Today

- Homework 1 due 11:59 pm next Wednesday, April 6
- Review math essential to algorithm analysis
  - Proof by induction (review example)
  - Exponents and logarithms
  - Floor and ceiling functions

- Begin algorithm analysis

CSE373: Data Structures and Algorithms

Lecture 3: Math Review; Algorithm Analysis

Linda Shapiro

Spring 2016

Mathematical induction

Suppose *P(n)* is some statement (mentioning integer *n*)

Example: $n \geq n/2 + 1$

We can use induction to prove P(n) for all integers $n \geq n_0$.

We need to

1. Prove the "base case" i.e. $P(n_0)$. For us n_0 is usually 1.
2. Assume the statement holds for P(k).
3. Prove the "inductive case" i.e. if P(k) is true, then P(k+1) is true.

Why we care:

To show an algorithm is correct or has a certain running time

no matter how big a data structure or input value is

(Our "*n*" will be the data structure or input size.)

Review Example

P(n) = "the sum of the first *n* powers of 2 (starting at 2^0) is 2^n-1"

$$2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1.$$

in other words: $1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1.$

Review Example

P(n) = "the sum of the first *n* powers of 2 (starting at 2^0) is 2^n-1"

We will show that *P(n)* holds for all $n \geq 1$

Proof: By induction on *n*

- Base case: *n*=1. Sum of first 1 power of 2 is 2^0 , which equals 1.
  And for *n*=1, 2^n-1 equals 1.

Review Example

P(n) = “the sum of the first *n* powers of 2 (starting at 2^0) is 2^n-1”

- Inductive case:
  - Assume P(k) is true i.e. the sum of the first *k* powers of 2 is 2^k-1
  - Show P(k+1) is true i.e. the sum of the first (*k*+1) powers of 2 is 2^{k+1}-1

Using our assumption, we know the first k powers of 2 is

$$2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = 2^k - 1$$

Add the next power of 2 to both sides…

$$2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} + 2^k = 2^k - 1 + 2^k$$

We have what we want on the left; massage the right a bit:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} + 2^k = 2(2^k) - 1$$
$$= 2^{k+1} - 1$$

Success!

Mathematical Preliminaries

- The following N slides contain basic mathematics needed for analyzing algorithms.
- You should actually know this stuff.
- Hang in there!

Logarithms and Exponents

- Definition: $x = 2^y$ **if** $\log_2 x = y$
  - $8 = 2^3$, so $\log_2 8 = 3$
  - $65536 = 2^{16}$, so $\log_2 65536 = 16$

- The exponent of a number says how many times to use the number in a multiplication. e.g. $2^3 = 2 \times 2 \times 2 = 8$

  (2 is used 3 times in a multiplication to get 8)

- A logarithm says how many of one number to multiply to get another number. It asks "what exponent produced this?"

  e.g. $\log_2 8 = 3$ *(2 makes 8 when used 3 times in a multiplication)*

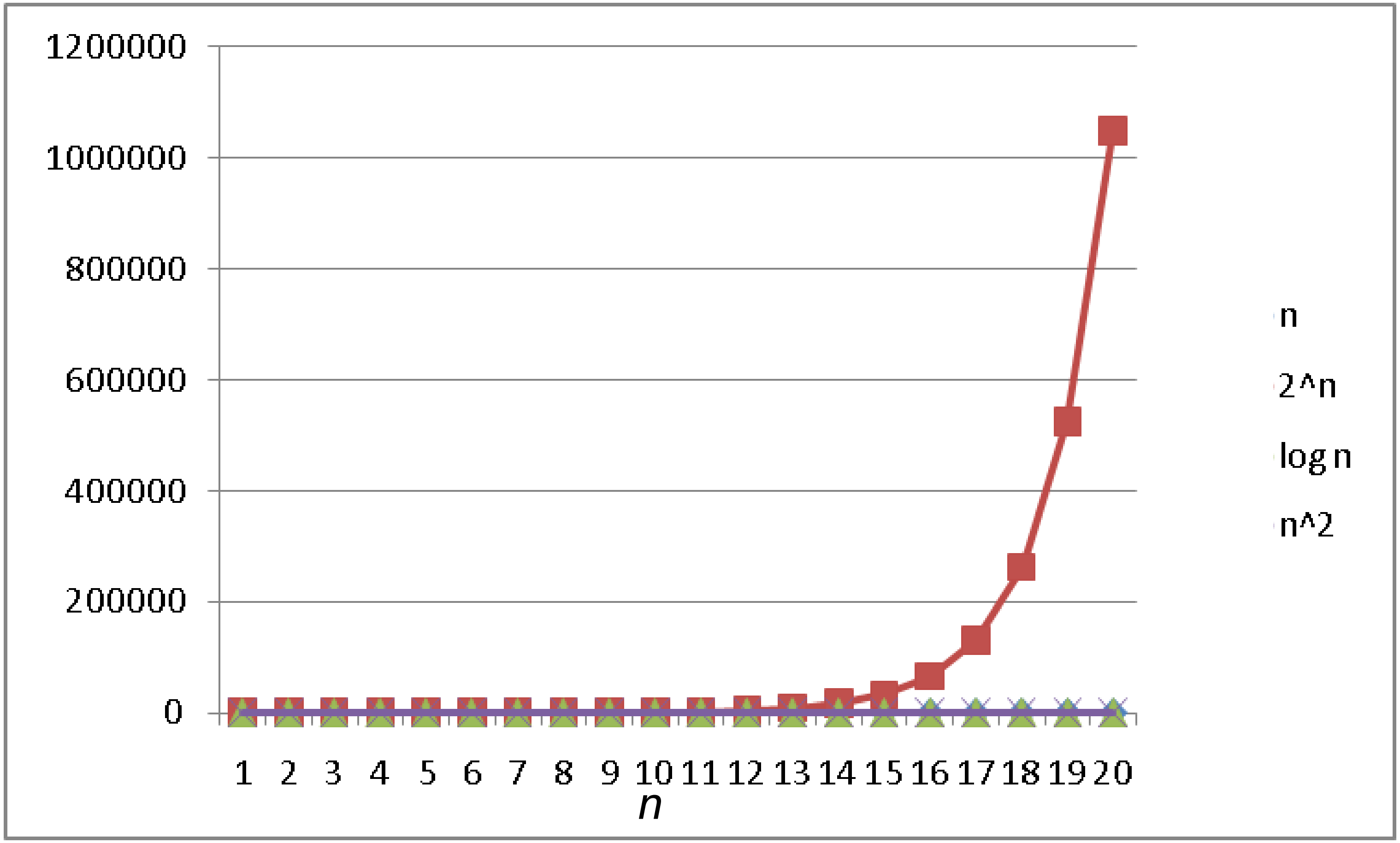
Logarithms and Exponents

- Definition: $\mathbf{x = 2^y \text{ if } \log_2 x = y}$
  - $8 = 2^3$, so $\log_2 8 = 3$
  - $65536 = 2^{16}$, so $\log_2 65536 = 16$

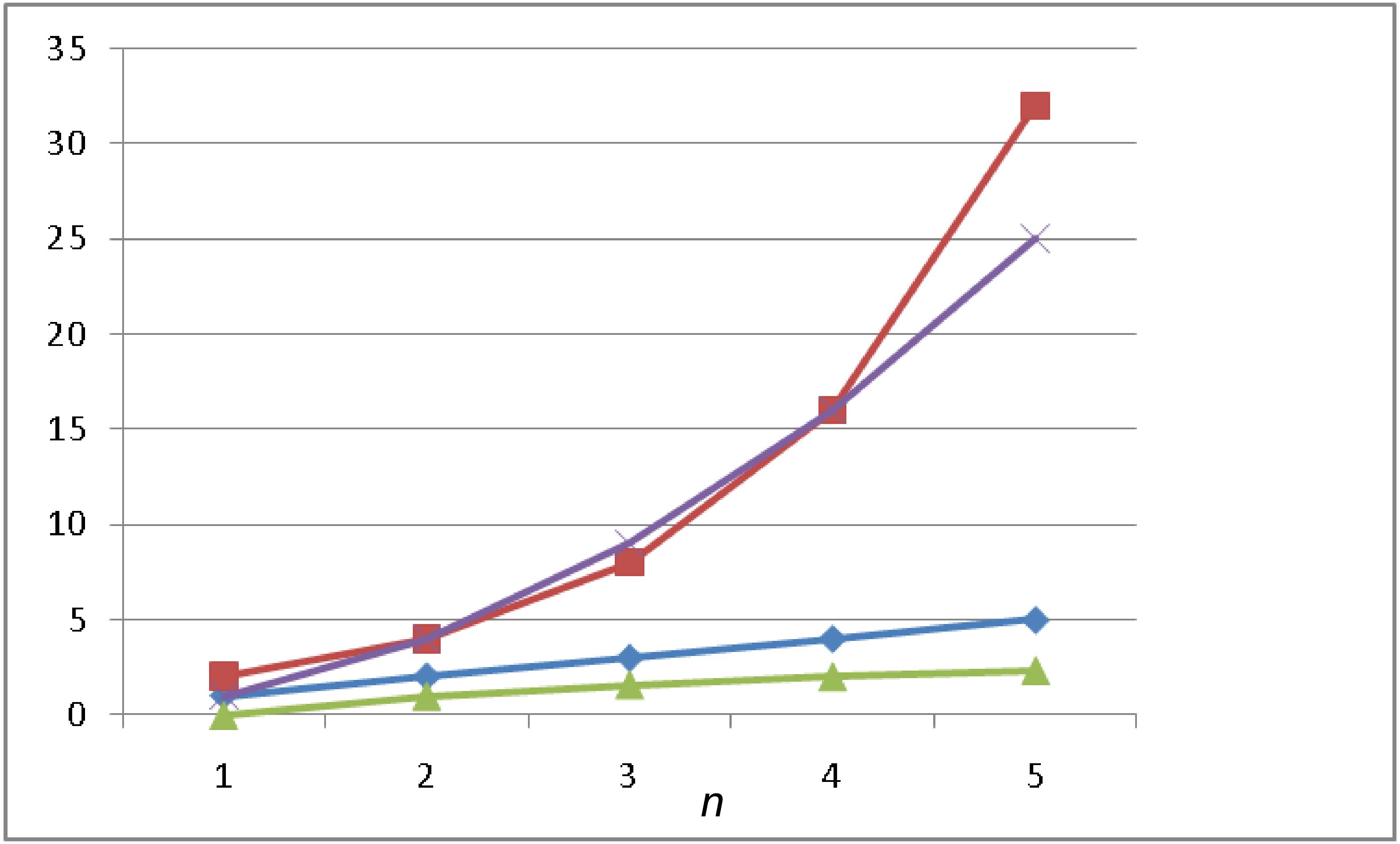
- Since so much is binary in CS, **log** almost always means $\log_2$
- $\log_2$ n tells you how many bits needed to represent n combinations.
- So, $\log_2$ 1,000,000 = "a little under 20" (19.9336)

- Logarithms and exponents are inverse functions. Just as exponents grow *very* quickly, logarithms grow *very* slowly.

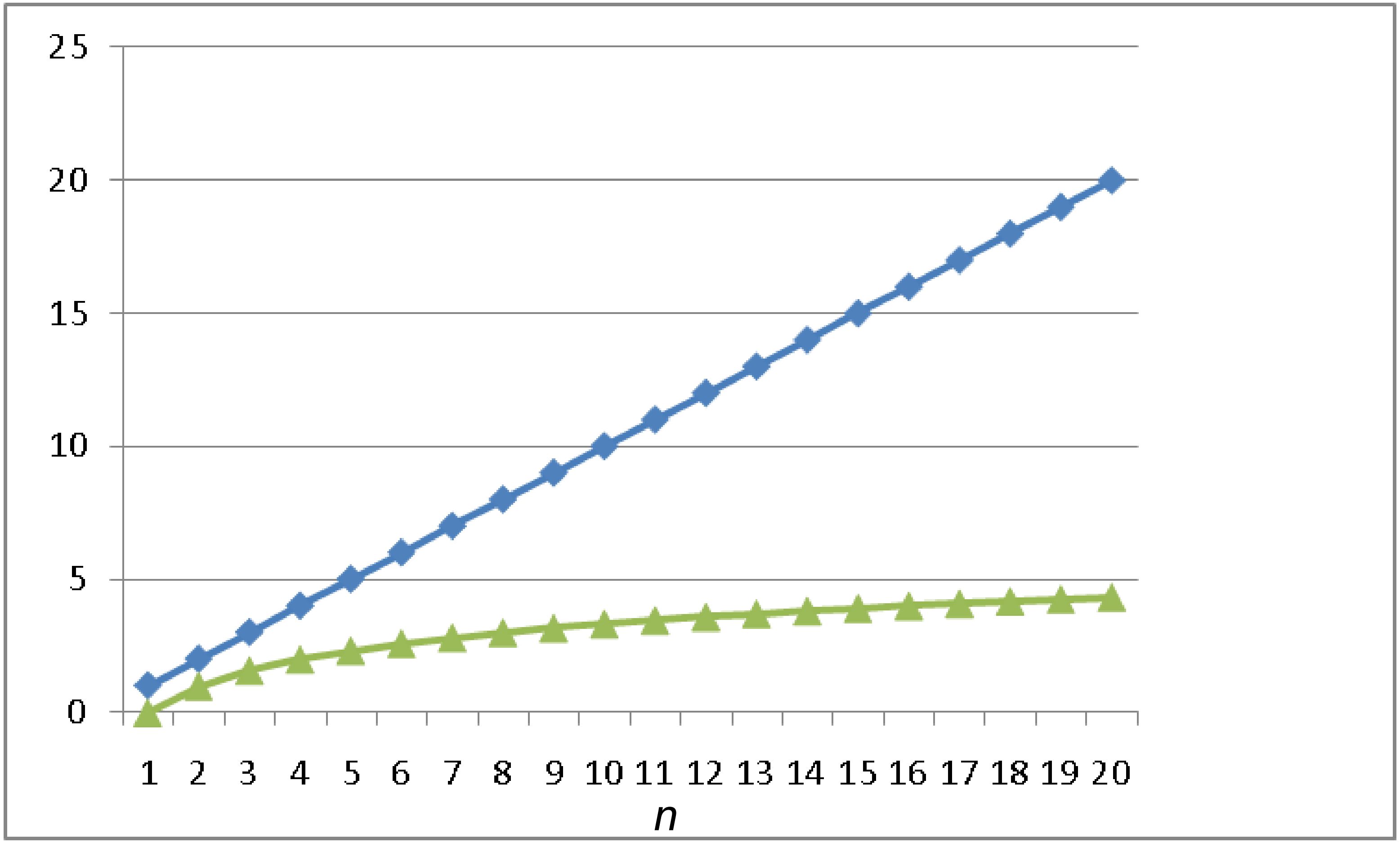
Logarithms and Exponents: Big View



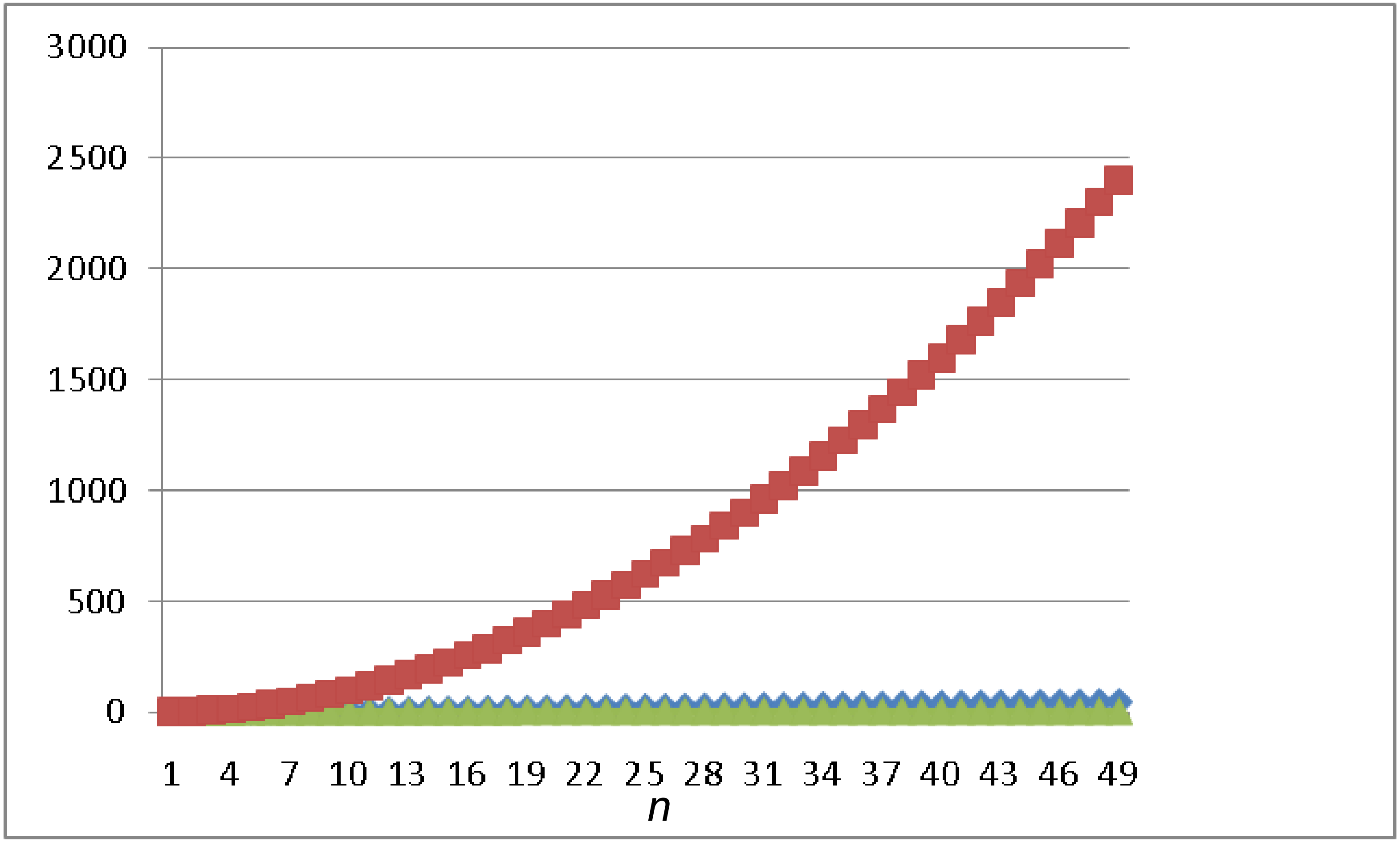
Logarithms and Exponents: Zoom in



Logarithms and Exponents: just n, log n



Logarithms and Exponents: n, log n, n^2



Properties of logarithms

- **`log(A*B) = log A + log B`**
- **`log(`N^k`)= k log N`**
- **`log(A/B) = log A – log B`**
- **`log(log x)`** is written **`log log x`**
  - Grows as slowly as 2^{2^y} grows quickly
- **`(log x)(log x)`** is written **`log`^{2}`x`**
  - It is greater than **`log x`** for all **`x > 2`**
  - It is not the same as **`log log x`**

Log base doesn't matter much!

"Any base *B* log is equivalent to base 2 log within a constant factor"

- And we are about to stop worrying about constant factors!
- In particular, $\log_2 x = 3.22 \log_{10} x$
- In general we can convert log bases via a constant multiplier
- To convert from base B to base A:

$$\log_B x = (\log_A x) / (\log_A B)$$
$$\log_2 1{,}000{,}000 = (\log_{10} 1{,}000{,}000) / (\log_{10} 2)$$
$$\log_2 1{,}000{,}000 = 6 \;/\; .3010 = 19.9336$$

- **I use this because my calculator doesn't have $\log_2$.**

Floor and ceiling

$\lfloor X \rfloor$ Floor function: the largest integer $\leq$ X

$$\lfloor 2.7 \rfloor = 2 \qquad \lfloor -2.7 \rfloor = -3 \qquad \lfloor 2 \rfloor = 2$$

$\lceil X \rceil$ Ceiling function: the smallest integer $\geq$ X

$$\lceil 2.3 \rceil = 3 \qquad \lceil -2.3 \rceil = -2 \qquad \lceil 2 \rceil = 2$$

Facts about floor and ceiling

1. $X - 1 < \lfloor X \rfloor \le X$
2. $X \le \lceil X \rceil < X + 1$
3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n \quad \text{if } n \text{ is an integer}$

$\lfloor 5/2 \rfloor \quad + \quad \lceil 5/2 \rceil$

2 3

5

Algorithm Analysis

As the "size" of an algorithm's input grows (integer, length of array, size of queue, etc.), we want to know

- How much longer does the algorithm take to run? (time)
- How much more memory does the algorithm need? (space)

Because the curves we saw are so different, often care about only "which curve we are like"

Separate issue: Algorithm *correctness* – does it produce the right answer for all inputs

- Usually more important, naturally

Algorithm Analysis: A first example

- Consider the following program segment:

```
x:= 0;
for i = 1 to n do
   for j = 1 to i do
        x := x + 1;
```

- What is the value of x at the end?

i	**j**	**x**
1	1 to 1	1
2	1 to 2	3
3	1 to 3	6
4	1 to 4	10
…		
n	1 to n	?

Number of times x gets incremented is

$= 1 + 2 + 3 + \ldots + (n-1) + n$

$= n*(n+1)/2$

Analyzing the loop

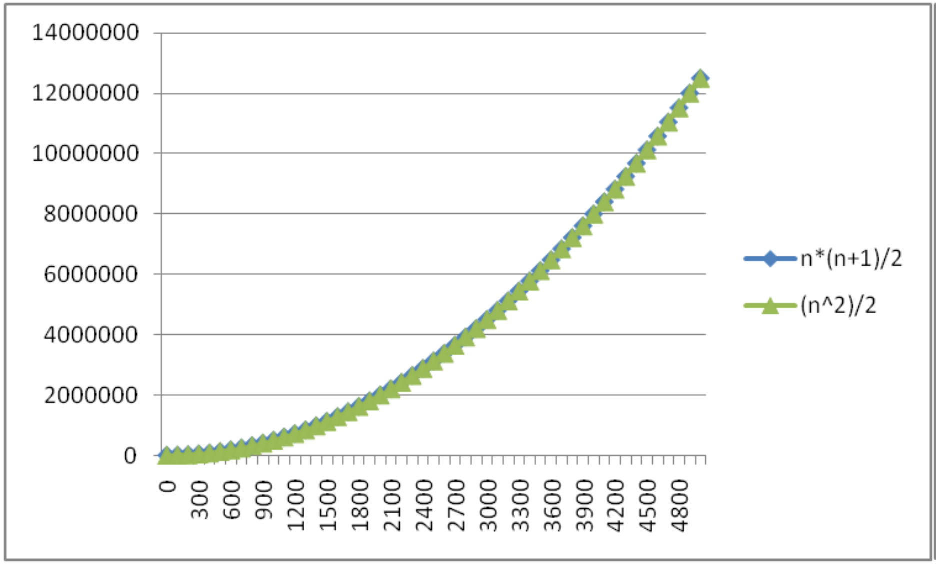
- Consider the following program segment:

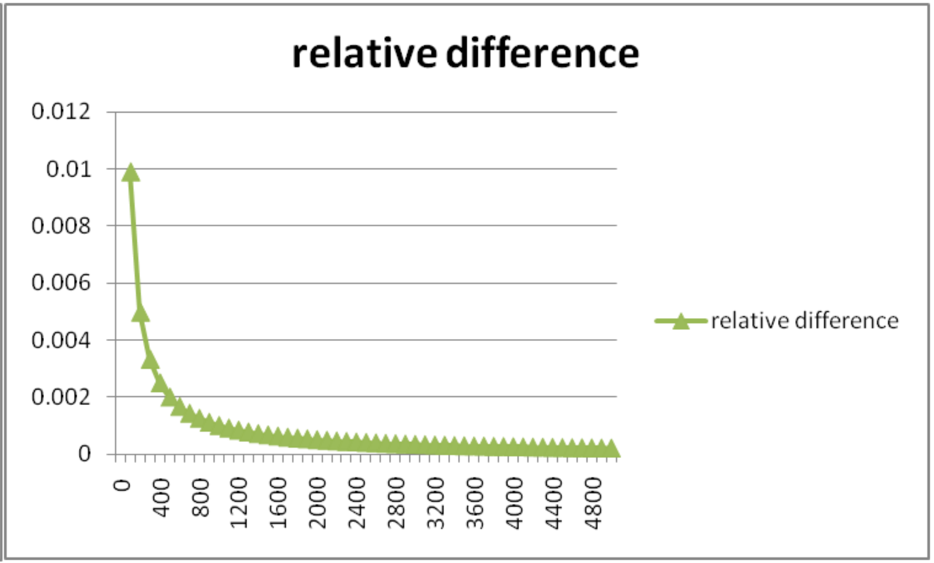
```
x:= 0;
for i = 1 to n do
   for j = 1 to i do
      x := x + 1;
```

- The total number of loop iterations is $n*(n+1)/2$
  - This is a very common loop structure, worth memorizing
  - This is *proportional to* n^2 , and we say $O(n^2)$, “big-Oh of n^2 ”
    - $n*(n+1)/2 = (n^2+ n)/2 = 1/2\mathbf{n^2} + 1/2n$
    - The n^2 term dominates the n term.
    - For large enough n, the lower order and constant terms are irrelevant, as are the assignment statements
    - See plot… $(n^2+ n)/2$ vs. just $n^2/2$

Lower-order terms don't matter

n*(n+ 1)/2   vs. just $n^2/2$





We just say $O(n^2)$

Big-O: Common Names

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\mathbf{log}\ n)$	logarithmic
$O(n)$	linear
$O(n\ \mathbf{log}\ n)$	"n **log** *n*"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant)
$O(k^n)$	exponential (where k is any constant > 1)
$O(n!)$	factorial

Note: "exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1"

Big-O running times

- For a processor capable of one million instructions per second

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

For a modern processor, how many instructions per second?
Something like 2 billion.

Analyzing code

Basic operations take "some amount of" constant time

- Arithmetic (fixed-width)
- Assignment
- Access one Java field **or array index**
- Etc.

(This is an *approximation of reality*: a very useful "lie".)

Consecutive statements	Sum of times
Conditionals	Time of test plus slower branch
Loops	Sum of iterations
Calls	Time of call's body
Recursion	Solve *recurrence equation (next lecture)*

Analyzing code

1. Add up time for all parts of the algorithm

   e.g. number of iterations = $(n^2 + n)/2$
2. Eliminate low-order terms i.e. eliminate n: $(n^2)/2$
3. Eliminate coefficients i.e. eliminate 1/2: (n^2) : Result is $O(n^2)$

Examples:

- $4n + 5$ $= O(n)$
- $0.5n \,\mathbf{log}\, n + 2n + 7$ $= O(n \log n)$
- $n^3 + 2^n + 3n$ $= O(2^n)$
- 365 $= O(1)$

Try a Java sorting program

```
private static void bubbleSort(int[] intArray) {
        int n = intArray.length;
        int temp = 0;

        for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){

                if(intArray[j-1] > intArray[j]){
                    //swap the elements!
                    temp = intArray[j-1];
                    intArray[j-1] = intArray[j];
                    intArray[j] = temp;
                }}}}
```

i j

| 5 | 3 | 10 | 4 |

i j

| 3 | 5 | 10 | 4 |

i j

| 3 | 5 | 10 | 4 |

i,j

| 3 | 5 | 4 | 10 |

i j

| 3 | 5 | 4 | 10 |

j i

| 3 | 4 | 5 | 10 |

and so on

Let's analyze it by counting

```
private static void bubbleSort(int[] intArray) {
        int n = intArray.length;
        int temp = 0;

        for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){

                if(intArray[j-1] > intArray[j]){
                    //swap the elements!
                    temp = intArray[j-1];
                    intArray[j-1] = intArray[j];
                    intArray[j] = temp;
                }}}}
```

1

1

outer loop	inner count
i=0:	n-1
i=1:	n-2
.	.
i=n-3	2
i=n-2	1
i=n-1	0 X

$1 + 2 + \ldots + (n-2) + (n-1)$

$= ((n-1)*n)/2$

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

$= \mathbf{O(n^2)}$

Another Exercise

```
class CoinFlip {
  static boolean heads()
   { return Math.random() < 0.5; }
  public static void main(String[] args)
   { int i, j, cnt;
     int  N = Integer.parseInt(args[0]);
     int  M = Integer.parseInt(args[1]);
     int [ ] hist = new int[N+1];
     for (j = 0; j <= N; j++) hist[j] = 0;
     for (i = 0; i < M; i++, hist[cnt]++)
        for (cnt = 0, j = 0; j <= N; j++)
           if (heads()) cnt++;
     for (j = 0; j <=N; j++) {
        if (hist[j] = = 0) system.out.print(".");
        for (i = 0; i < hist[j]; i+=1)
           system.out.print("*");
        system.out.println(); }}}
```

hist

0	1	2	3	4	5	6

hist[j] is the count of how many of the M trials got j heads

What is the max number of heads you can get in M trials?

```
0.
1*
2***
3**********
4****************
5**********************
6*******************
7******************
8*********
9**
10.
```