That means how much memory, in the worst case, is needed at any point in the algorithm.

- As with time complexity, we‘re mostly concerned with how the space needs grow, in big-Oh terms, as the size N of the input problem grows.

Space Complexity

```
int sum(int x, int y, int z) {
    int r = x + y + z;
    return r;
}
```

requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so this is O(1).

Space Complexity

```
int sum(int a[], int n) {
   int r = 0;
   for (int i = 0; i < n; ++i) {
        r += a[i];
     }
   return r;
 }
```

requires N units for a, plus space for n, r and i, so it's O(N).