CPSC 259: Data Structures and Algorithms for Electrical Engineers

Algorithm Analysis

Textbook References:
(a) Thareja (first edition) 4.6-4.7
(b) Thareja (second edition): 2.8 – 2.12

Learning Goals

- Justify which operation(s) we should measure in an algorithm/program in order to estimate its "efficiency".
- Define the "input size" n for various problems, and determine the effect (in terms of performance) that increasing the value of n has on an algorithm.
- Given a fragment of code, write a formula which measures the number of steps executed, as a function of n.
- Define the notion of Big-O complexity, and explain pictorially what it represents.
- Compute the worst-case asymptotic complexity of an algorithm in terms of its input size n, and express it in Big-O notation.

Learning Goals (cont)

- Compute an appropriate Big-O estimate for a given function $T(n)$.
- Discuss the pros and cons of using best-, worst-, and average-case analysis, when determining the complexity of an algorithm.
- Describe why best-case analysis is rarely relevant and how worst-case analysis may never be encountered in practice.
- Given two or more algorithms, rank them in terms of their time and space complexity.

Introduction

- What is Algorithm?
  - a clearly specified set of simple instructions to be followed to solve a problem
    - Takes a set of values, as input and
    - produces a value, or set of values, as output
  - May be specified
    - In English
    - As a computer program
    - As a pseudo-code
- Data structures
  - Methods of organizing data
- Program = algorithms + data structures

Introduction

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

Example: Selection Problem

- Given a list of N numbers, determine the *k*th largest, where $k \leq N$.
- Algorithm 1:
  (1) Read N numbers into an array
  (2) Sort the array in decreasing order by some simple algorithm
  (3) Return the element in position k

Example: Selection Problem…

- Algorithm 2:
  (1) Read the first k elements into an array and sort them in decreasing order
  (2) Each remaining element is read one by one
    - If smaller than the kth element, then it is ignored
    - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.
  (3) The element in the kth position is returned as the answer.

Example: Selection Problem…

- Which algorithm is better when
  - N =100 and k = 100?
  - N =100 and k = 1?
- What happens when N = 1,000,000 and k = 500,000?
- There exist better algorithms

Algorithm Analysis

- We only analyze *correct* algorithms
- An algorithm is correct
  - If, for every input instance, it halts with the correct output
- Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer
- Analyzing an algorithm
  - Predicting the resources that the algorithm requires
  - Resources include
    - Memory
    - Communication bandwidth
    - Computational time (usually most important)

Algorithm Analysis…

- Factors affecting the running time
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - typically, the *input size* (number of items in the input) is the main consideration
      - E.g. sorting problem ⇒ the number of items to be sorted
      - E.g. multiply two matrices together ⇒ the total number of elements in the two matrices
- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations ⇒ Not parallel computers

Efficiency

- Complexity theory addresses the issue of how *efficient* an algorithm is, and in particular, how well an algorithm *scales* as the problem size increases.
- Some measure of efficiency is needed to compare one algorithm to another (assuming that both algorithms are correct and produce the same answers). Suggest some ways of how to measure efficiency.
  - Time (How long does it take to run?)
  - Space (How much memory does it take?)
  - Other attributes?
    - Expensive operations, e.g. I/O
    - Elegance, Cleverness
    - Energy, Power
    - Ease of programming, legal issues, etc.

Analyzing Runtime

```
old2 = 1;
old1 = 1;
for (i=3; i<n; i++) {
    result = old2+old1;
    old1 = old2;
    old2 = result;
}
```

How long does this take?

Analyzing Runtime

```
old2 = 1;
old1 = 1;
for(i=3; i<n; i++){
  result = old2+old1;
  old1 = old2;
  old2 = result;
}
```

Wouldn't it be nice if it didn't depend on so many things?

How long does this take?

IT DEPENDS

- What is n?
- What machine?
- What language?
- What compiler?
- How was it programmed?

Number of Operations

- Let us focus on one complexity measure: the **number of operations** performed by the algorithm on an input of a given **size**.
- What is meant by "number of operations"?
  - # instructions executed
  - # comparisons
- Is the "number of operations" a precise indicator of an algorithm's running time (time complexity)? Compare a "shift register" instruction to a "move character" instruction, in assembly language.
  - No, some operations are more costly than others
- Is it a fair indicator?
  - Good enough

Analyzing Runtime

```
old2 = 1
old1 = 1
for(i=3; i<n; i++){
  result = old2+old1
  old1 = old2
  old2 = result
}
```

How many operations does this take?

IT DEPENDS

- What is n?

- Running time is a function of n such as T(n)
- This is really nice because the runtime analysis doesn't depend on hardware or subjective conditions anymore

Input Size

- What is meant by the input size n? Provide some application-specific examples.
  - Dictionary:
    - # words
  - Restaurant:
    - # customers or # food choices or # employees
  - Airline:
    - # flights or # luggage or # costumers

- We want to express the number of operations performed as a function of the input size n.

Run Time as a Function of **Size of** Input

- But, **which** input?
  - Different inputs of same size have different run times

  E.g., what is run time of linear search in a list?
  - If the item is the first in the list?
  - If it's the last one?
  - If it's not in the list at all?

What should we report?

Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.

Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.

Mostly useless

Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.

Useful, pessimistic

Which Run Time?

Useful, hard to do right

- Average Case (Expected Time)
  - Requires a notion of an "average" input to an algorithm, which uses a probability distribution over possible inputs.
  - Allows discriminating among algorithms with the same worst case complexity
    - Classic example: Insertion Sort vs QuickSort

Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.

Very useful, but ill-defined

Scalability!

- What's more important?
  - At n=5, plain recursion version is faster.
  - At n=3500, complex version is faster.
- Computer science is about solving problems people couldn't solve before. Therefore, the emphasis is almost always on solving the big versions of problems.
- (In computer systems, they always talk about "scalability", which is the ability of a solution to work when things get really big.)

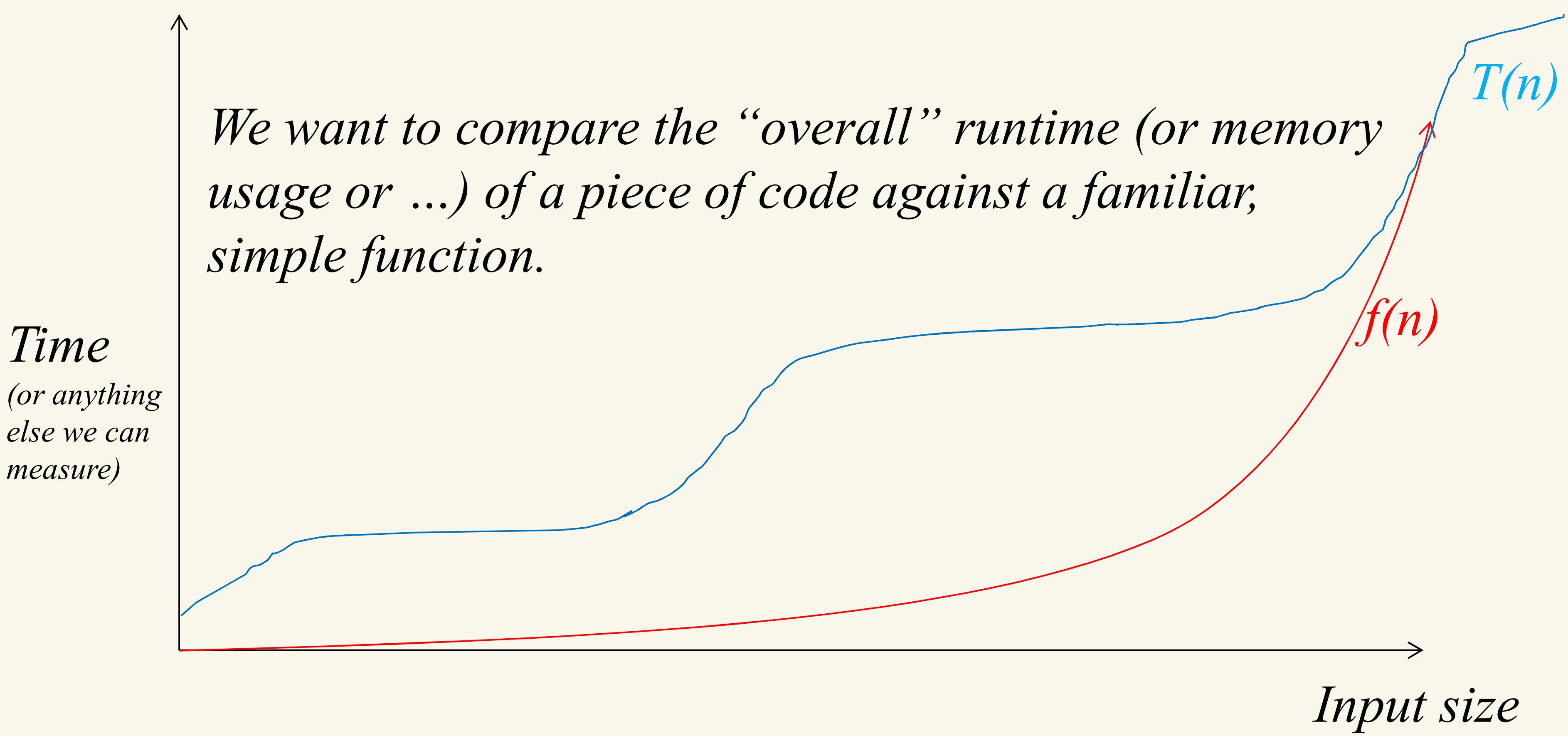
Asymptotic Analysis

- Asymptotic analysis is analyzing what happens to the run time (or other performance metric) as the input size n goes to infinity.
  - The word comes from "asymptote", which is where you look at the limiting behavior of a function as something goes to infinity.
- This gives a solid mathematical way to capture the intuition of emphasizing scalable performance.
- It also makes the analysis a lot simpler!

Big-O (Big-Oh) Notation

- Let $T(n)$ and $f(n)$ be functions mapping $Z^+ \rightarrow R^+$.

Positive integers *Positive real numbers*



Big-O Notation

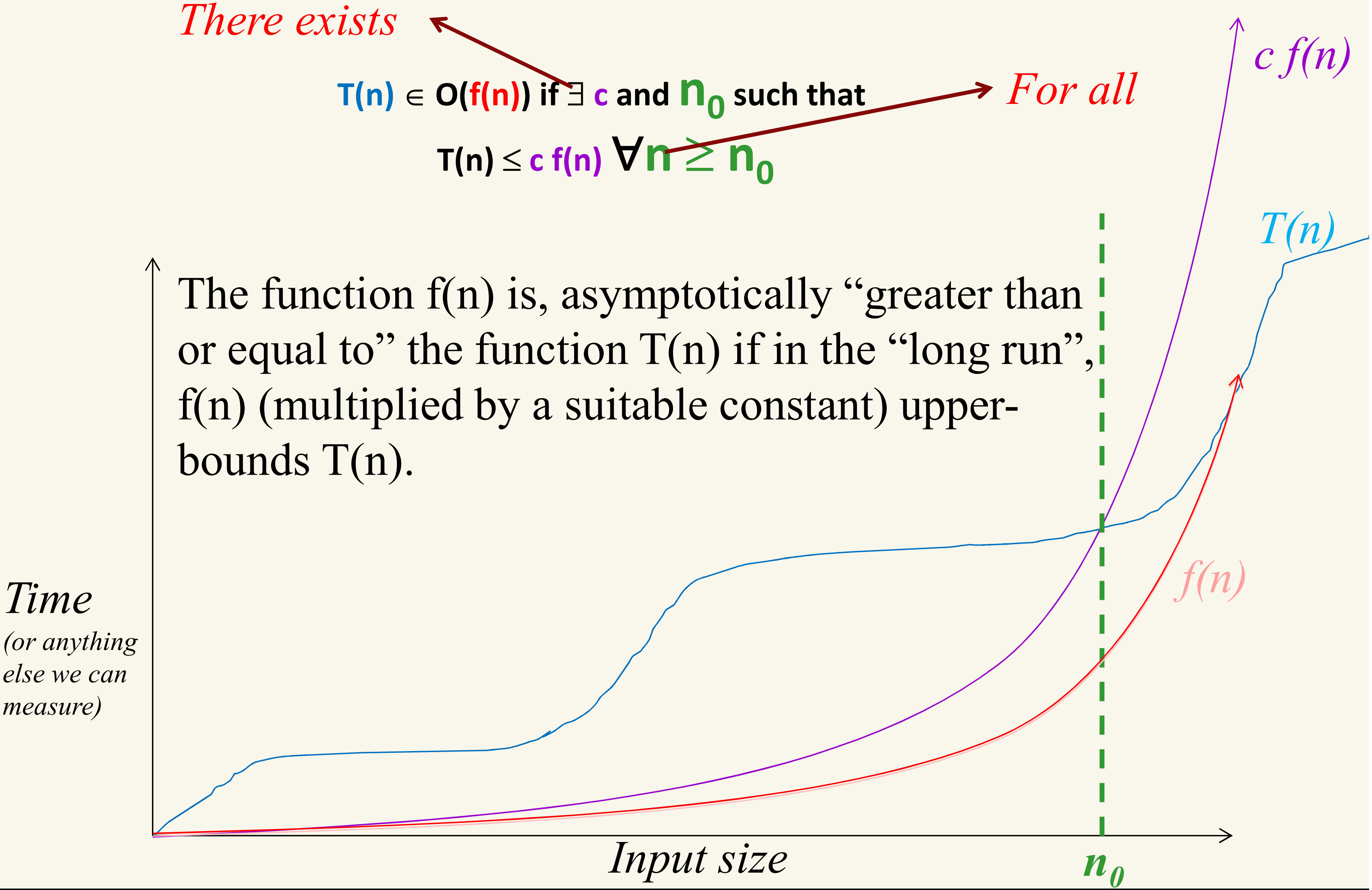
There exists

For all

$T(n) \in O(f(n))$ if $\exists$ c and n_0 such that

$$T(n) \le c\,f(n) \;\; \forall n \ge n_0$$

The function f(n) is, asymptotically "greater than or equal to" the function T(n) if in the "long run", f(n) (multiplied by a suitable constant) upper-bounds T(n).



Big-O Notation

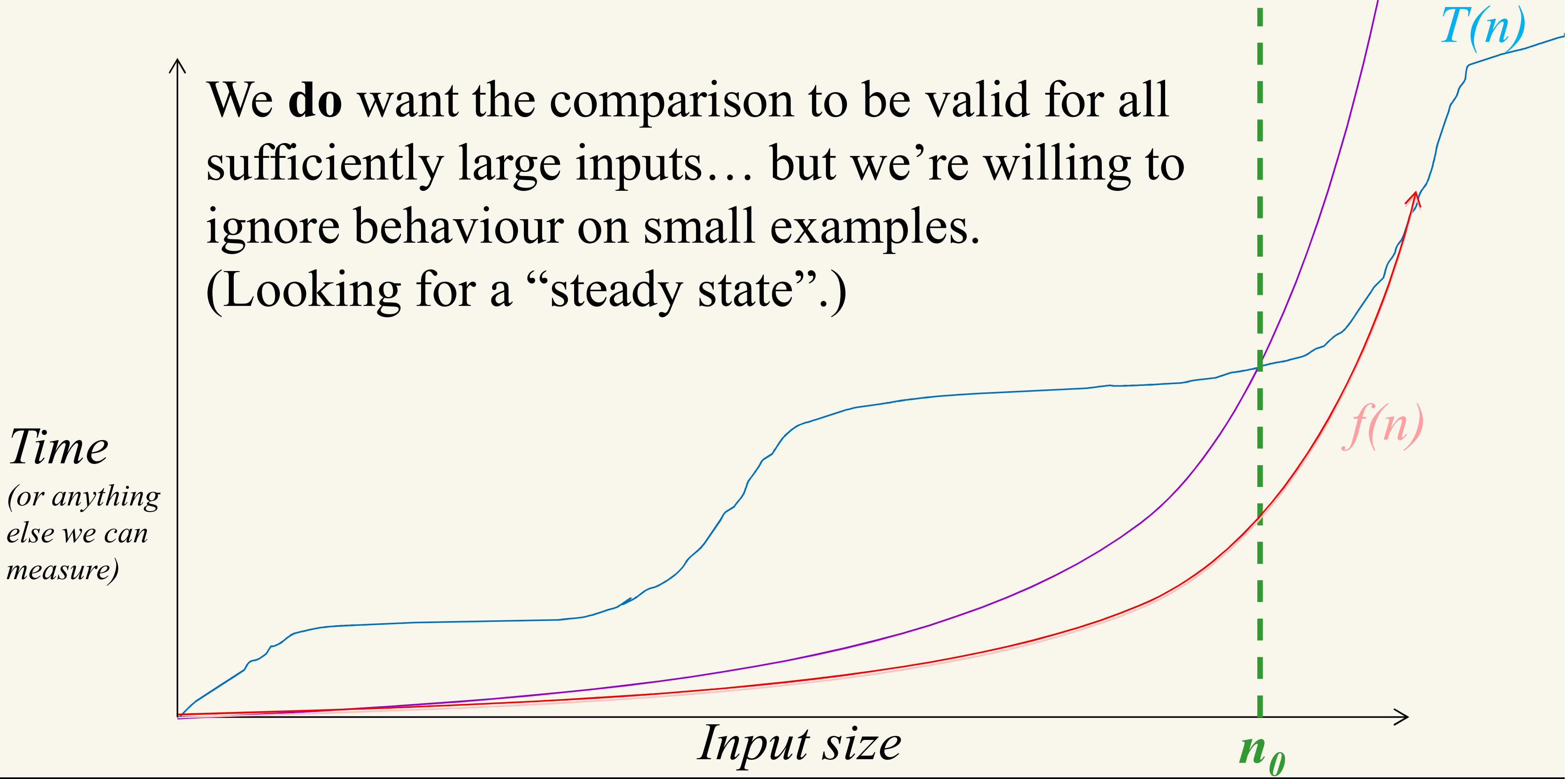
There exists

$T(n) \in O(f(n))$ if $\exists$ c and n_0 such that

$T(n) \leq c\, f(n)\ \forall n \geq n_0$

For all

We **do** want the comparison to be valid for all sufficiently large inputs… but we're willing to ignore behaviour on small examples. (Looking for a "steady state".)



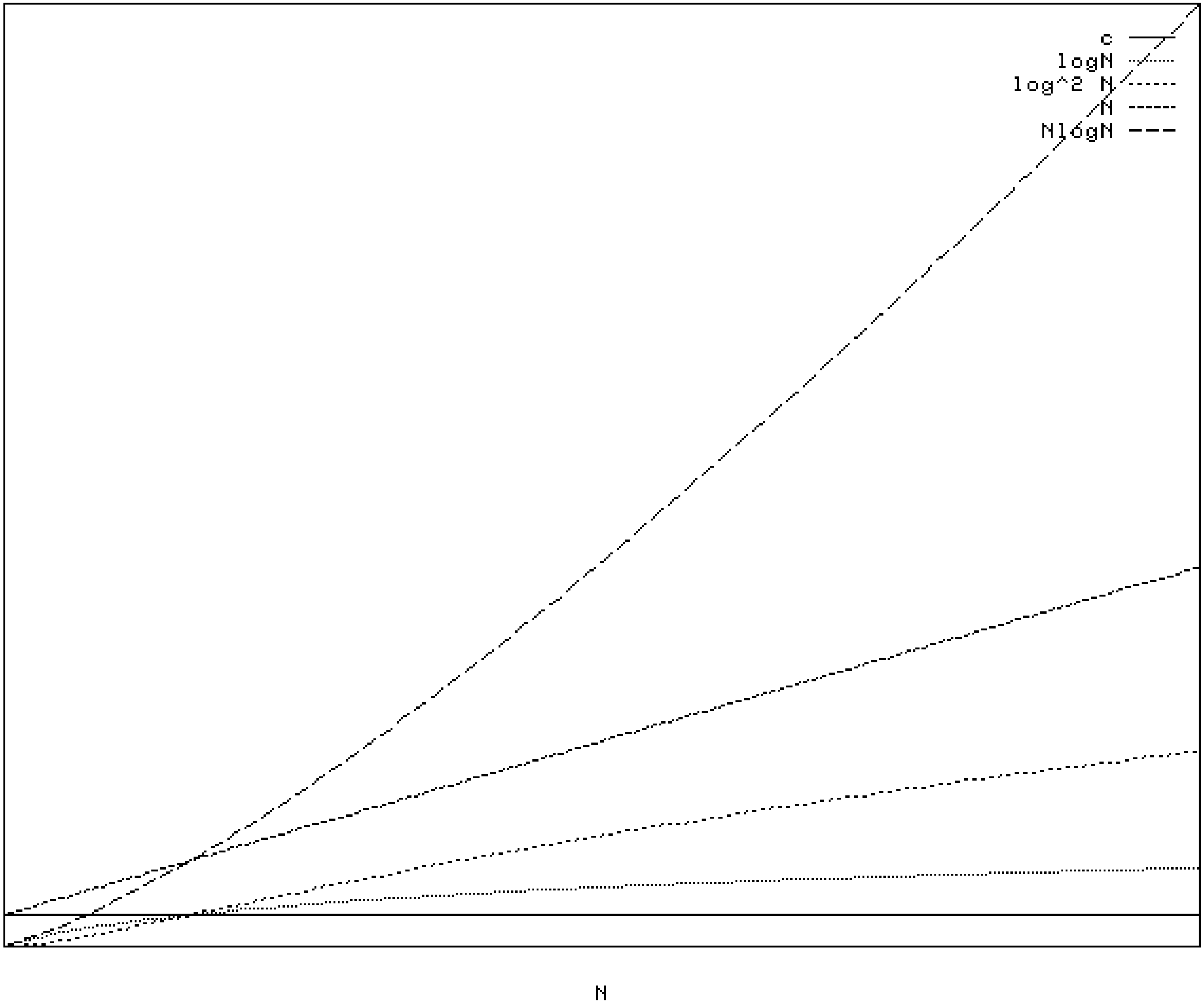
Big-O Notation (cont.)

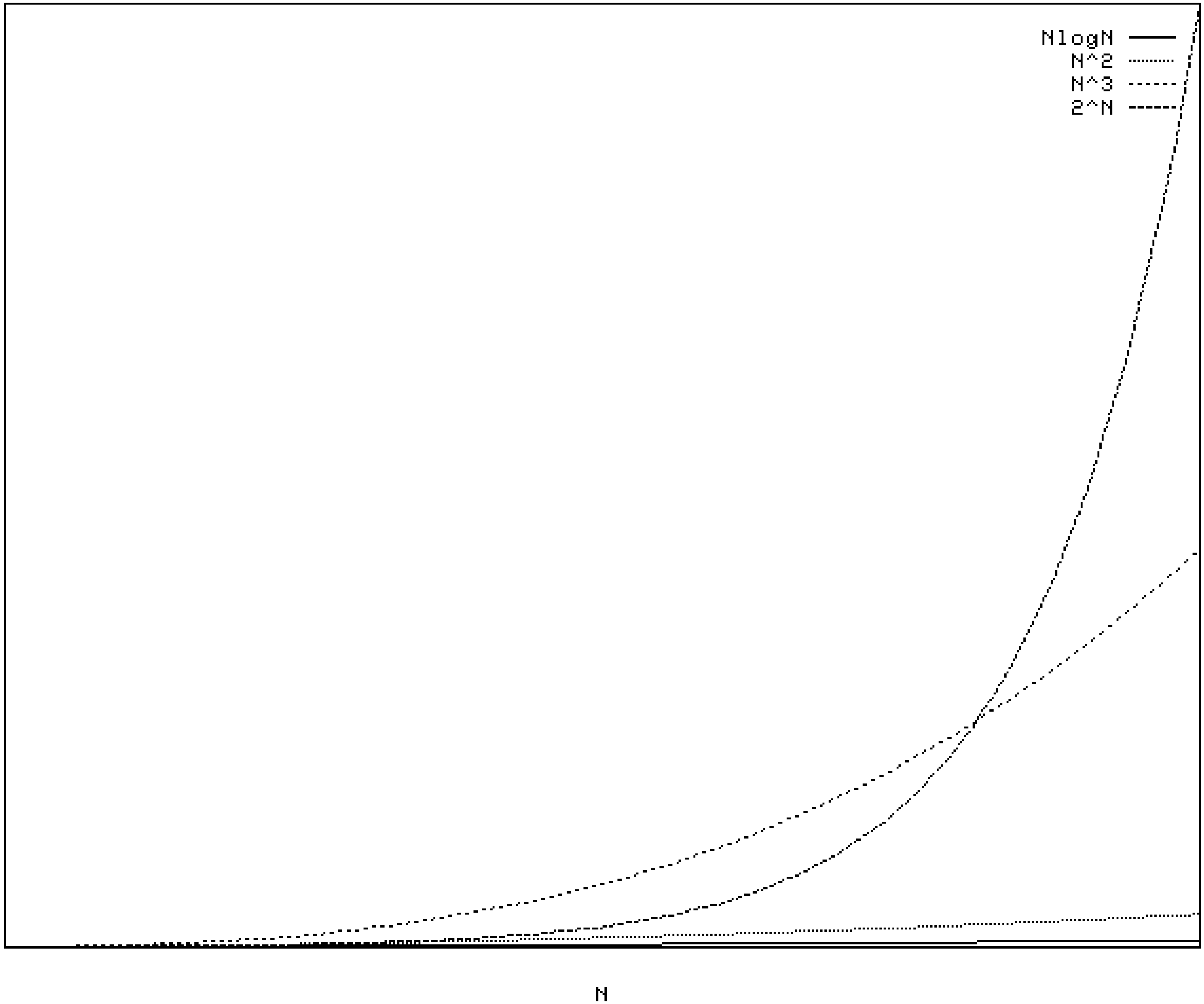
- Using Big-O notation, we might say that Algorithm A "runs in time Big-O of $n \log n$", or that Algorithm B "is an order n-squared algorithm". We mean that the number of operations, as a function of the input size n, is $O(n \log n)$ or $O(n^2)$ for these cases, respectively.
- Constants don't matter in Big-O notation because we're interested in the asymptotic behavior as n grows arbitrarily large; but, be aware that large constants can be very significant in an actual implementation of the algorithm.

Typical Growth Rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates


c
logN
log^2 N
N
NlogN
N


NlogN
N^2
N^3
2^N
N

Rates of Growth

- Suppose a computer executes 10^{12} ops per second:

n =	**10**	**100**	**1,000**	**10,000**	**10^{12}**
n	10^{-11}s	10^{-10}s	10^{-9}s	10^{-8}s	1s
n lg n	10^{-11}s	10^{-9}s	10^{-8}s	10^{-7}s	40s
n^2	10^{-10}s	10^{-8}s	10^{-6}s	10^{-4}s	10^{12}s
n^3	10^{-9}s	10^{-6}s	10^{-3}s	1s	10^{24}s
2^n	10^{-9}s	10^{18}s	10^{289}s		

10^4s = 2.8 hrs ***10^{18}s = 30 billion years***

Example

- Calculate

$$\sum_{i=1}^{N} i^3$$

```
int sum(int n)
{
      int partialSum;

1     partialSum=0;                  1
2     for (int i=1;i<=n;i++)         2N+2
3        partialSum += i*i*i;        4N
4     return partialSum;             1
}
```

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time four units
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 6N + 4 $\Rightarrow$ O(N)

General Rules

- For loops
  - at most the running time of the statements inside the for-loop (including tests) times the number of iterations.
- Nested for loops

```
for (i=0;i<n;i++)
      for (j=0;j<n;j++)
            k++;
```

  - the running time of the statement multiplied by the product of the sizes of all the for-loops.
  - $O(N^2)$

General rules (cont'd)

- Consecutive statements

```
for (i=0;i<n;i++)
        a[i]=0;
for (i=0;i<n;i++)
        for (j=0;j<n;j++)
                a[i] += a[j]+i+j;
```

  - These just add
  - $O(N) + O(N^2) = O(N^2)$
- If/Else
  - never more than the running time of the test plus the larger of the running times of S1 and S2.

Asymptotic Analysis Hacks

- Eliminate low order terms
  - $4n + 5 \Rightarrow 4n$
  - $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
  - $2^n + n^3 + 3n \Rightarrow 2^n$
- Eliminate coefficients
  - $4n \Rightarrow n$
  - $0.5\ n \log n \Rightarrow n \log n$
  - $n \log (n^2) = 2\ n \log n \Rightarrow n \log n$

Silicon Downs

Post #1	*Post #2*
$n^3 + 2n^2$	$100n^2 + 1000$
$n^{0.1}$	$\log n$
$n + 100n^{0.1}$	$2n + 10 \log n$
$5n^5$	$n!$
$n^{-15}2^n/100$	$1000n^{15}$
$8^{2\lg n}$	$3n^7 + 7n$
mn^3	$2^m n$

For each race, which "horse" is "faster". Note that faster means smaller, not larger!

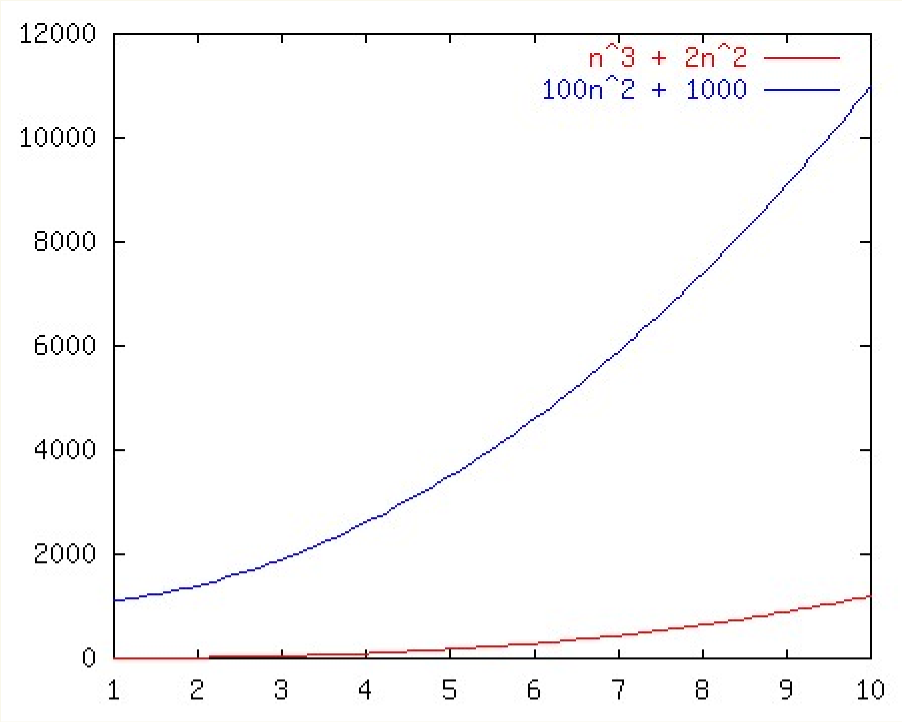
All analysis are done asymptotically

a. Left
b. Right
c. Tied
d. It depends

Race I

a. Left
b. Right
c. Tied
d. It depends

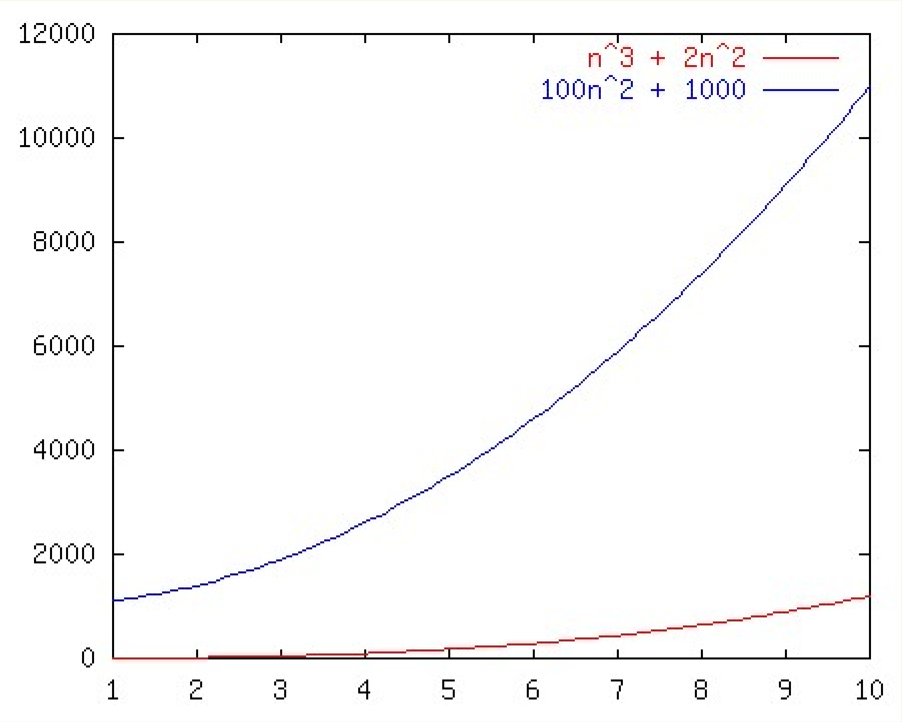
$n^3 + 2n^2$ vs. $100n^2 + 1000$

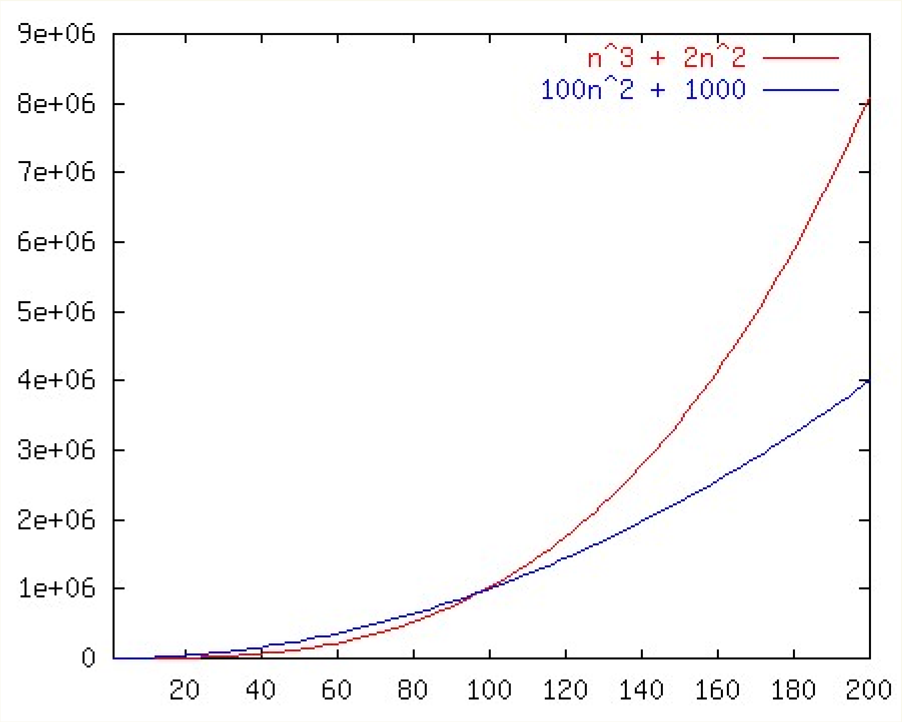


Race I

a. *Left*
b. *Right*
c. *Tied*
d. *It depends*

$n^3 + 2n^2$ vs. $100n^2 + 1000$

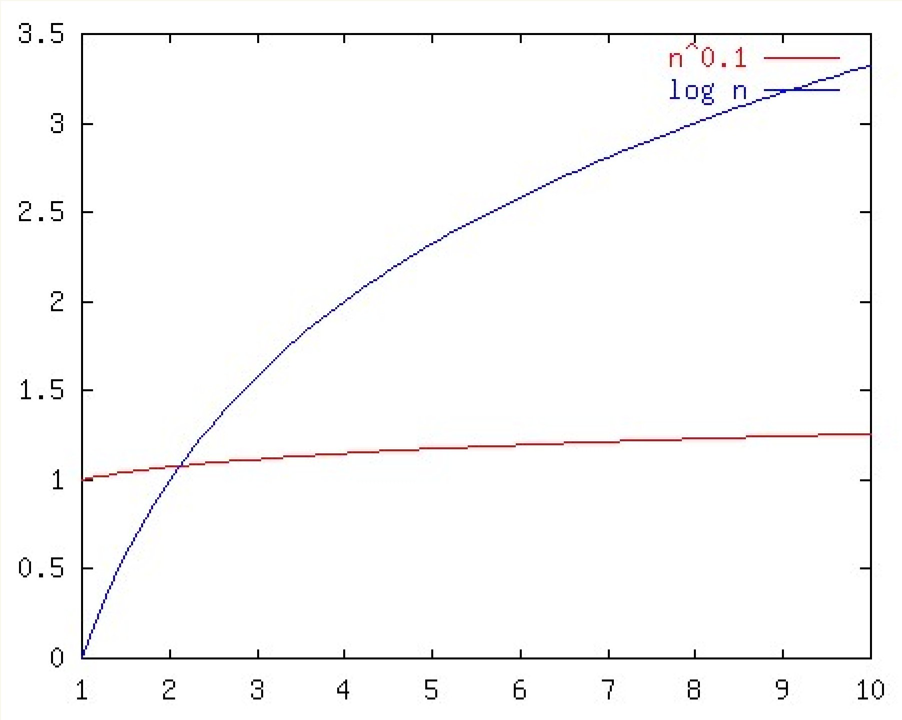




Race II

a. Left
b. Right
c. Tied
d. It depends

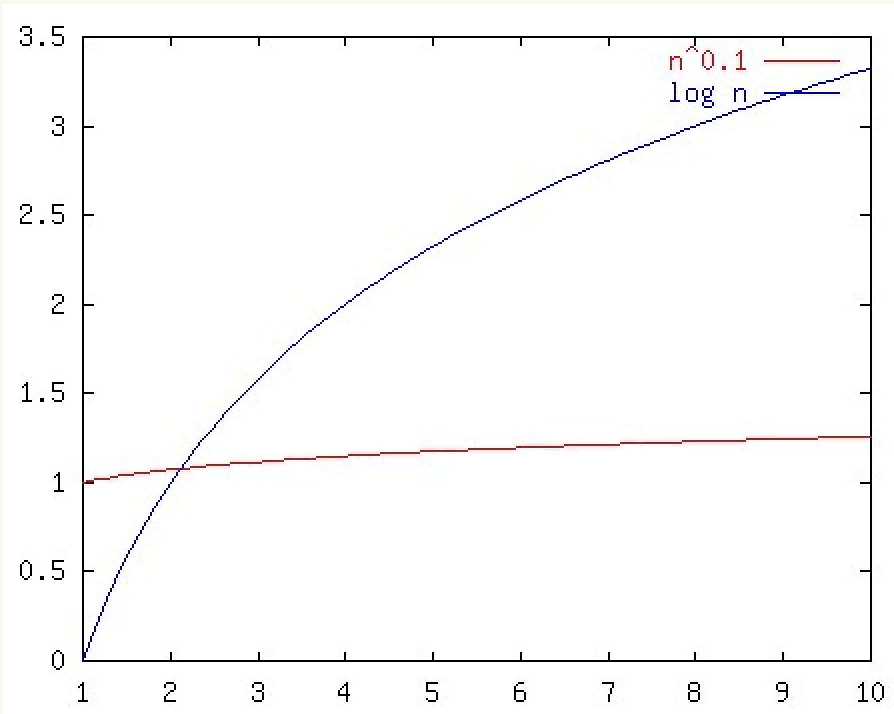
$n^{0.1}$ vs. `log n`

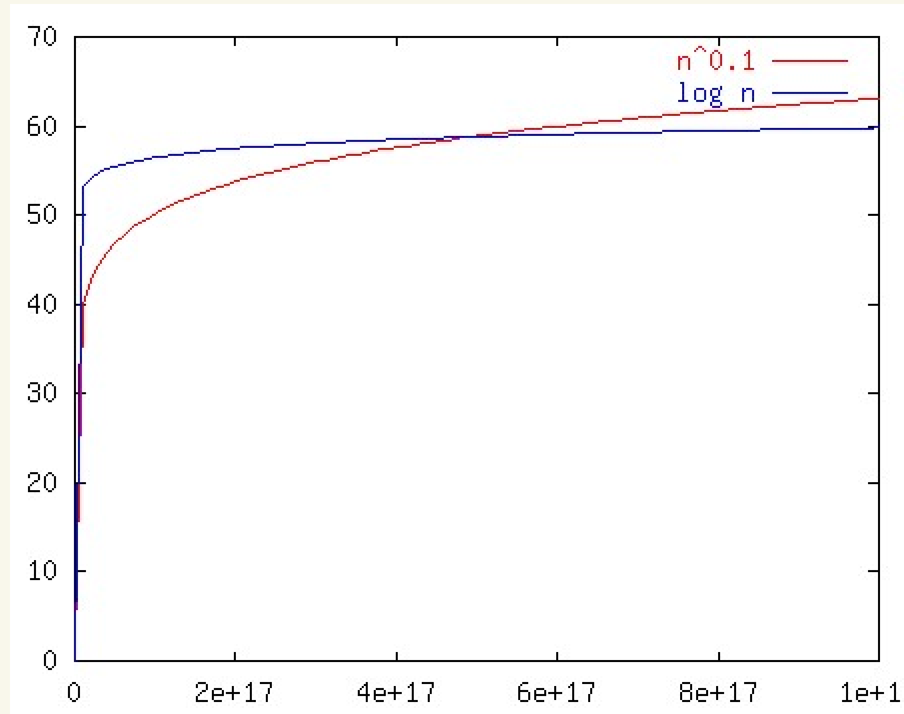


Race II

a. Left
b. Right
c. Tied
d. It depends

$n^{0.1}$ vs. **log n**



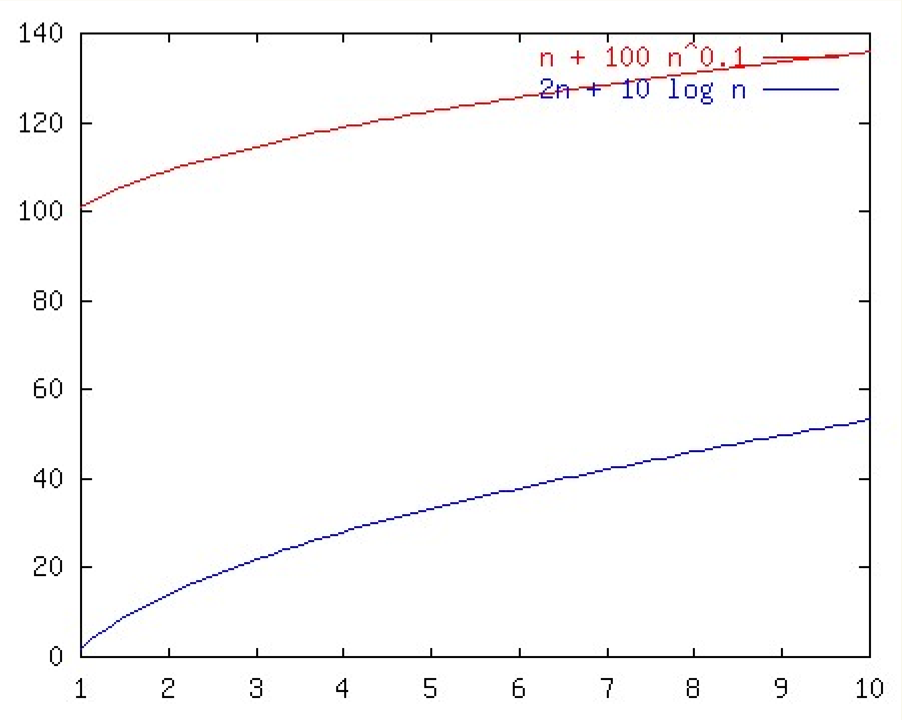


Moral of the story? n^{ϵ} *is slower than log n for any* $\epsilon > 0$

Race III

a. *Left*
b. *Right*
c. *Tied*
d. *It depends*

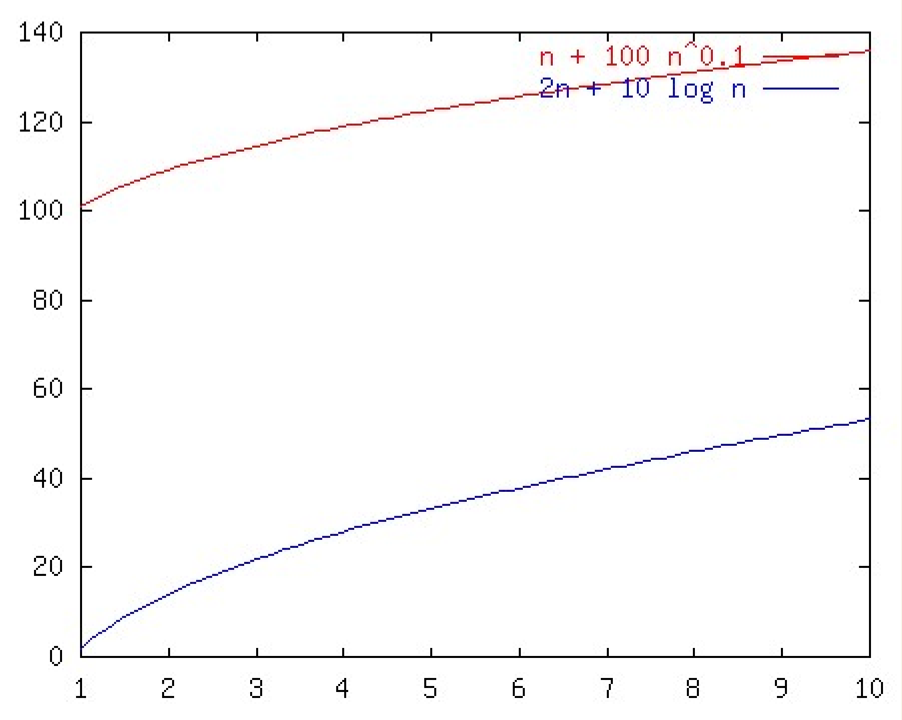
$n + 100n^{0.1}$ vs. $2n + 10 \log n$

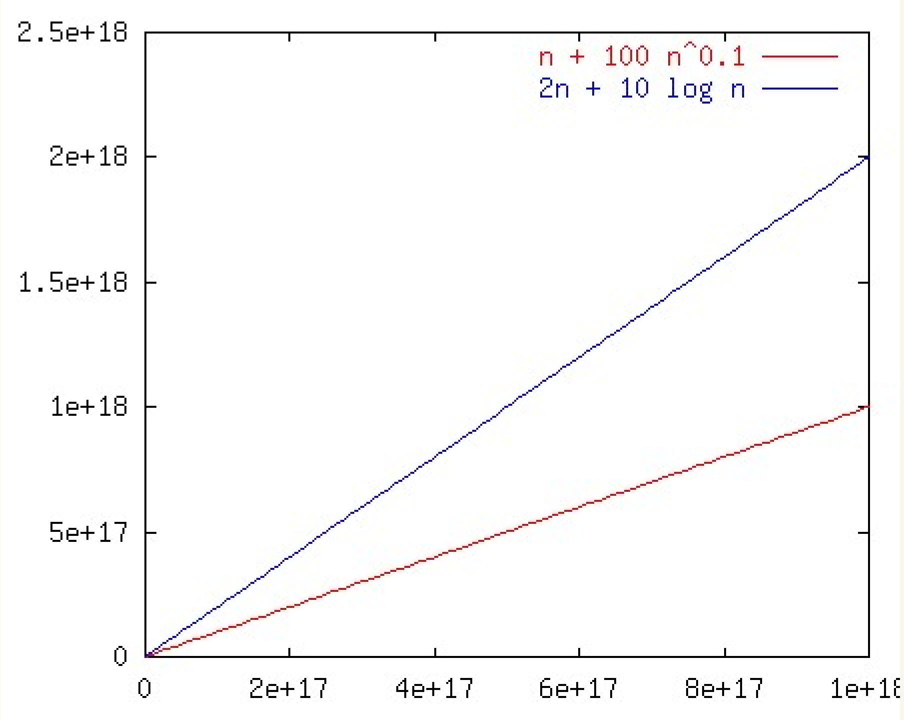


Race III

a. Left
b. Right
c. Tied
d. It depends

n + 100n^0.1 vs. **2n + 10 log n**



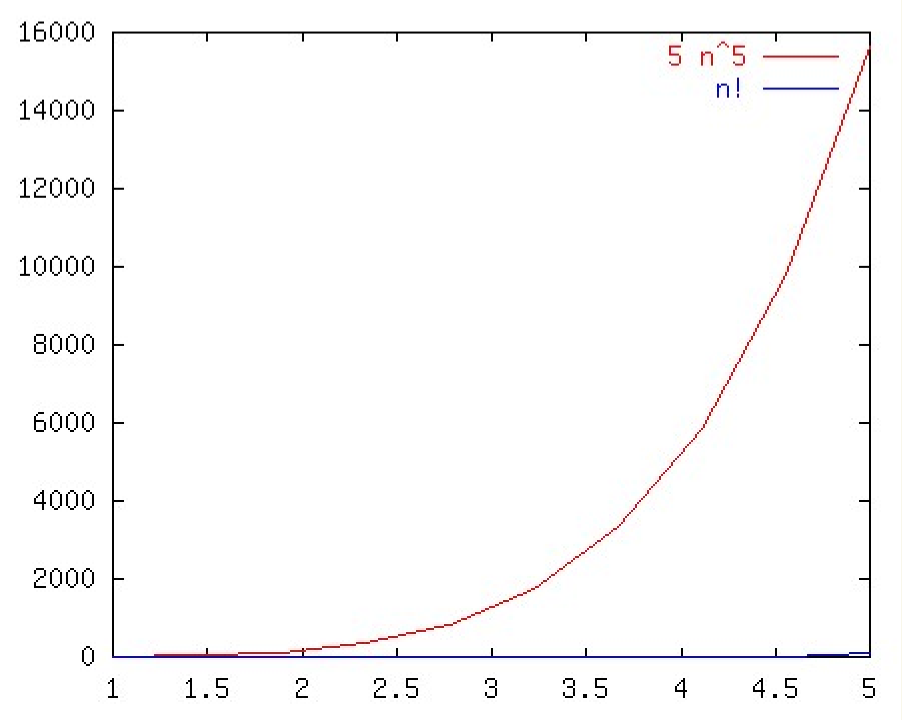


Although left seems faster, asymptotically it is a TIE

Race IV

a. Left
b. Right
c. Tied
d. It depends

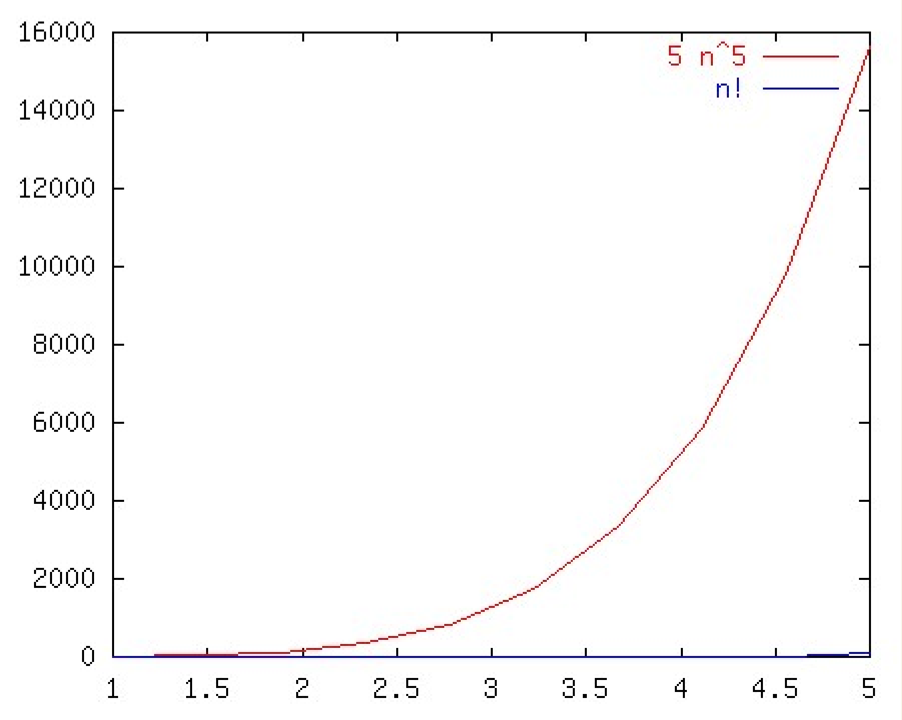
$5n^5$ vs. `n!`

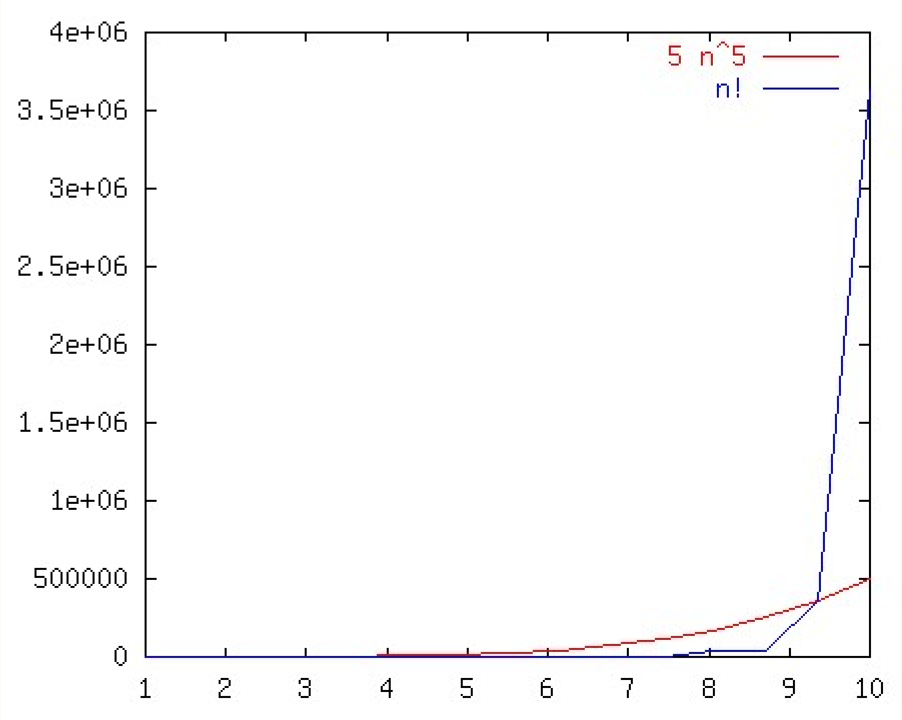


Race IV

a. *Left*
b. *Right*
c. *Tied*
d. *It depends*

$5n^5$ vs. `n!`

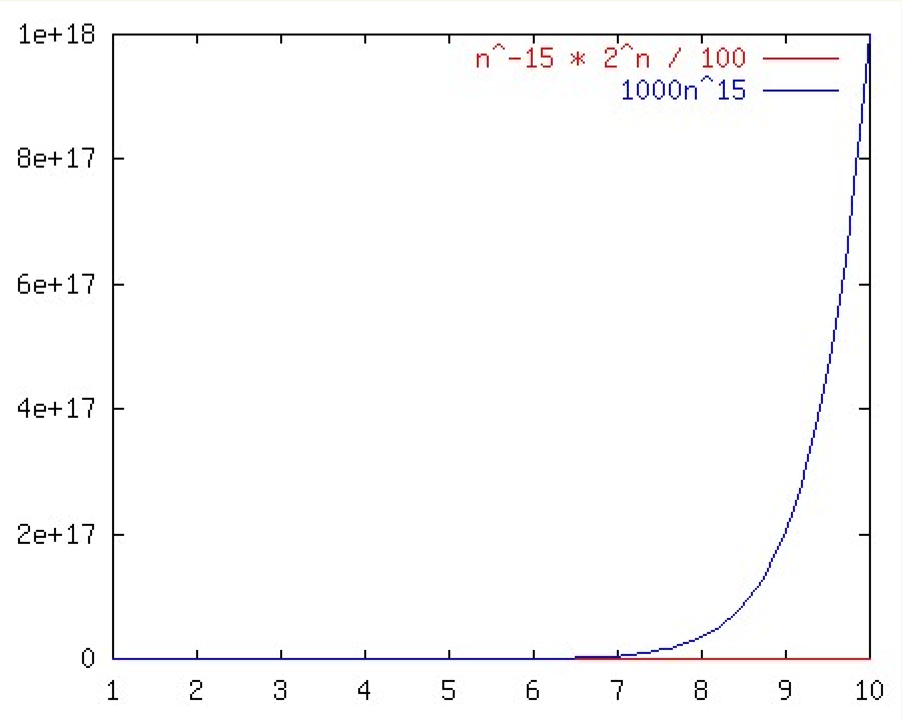




Race V

a. *Left*
b. *Right*
c. *Tied*
d. *It depends*

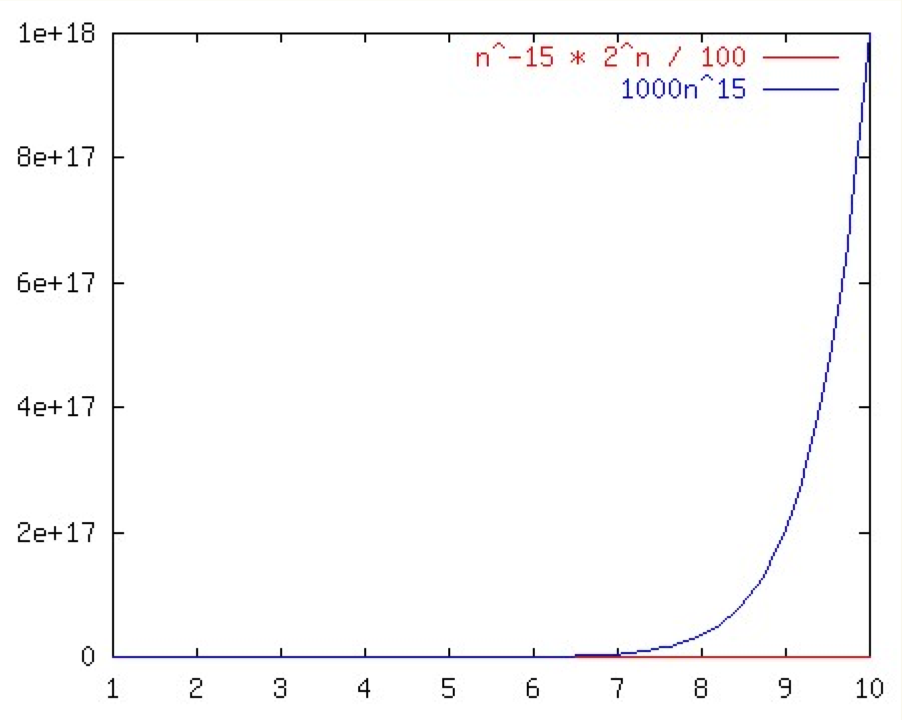
$n^{-15}2^n/100$ vs. $1000n^{15}$

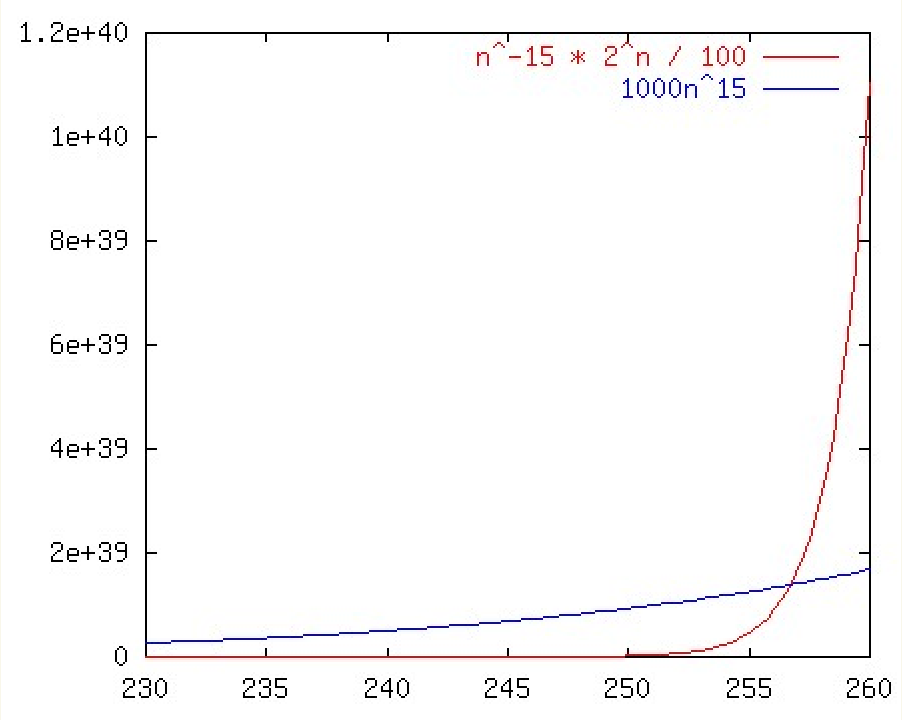


Race V

a. Left
b. Right
c. Tied
d. It depends

$n^{-15}\, 2^n/100$ vs. $1000n^{15}$



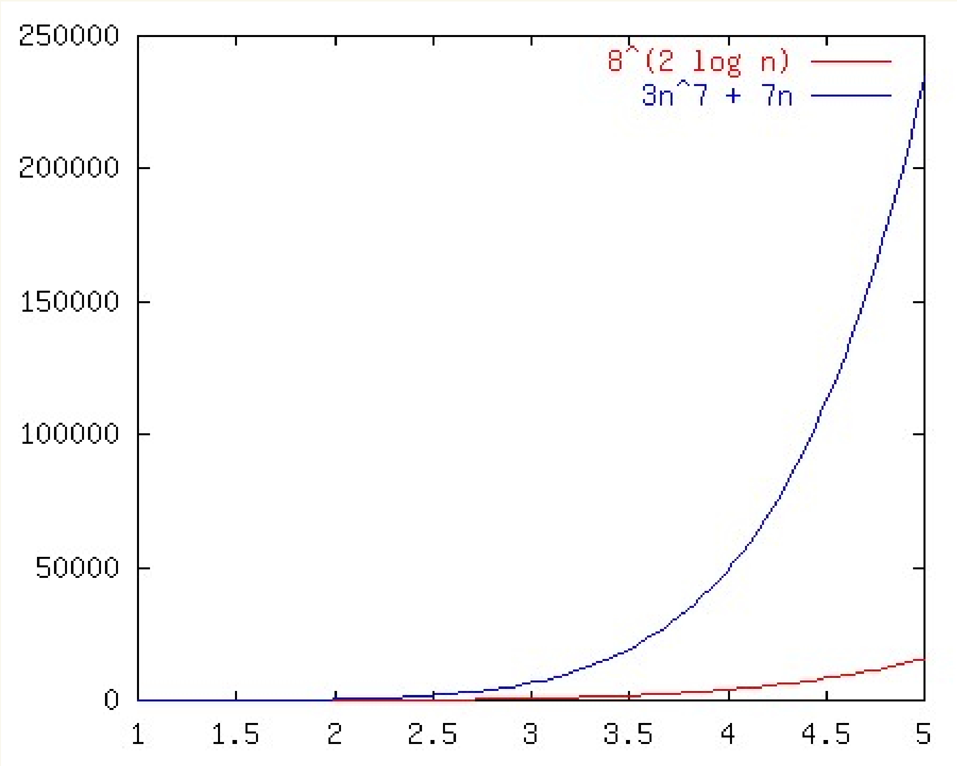


Any exponential is slower than any polynomial.
It doesn't even take that long here (~250 input size)

Race VI

a. *Left*
b. *Right*
c. *Tied*
d. *It depends*

$8^{2\ \log_2(n)}$ vs. $3n^7 + 7n$



Log Rules:
1) log(mn) = log(m) + log(n)
2) log(m/n) = log(m) – log(n)
3) $\log(m^n) = n \cdot \log(m)$
4) $n = 2^k \rightarrow \log_2 n = k$

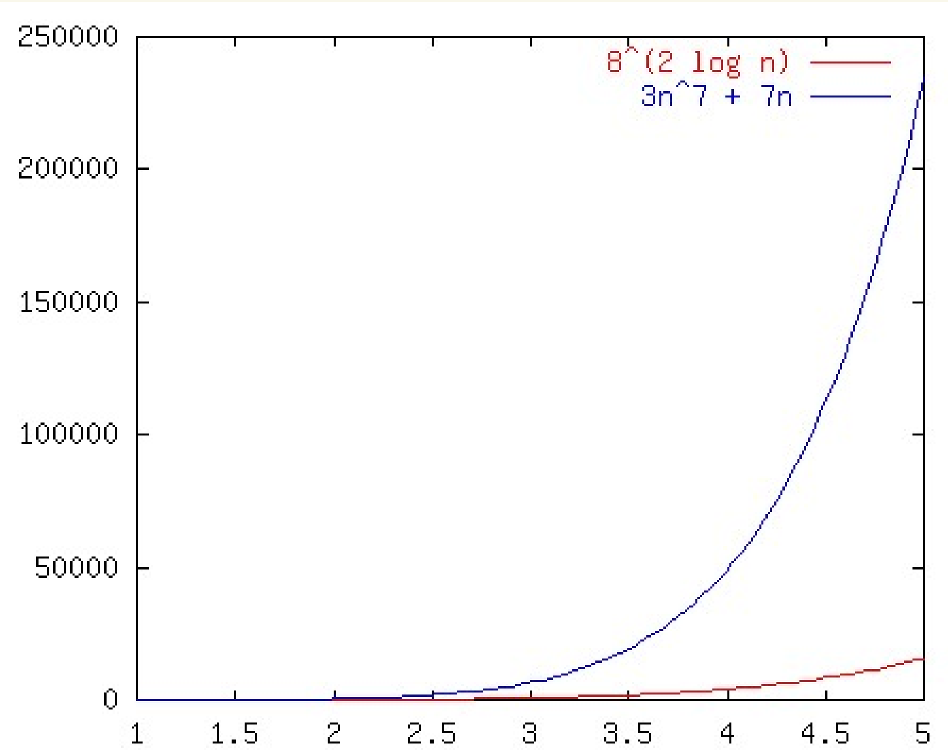
Log Rules:

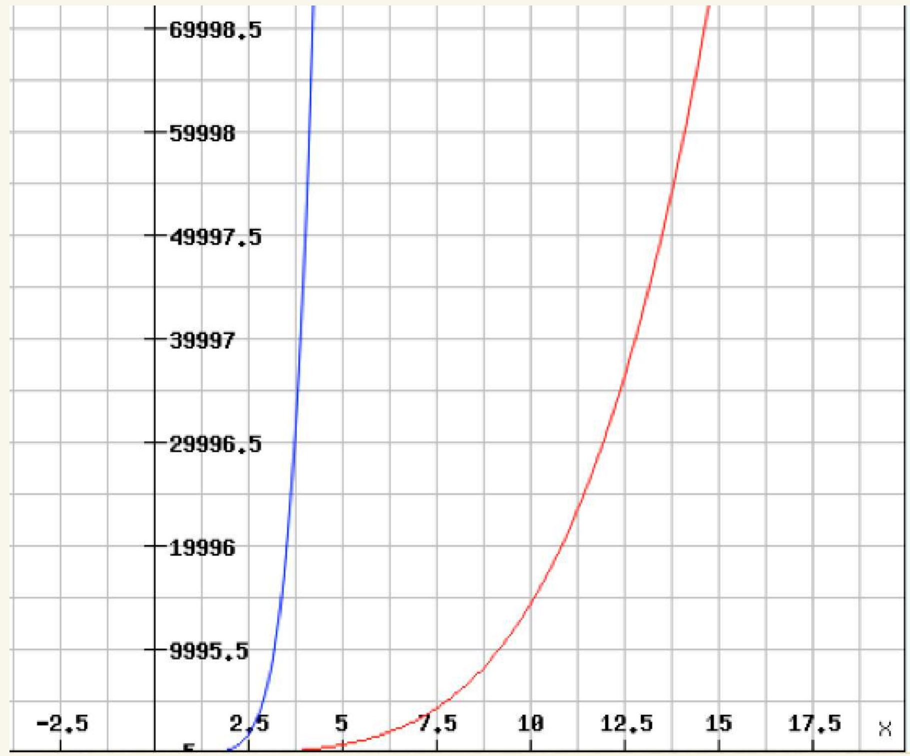
1) log(mn) = log(m) + log(n)
2) log(m/n) = log(m) – log(n)
3) $\log(m^n) = n \cdot \log(m)$
4) $n = 2^k \rightarrow \log n = k$

Race VI

a. Left
b. Right
c. Tied
d. It depends

$8^{2\ \log_2(n)}$ vs. $3n^7 + 7n$





$$8^{2\lg(n)} = 8^{\lg(n^2)} = (2^3)^{\lg(n^2)} = 2^{3\lg(n^2)} = 2^{\lg(n^6)} = n^6$$

Log Aside

$\log_a b$ means "the exponent that turns **a** into **b**"

lg x means "**$\log_2 x$**" (the usual log in CS)

log x means "**$\log_{10} x$**" (the common log)

ln x means "**$\log_e x$**" (the natural log)

- There's just a constant factor between the three main log bases, and asymptotically they behave equivalently.

Race VII

a. *Left*
b. *Right*
c. *Tied*
d. *It depends*

mn^3 vs. $2^m n$

Race VII

a. Left
b. Right
c. Tied
d. It depends

mn^3 vs. $2^m n$

It depends on values of m and n

Silicon Downs

Post #1	*Post #2*	*Winner*
$n^3 + 2n^2$	$100n^2 + 1000$	$O(n^2)$
$n^{0.1}$	$\log n$	$O(\log n)$
$n + 100n^{0.1}$	$2n + 10 \log n$	***TIE*** $O(n)$
$5n^5$	$n!$	$O(n^5)$
$n^{-15}2^n/100$	$1000n^{15}$	$O(n^{15})$
$8^{2\lg n}$	$3n^7 + 7n$	$O(n^6)$
mn^3	$2^m n$	***IT DEPENDS***

The fix sheet

- The fix sheet (typical growth rates in order)
  - constant: $O(1)$
  - logarithmic: $O(\log n)$ ($\log_k n$, $\log n^2 \in O(\log n)$)
  - Sub-linear: $O(n^c)$ (c is a constant, $0 < c < 1$)
  - linear: $O(n)$
  - (log-linear): $O(n \log n)$ (usually called "n log n")
  - (superlinear): $O(n^{1+c})$ (c is a constant, $0 < c < 1$)
  - quadratic: $O(n^2)$
  - cubic: $O(n^3)$
  - polynomial: $O(n^k)$ (k is a constant)
  - exponential: $O(c^n)$ (c is a constant > 1)

Tractable

Intractable!

Name-drop your friends

- constant: $O(1)$
- Logarithmic: $O(\log n)$
- Sub-linear: $O(n^c)$
- linear: $O(n)$
- (log-linear): $O(n \log n)$
- (superlinear): $O(n^{1+c})$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$
- exponential: $O(c^n)$

Casually name-drop the appropriate terms in order to sound bracingly cool to colleagues: “Oh, linear search? I hear it’s sub-linear on quantum computers, though. Wild, eh?”

Clicker Question

Which of the following functions is likely to grow the fastest, meaning that the algorithm is likely to take the most steps, as the input size, n, grows sufficiently large?

A. O(n)
B. O(sqrt (n))
C. O (log n)
D. O (n log n)
E. They would all be about the same.

Clicker Question (answer)

Which of the following functions is likely to grow the fastest, meaning that the algorithm is likely to take the most steps, as the input size, n, grows sufficiently large?

A. O(n)
B. O(sqrt (n))
C. O (log n)
D. O (n log n)
E. They would all be about the same.

Clicker Question

Suppose we have 4 programs, A-D, that run algorithms of the time complexities given. Which program will finish first, when executing the programs on input size n=10?

A. O(n)
B. O(sqrt (n))
C. O (log n)
D. O (n log n)
E. Impossible to tell

Clicker Question (Answer)

Suppose we have 4 programs, A-D, that run algorithms of the time complexities given. Which program will finish first, when executing the programs on input size n=10?

A. O(n)
B. O(sqrt (n))
C. O (log n)
D. O (n log n)
E. Impossible to tell

Clicker Question

Which of the following statements is true? Choose the best answer

A. The set of functions in $O(n^4)$ have a fairly slow growth rate

A. O(lg n) doesn‘t grow very quickly

A. Big-O functions with the fastest growth rate represent the fastest algorithms, most of the time

A. Asymptotic complexity deals with relatively small input sizes

Clicker Question (answer)

Which of the following statements is true? Choose the best answer

A. The set of functions in $O(n^4)$ have a fairly slow growth rate

A. O(lg n) doesn't grow very quickly

A. Big-O functions with the fastest growth rate represent the fastest algorithms, most of the time

A. Asymptotic complexity deals with relatively small input sizes

Computing Big-O

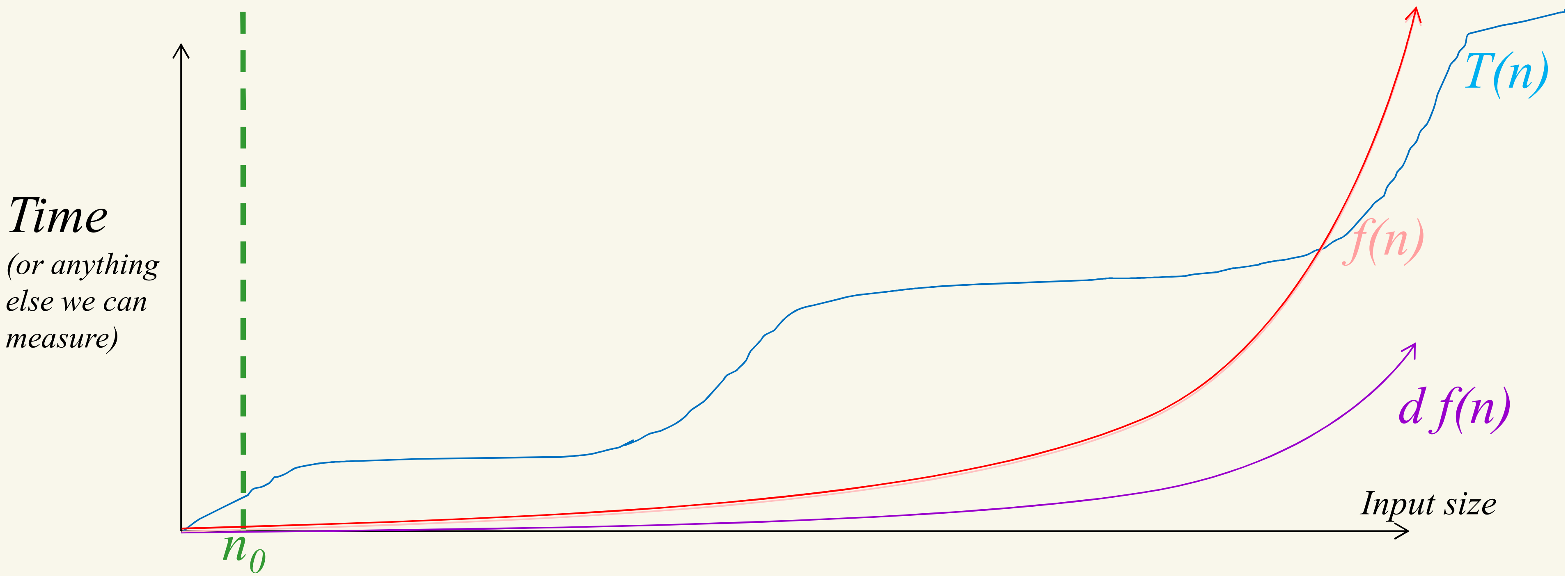
- If $T(n)$ is a polynomial of degree d
  - (i.e., $T(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$),
- then its Big-O estimate is simply the largest term without its coefficient, that is, $T(n) \in O(n^d)$.

- If $T_1(n) \in O(f(n))$ and $T_2(n) \in O(g(n))$, then
  - $T_1(n) + T_2(n) \in O(\max(f(n), g(n)))$.
  - $T_1(n) = 4\, n^{3/2} + 9$
  - $T_2(n) = 30\, n \lg n + 17n$
  - $T(n) = T_1(n) + T_2(n) \in O(\max(n^{3/2}, n \lg n) = O(n^{3/2})$

Big-Omega (Ω) notation

- Just as Big-O provides an *upper* bound, there exists Big-Omega (Ω) notation to estimate the *lower* bound of an algorithm, meaning that, in the worst case, the algorithm takes at least so many steps:

$$T(n) \in \Omega(f(n)) \text{ if } \exists\ d \text{ and } n_0 \text{ such that}$$
$$d\, f(n) \leq T(n)\ \forall\ n \geq n_0$$

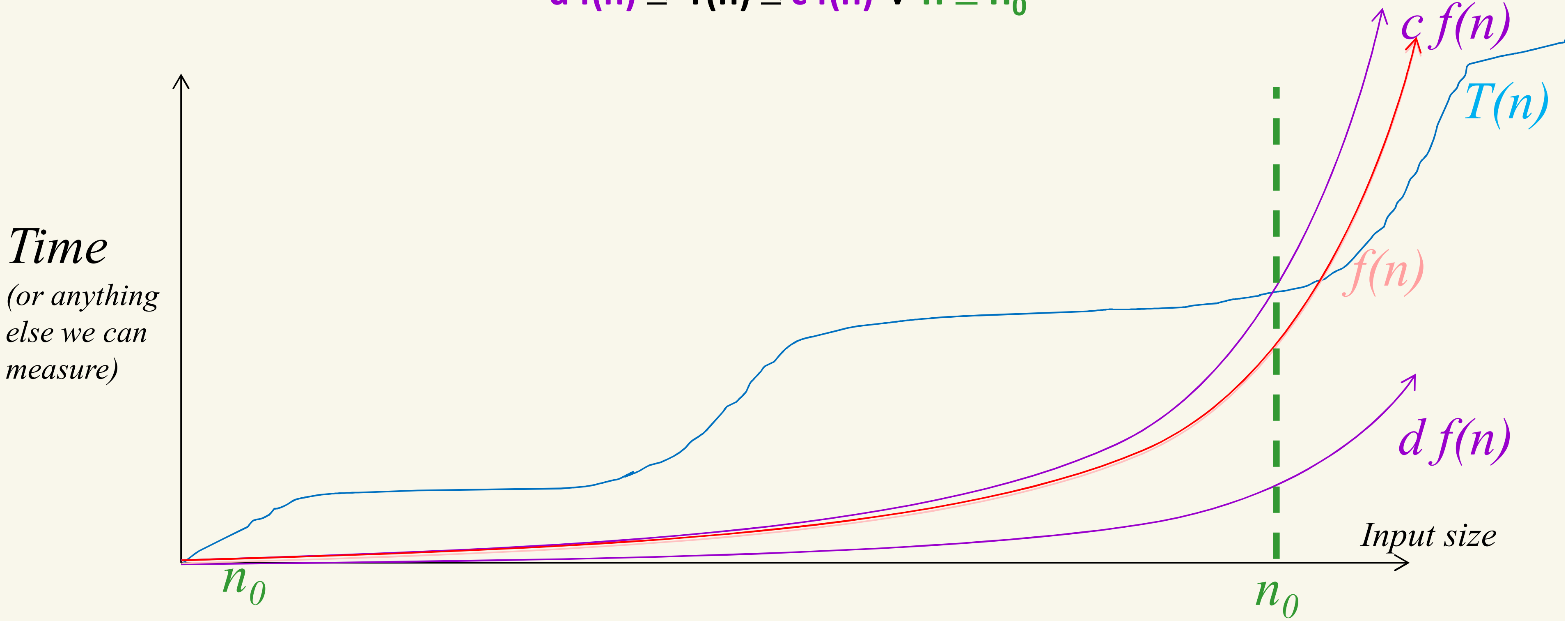


Big-Theta (Θ) notation

- Furthermore, each algorithm has both an upper bound and a lower bound, and when these correspond to the same growth order function, the result is called Big-Theta (Θ) notation.

$T(n) \in \Theta(f(n))$ if $\exists$ c, d and n_0 such that

$d\,f(n) \leq T(n) \leq c\,f(n)$ $\forall$ $n \geq n_0$



Examples

$10{,}000\ n^2 + 25\ n \in \Theta(n^2)$

$10^{-10}\ n^2 \in \Theta(n^2)$

$n \log n \in O(n^2)$

$n \log n \in \Omega(n)$

$n^3 + 4 \in O(n^4)$ but not $\Theta(n^4)$

$n^3 + 4 \in \Omega(n^2)$ but not $\Theta(n^2)$

Analyzing Code

- But how can we obtain T(n) from an algorithm/code

• C operations	- constant time
• consecutive stmts	- sum of times
• conditionals	- max of branches, condition
• loops	- sum of iterations
• function calls	- cost of function body

Analyzing Code

```
find(key, array)
   for i = 1 to length(array) do
      if array[i] == key
         return i

   return -1
```

- Step 1: What's the input size **n**?
- Step 2: What kind of analysis should we perform?
  - Worst-case? Best-case? Average-case?
- Step 3: How much does each line cost? (Are lines the right unit?)

Analyzing Code

```
find(key, array)
   for i = 1 to length(array) do
      if array[i] == key
         return i

   return -1
```

- Step 4: What's $T(n)$ in its raw form?
- Step 5: Simplify $T(n)$ and convert to order notation. (Also, which order notation: O, Θ, Ω?)
- Step 6: **Prove** the asymptotic bound by finding constants c and n_0 such that
  - for all $n \geq n_0$, $T(n) \leq cn$.

Example 1

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

1
1
1

n times
n times

- This example is pretty straightforward. Each loop goes *n* times, and a constant amount of work is done on the inside.

$$T(n) = \sum_{i=1}^{n}(1 + \sum_{j=1}^{n} 2) = \sum_{i=1}^{n}(1 + 2n) = n + 2n^2 = O(n^2)$$

Example 1 (simpler version)

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

1
1
1

n times
n times

- Count the number of times sum = sum + 1 occurs

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2 = O(n^2)$$

Example 2

```
i = 1
  while i < n do
    for j = i to n do
      sum = sum + 1
    i++
```

Time complexity:

a. $\Theta(n)$
b. $\Theta(n \lg n)$
c. $\Theta(n^2)$
d. $\Theta(n^2 \lg n)$
e. None of these

Example 2 (Pure Math Approach)

```
i = 1                              takes "1" step
while i < n do                     i varies 1 to n-1
  for j = i to n do                j varies i to n
    sum = sum + 1                  takes "1" step
  i++                              takes "1" step
```

Now, we write a function T(n) that adds all of these up, summing over the iterations of the two loops:

$$T(n) = 1 + \sum_{i=1}^{n-1} \left(1 + \sum_{j=i}^{n} 1 \right)$$

Example 2 (Pure Math Approach)

Here's our function for the runtime of the code:

$$T(n) = 1 + \sum_{i=1}^{n-1}\left(1 + \sum_{j=i}^{n} 1\right)$$

Summing 1 for *j* from *i* to *n* is just going to be 1 added together (n-i+1) times, which is (n-i+1):

$$T(n) = 1 + \sum_{i=1}^{n-1}(1 + n - i + 1) = 1 + \sum_{i=1}^{n-1}(n - i + 2)$$

Example 2 (Pure Math Approach)

Here's our function for the runtime of the code:

$$T(n) = 1 + \sum_{i=1}^{n-1}(1 + n - i + 1) = 1 + \sum_{i=1}^{n-1}(n - i + 2)$$

The n and 2 terms don't change as i changes. So, we can pull them out (and multiply by the number of times they're added):

$$T(n) = 1 + n(n-1) + 2(n-1) - \sum_{i=1}^{n-1} i$$

And, we know that $\sum_{i=1}^{k} i = k(k+1)/2$, so:

$$T(n) = 1 + n^2 - n + 2n - 2 - \frac{(n-1)n}{2}$$

Example 2 (Pure Math Approach)

Here's our function for the runtime of the code:

$$T(n) = 1 + n^2 - n + 2n - 2 - \frac{(n-1)n}{2}$$

$$= n^2 + n - 1 - \frac{n^2}{2} + \frac{n}{2} = \frac{n^2}{2} + \frac{3n}{2} - 1$$

So, $T(n) = \frac{n^2}{2} + \frac{3n}{2} - 1$.

Drop low-order terms and the ½ coefficient, and we find:

$T(n) \in \Theta(n^2)$.

Yay!!!

Example 2 (Simplified Math Approach)

```
i = 1
  while i < n do
    for j = i to n do
      sum = sum + 1
    i++
```

Count this line

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i}^{n} 1$$

The second sigma is n-i+1

$$T(n) = \sum_{i=1}^{n-1} (n - i + 1) = n + n - 1 + \ldots + 2$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$T(n) = n(n+1)/2 \in \Theta(n^2)$$

Example 2 Pretty Pictures Approach

```
i = 1                          /* takes "1" step */
 while i < n do                /* i varies 1 to n-1 */
  for j = i to n do            /* j varies i to n */
    sum = sum + 1              /* takes "1" step */
  i++                          /* takes "1" step */
```

- Imagine drawing one point for each time the "sum=sum+1" line gets executed. In the first iteration of the outer loop, you'd draw n points. In the second, n-1. Then n-2, n-3, and so on down to (about) 1. Let's draw that picture…

```
* * * * * * * * * *
  * * * * * * * * *
    * * * * * * * *
      * * * * * * *
        * * * * * *
          * * * * *
            * * * *
              * * *
                * *
                  *
```

Example 2 Pretty Pictures Approach

```
         n columns
* * * * * * * * * *
  * * * * * * * * *
    * * * * * * * *
      * * * * * * *     n rows
        * * * * * *
          * * * * *
            * * * *
              * * *
                * *
                  *
```

- It is a triangle and its area is proportional to runtime

$$T(n) = \frac{Base \times Height}{2} = \frac{n^2}{2} \in \Theta(n^2)$$

Example 3

```
for (i=1; i <= n; i++)
  for (j=1; j <= n; j=j*2)
     sum = sum + 1
```

Example 3

```
for (i=1; i <= n; i++)
  for (j=1; j <= n; j=j*2)
     sum = sum + 1
```

Time complexity:

a. $\Theta(n)$
b. $\Theta(n \lg n)$
c. $\Theta(n^2)$
d. $\Theta(n^2 \lg n)$
e. None of these

Example 3

```
for (i=1; i <= n; i++)
  for (j=1; j <= n; j=j*2)
     sum = sum + 1
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{?} 1$$

$j = 1, \; 2, \; 4, \; \ldots, \; x$

$= 2^0, 2^1, 2^2, \ldots, 2^k$

x <= n < 2x

$2^k <= 2^{\lg n} < 2^{k+1}$

k <= lg n< k+1

$$k = \lfloor \lg n \rfloor$$

$$T(n) = \sum_{i=1}^{n} \sum_{j=0}^{\lfloor \lg n \rfloor} 1 = \sum_{i=0}^{n} \lg n = (n+1)\lg n \in O(n \lg n)$$

Asymptotically flooring doesn't matter

Example 4

- Conditional

`if` C `then` S_1 `else` S_2

O(c) + max (O(s_1), O(s_2))

or

O(c) + O(s_1) + O(s_2)

- Loops

`while` C `do` S

max(O(c), O(s)) * # iterations

Learning Goals revisited

- Justify which operation(s) we should measure in an algorithm/program in order to estimate its "efficiency".
- Define the "input size" n for various problems, and determine the effect (in terms of performance) that increasing the value of n has on an algorithm.
- Given a fragment of code, write a formula which measures the number of steps executed, as a function of n.
- Define the notion of Big-O complexity, and explain pictorially what it represents.
- Compute the worst-case asymptotic complexity of an algorithm in terms of its input size n, and express it in Big-O notation.

Learning Goals (revisited)

- Compute an appropriate Big-O estimate for a given function $T(n)$.
- Discuss the pros and cons of using best-, worst-, and average-case analysis, when determining the complexity of an algorithm.
- Describe why best-case analysis is rarely relevant and how worst-case analysis may never be encountered in practice.
- Given two or more algorithms, rank them in terms of their time and space complexity.