COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures” by Douglas Wilhelm Harder

Analysis of Algorithms

Algorithm run times

Once we have chosen a data structure to store both the objects and the relationships, we must implement the queries or operations as algorithms

- The Abstract Data Type will be implemented as a class
- The data structure will be defined by the member variables
- The methods will implement the algorithms

The question is, how do we determine the efficiency of the algorithms?

Asymptotic Analysis

The next topic, asymptotic analysis, will provide the mathematics that will allow us to measure the efficiency of algorithms

It will also allow us to measure the memory requirements of both the data structure and any additional memory required by the algorithms

Outline

- In this topic, we will look at:
  - Justification for analysis
  - Quadratic and polynomial growth
  - Counting machine instructions
  - Big-**O**
  - Big-Θ

Background

- Suppose we have two algorithms, how can we tell which one is better?
- We could implement both algorithms, run them both
  - Expensive and error prone
- Preferably, we should analyze them mathematically
  - *Algorithm analysis*

Asymptotic Analysis

- In general, we will always analyze algorithms with respect to one or more variables
- This variable is the input size of the algorithm
  - The number of items n currently stored in an array or other data structure
  - The number of items expected to be stored in an array or other data structure
  - The dimensions of an $n \times n$ matrix
- Examples with multiple variables:
  - Dealing with n objects stored in m memory locations
  - Multiplying a $k \times m$ and an $m \times n$ matrix
  - Dealing with sparse matrices of size $n \times n$ with m non-zero entries

Maximum Value

For example, the time taken to find the largest object in an array of n random integers needs how many operations?

```
private static int find_max(int[] arr) {
    int max = arr[0];
    for (int i = 0; i < arr.length; i++) {
            if (arr[i] > max) {
                    max = arr[i];
            }
    }
    return max;
}
```

How many simple instructions we need to run the algorithm?

Maximum Value

For example, the time taken to find the largest object in an array of N random integers needs how many operations?

```
     private static int find_max(int[] arr) {
1       int max = arr[0];
3*N     for (int i = 0; i < arr.length; i++) {
           N    if (arr[i] > max) {
   2 <? < N         max = arr[i];
                }
        }
1       return max;
     }
```

$cN+c'$ instructions

c, c' constants

How many simple instructions we need to run the algorithm?

Example

- Suppose an algorithm for processing a retail store's inventory takes:
    - 10,000 milliseconds to read the initial inventory from disk, and then
    - 10 milliseconds to process each transaction (items acquired or sold).
- Processing n transactions takes (10,000 + 10 n) ms. Even though 10,000 >> 10, we sense that the "10 n" term will be more important if the number of transactions is very large.

Example

- We also know that these coefficients will change if we buy a faster computer or disk drive, or use a different language or compiler. We want a way to express the speed of an algorithm independently of a specific implementation on a specific machine--specifically, we want to ignore constant factors (which get smaller and smaller as technology improves).

Asymptotic Analysis

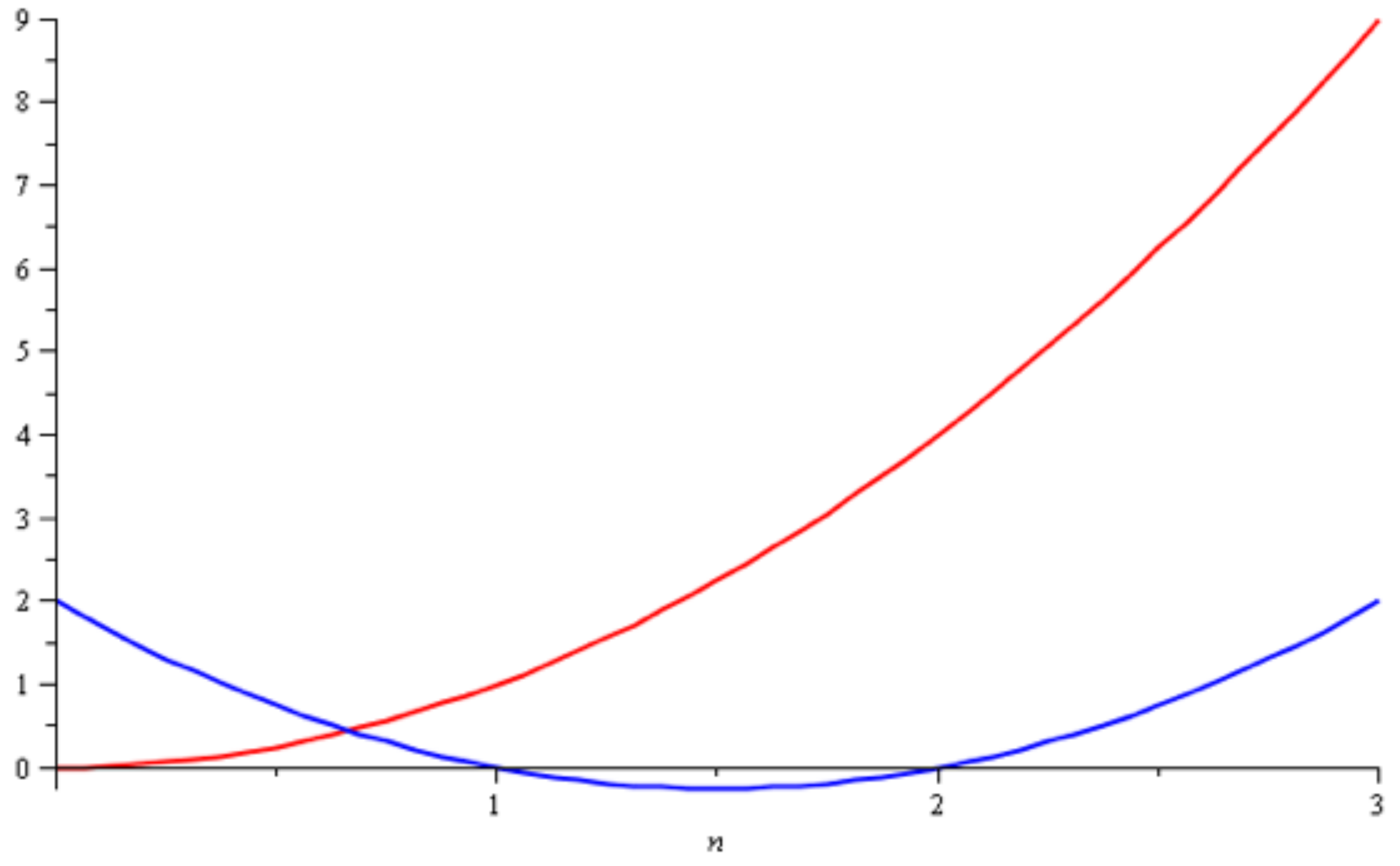
- Given an algorithm:
    - We need to be able to describe these values mathematically
    - We need a systematic means of using the description of the algorithm together with the properties of an associated data structure
    - We need to do this in a machine-independent way
- For this, we need asymptotic analysis

Quadratic Growth

Consider the two functions

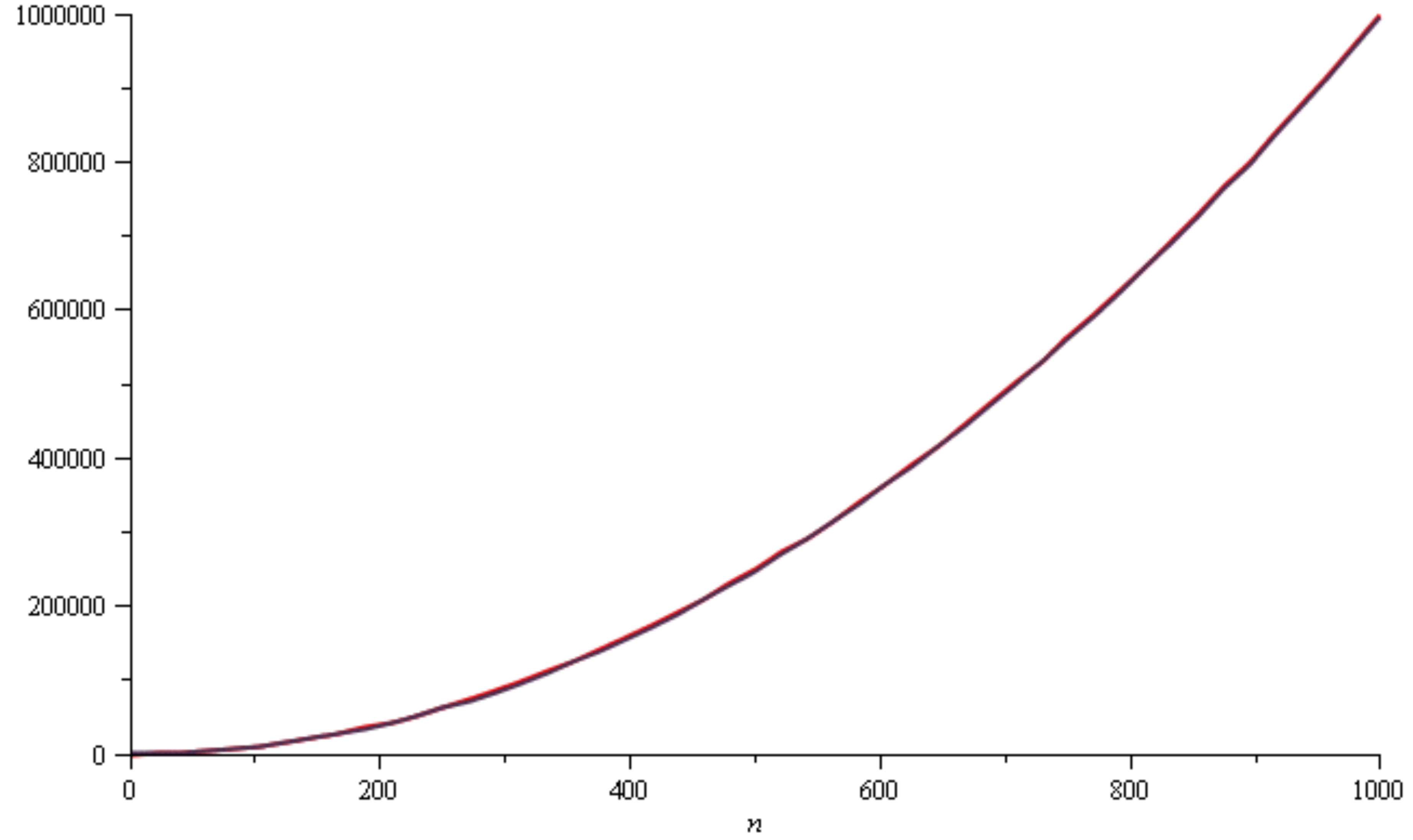
$f(n) = n^2$ and $g(n) = n^2 - 3n + 2$

Around $n = 0$, they look very different



Quadratic Growth

Yet on the range $n = [0, 1000]$, they are (relatively) indistinguishable:



Quadratic Growth

The absolute difference is large, for example,

$$f(1000) = 1\ 000\ 000$$

$$g(1000) = \quad 997\ 002$$

but the relative difference is very small

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

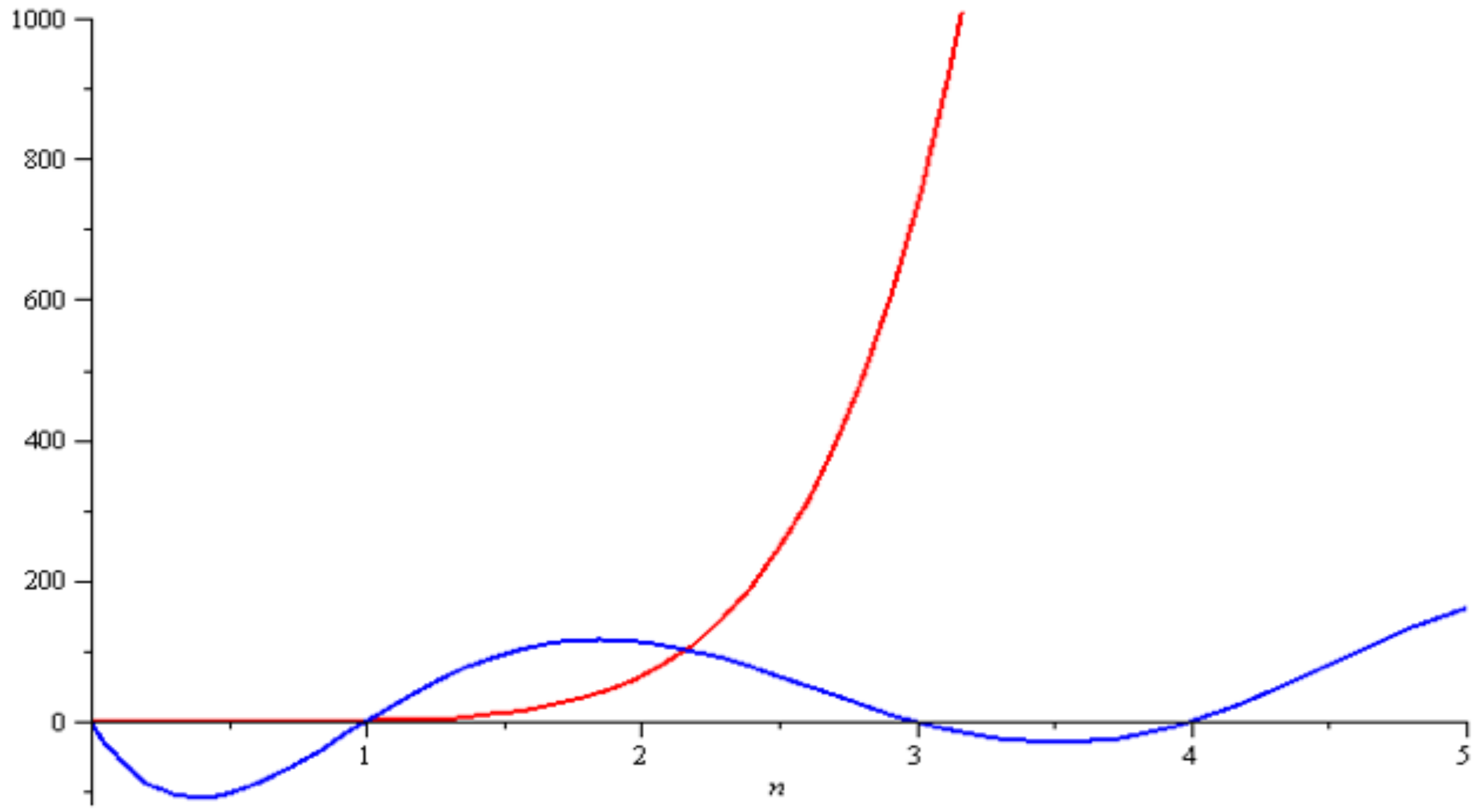
and this difference goes to zero as $n \to \infty$

Polynomial Growth

To demonstrate with another example,

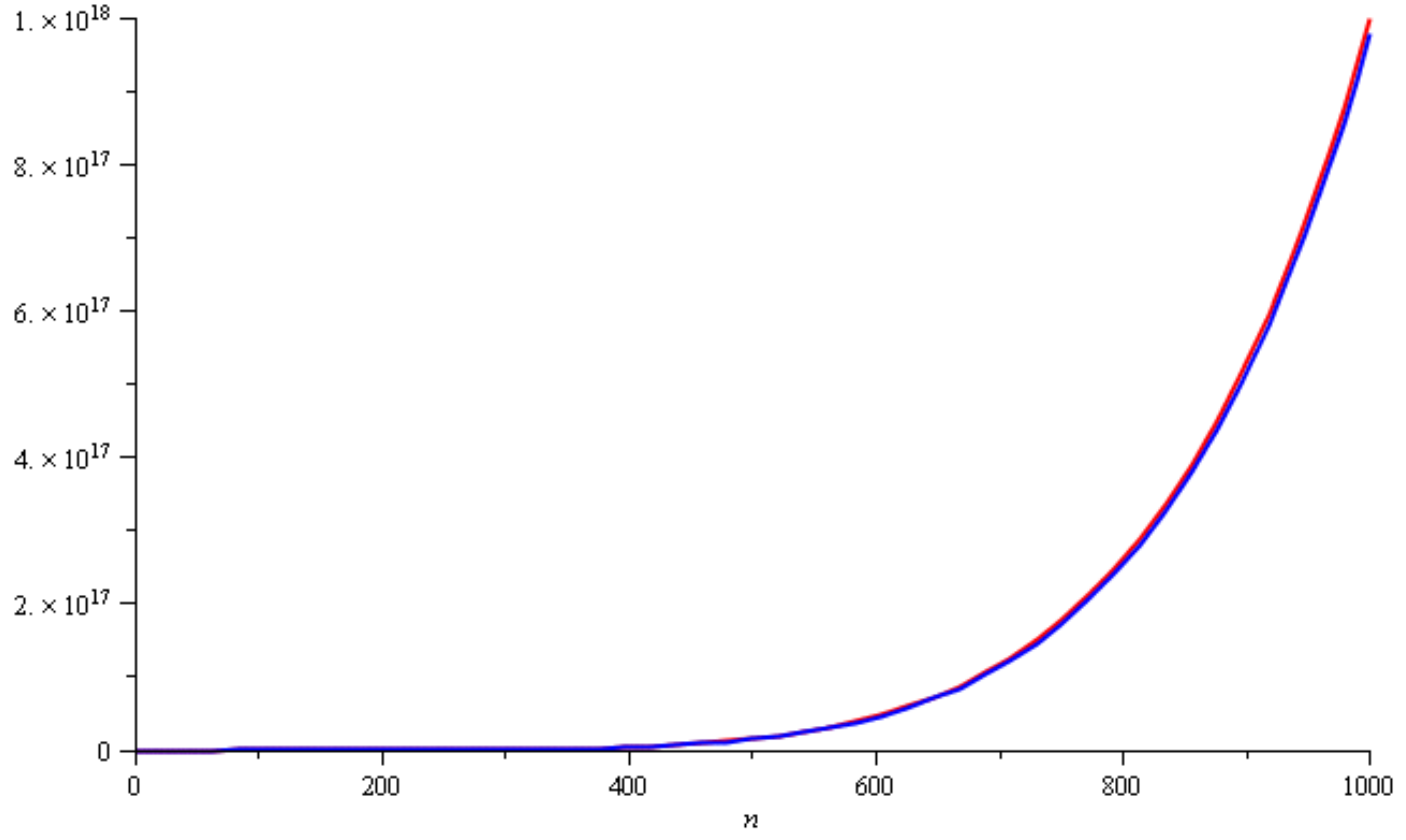
$f(n) = n^6$ and $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$

Around $n = 0$, they are very different



Polynomial Growth

Still, around $n = 1000$, the relative difference is less than 3%



Polynomial Growth

The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:

n^2 in the first case, n^6 in the second

Suppose however, that the coefficients of the leading terms were different

– In this case, both functions would exhibit the same rate of growth, however, one would always be proportionally larger

Big **O**

(upper bound)

- A function f(n) is in **O**(g(n)) if there exists N and c such that

$$f(n) < c\, g(n)$$

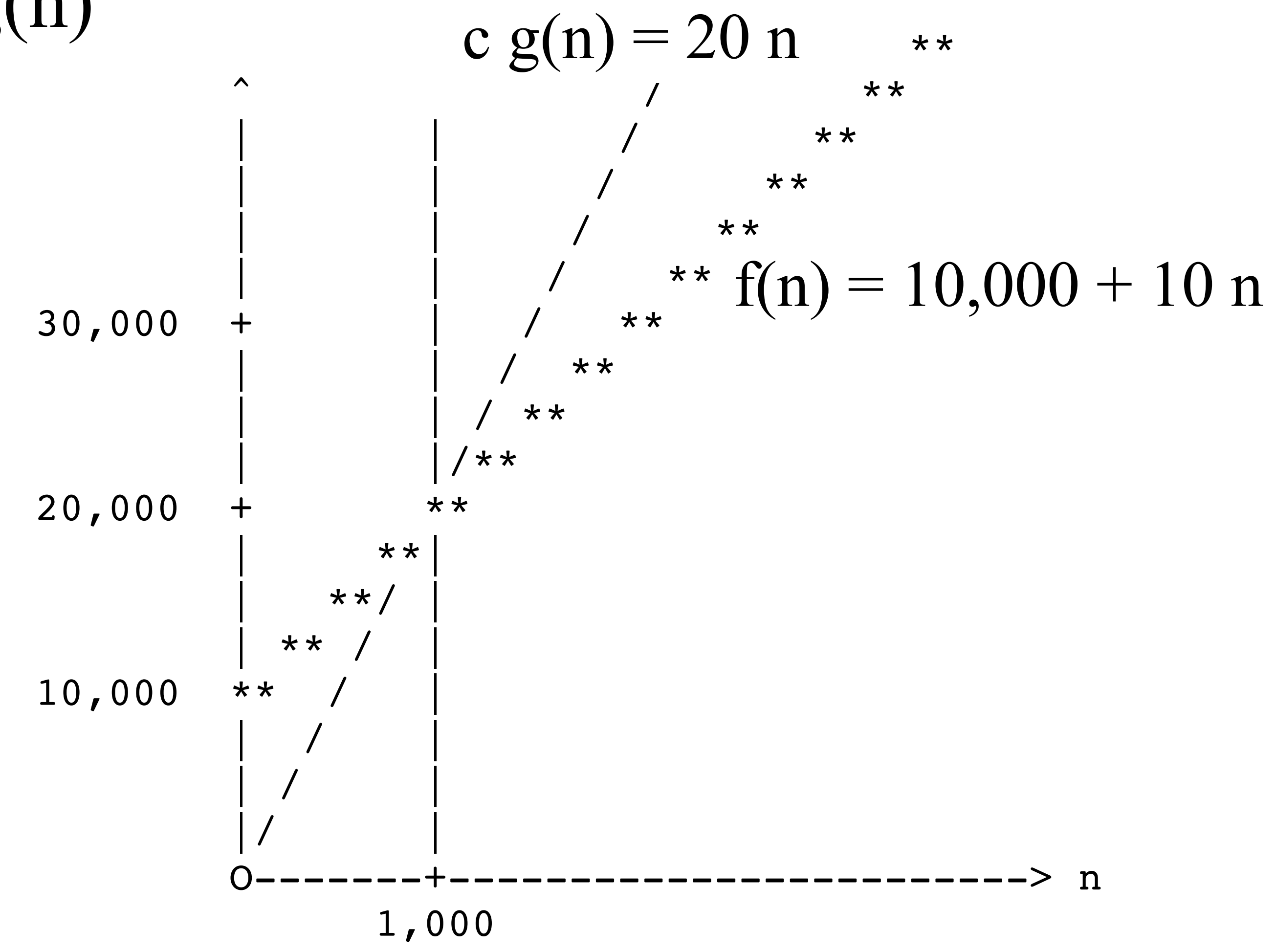
whenever $n > N$

  – The function f(n) has a rate of growth no greater than that of g(n)

EXAMPLE: Inventory

- For our inventory example $f(n) = 10{,}000 + 10\ n$.
- Let's try out $g(n) = n$
  - We can choose c as large as we want (and we're trying to make $f(n)$ fit underneath $c\ g(n)$
  - $c=20$, $N=1000$

For large n (any n bigger than 1,000)
f(n) <= c g(n)



Example

- $f(n)=1{,}000{,}000\ n$ is in $O(n)$.
- Proof:
    - set $c = 1{,}000{,}000$, $N = 0$.
    - Therefore, Big-O doesn't care about (most) constant factors.
- We generally leave constants out; it's unnecessary to write $O(2n)$, because $O(2n) = O(n)$. (The 2 is not wrong; just unnecessary. We could chose any linear function ($3n$, $n+1000$, …), but we always choose the simplest one which is n)

Example

- f(n)= n^3+n^2+n is in O(n^3)?

Example

- $f(n) = n^3 + n^2 + n$ is in $O(n^3)$?
- Proof: set c = 3, N = 1.
- Big-O is usually used only to indicate the dominating (largest and most displeasing) term in the function. The other terms become insignificant when n is really big.

Table of Important Big-Oh Sets

	$O(1)$	constant
is a subset of	$O(\ln(n))$	logarithmic
is a subset of	$O(\mathrm{root}(n))$	root
is a subset of	$O(n)$	linear
is a subset of	$O(n \log(n))$	“n log n”
is a subset of	$O(n^2)$	quadratic
is a subset of	$O(n^3)$	cubic
is a subset of	$2^n, e^n, 4^n, \ldots$	exponential

General Facts

- Algorithms that run in $n \log(n)$ time or faster are considered efficient.

- Algorithms that take n^7 time or more are usually considered useless.

- In the region between $n \log(n)$ and n^7, the usefulness of an algorithm depends on the typical input sizes and the associated constants hidden by the Big-O.

Fallacious Proof

- n^2 is in $O(n)$,
- Proof: if we choose $c = n$, we get $n^2 <= n^2$

Fallacious Proof

- n^2 is in $O(n)$,
- Proof: if we choose $c = n$, we get $n^2 <= n^2$
- *WRONG! c must be a constant; it <u>cannot depend on n</u>.*

Question

- Is it correct: e^{3n} is in $\mathrm{O}(e^n)$ because constant factors don't matter?

Question

- Is it correct: e^{3n} is in $O(e^n)$ because constant factors don't matter.

- Big-O doesn't care about (most) constant factors. Here are some of the exceptions.

    - A constant factor in an exponent is not the same as a constant factor in front of a term.

    - e^{3n} is not bigger than e^n by a constant factor; it's bigger by a factor of e^{2n}, which is damn big.

Question

- Is it correct: "10^n is in $O(2^n)$ because constant factors don't matter?

Question

- Is it correct: "10^n is in $O(2^n)$ because constant factors don't matter?

- 10^n is not bigger than 2^n by a constant factor; it's bigger by a factor of 5^n, which is not constant.

Arbitrary entry in a sorted list

The time taken to find an entry in a sorted array of size N

```
private static int find_max(int[] arr, int val) {
   int index = -1;
   for (int i = 0; i < arr.length; i++) {
           if (arr[i] == val) {
                   index = i;
           }
   }
   return index;
}
```

O(?)

Maximum Value

The time taken to find an entry in a sorted array of size N

```
      private static int find_max(int[] arr) {
  1     int max = arr[0];
1+2*N   for (int i = 0; i < arr.length; i++) {
            N  if (arr[i] > max) {
     2 <? < N       max = arr[i];
               }
        }
  1     return max;
      }
```

$cN+c'$ instructions

c, c' constants

How many simple instructions we need to run the algorithm?

Arbitrary entry in a sorted list

The time taken to find an entry in a sorted array of size N

```
private static int find_max(int[] arr, int val) {
   int index = -1;
   for (int i = 0; i < arr.length; i++) {
           if (arr[i] == val) {
                   index = i;
           }
   }
   return index;
}
```

$cN+c'$ operations

c, c' constants

O(N)

Warning

- Big-Oh notation doesn't tell the whole story, because it leaves out the constants.
- If one algorithm runs in time T(n) = n log_2 n, and another algorithm runs in time U(n) = 100 n, then Big-Oh notation suggests you should use U(n), because T(n) dominates U(n) asymptotically.
- However, U(n) is only faster than T(n) in practice if your input size is greater than current estimates of the number of subatomic particles in the universe. The base-two logarithm log_2 n < 50 for any input size n you are ever likely to encounter.
- Nevertheless, Big-Oh notation is still a good rule of thumb, because the hidden constants in real-world algorithms usually aren't that big

Upper Bound

- f(n)= n is in $O(n^3)$?

Upper Bound

- f(n)= n is in $O(n^3)$?
- Proof:
    - set c = 1, N = 1.
- Therefore, Big-O can be misleading.
- Just because an algorithm’s running time is in $O(n^3)$ doesn't mean it's slow; it might also be in $O(n)$.
- Big-O only gives us an UPPER BOUND on a function.

Big Θ

A function $f(n)$ is in $\Theta(g(n))$ if there exist positive N, c_1, and c_2 such that

$$c_1\, g(n) < f(n) < c_2\, g(n)$$

whenever $n > N$

– The function $f(n)$ has a rate of growth equal to that of $g(n)$

Example

- $f(n)=1{,}000{,}000\, n$ is in $\Theta(n)$.
- Proof:
  - set $c_1 = 1$, $c_2 = 1{,}000{,}001$ $N = 1$

$$n < 1{,}000{,}000\, n < 1{,}000{,}001\, n$$

- We generally leave constants out; it's unnecessary to write $\Theta(2n)$, because $\Theta(2n) = \Theta(n)$. (The 2 is not wrong; just unnecessary. We could chose any linear function ($3n$, $n+1000$,…), but we always choose the simplest one which is n)

Example

- f(n)= n is in $O(n^3)$ Correct
- f(n)= n is in $\Theta(n^3)$ Wrong

Example

- f(n)= n^3+n^2+n is in $\Theta(n^3)$?

Example

- f(n)= n^3+n^2+n is in $\Theta(n^3)$?
    - set $c_1 = 1, c_2 = 3, N = 1$

$$n^3 < n^3+n^2+n < 3n^3$$

Big-Θ as an Equivalence Relation

The most common classes are given names:

$\Theta(1)$	constant
$\Theta(\ln(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \ln(n))$	"n log n"
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$2^n, e^n, 4^n, \ldots$	exponential

Logarithms and Exponentials

Recall that all logarithms are scalar multiples of each other

- Therefore $\log_b(n) = \Theta(\ln(n))$ for any base b

But, there is no single equivalence class for exponential functions:

- If $1 < a < b$, $\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} \left(\frac{a}{b}\right)^n = 0$
- Therefore we **cannot** say $a^n = \Theta(b^n)$

However, we will see that it is almost universally undesirable to have an exponentially growing function!

Logarithms and Exponentials

Plotting 2^n, e^n, and 4^n on the range $[1, 10]$ already shows how significantly different the functions grow

Note:

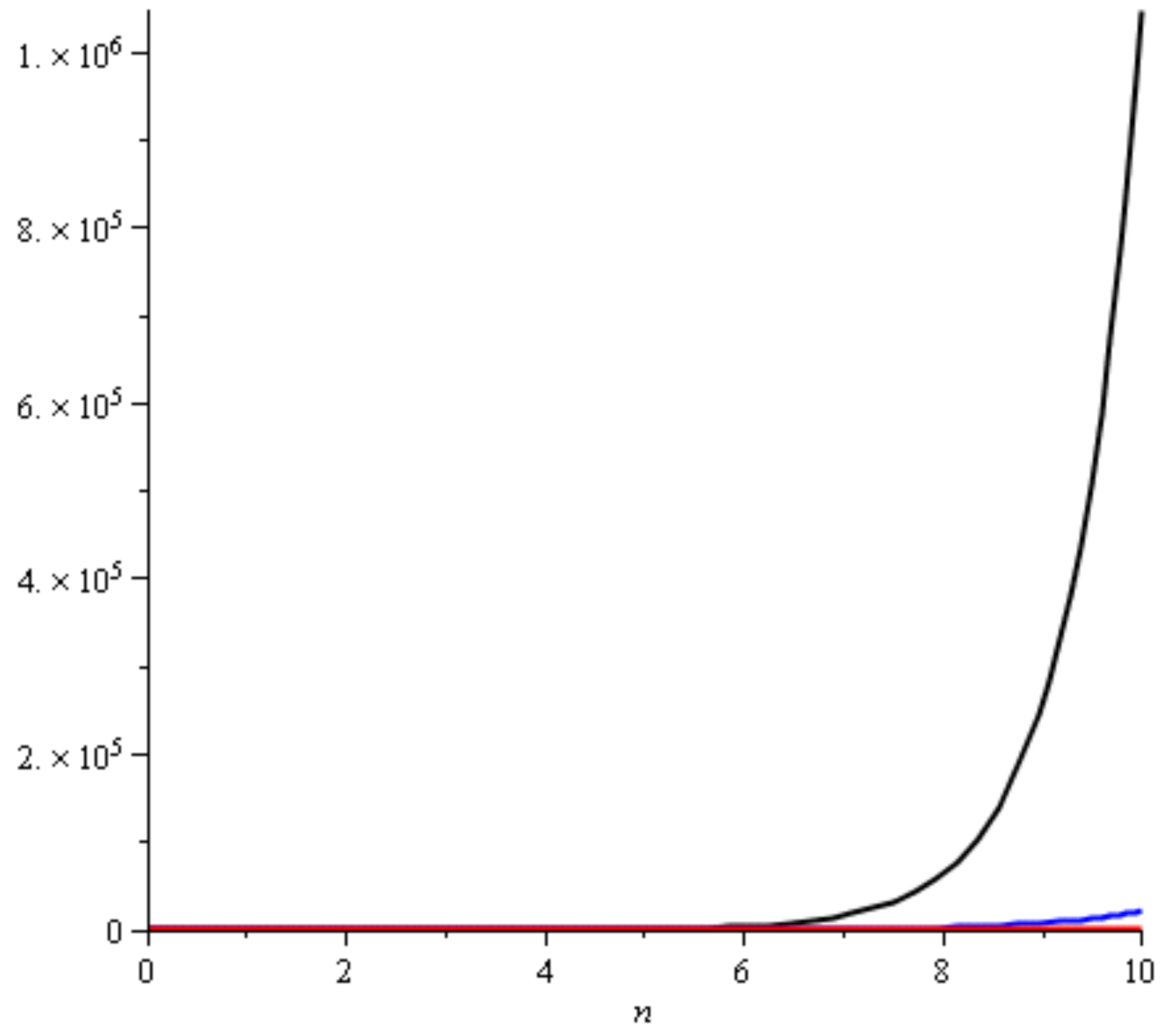
$2^{10} = 1024$

$e^{10} \approx 22\,026$

$4^{10} = 1\,048\,576$

Therefore

$a^n = \Theta(b^n)$ is wrong!



Example

For this code fragment, give a big-O analysis of the running time.

```
    1 int sum = 0;
1+2*n for (int i = 0; i < n; i++)
          n  sum++;
```

$3*n+2$ instructions

3, 2 constants

O(n)

Example

For this code fragment, give a big-O analysis of the running time.

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

Example

For this code fragment, give a big-O analysis of the running time.

```
          1  int sum = 0;
     1+2*?   for (int i = 0; i < n; i++)
?*(1+2*?)      for (int j = 0; j < n; j++)
        1*?      sum++;
```

Example

For this code fragment, give a big-O analysis of the running time.

```
          1  int sum = 0;
      1+2*n  for (int i = 0; i < n; i++)
n*(1+2*n)      for (int j = 0; j < n; j++)
   n*(1*n)       sum++;
```

$3*n^2 + 3*n + 2$ instructions

$O(n^2)$

Example

For this code fragment, give a big-O analysis of the running time.

```
int sum = 0;
for (int i = 1; i < n; i = i*2)
   sum++;
```

Example

For this code fragment, give a big-O analysis of the running time.

```
    1   int sum = 0;
1+2*?   for (int i = 1; i < n; i = i*2)
  1*?       sum++;
```

Example

For this code fragment, give a big-O analysis of the running time.

```
         1  int sum = 0;
1+2*log(n)  for (int i = 1; i < n; i = i*2)
  1*log(n)    sum++;
```

$3*\log(n)+2$ instructions

$O(\log(n))$

Example

- Given a set of p points, find the pair closest to each other.
- Algorithm #1: Calculate the distance between each pair; return the minimum.
- There are p (p - 1) / 2 pairs, and each pair takes constant time to examine.

```
/* Visit a pair (i, j) of points. */
  for (int i = 0; i < numPoints; i++) {
    /* We require that j > i so that each pair is visited only once. */
    for (int j = i + 1; j < numPoints; j++) {
      double thisDistance = point[i].distance(point[j]);
      if (thisDistance < minDistance) {
        minDistance = thisDistance;
      }
    }
  }
```

Example

- Given a set of p points, find the pair closest to each other.
- Algorithm #1: Calculate the distance between each pair; return the minimum.
- There are p (p - 1) / 2 pairs, and each pair takes constant time to examine.

```
/* Visit a pair (i, j) of points. */
  for (int i = 0; i < numPoints; i++) {
    /* We require that j > i so that each pair is visited only once. */
    for (int j = i + 1; j < numPoints; j++) {
      double thisDistance = point[i].distance(point[j]);
      if (thisDistance < minDistance) {
        minDistance = thisDistance;
      }
    }
  }
```

$\Theta(n^2)$

Example

- Given an array of integers, write an algorithm to remove consecutive duplicates.
  - for example [1,2,2,3,3,3,4,5,5,5,5] should be converted to [1,2,3,4,5,-1,-1,-1,-1,-1,-1]

```
int i = 0, j = 0;

  while (i < ints.length) {
    ints[j] = ints[i];
    do {
      i++;
    } while ((i < ints.length) && (ints[i] == ints[j]));
    j++;
  }
  // Code to fill in -1's at end of array omitted.
```

Example

- Given an array of integers, write an algorithm to remove consecutive duplicates.
  - for example [1,2,2,3,3,3,4,5,5,5,5] should be converted to [1,2,3,4,5,-1,-1,-1,-1,-1,-1]

```
int i = 0, j = 0;

  while (i < ints.length) {
    ints[j] = ints[i];
    do {
      i++;
    } while ((i < ints.length) && (ints[i] == ints[j]));
    j++;
  }
  // Code to fill in -1's at end of array omitted.
```

$\Theta(n)$!!

But doubly-nested loops don't

always mean quadratic running time!

Example

- The outer loop can iterate up to ints.length times, and so can the inner loop.
- But the index "i" advances on **every** iteration of the inner loop. It can’t advance more than ints.length times before both loops end.
- So the running time of this algorithm is in Θ(ints.length).

```
int i = 0, j = 0;

  while (i < ints.length) {
    ints[j] = ints[i];
    do {
      i++;
    } while ((i < ints.length) && (ints[i] == ints[j]));
    j++;
  }
  // Code to fill in -1's at end of array omitted.
```

Algorithms Analysis

We will use Θ,O symbols to describe the complexity of algorithms

– E.g., adding a list of n numbers will be said to be a $\Theta(n)$ algorithm

An algorithm is said to have ***polynomial time complexity*** if its run-time may be described by $\mathrm{O}(n^d)$ for some <u>fixed</u> $d \geq 0$

– We will consider such algorithms to be ***efficient***

Problems that have no known polynomial-time algorithms are said to be ***intractable***

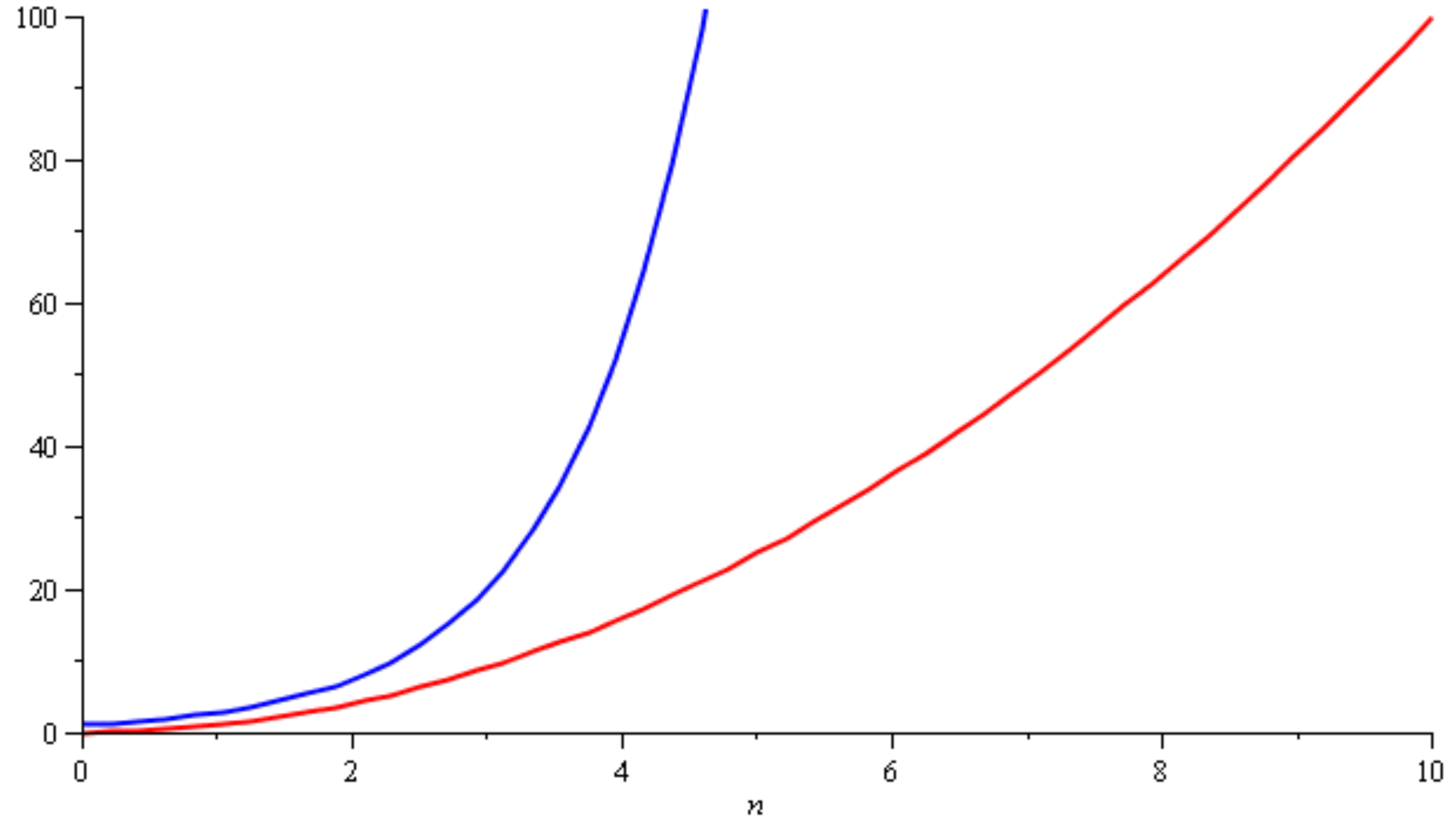
– Traveling salesman problem: find the shortest path that visits n cities

– Best run time: $\Theta(n^2\,2^n)$

Algorithm Analysis

In general, you don't want to implement exponential-time or exponential-memory algorithms

- Warning: don't call a quadratic curve "exponential", either...please



Reading

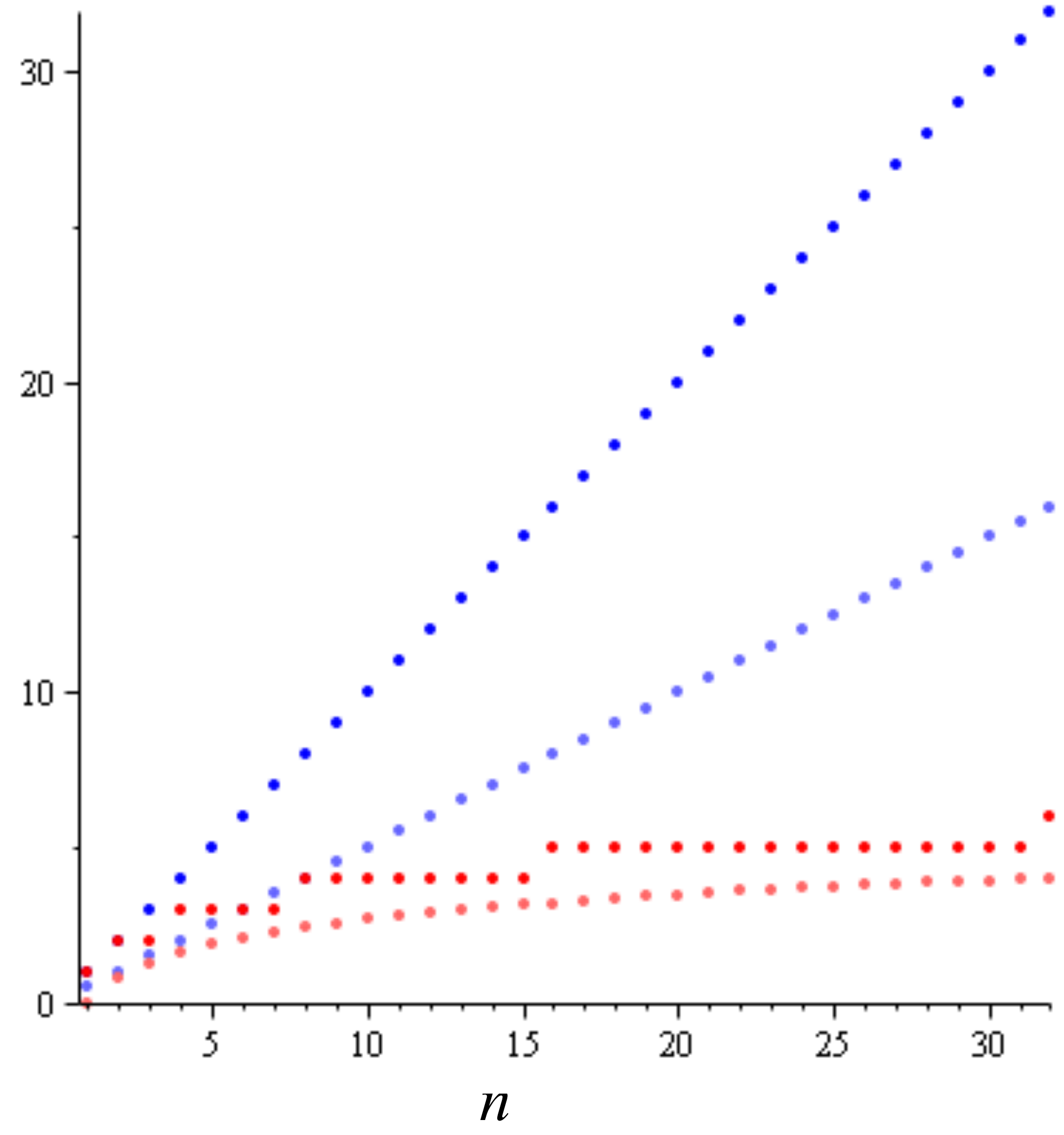
- Chapter 5 of the textbook
- Goodrich & Tamassia, Chapter 4 (especially 4.2 and 4.3)

Extra Slides

Linear and binary search

- There are other algorithms which are significantly faster as the problem size increases

- This plot shows maximum and average number of comparisons to find an entry in a sorted array of size n
  - Linear search
  - Binary search



Examples

We will now look at two examples:

- A comparison of selection sort and bubble sort
- A comparison of insertion sort and quicksort

Counting Instructions

Suppose we had two algorithms which sorted a list of size n and the run time (in μs) is given by

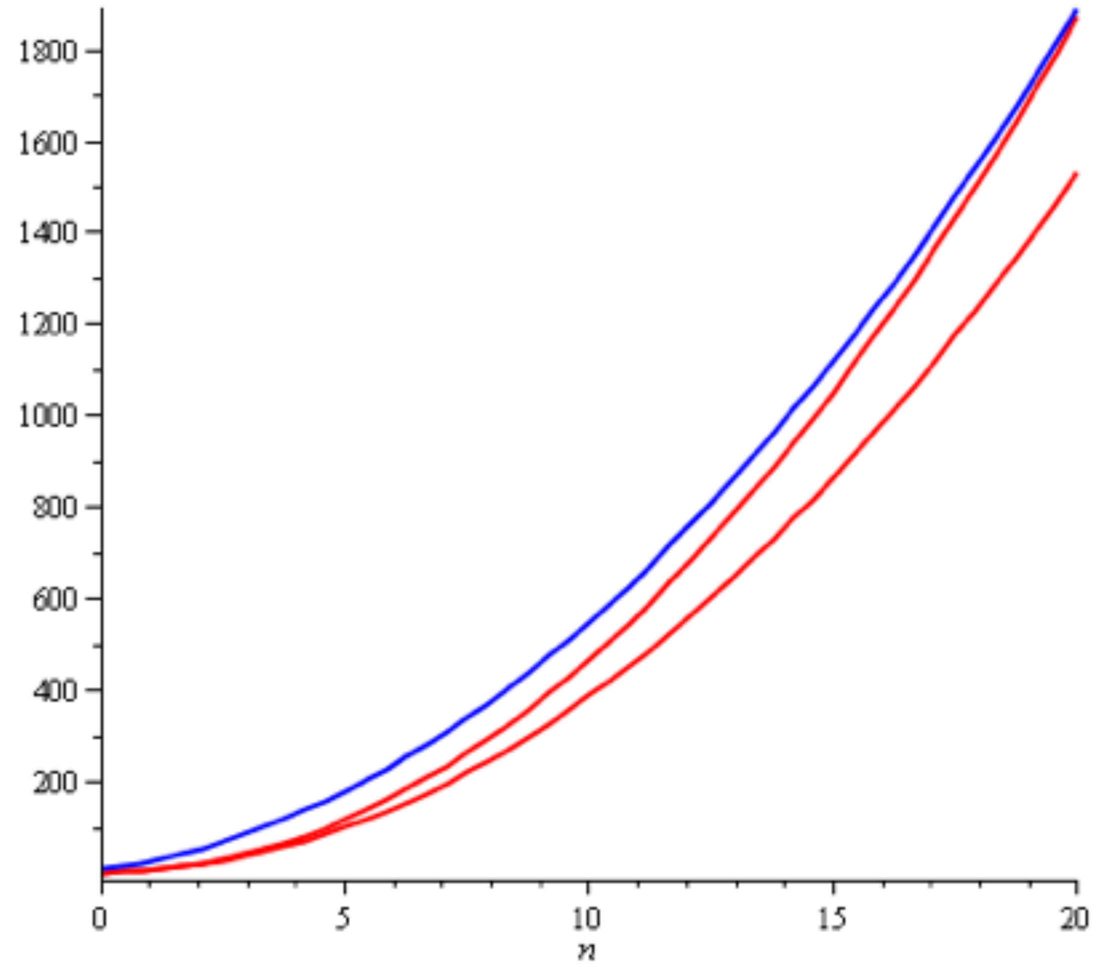
$b_{\text{worst}}(n) = 4.7n^2 - 0.5n + 5$	Bubble sort
$b_{\text{best}}(n) = 3.8n^2 + 0.5n + 5$	
$s(n) = 4n^2 + 14n + 12$	Selection sort

The smaller the value, the fewer instructions are run

- For $n \le 21$, $b_{\text{worst}}(n) < s(n)$
- For $n \ge 22$, $b_{\text{worst}}(n) > s(n)$

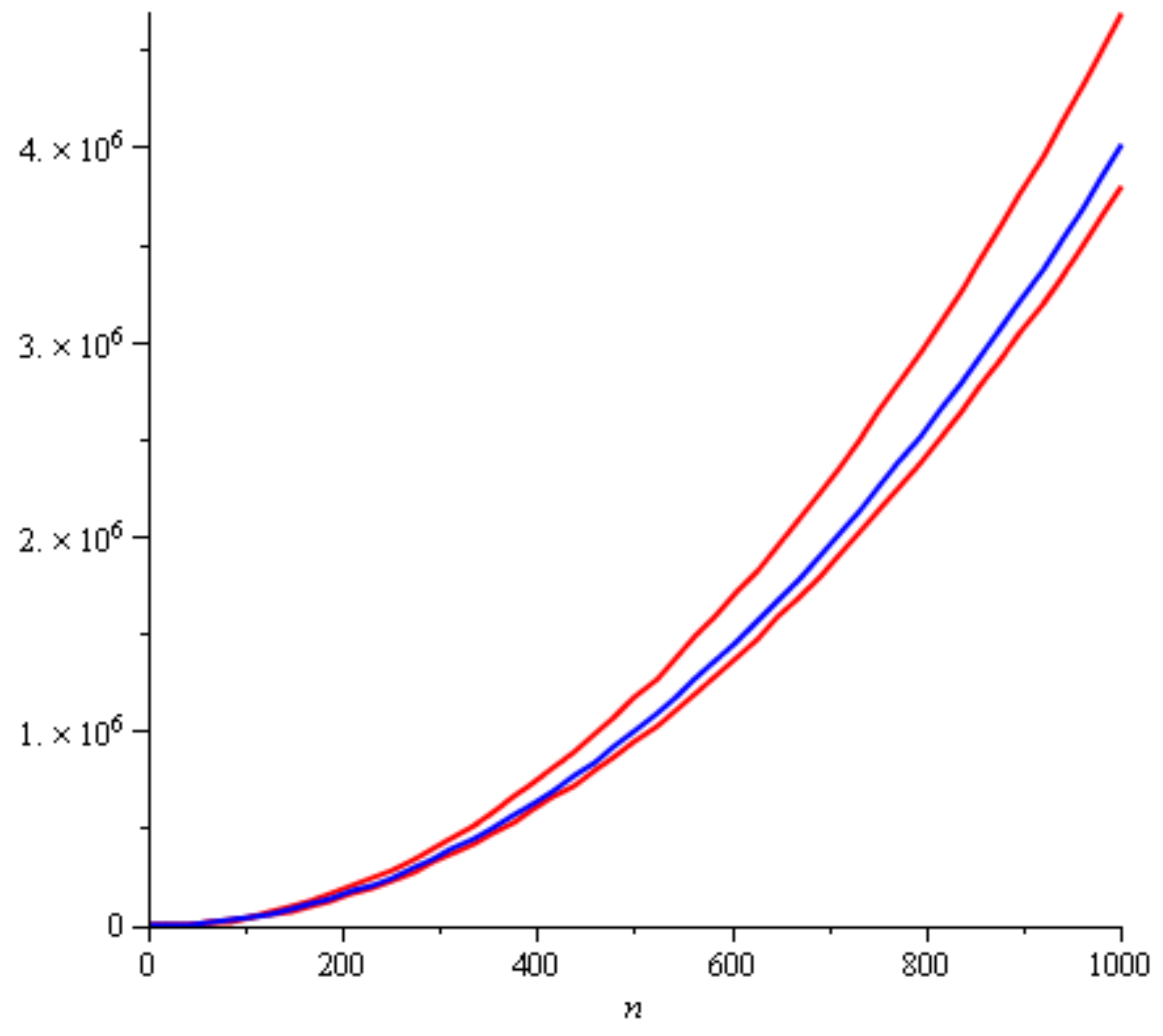
Counting Instructions

With small values of n, the algorithm described by $s(n)$ requires more instructions than even the worst-case for bubble sort



Counting Instructions

Near $n = 1000$, $b_{\text{worst}}(n) \approx 1.175\, s(n)$ and $b_{\text{best}}(n) \approx 0.95\, s(n)$



Counting Instructions

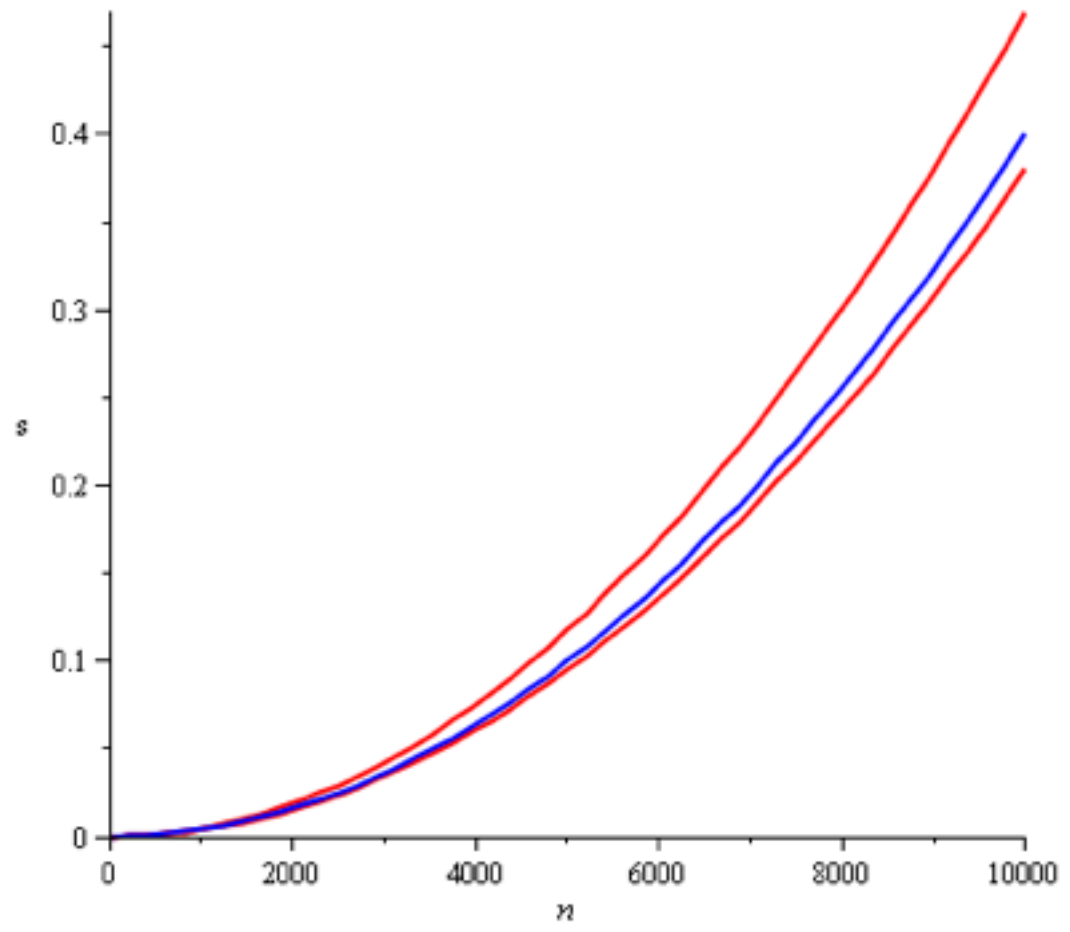
Is this a serious difference between these two algorithms?

Because we can count the number instructions, we can also estimate how much time is required to run one of these algorithms on a computer

Counting Instructions

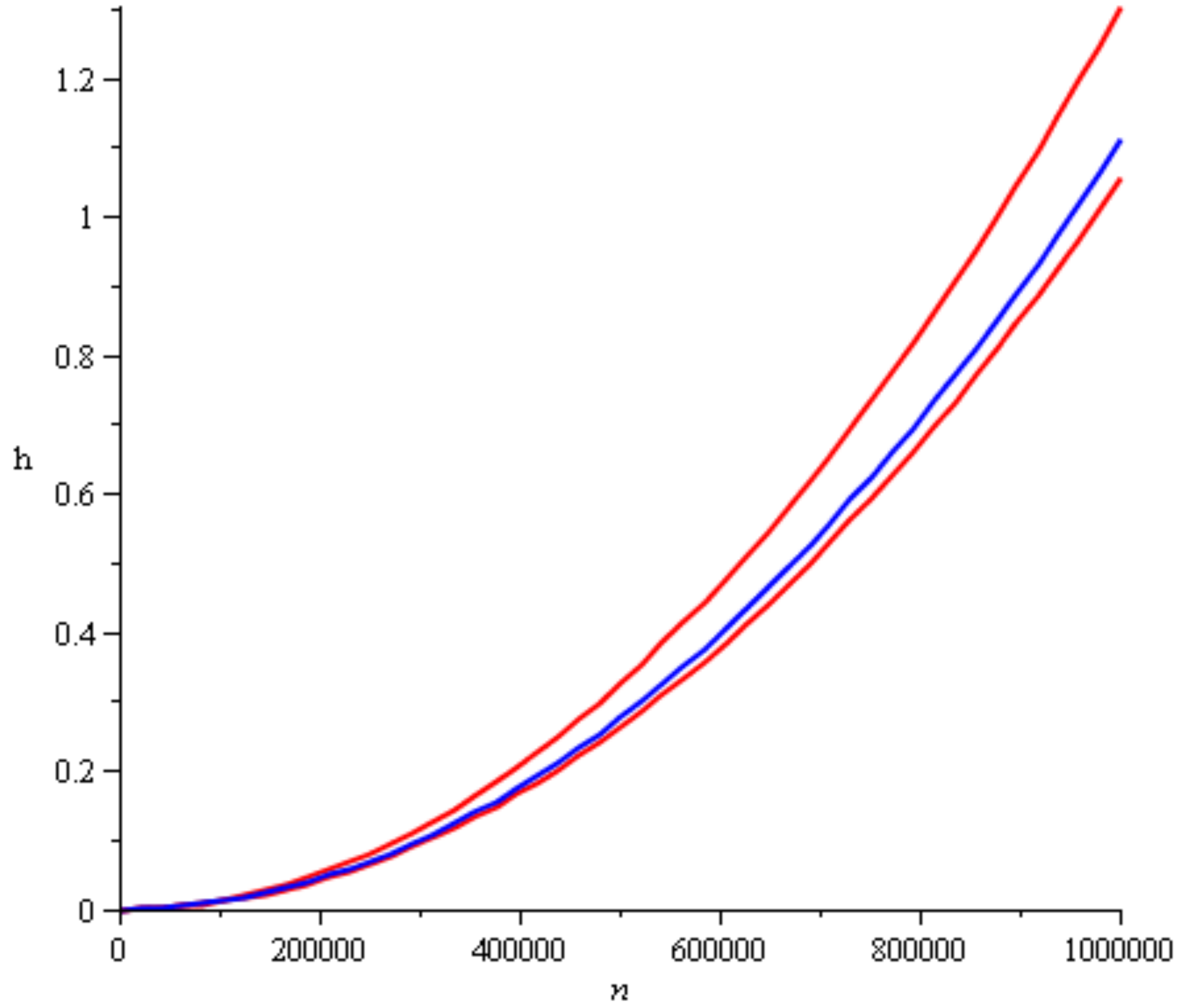
Suppose we have a $1\ \mathrm{GHz}$ computer

- The time (s) required to sort a list of up to $n = 10\ 000$ objects is under half a second



Counting Instructions

To sort a list with one million elements, it will take about 1 h

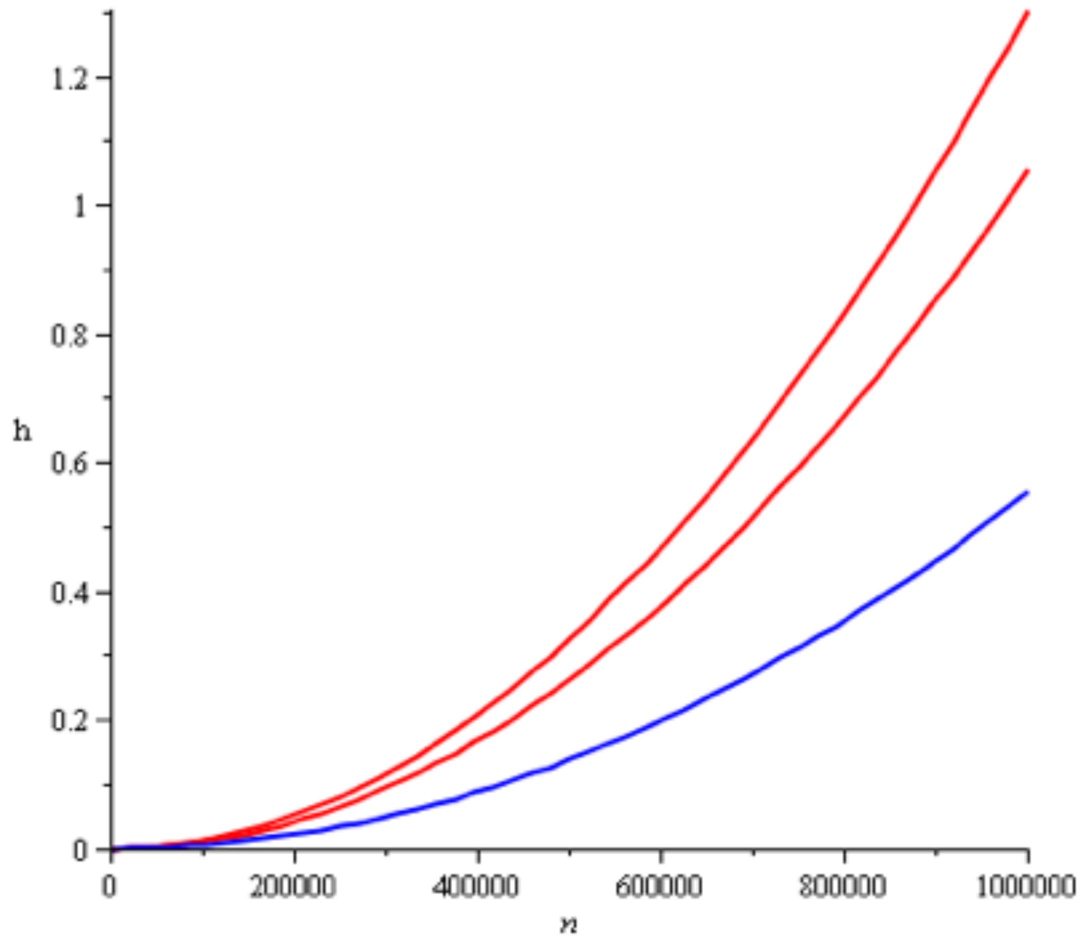


Bubble sort could, under some conditions, be 200 s faster

Counting Instructions

How about running selection sort on a faster computer?

- For large values of n, selection sort on a faster computer will always be faster than bubble sort



Counting Instructions

- Justification?
  - If $f(n) = a_k n^k + \cdots$ and $g(n) = b_k n^k + \cdots$, for large enough n, it will always be true that

    $$f(n) < M g(n)$$

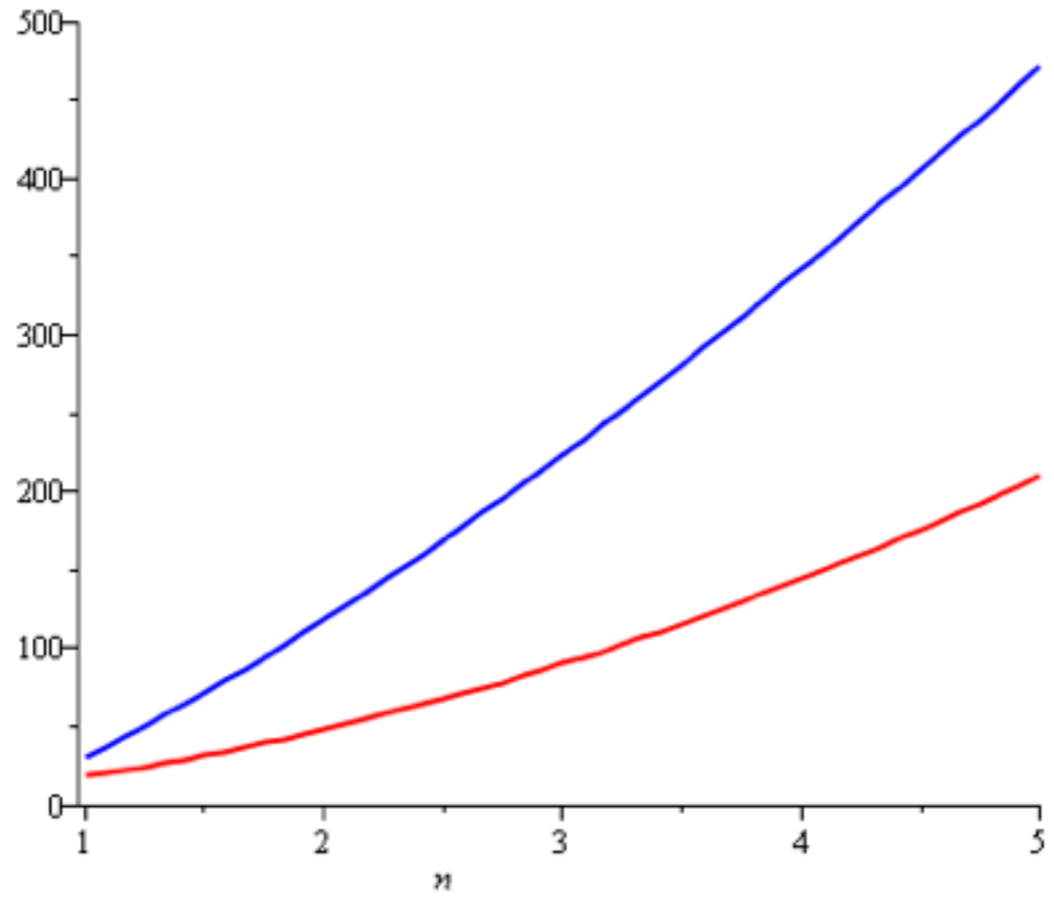
    where we choose

    $M = a_k / b_k + 1$
- In this case, we only need a computer which is M times faster (or slower)

Counting Instructions

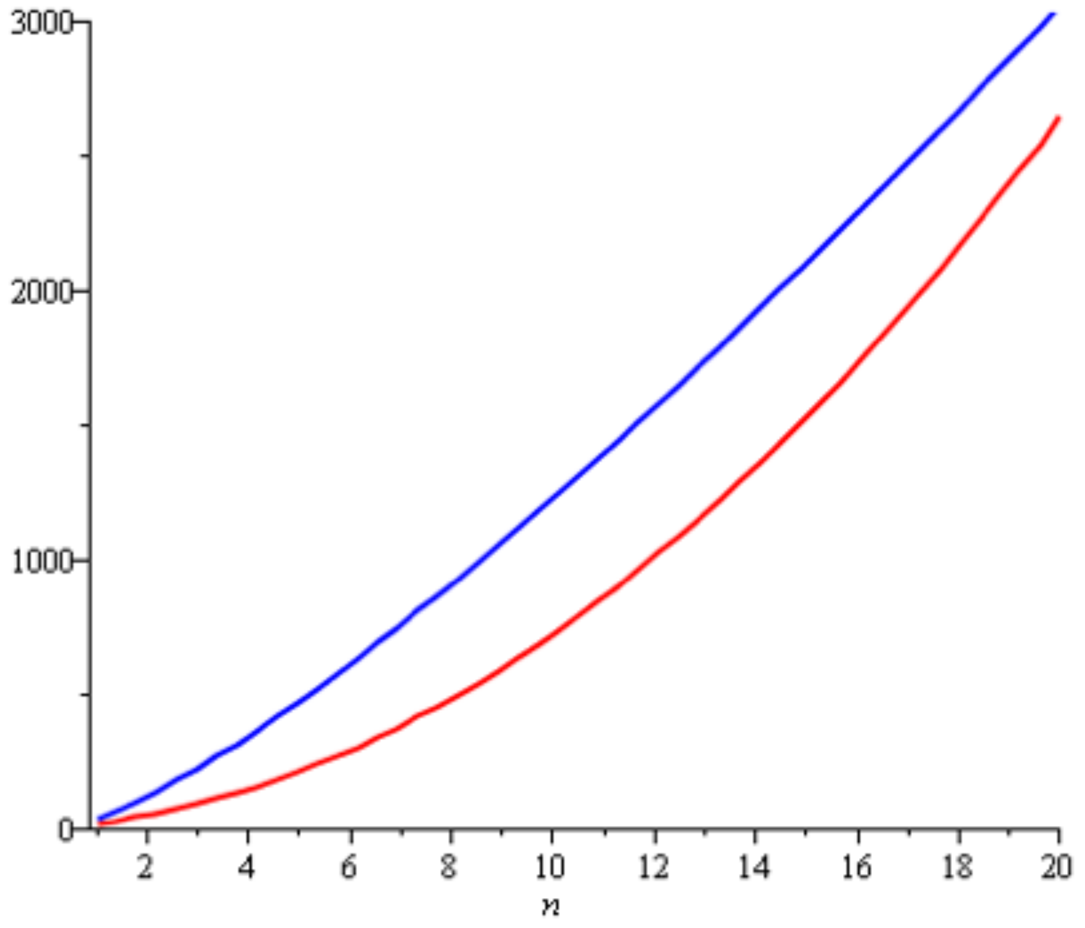
As another example:

- Compare the number of instructions required for insertion sort and for quicksort
- Both functions are concave up, although one more than the other



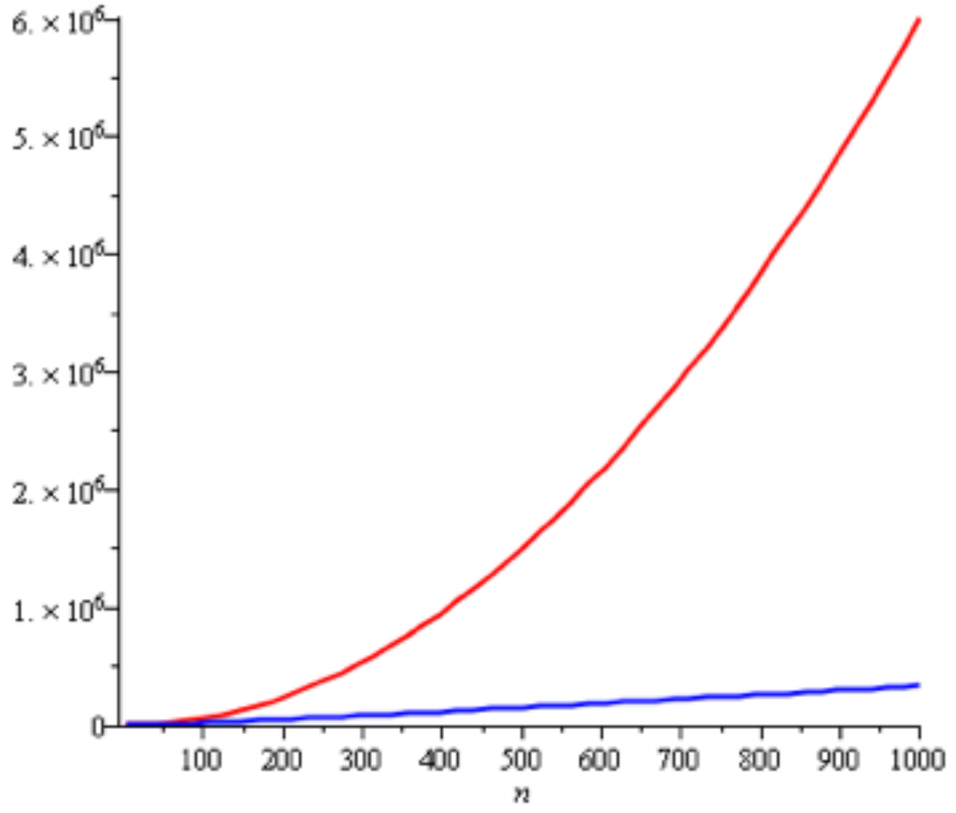
Counting Instructions

- Insertion sort, however, is growing at a rate of n^2 while quicksort grows at a rate of $n \lg(n)$
  - Never-the-less, the graphic suggests it is more useful to use insertion sort when sorting small lists—quicksort has a large overhead



Counting Instructions

- If the size of the list is too large (greater than 20), the additional overhead of quicksort quickly becomes insignificant
  - The quicksort algorithm becomes significantly more efficient
  - Question: can we just buy a faster computer?



Simplification

These definitions are often unnecessarily tedious

Note, however, that if $f(n)$ and $g(n)$ are polynomials of the same degree with positive leading coefficients:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \quad \text{where} \quad 0 < c < \infty$$

Big-Θ as an Equivalence Relation

If we look at the first relationship, we notice that $f(n) = \Theta(g(n))$ seems to describe an equivalence relation:

1. $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
2. $f(n) = \Theta(f(n))$
3. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, it follows that $f(n) = \Theta(h(n))$

Consequently, we can group all functions into equivalence classes, where all functions within one class are big-theta Θ of each other

Big-Θ as an Equivalence Relation

For example, all of

n^2 $\qquad$ $100000n^2 - 4\,n + 19$ $\qquad$ $n^2 + 1000000$

$323n^2 - 4\,n\ln(n) + 43\,n + 10$ $\qquad$ $42n^2 + 32$

$$n^2 + 61\,n\ln^2(n) + 7n + 14\ln^3(n) + \ln(n)$$

are big-Θ of each other

E.g., $42n^2 + 32 = \Theta(\,323\,n^2 - 4\,n\ln(n) + 43\,n + 10\,)$

Big-Θ as an Equivalence Relation

Recall that with the equivalence class of all 19-year olds, we only had to pick one such student?

Similarly, we will select just one element to represent the entire class of these functions: n^2

– We could chose any function, but this is the simplest

Weak ordering

Consider the following definitions:

- We will consider two functions to be equivalent, $f \sim g$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ where } \quad 0 < c < \infty$$

- We will state that $f < g$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

For functions we are interested in, these define a weak ordering

Weak ordering

- In general, there are functions such that
  - If $f(n) \sim g(n)$, then it is always possible to improve the performance of one function over the other by purchasing a faster computer
  - If $f(n) < g(n)$, then you can never purchase a computer fast enough so that the second function always runs in less time than the first

- Note that for small values of n, it may be reasonable to use an algorithm that is asymptotically more expensive, but we will consider these on a one-by-one basis

Landau Symbols

We will at times use five possible descriptions

$$\mathrm{f}(n) = \mathbf{o}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = 0$$

$$\mathrm{f}(n) = \mathbf{O}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} < \infty$$

$$\mathrm{f}(n) = \mathbf{\Theta}(\mathrm{g}(n)) \qquad 0 < \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} < \infty$$

$$\mathrm{f}(n) = \mathbf{\Omega}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} > 0$$

$$\mathrm{f}(n) = \boldsymbol{\omega}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = \infty$$

Landau Symbols

For the functions we are interested in, it can be said that

$f(n) = \mathbf{O}(g(n))$ is equivalent to $f(n) = \Theta(g(n))$ or

$f(n) = \mathbf{o}(g(n))$

and

$f(n) = \Omega(g(n))$ is equivalent to $f(n) = \Theta(g(n))$ or

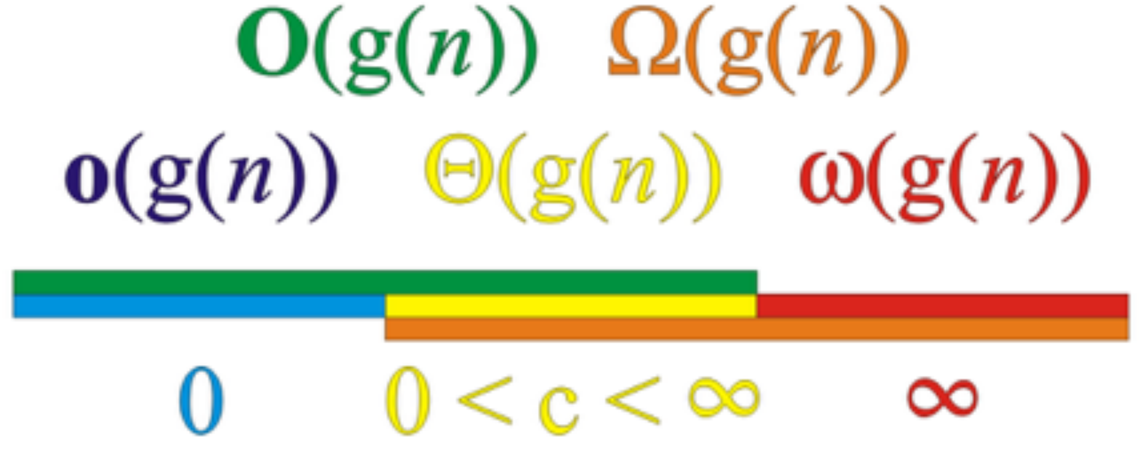
$f(n) = \omega(g(n))$

Landau Symbols

Graphically, we can summarize these as follows:

We say $f(n) =$

if $\lim_{n \to \infty} \frac{f(n)}{g(n)} =$



Landau Symbols

Some other observations we can make are:

$$\mathrm{f}(n) = \Theta(\mathrm{g}(n)) \Leftrightarrow \mathrm{g}(n) = \Theta(\mathrm{f}(n))$$

$$\mathrm{f}(n) = \mathbf{O}(\mathrm{g}(n)) \Leftrightarrow \mathrm{g}(n) = \Omega(\mathrm{f}(n))$$

$$\mathrm{f}(n) = \mathbf{o}(\mathrm{g}(n)) \Leftrightarrow \mathrm{g}(n) = \omega(\mathrm{f}(n))$$