CSE373: Data Structures and Algorithms

Lecture 3: Math Review; Algorithm Analysis

Nicki Dell

Spring 2014

Today

- Registration should be done.
- Homework 1 due 11pm next Wednesday, April 9th.

- Review math essential to algorithm analysis
  - Proof by induction (another example)
  - Exponents and logarithms
  - Floor and ceiling functions

- Begin algorithm analysis

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.)

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

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.

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!

Logarithms and Exponents

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

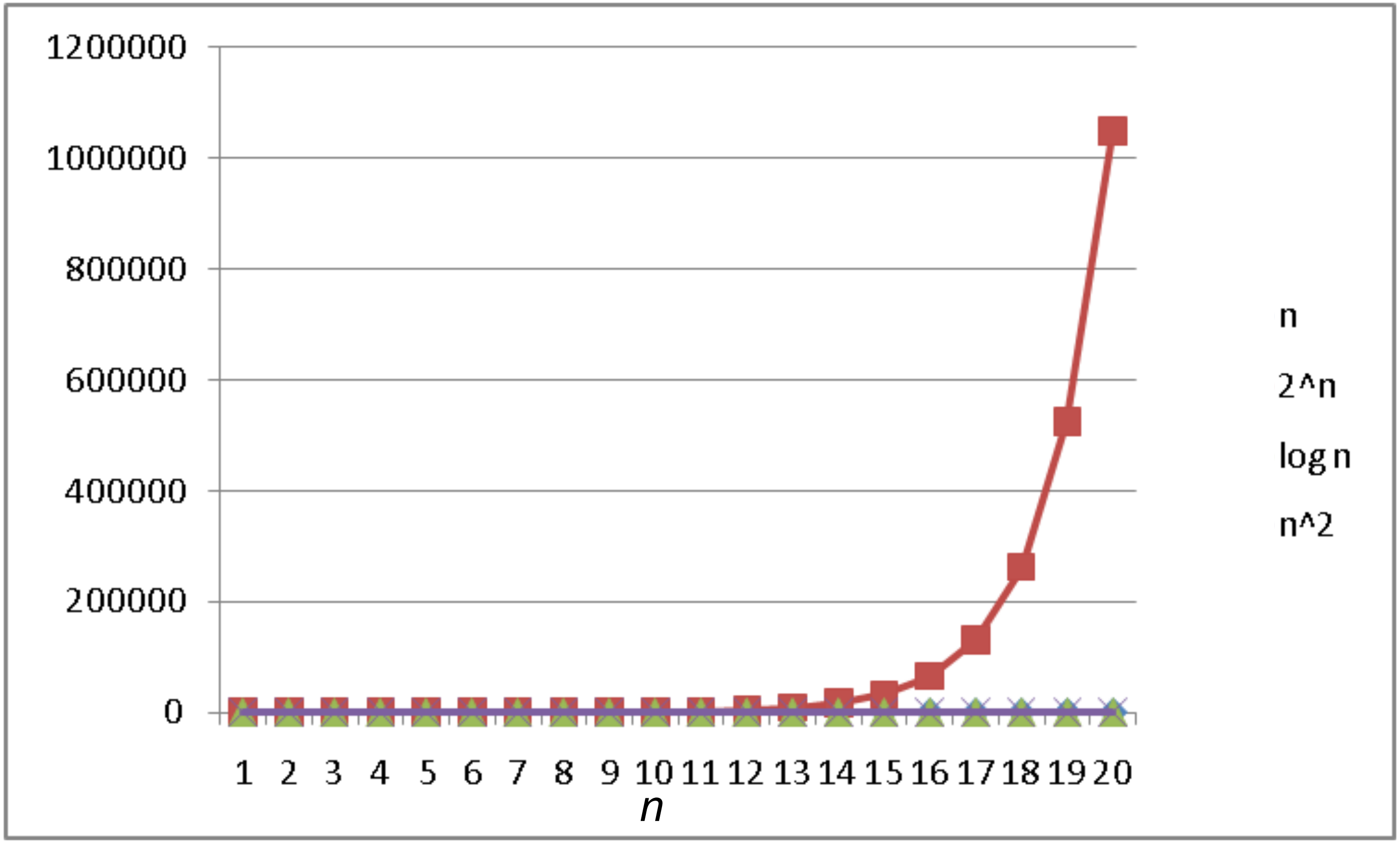
- Definition: $x = 2^y$ if $\log_2 x = y$
  - $8 = 2^3$, so $\log_2 8 = 3$
  - $65536 = 2^{16}$, so $\log_2 65536 = 16$

- 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"

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

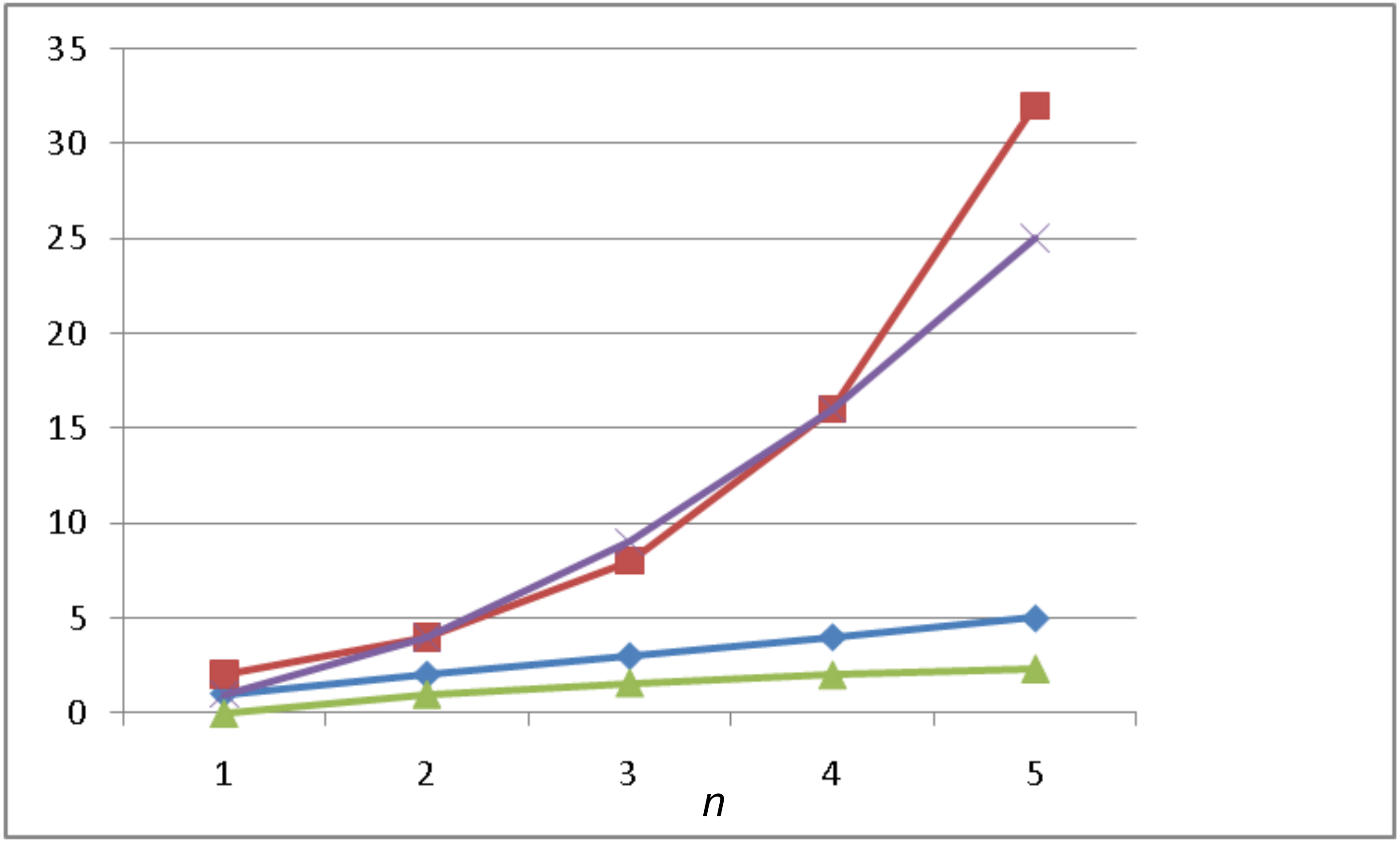
Logarithms and Exponents

See Excel file for plot data – play with it!



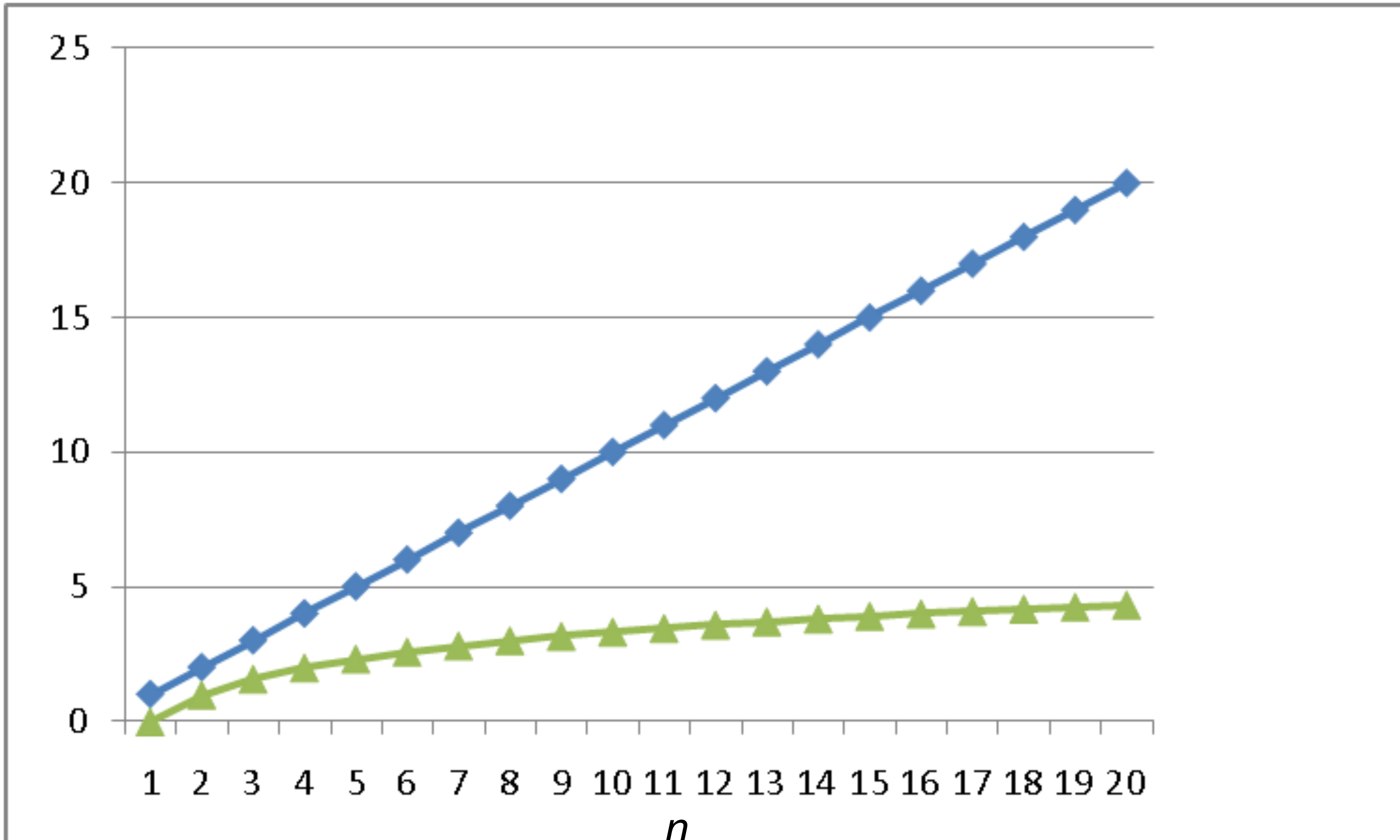
Logarithms and Exponents

See Excel file for plot data – play with it!



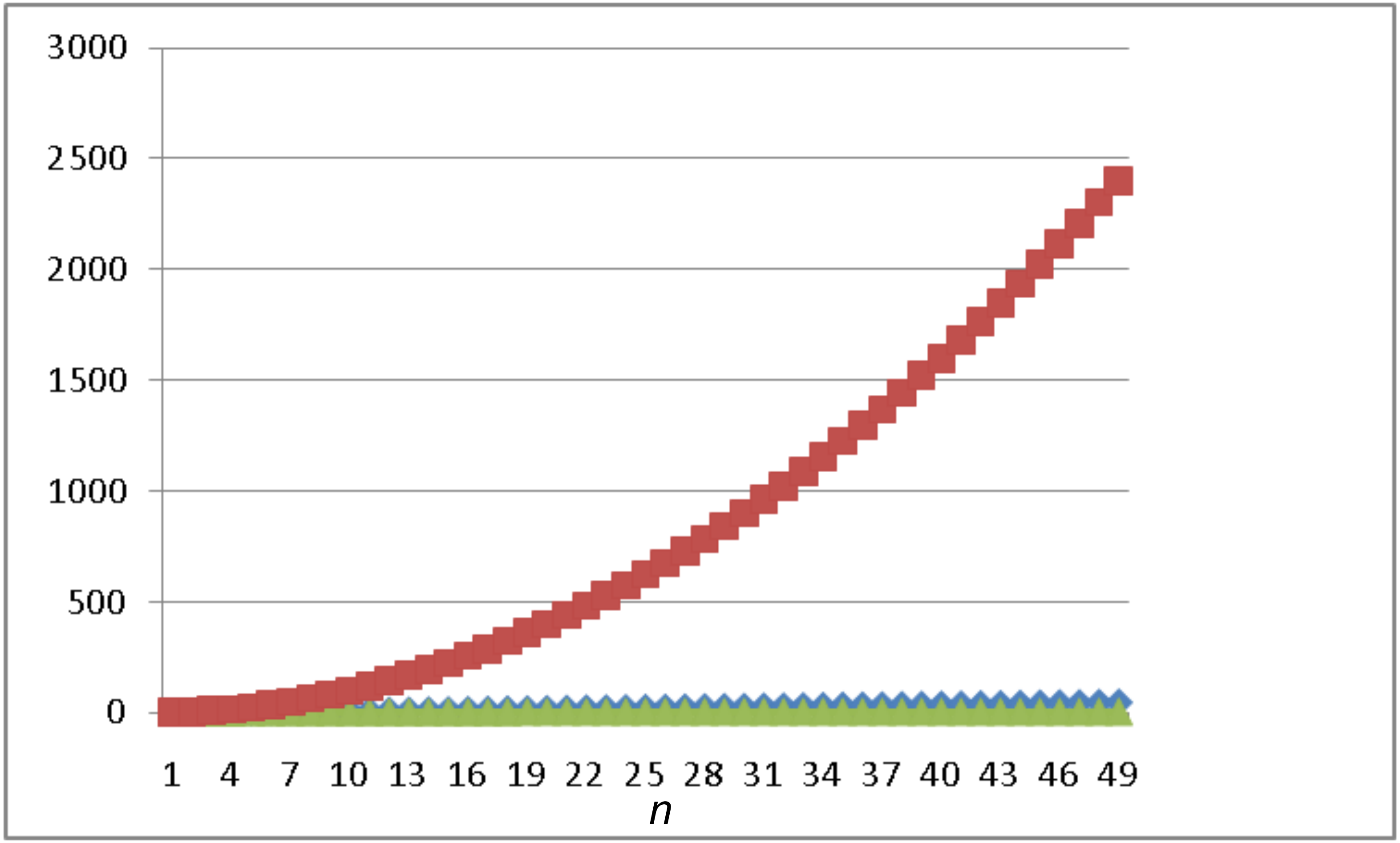
Logarithms and Exponents

See Excel file for plot data – play with it!



Logarithms and Exponents

See Excel file for plot data – play with it!



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:

$$\log_B x = (\log_A x) / (\log_A B)$$

Floor and ceiling

$\lfloor X \rfloor$ Floor function: the largest integer ≤ 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 ≥ X

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

Facts about floor and ceiling

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

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 + … + (n-1) + n

= n*(n+1)/2

Analyzing the loop

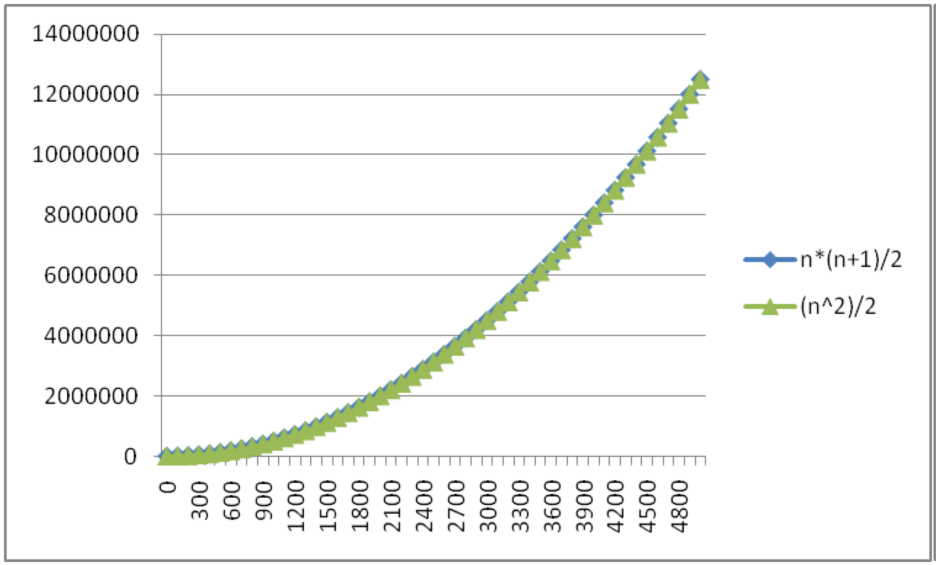
- Consider the following program segment:

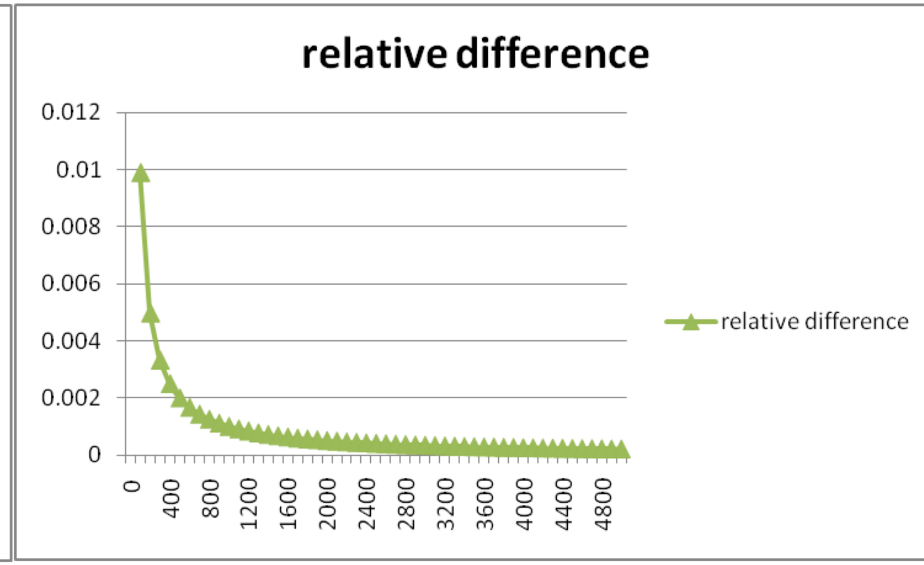
```
x:= 0;
for i = 1 to n do
   for j = 1 to i do
      x := x + 1;
```

- 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”
    - $n*(n+1)/2 = (n^2+ n)/2$
    - 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)
$O(\texttt{log}\ n)$	logarithmic
$O(n)$	linear
$O(n\ \texttt{log}\ n)$	"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

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)

Examples:

- $4n + 5$ = O(n)
- $0.5n \log n + 2n + 7$ = O(n log *n)*
- $n^3 + 2^n + 3n$ = $O(2^n)$
- $n \log (10n^2)$
  - $2n \log (10n)$ = O(n log *n)*