INFS7901 Database Principles

Asymptotic Analysis

Hassan Khosravi & Ash Rahimi

Module 2

- Data Access methods
  - Asymptotic analysis
    - An ML model is trained for 1 hour for 1000 instances with 10k features, how long does it take to train it on 100k instances with 100k features?)
  - Sorting algorithms
  - Binary trees
  - Hashing
- Need these to understand how DBMS works, also need it for your job interviews, you want more practice for your job interviews? check hackerrank.com or leetcode.com)

- Taught and delivered with data science rather computer science in mind.
  - Resources
    - Problem Solving with Algorithms and Data Structures using Python¶
    - Introduction to Algorithms (CLRS)

- Jupyter notebooks to give you insight as a data scientist

Learning Objectives

Description	Tag
Define which program operations we measure in an algorithm in order to approximate its efficiency.	Big-O
Define "input size" and determine the effect (in terms of performance) that input size has on an algorithm.	
Give examples of common practical limits of problem size for each complexity class.	
Define the big-O notation.	
Give examples of tractable problems.	
Give examples of intractable problems.	
Categorize an algorithm into one of the common complexity classes.	
Describe best-case analysis and reason why it is rarely relevant	
Describe worst-case analysis and reason why it may never be encountered	
Describe average-case analysis and reason why it is often hard to compute	
Compare and contrast best-, worst-, and average-case analysis.	
Given code, write a formula which measures the number of steps executed as a function of the size of the input (N).	
Given two or more algorithms, rank them in terms of their time complexity.	

A Task to Solve and Analyse

- Find a student’s name in a class given her student ID

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;
}
```

How long does this take?

IT DEPENDS

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

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

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 input
  - 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.

- Data 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!

Motivation

Big-O Notation

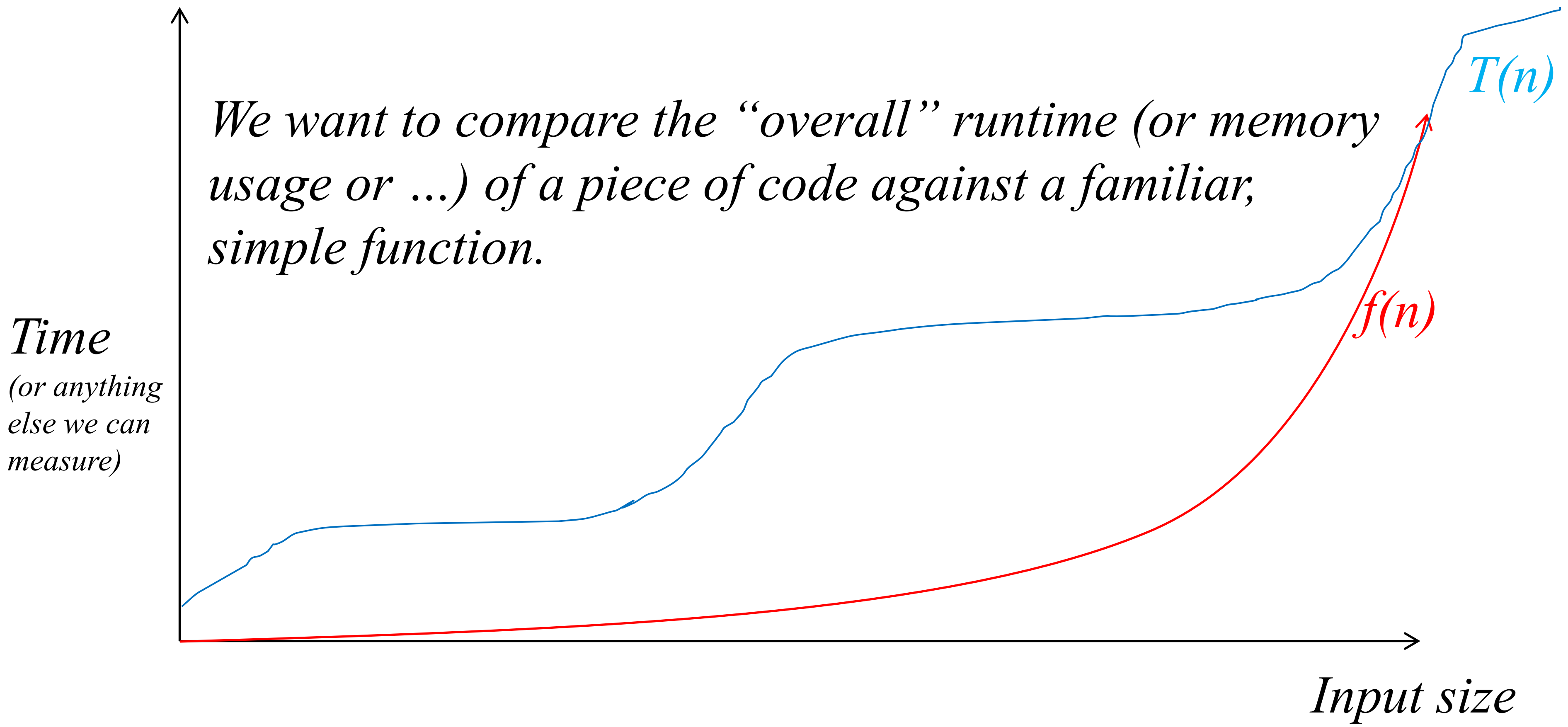
Analyzing Code

Big-O (Big-Oh) Notation

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

Positive integers

Positive real numbers



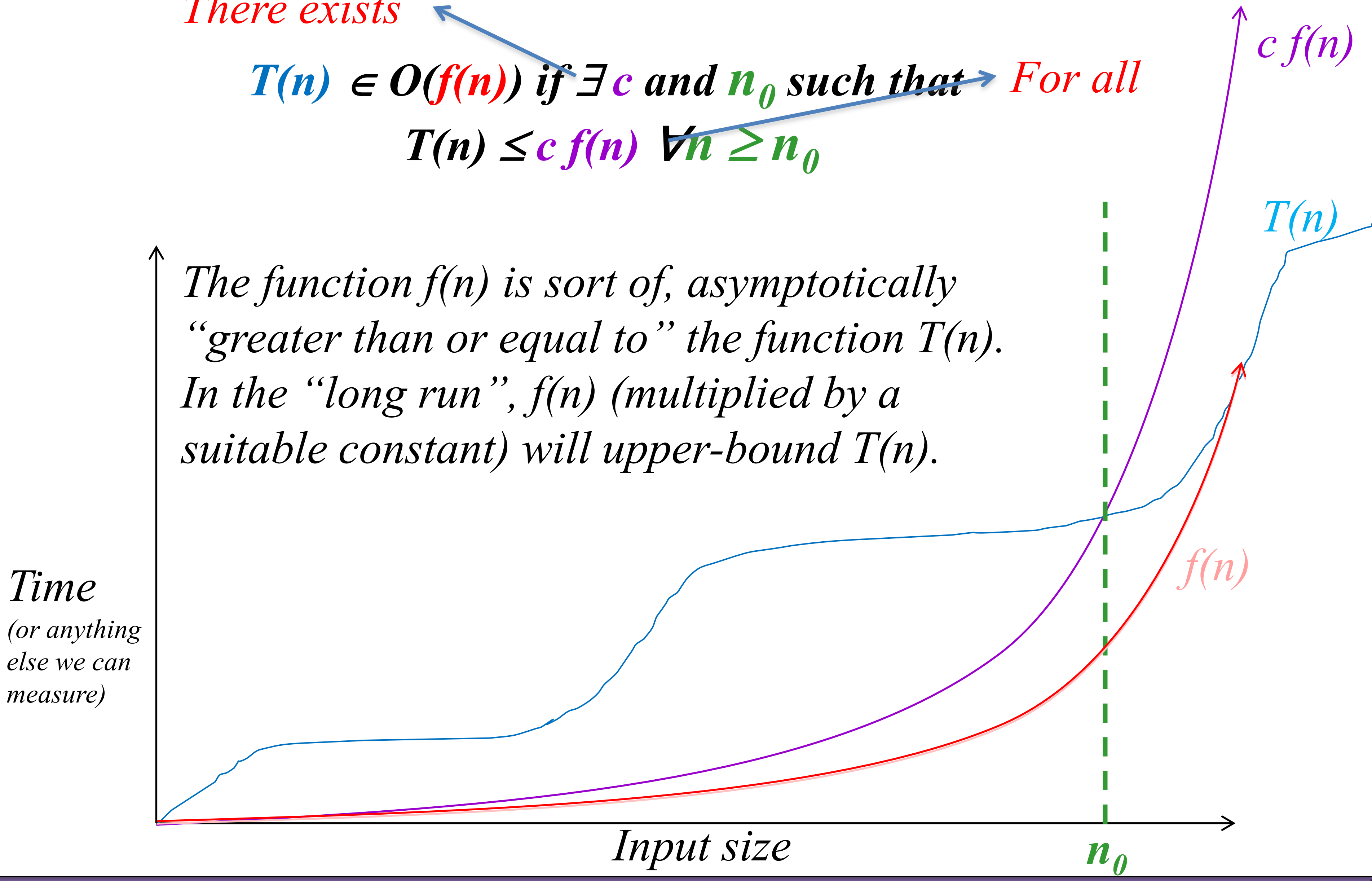
We want to compare the "overall" runtime (or memory usage or …) of a piece of code against a familiar, simple function.

Big-O Notation

There exists

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

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



The function f(n) is sort of, asymptotically "greater than or equal to" the function T(n). In the "long run", f(n) (multiplied by a suitable constant) will upper-bound T(n).

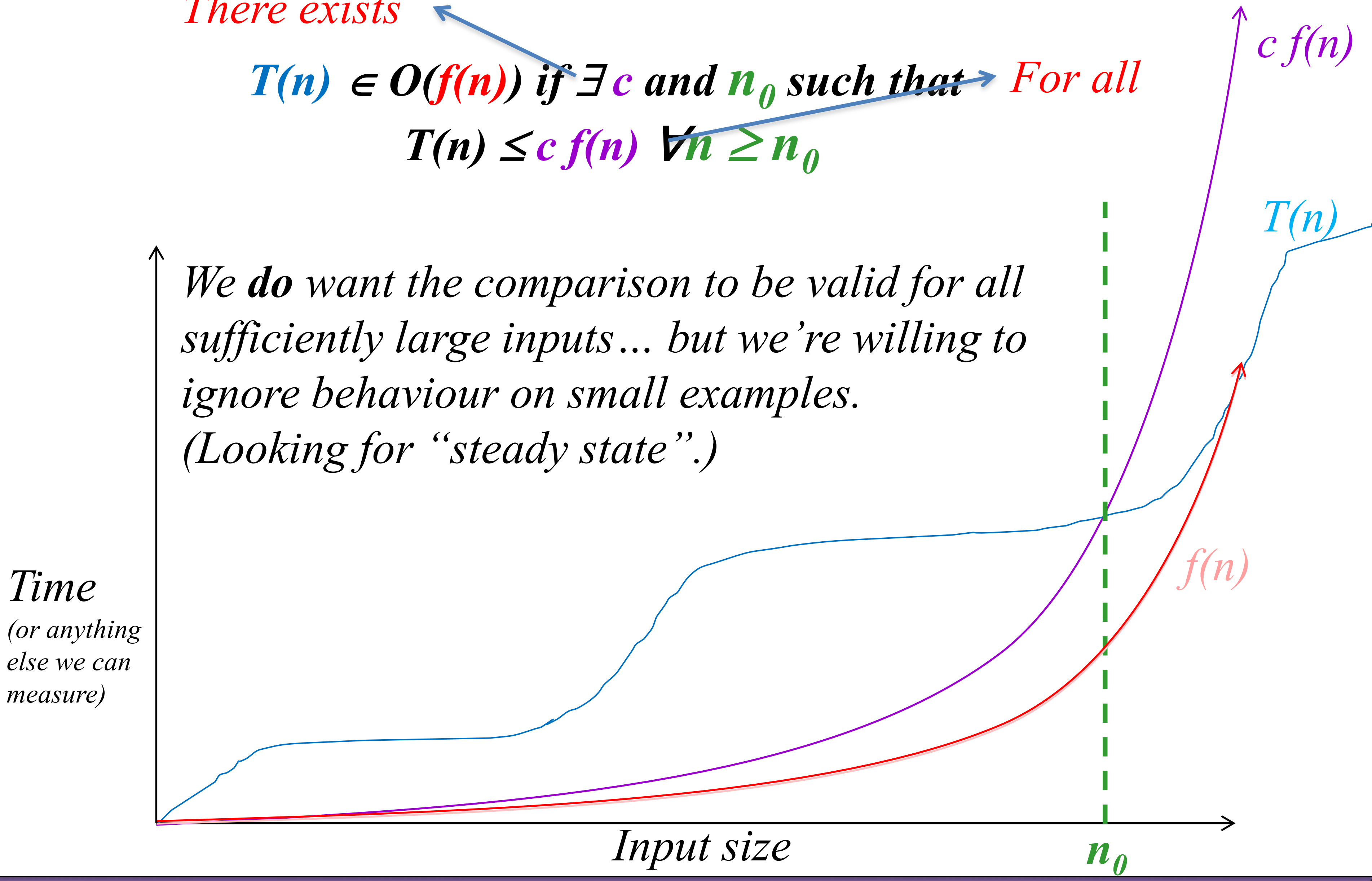
Big-O Notation

There exists

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

For all

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



*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 "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.

Other Bounds

- We have shown that Big-O is an upper bound for an algorithm.
- Big-Ω is a lower bound with analogous definition.
- Big-Θ is a tight bound, or put simply, a combination of Big-O and Big-Ω

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***

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!

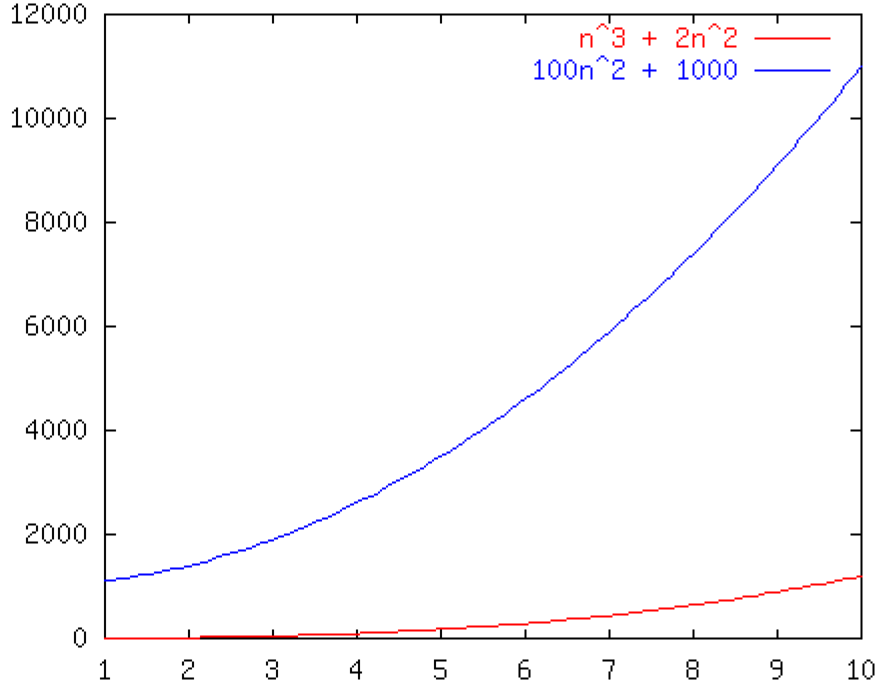
All analysis are done asymptotically

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

Race I

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

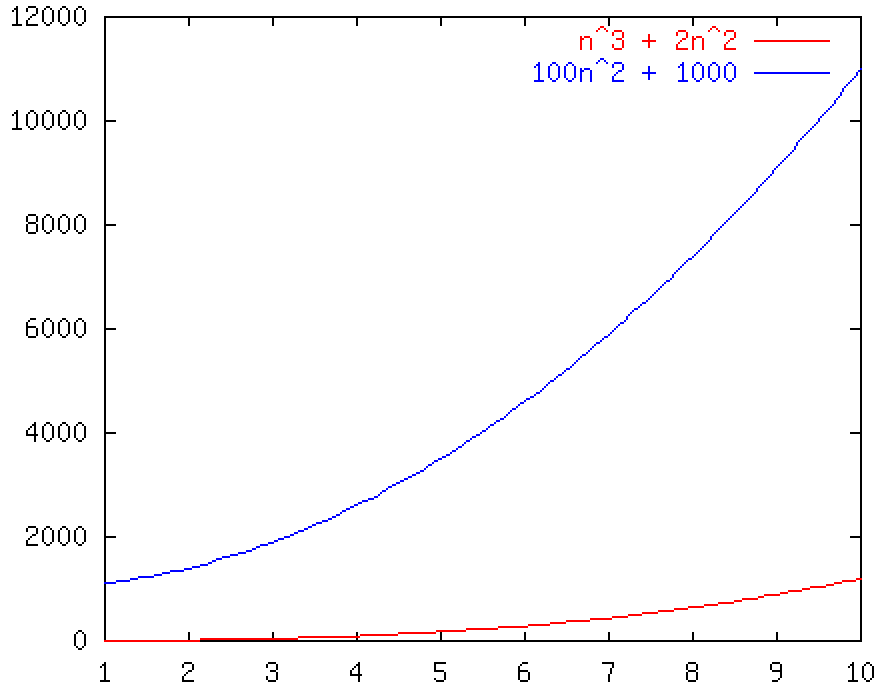
$n^3 + 2n^2$ vs. $100n^2 + 1000$

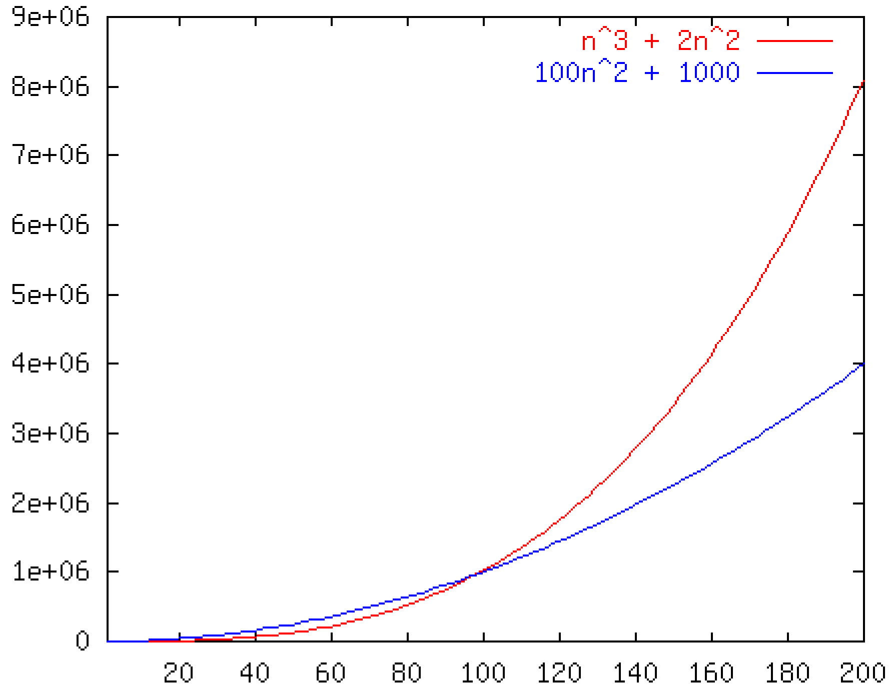


Race I

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

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

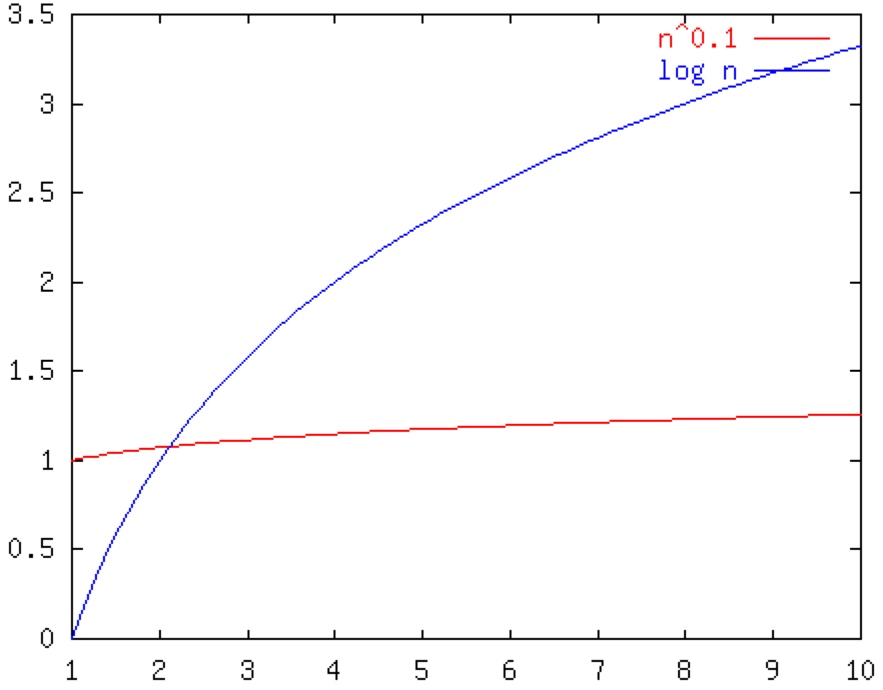




Race II

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

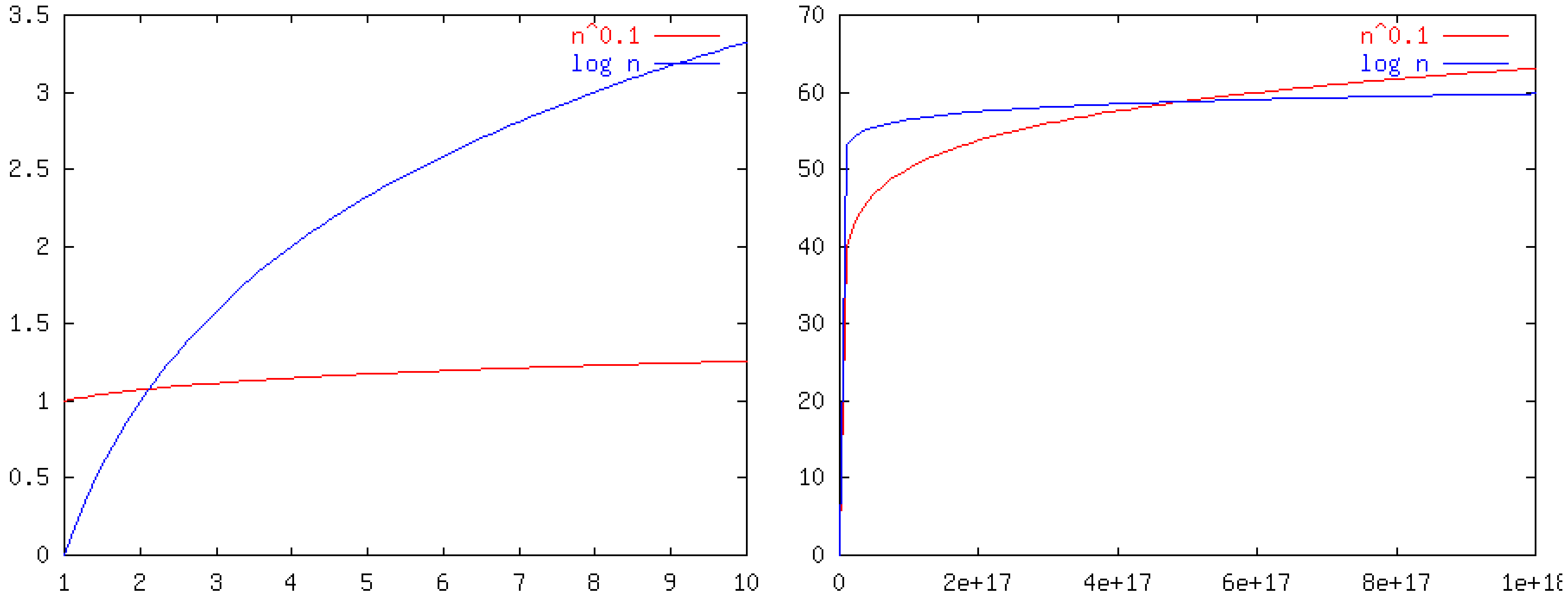
$n^{0.1}$ *vs.* $\log n$



Race II

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

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

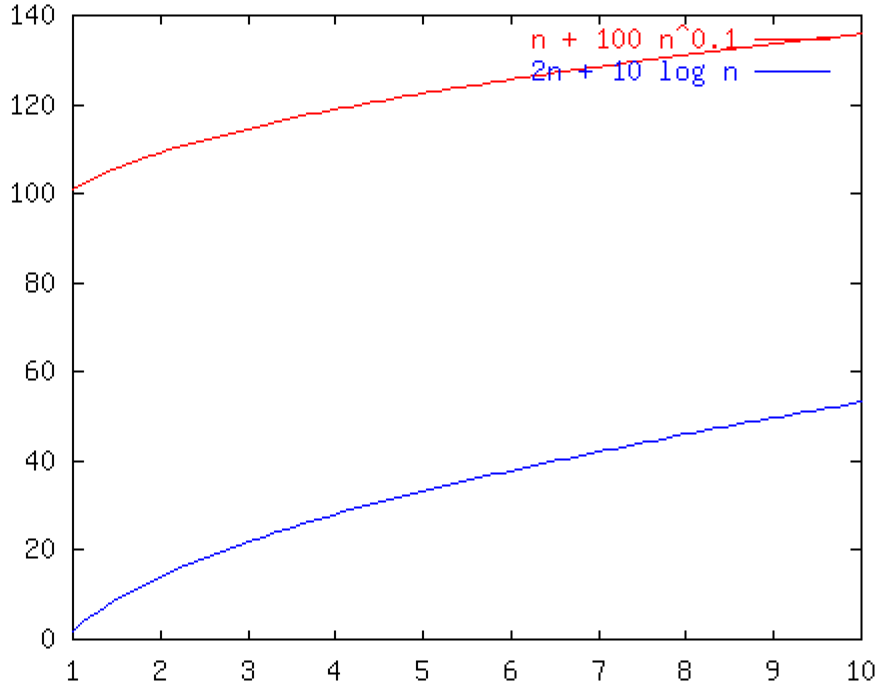


Moral of the story? n^{ϵ} *is slower than* $\log n$ *for any* $\epsilon > 0$

Race III

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

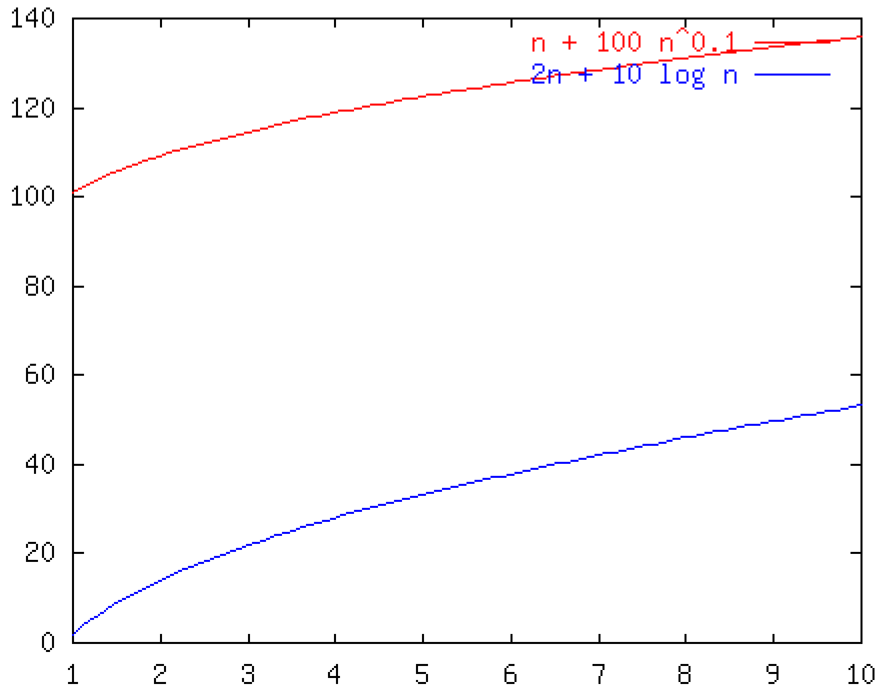
$n + 100n^{0.1}$ *vs.* $2n + 10 \log n$

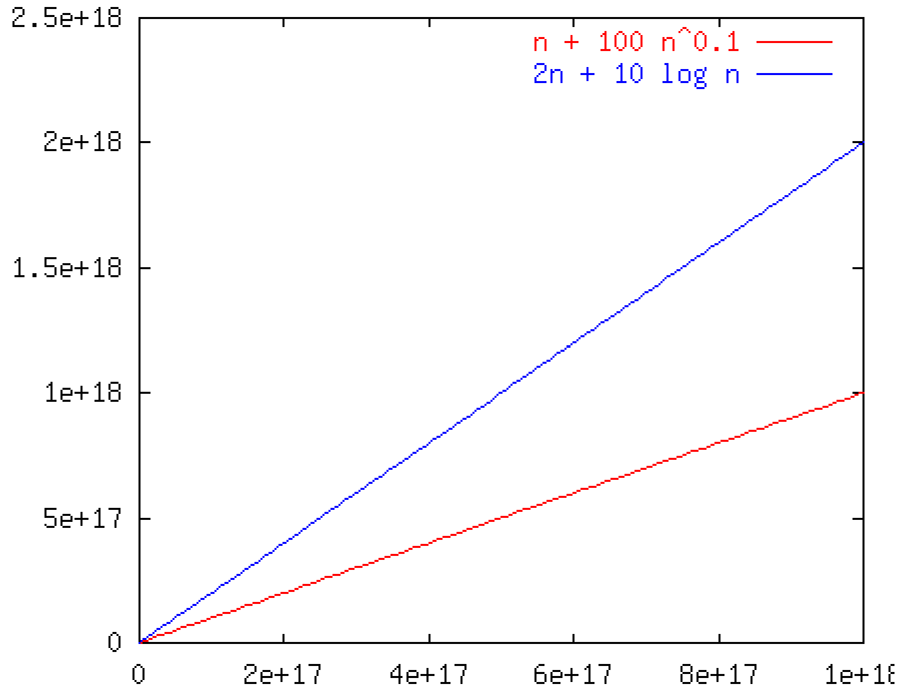


Race III

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

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



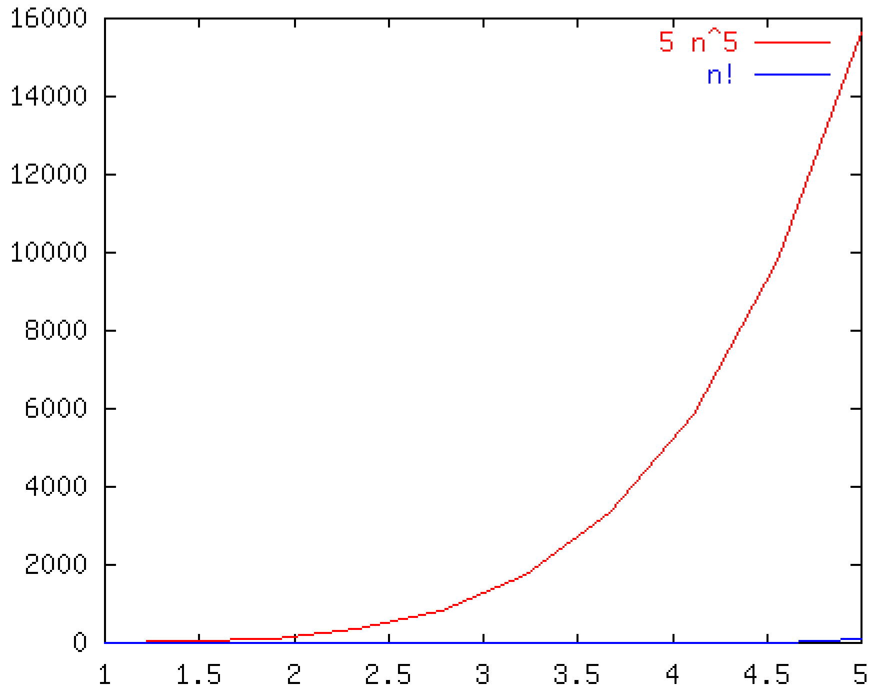


Although left seems faster, asymptotically it is a TIE

Race IV

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

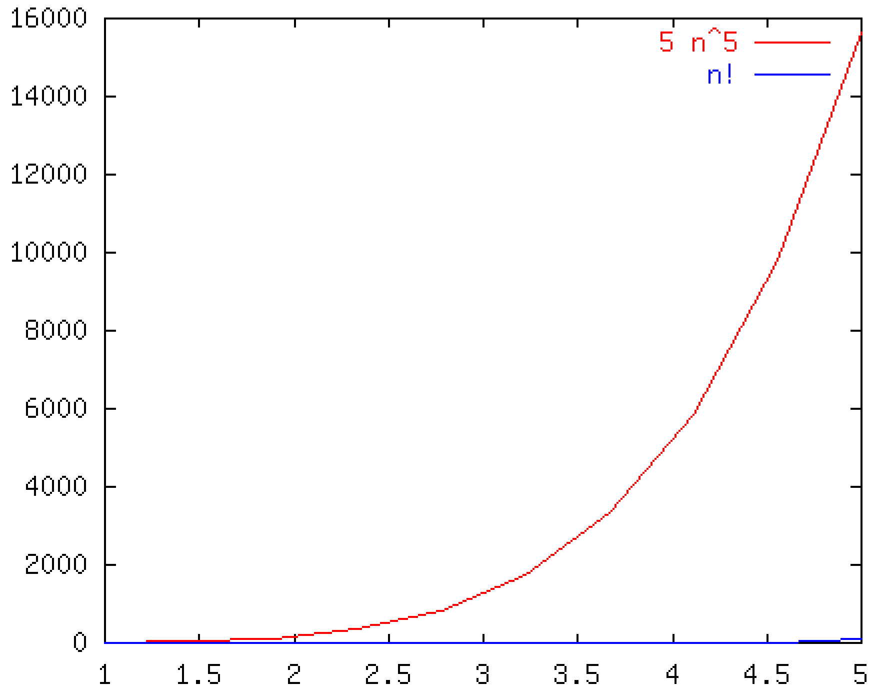
$5n^5$ *vs.* **$n!$**

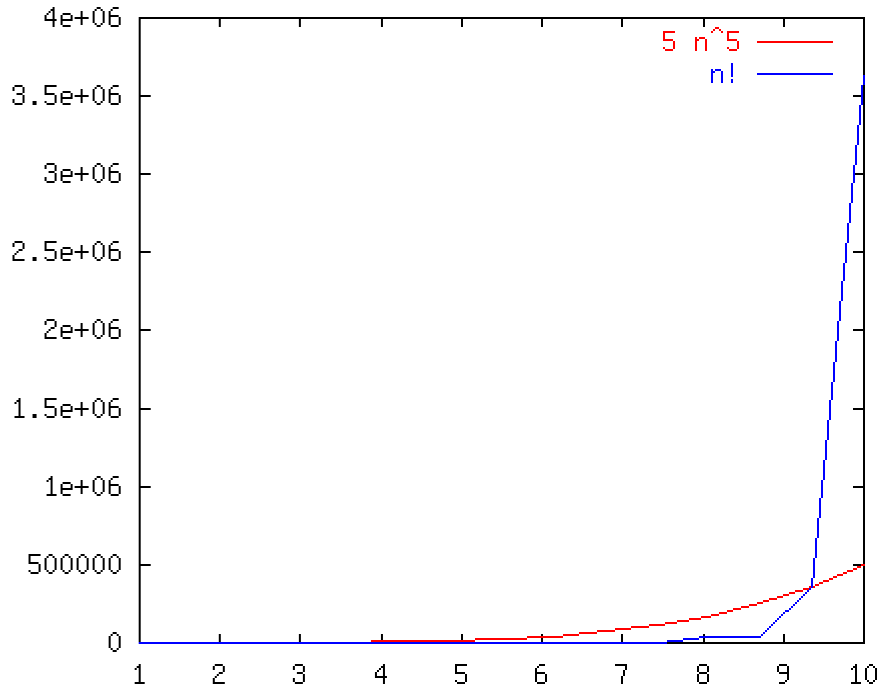


Race IV

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

$5n^5$ *vs.* $n!$

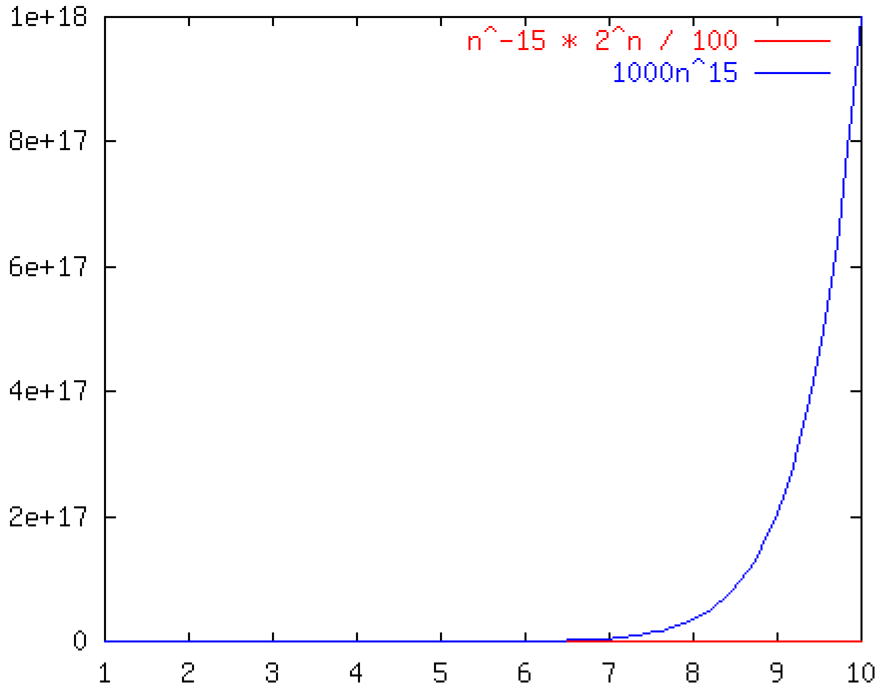




Race V

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

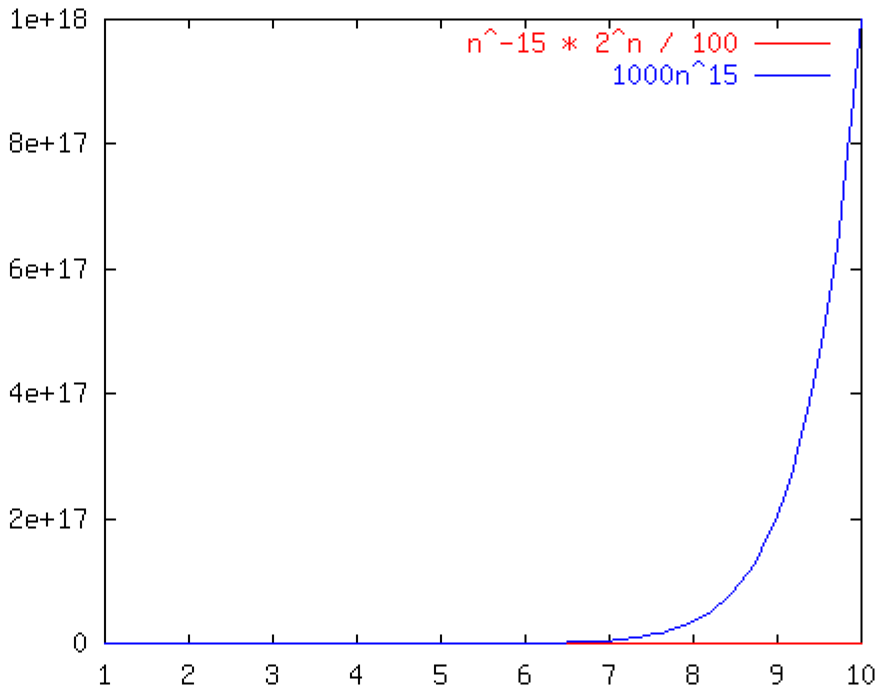
$n^{-15}2^n/100$ vs. $1000n^{15}$

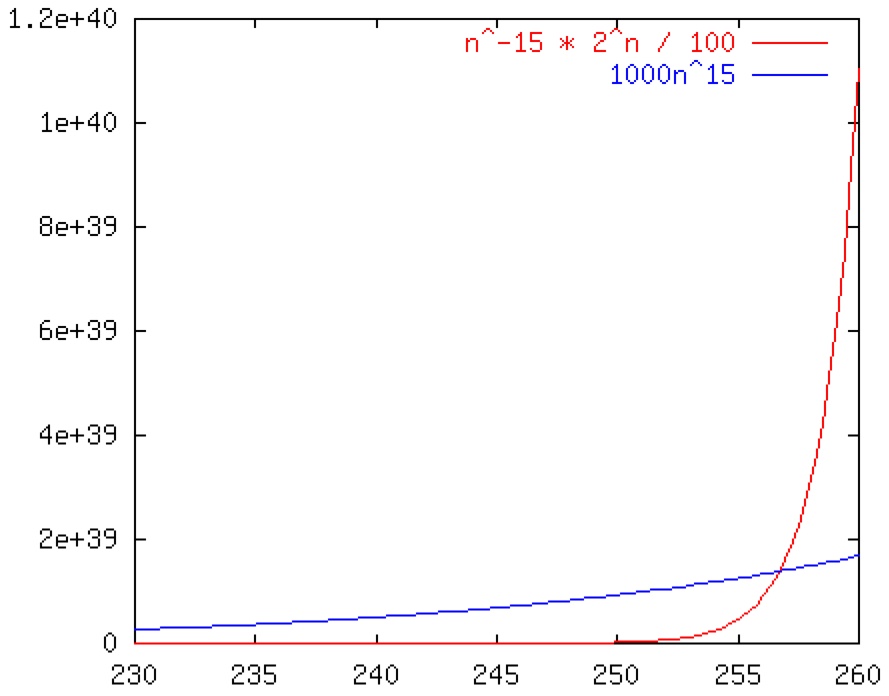


Race V

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

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



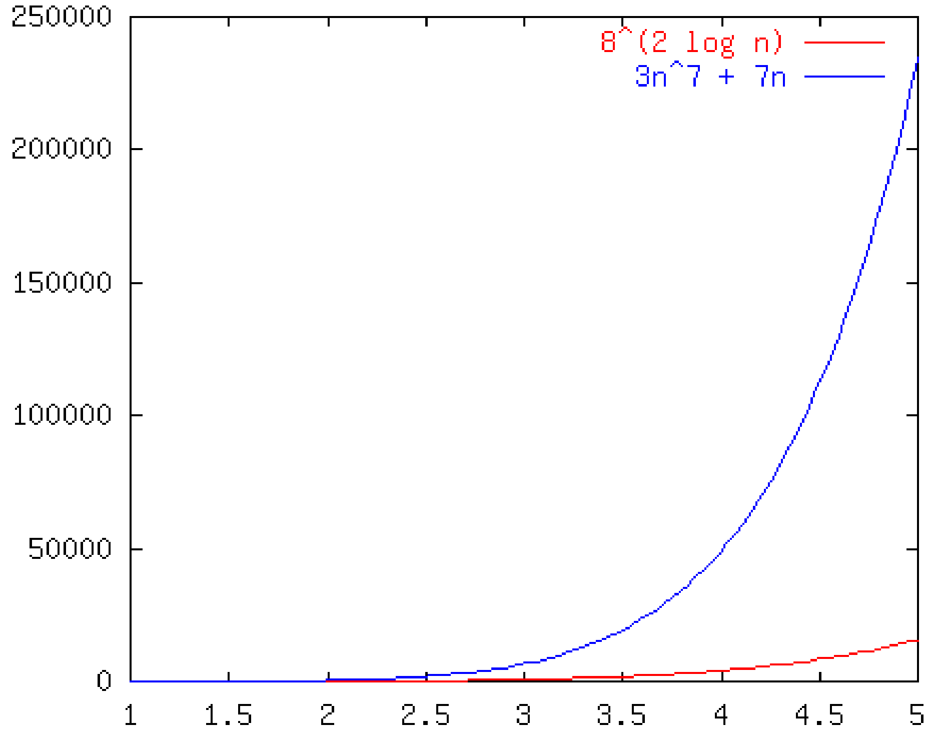


Any exponential is slower than any polynomial.
It doesn't even take that long here (~250 input size)

Race VI

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

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



Log Rules:

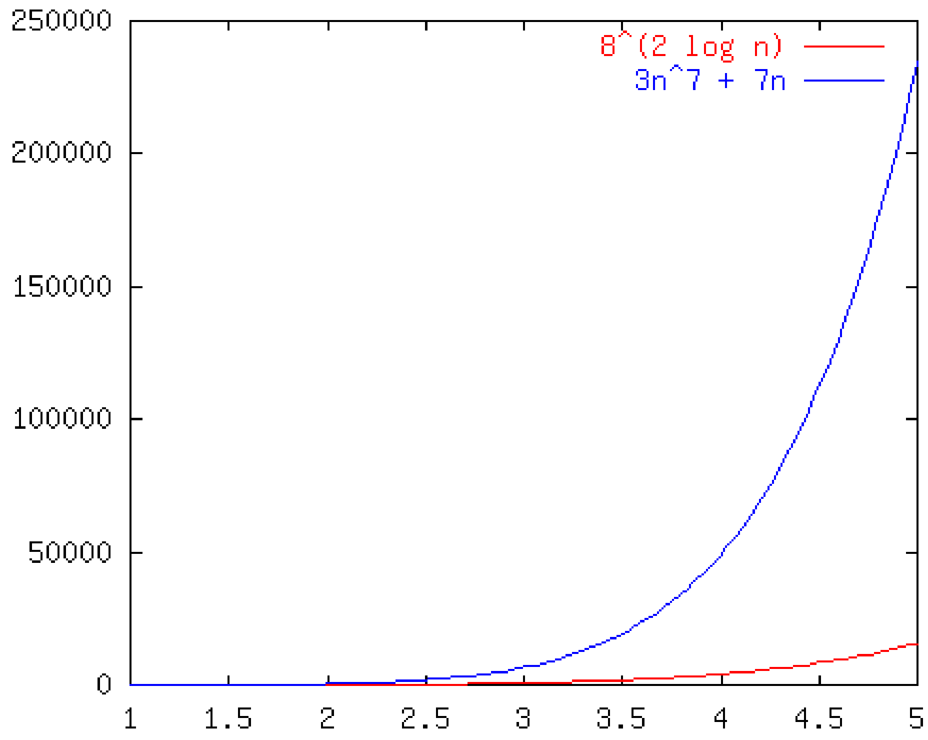
1) $\log(mn) = \log(m) + \log(n)$
2) $\log(m/n) = \log(m) - \log(n)$
3) $\log(m^n) = n \cdot \log(m)$
4) $n = 2^k \rightarrow \log_2 n = k$

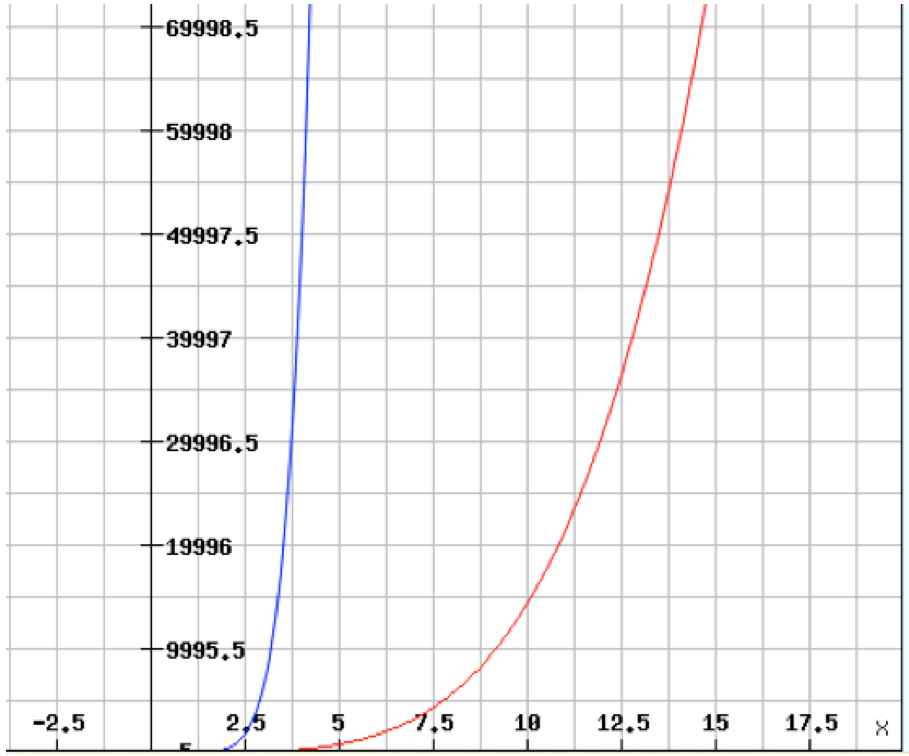
Log Rules:
1) log(mn) = log(m) + log(n)
2) log(m/n) = log(m) – log(n)
3) $\log(m^n) = n \cdot \log(m)$
4) $n = 2^k \rightarrow \log n = k$

Race VI

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

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





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

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 and what we care about most

- The fix sheet (typical growth rates in order)
  - constant: $O(1)$
  - **logarithmic: $O(\log n)$ ($\log_k n$, $\log n^2 \in O(\log n)$)**
  - poly-log: $O(\log^k n)$ (k is a constant >1)
  - 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: $O(\log n)$
- poly-log: $O(\log^k n)$
- 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

Examples

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

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

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

$n^3 + 4 \in O(n^3)$

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? 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.

Example 1

```
for i = 1 to n do                1  ⎤
  for j = 1 to n do              1  ⎥ ]n times   ]n times
    sum = sum + 1                1  ⎦
```

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

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

Example 1 (simpler version)

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

$$\left.\begin{matrix}1\\ \left.\begin{matrix}1\\1\end{matrix}\right] n \text{ times}\end{matrix}\right] n \text{ times}$$

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

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

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

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

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

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 2 (Faster/Slower Code 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
```

- This code is "too hard" to deal with. So, let's find just an upper bound.
  - In which case we get to change the code so in any way that makes it run no faster (even if it runs slower).
  - We'll let j go from 1 to n rather than i to n. Since i ≥ 1, this is no less work than the code was already doing…

Example 2 (Faster/Slower Code Approach)

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

•But this is just an upper bound $O(n^2)$, since we made the code run slower.

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

•Could it actually run faster?

Example 2 (Faster/Slower Code 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
```

- Let's do a lower-bound, in which case we can make the code run faster if we want.
  - Let's make j start at n-1. Does the code run faster? Is that helpful?

Runs faster but you get $\Omega(n)$ which is not what we want

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

Example 2 (Faster/Slower Code 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
```

- Let's do a lower-bound, in which case we can make the code run faster if we want.
  - Let's make j start at n/2. Does the code run faster? Is that helpful?

Hard to argue that it is faster. Every inner loop now runs n/2 times

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

Example 2(Faster /Slower Code Approach)

```
i = 1                   takes "1" step
while i < n/2 +1 do            goes n/2 times
  for j = n/2 +1 to n do       goes n/2 times
    sum = sum + 1         takes "1" step
  i++                   takes "1" step
```

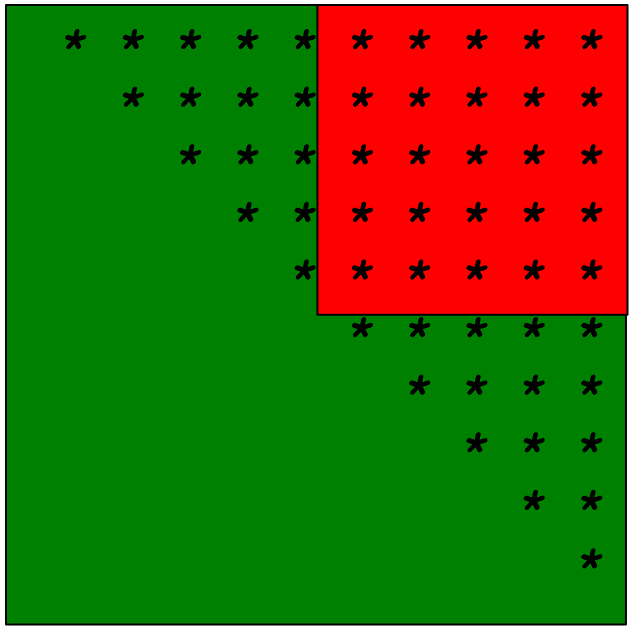
- Let's change the bounds on both i and j to make both loops faster.

$$T(n) = \sum_{i=1}^{n/2} \sum_{j=1}^{n/2} 1 = \sum_{i=1}^{n/2} (n/2) = n^2/4 \in \Omega(n^2)$$

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

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

Note Pretty Pictures and Faster/Slower are the Same(ish) Picture



- Both the overestimate (upper-bound) and the underestimate (lower-bound) are proportional to n^2

Example 2.5

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

Time complexity:

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

Example 2.5

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

j= 1, 2, 4, 8, 16, 32, …

x <= n < 2x

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

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

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 3

- Conditional

```
if C then S₁ else S₂
```

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

or

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

- Loops

```
while C do S
```

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

Learning Objectives Revisited

Description	Tag
Define which program operations we measure in an algorithm in order to approximate its efficiency.	Big-O
Define "input size" and determine the effect (in terms of performance) that input size has on an algorithm.	
Give examples of common practical limits of problem size for each complexity class.	
Define the big-O notation.	
Give examples of tractable problems.	
Give examples of intractable problems.	
Categorize an algorithm into one of the common complexity classes.	
Describe best-case analysis and reason why it is rarely relevant	
Describe worst-case analysis and reason why it may never be encountered	
Describe average-case analysis and reason why it is often hard to compute	
Compare and contrast best-, worst-, and average-case analysis.	
Given code, write a formula which measures the number of steps executed as a function of the size of the input (N).	
Given two or more algorithms, rank them in terms of their time complexity.	