Algorithms

Introduction
Algorithm Examples
Pseudocode
Order of Growth

Algorithms - what are they

- series of computational steps given an input, to produce an output
- desirable properties:
  - correctness
  - efficient (time, space)
  - elegant
  - easy to implement

Week 1 Objectives

- understand the importance of algorithms
- Example of different solutions for the same problem
  - emphasize running time
- Example of being good at math
- Example of being smart
- Running time analysis, intro
  - Order of growth
  - Big-O notation

Example 1 : MAX

- given an array A, find maximum value

```
input A[1:n]
maxi=1
for i=2:n
    if A[i]>A[maxi] then maxi=i
endfor
return (maxi, A[maxi])
```

- number of comparisons: n-1. Running Time o(n)
- observe correctness
- observe pseudocode

Example 2 : Fibonacci

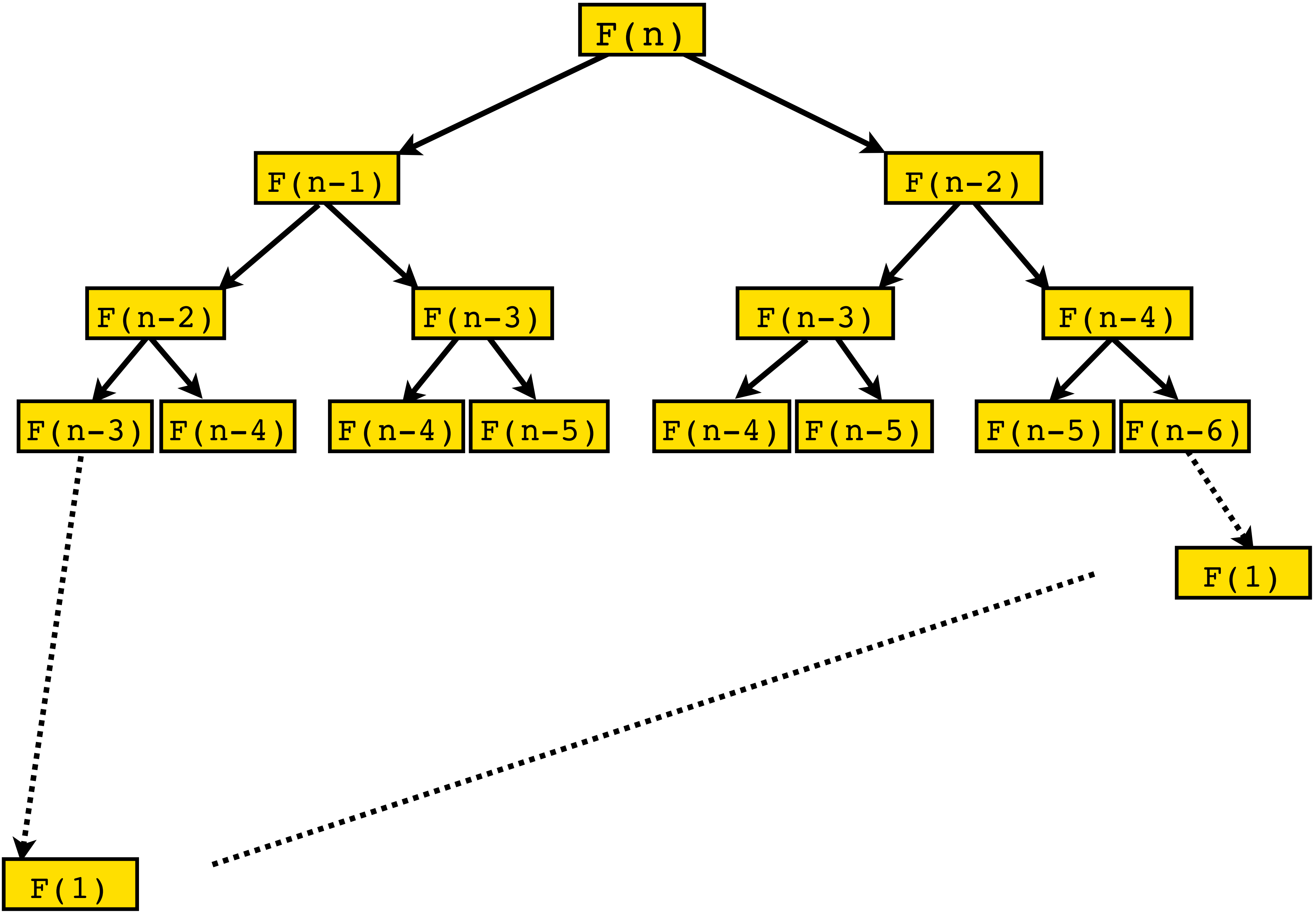
- Fibonacci numbers are defined as
  - $F(0)=0; F(1)=1;$
  - $F(n) = F(n-1) + F(n-2)$ for all $n>1$
- Observe the recursive definition
- Task: given n, calculate F(n)

Fibonacci - recursive solution

```
Fib(n)
    if n<2
        return n
    endif
    val = Fib(n-1) + Fib(n-2)
return val
```

- correct
- exponential running time (bad)
    - see recursion tree

Fibonacci - recursive solution tree



- tree= stack of function calls
- n levels on the left
- n/2 levels on the right
- at least n/2 levels full binary tree
  - at least $2^{(n/2)}$ calls

Fibonacci – array solution

```
Fib(n)
    array A[0..n] initialized
    A[0]=0; A[1]=1;
    for i=2:n
        A[i]=A[i-1] + A[i-2];
    endfor
return A[n];
```

- one for loop runs across the array
- inside the loop a constant time operation O(1)
- overall linear time O(n)

Fibonacci - Matrix Multiplication

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \qquad M^k = \begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix}$$

- proof by mathematical induction

$$M^{k+1} = M * M^k = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix}$$

$$= \begin{bmatrix} F_{k+1} + F_k & F_k + F_{k-1} \\ F_{k+1} & F_k \end{bmatrix} = \begin{bmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{bmatrix}$$

- so we have to multiply M with itself n times
  - how fast can it be done?
  - naively : each multiplication of 2x2 matrixes takes constant O(1) time so linear time O(n) total

Fibonacci - Matrix Multiplication

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

- want to compute M^n : multiply M with itself n times
  - each multiplication of 2x2 matrixes takes constant O(1) time
- idea: repeated squaring
  - $M^2=M*M$; $M^4=M^2*M^2$; $M^8=M^4*M^4$ etc
- then multiply only the powers needed
  - for example n=13 gives $M^{13}=M^8*M^4*M$

Fibonacci - Matrix Multiplication

```
Fib(n)
    init $M = \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}$ $A = \begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}$
    for i=0:log(n)
        if(n%2 ==1) A=A*M; endif //add this power of M only if the respective bit in n is 1
        M=M*M; // get the next squaring M
        n=n/2 // move on to the next bit(right to left) in n - think of n represented in binary
    endfor
return A[1,1]
```

- only log(n) iterations in for loop, each constant time
    - logarithmic time O(log n) total

Fibonacci - generative function

- use the generative function (requires analytic solution)

$$F_n = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

- where

$$\phi = \frac{1+\sqrt{5}}{2}; \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

- practically constant time, if scalar expoential is done with a dedicated math processor

Conclusions

- Algorithms matter, even if the problem is very simple
- They matter a lot if the problem is BIG
  - think of big data today, or the web search
- Analysis of algorithms: running time, space requirements, bottlenecks
- Implementation makes a difference too

CheckPoint

- Consider the first Fibonacci solution (recursion) vs the second (array)
  - how is it possible to reduce an exponential number of computations to a linear number?
  - are some of the computations in the first solutions not necessary?
  - can you speed up the recursion for the first solution?

Matrix multiplication

- multiply nxn matrices

$$C = AB$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

- running time $\Theta(n^3)$
  - $\Theta(n^3)$ means actual number of multiplications $T(n)$ is about n^3
  - $C_1 * n^3 \leq T(n) \leq C_2 * n^3$, for fixed constants C_1 and C_2
  - count the number of multiplications

Strassen's Algorithm

- n=2: Multiplay 2x2 matrix using 7 multiplications instead of 8

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & g \\ f & h \end{bmatrix}$$

- Strassen's equations

$$\begin{aligned} P1 &= a(g-h) \\ P2 &= (a+b)h \\ P3 &= (c+d)e \\ P4 &= d(f-e) \\ P5 &= (a+d)(e+h) \\ P6 &= (b-d)(f+h) \\ P7 &= (a-c)(e+g) \end{aligned}$$

$$\begin{aligned} r &= P5+P4-P2+P6 \\ s &= P1+P2 \\ t &= P3+P4 \\ u &= P5+P1-P3-P7 \end{aligned}$$

Strassen's Algorithm

- divide : partition A, B each in four $^n/_2 \times {}^n/_2$ matrices
- conquer: perform 7 multiplications
  - each multiplication of 2 matrices of size $^n/_2$, done recursively with divide-conquer mechanism for n= $^n/_2$
- combine: find C=AxB using Strassen's equations
- T(n) = time to multiply nxn matrices
  - recursively: $T(n) = 7T(n/2) + \Theta(n^2)$
  - how to solve this recursion?

Running time

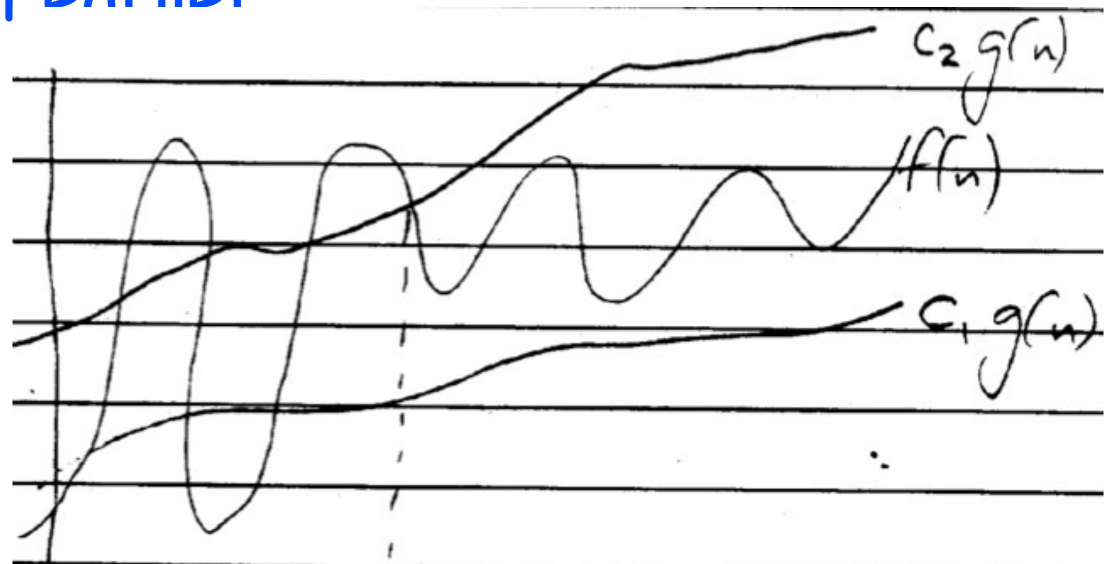
- Solve equation $T(n) = 7T(n/2) + \Theta(n^2)$ as order of growth
  - no interest in $T(1)$, $T(2)$ etc, but the general growth of the function
- solution next module: $T(n)$ is like $n^{\log(7)}$
  - approx $n^{2.81}$, better than n^3 the running time of multiplication
- that means $C_1 * n^{\log(7)} < T(n) < C_2 * n^{\log(7)}$, for some constants C_1 and C_2, for $n \geq n_0$ some starting value

checkpoint: matrix multiplication

- verify that Strassen's equation produce indeed the correct matrix multiplication

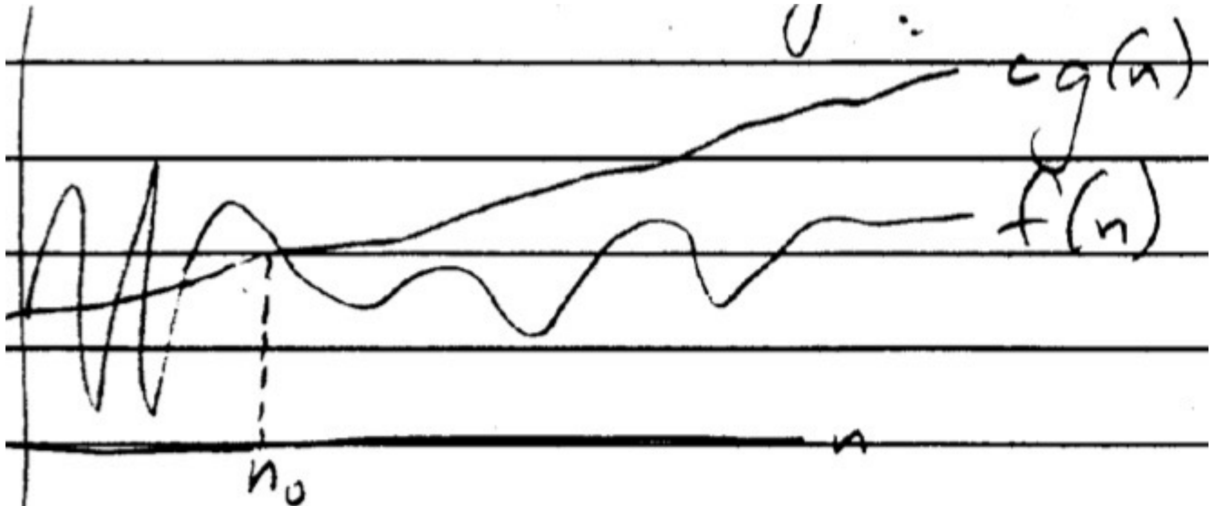
Asymptotic Notation : Θ

- $f(n) = \Theta(g(n))$ if $C_1 g(n) \le f(n) \le C_2 g(n)$
  - for some positive constants C_1 and C_2, and starting at $n \ge n_0$
  - $T(n)$ for Strassen's multiplication is $\Theta(n^{\log(7)})$
  - we cannot compute $T(n)$ exactly, but we know its growing like constant* $n^{\log(7)}$
  - example: $f(n)={}^{1}/{}_{2}*n^2-2n$ is $\Theta(n^2)$
- a simple loop through data = linear algorithm
  - $\Theta(n)$ or growing like constant*n
  - for example the MAX algorithm earlier



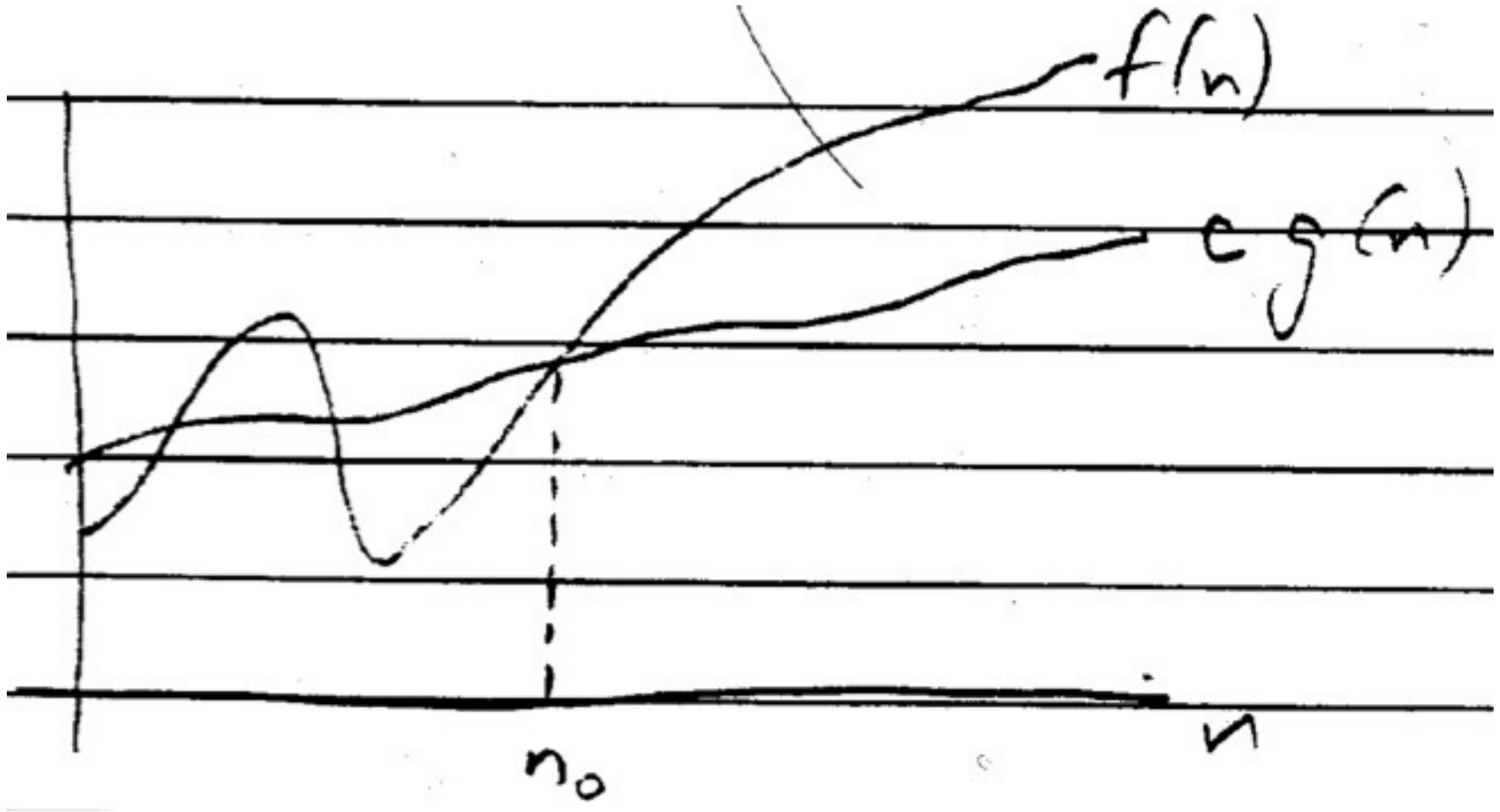
Asymptotic Notation : "big" O

- $f(n) = O(g(n))$ if $f(n) \le C_2 g(n)$
  - for some positive constant C_2, and starting at $n \ge n_0$
  - only bounding T(n) up, not down
  - "worst case" = longest running time
  - worst case not worse than g(n) growth
  - if T(n) is Θ(g(n)), then T(n) is also O(g(n)), but not the converse!
- expression $f(n) = n^2 + O(n)$, or n^2 plus "linear"
  - means $f(n) \le n^2 + C_2 n$, for some constant C_2, and initial n_0

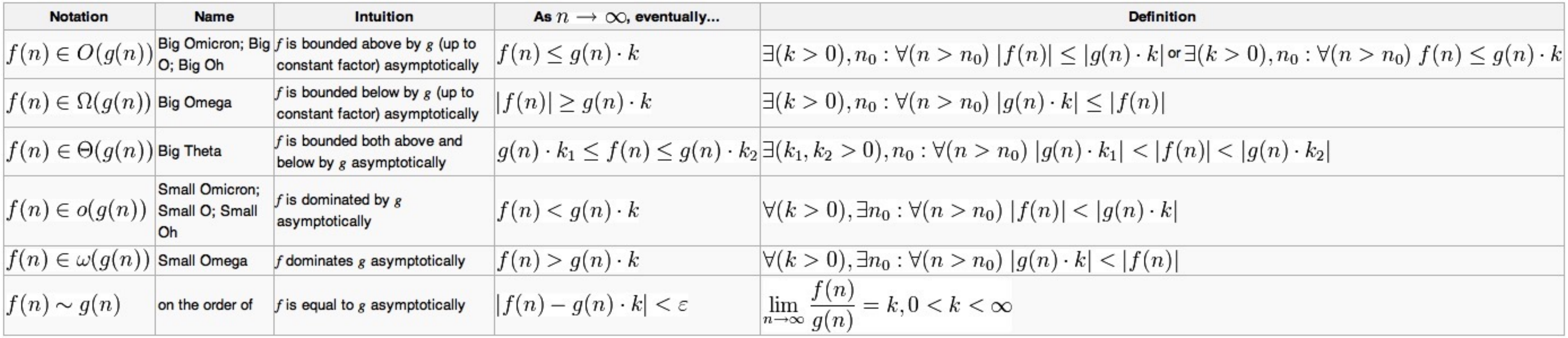


Asymptotic Notation : Ω

- lower bound: $f(n) = \Omega(g(n))$ if $f(n) > C_1 g(n)$
  - for some positive constant C_1, and starting at $n \geq n_0$
  - example: $f(n)=n^2$ is $\Omega(n\log(n))$

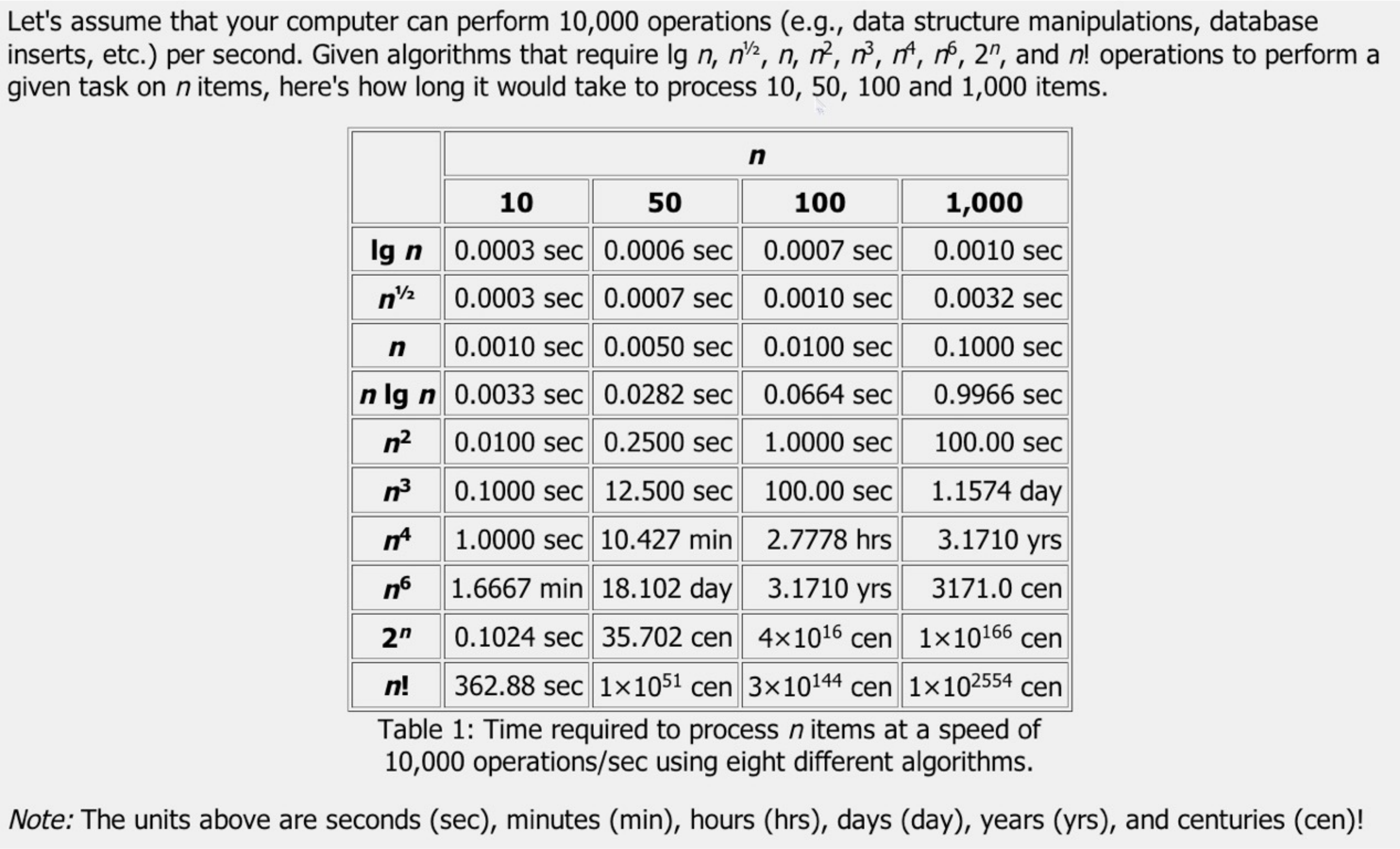


Asymptotic Notation : summary



Notation	Name	Intuition	As $n \to \infty$, eventually...	Definition
$f(n) \in O(g(n))$	Big Omicron; Big O; Big Oh	f is bounded above by g (up to constant factor) asymptotically	$f(n) \le g(n) \cdot k$	$\exists (k > 0), n_0 : \forall (n > n_0)\ \lvert f(n)\rvert \le \lvert g(n) \cdot k\rvert$ or $\exists (k > 0), n_0 : \forall (n > n_0)\ f(n) \le g(n) \cdot k$
$f(n) \in \Omega(g(n))$	Big Omega	f is bounded below by g (up to constant factor) asymptotically	$\lvert f(n)\rvert \ge g(n) \cdot k$	$\exists (k > 0), n_0 : \forall (n > n_0)\ \lvert g(n) \cdot k\rvert \le \lvert f(n)\rvert$
$f(n) \in \Theta(g(n))$	Big Theta	f is bounded both above and below by g asymptotically	$g(n) \cdot k_1 \le f(n) \le g(n) \cdot k_2$	$\exists (k_1, k_2 > 0), n_0 : \forall (n > n_0)\ \lvert g(n) \cdot k_1\rvert < \lvert f(n)\rvert < \lvert g(n) \cdot k_2\rvert$
$f(n) \in o(g(n))$	Small Omicron; Small O; Small Oh	f is dominated by g asymptotically	$f(n) < g(n) \cdot k$	$\forall (k > 0), \exists n_0 : \forall (n > n_0)\ \lvert f(n)\rvert < \lvert g(n) \cdot k\rvert$
$f(n) \in \omega(g(n))$	Small Omega	f dominates g asymptotically	$f(n) > g(n) \cdot k$	$\forall (k > 0), \exists n_0 : \forall (n > n_0)\ \lvert g(n) \cdot k\rvert < \lvert f(n)\rvert$
$f(n) \sim g(n)$	on the order of	f is equal to g asymptotically	$\lvert f(n) - g(n) \cdot k\rvert < \varepsilon$	$\lim_{n \to \infty} \frac{f(n)}{g(n)} = k, 0 < k < \infty$

Ten orders of growth



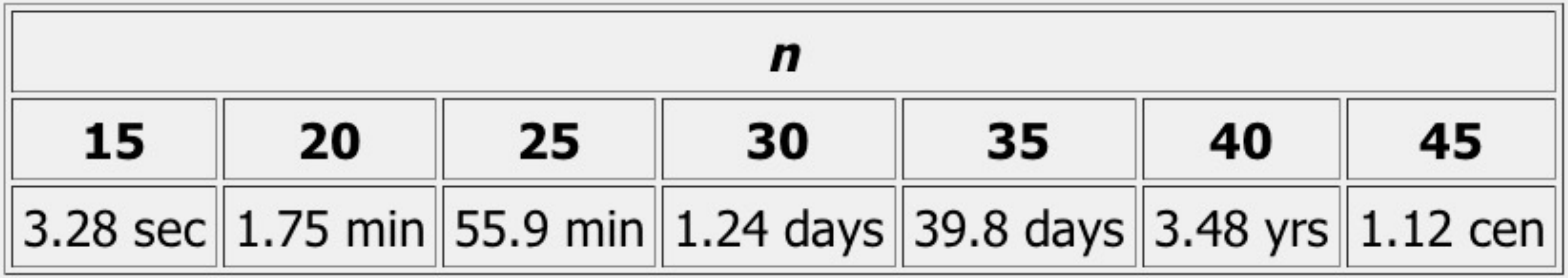
Let's assume that your computer can perform 10,000 operations (e.g., data structure manipulations, database inserts, etc.) per second. Given algorithms that require lg n, $n^{1/2}$, n, n^2, n^3, n^4, n^6, 2^n, and $n!$ operations to perform a given task on n items, here's how long it would take to process 10, 50, 100 and 1,000 items.

	n			
	10	**50**	**100**	**1,000**
lg *n*	0.0003 sec	0.0006 sec	0.0007 sec	0.0010 sec
$n^{1/2}$	0.0003 sec	0.0007 sec	0.0010 sec	0.0032 sec
n	0.0010 sec	0.0050 sec	0.0100 sec	0.1000 sec
n* lg *n	0.0033 sec	0.0282 sec	0.0664 sec	0.9966 sec
n^2	0.0100 sec	0.2500 sec	1.0000 sec	100.00 sec
n^3	0.1000 sec	12.500 sec	100.00 sec	1.1574 day
n^4	1.0000 sec	10.427 min	2.7778 hrs	3.1710 yrs
n^6	1.6667 min	18.102 day	3.1710 yrs	3171.0 cen
2^n	0.1024 sec	35.702 cen	4×10^{16} cen	1×10^{166} cen
$n!$	362.88 sec	1×10^{51} cen	3×10^{144} cen	1×10^{2554} cen

Table 1: Time required to process n items at a speed of 10,000 operations/sec using eight different algorithms.

Note: The units above are seconds (sec), minutes (min), hours (hrs), days (day), years (yrs), and centuries (cen)!

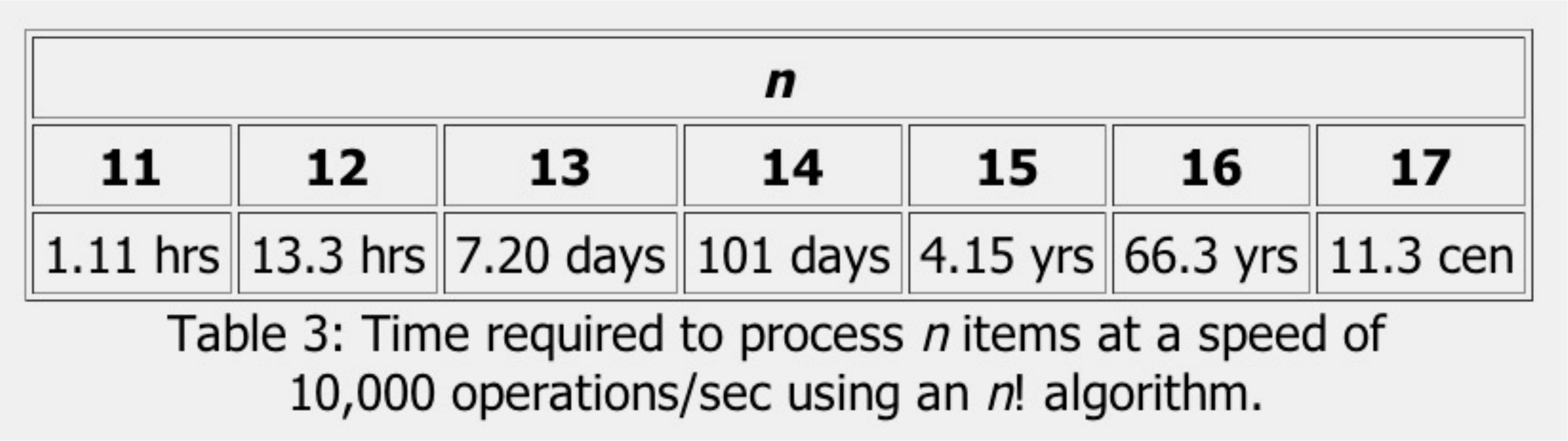
Explosive growth of exponential



n						
15	**20**	**25**	**30**	**35**	**40**	**45**
3.28 sec	1.75 min	55.9 min	1.24 days	39.8 days	3.48 yrs	1.12 cen

Table 2: Time required to process *n* items at a speed of 10,000 operations/sec using a 2^n algorithm.

Even more explosive n!



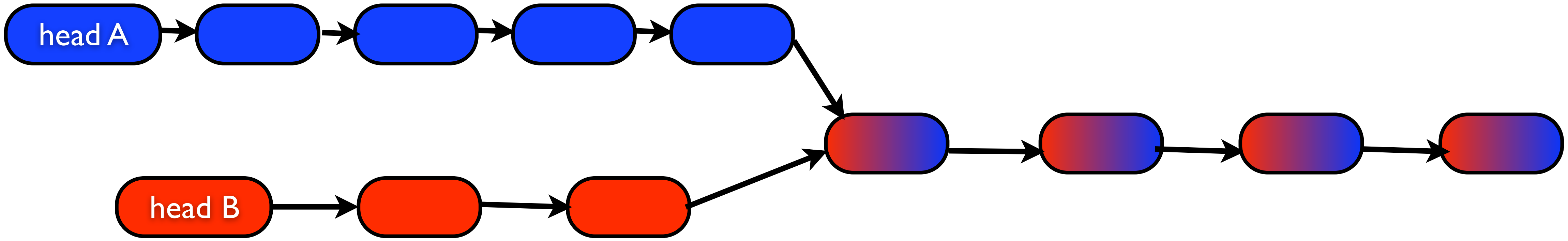
n						
11	**12**	**13**	**14**	**15**	**16**	**17**
1.11 hrs	13.3 hrs	7.20 days	101 days	4.15 yrs	66.3 yrs	11.3 cen

Table 3: Time required to process *n* items at a speed of 10,000 operations/sec using an *n*! algorithm.

CheckPoint: Order of growth

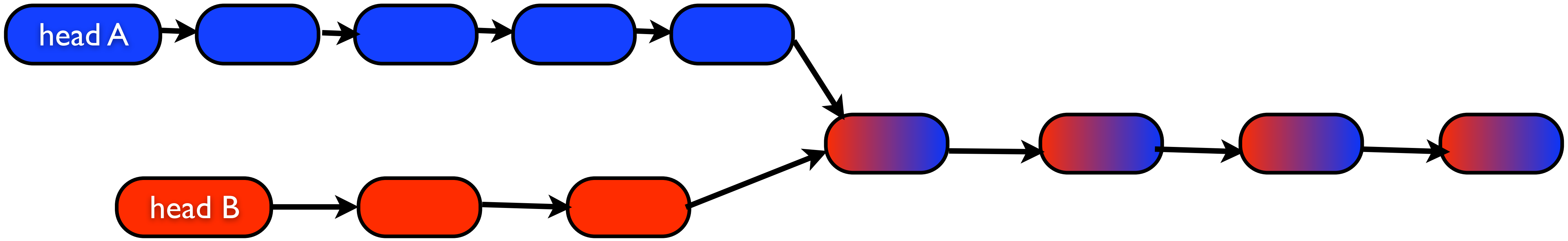
- who is growing faster ?
  - $f(n)=n^{1/2}$ or $g(n)=2\log(n)$
  - $f(n)=n^{1/3}$ or $g(n) = [\log(n)]^3$
  - $f(n) = 2^{(2^n)}$ or $g(n)=n!$
- explain equation $T(n) = 7T(n/2) + \Theta(n^2)$
- MergeSort (size n) : solve 2 problems of size T(n/2), then combine result in linear time. What is the recursive equation for the running time T(n) ?

Being Smart: list intersection



- You are given the two head-nodes `headA` and `headB` of two single-linked lists that are known to intersect
  - after intersection they are identical due to the linkage nature
- Task: find the intersection node
  - cannot modify the lists, or use auxiliary data structures

Being Smart: list intersection

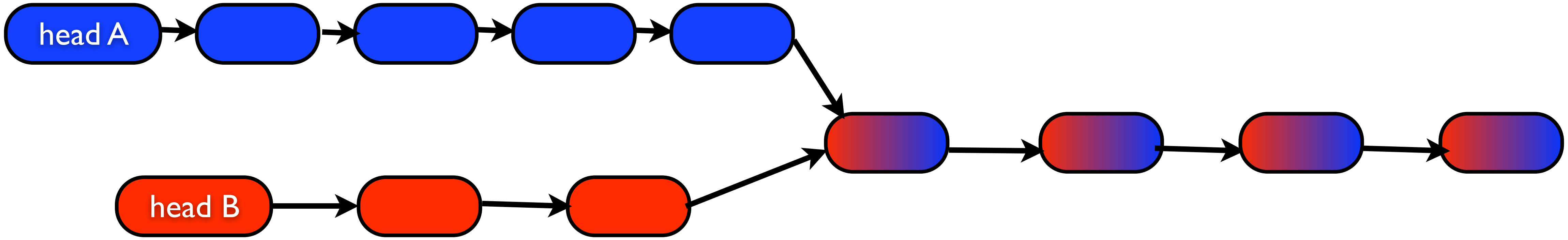


- naive solution:

```
for each a=node of first list (traversal)
    for each b=node of the second list (traversal)
        if a==b return “found intersection node: a”
    end second for
end first for
```

- such solution runs in O(mn) quadratic time, if m and n are the lengths of the two lists
    - the first loop takes up to m steps to iterate to the first list
    - the second loop takes n steps; it runs for each step of the first loop

Being Smart: list intersection



- smart solution:
  - `traverse the first list to count it, obtain m;` *m=9 in example*
  - `traverse the second list to count it, obtain n;` *n=7 in example*
  - `if m>n traverse first list for exactly m-n nodes;` *m-n=2 in example*
    - `if n>m traverse second list for exactly n-m elements`
  - `traverse the list simultaneously until the intersection node` *// in example this simultaneous traversal starts at third blue and first red*
- smart solution runs in O(m+n) linear time