CSE373: Data Structures and Algorithms

Lecture 3: Math Review; Algorithm Analysis

Lauren Milne

Summer 2015

Today

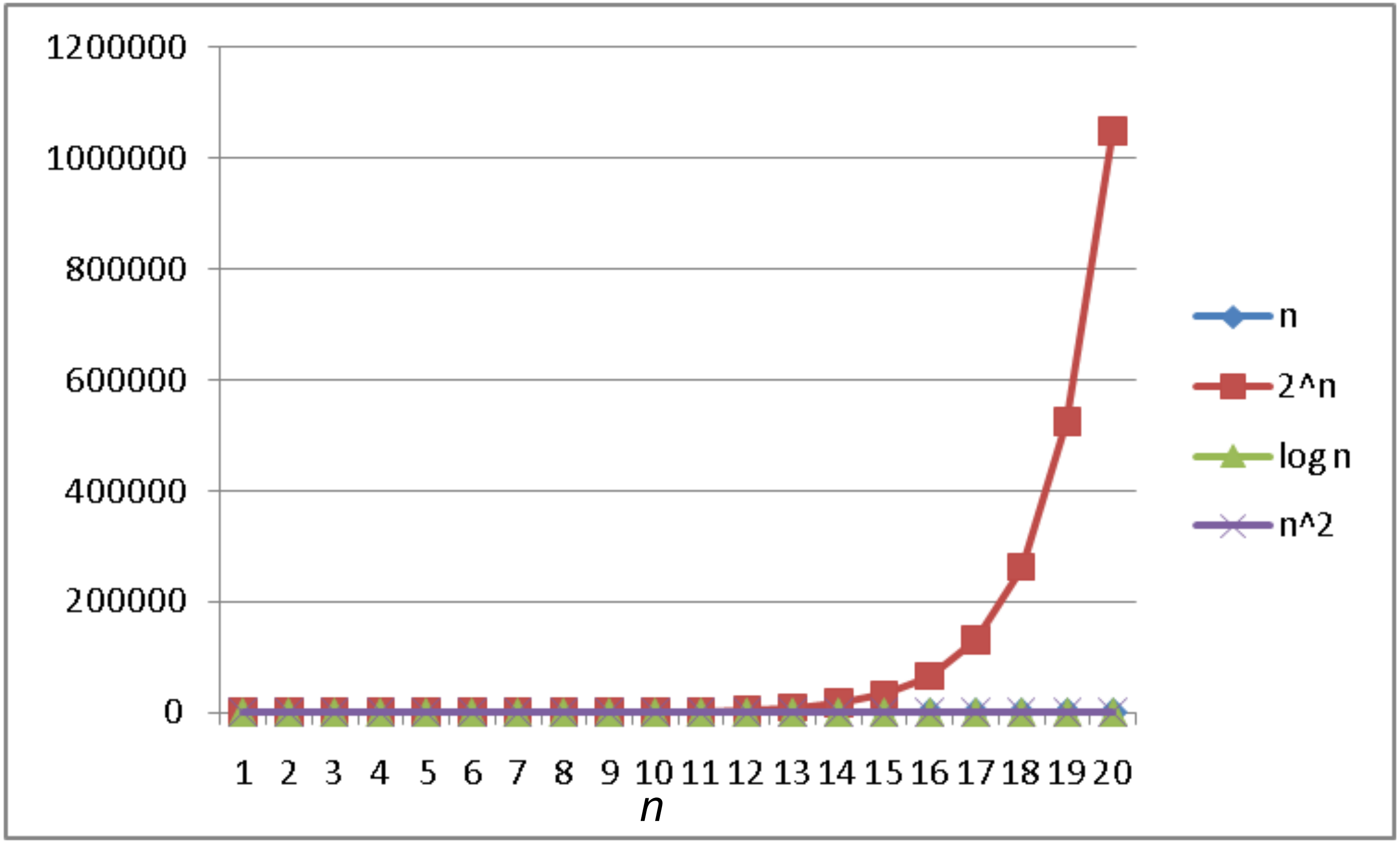
- Homework 1 due 10:59pm next Wednesday, July 1st.

- Review math essential to algorithm analysis
  - Exponents and logarithms
  - Floor and ceiling functions

- Algorithm analysis

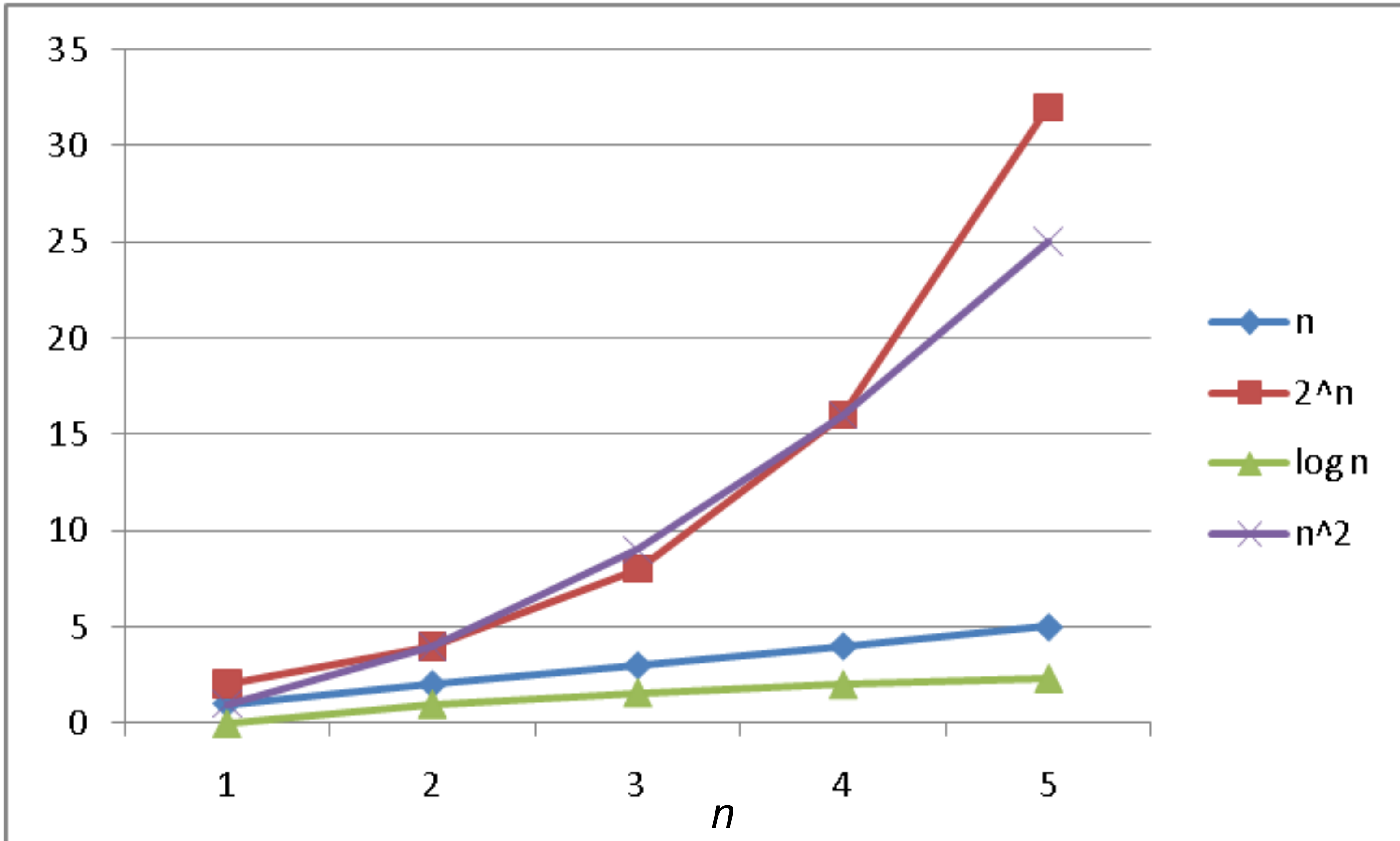
Logarithms and Exponents

See Excel file for plot data – play with it!



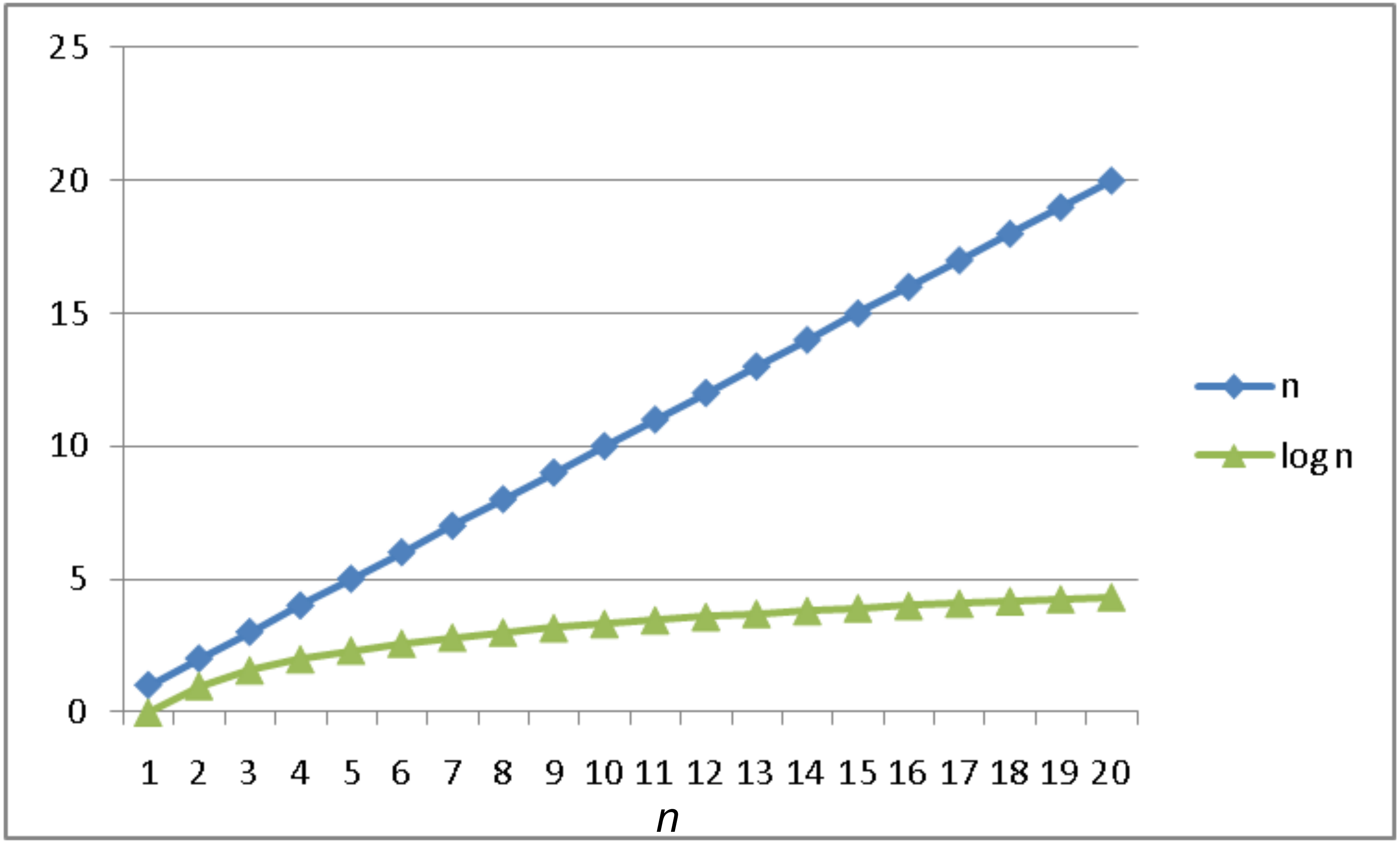
Logarithms and Exponents

See Excel file for plot data – play with it!



Logarithms and Exponents

See Excel file for plot data – play with it!



Properties of logarithms

- $\log(A*B) = \log A + \log B$
- $\log(A/B) = \log A - \log B$
- $\log(N^k) = k \log N$
- $\log(\log x)$ is written $\log \log x$
  - Grows as slowly as 2^{2^x} 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 $\log \log x$

Expand this:

$$\log(2a^2b/c)$$

$$= 1 + 2\log(a) + \log(b) - \log(c)$$

Log base doesn't matter much!

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

- Do we care about constant factors?
- $\mathbf{log_2\ x = 3.22\ log_{10}\ x}$
- 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 $\leq$ 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. $X - 1 < \lfloor X \rfloor \leq X$
2. $X \leq \lceil X \rceil < X + 1$
3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ if n is an integer

Algorithm Analysis

As the “size” of an algorithm’s input grows, we want to know

- Time it takes to run
- Space it takes to to run

Because the curves we saw are so different (2^n vs logn), often care about only which curve we are like

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

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

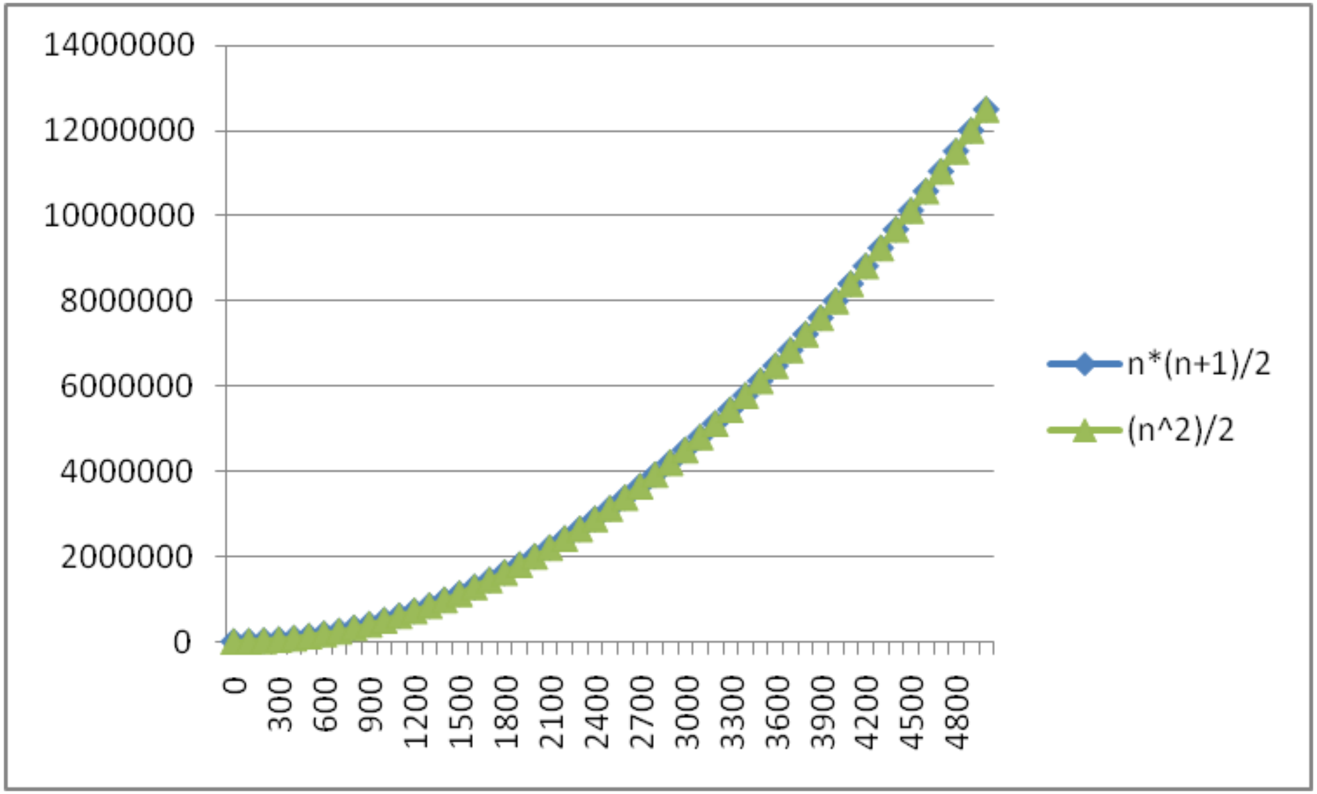
- Consider the following program segment:

```
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
  - $n*(n+1)/2 = (n^2+ n)/2$
  - For large enough n, the lower order and constant terms are irrelevant!
  - *So this is O*(n^2)

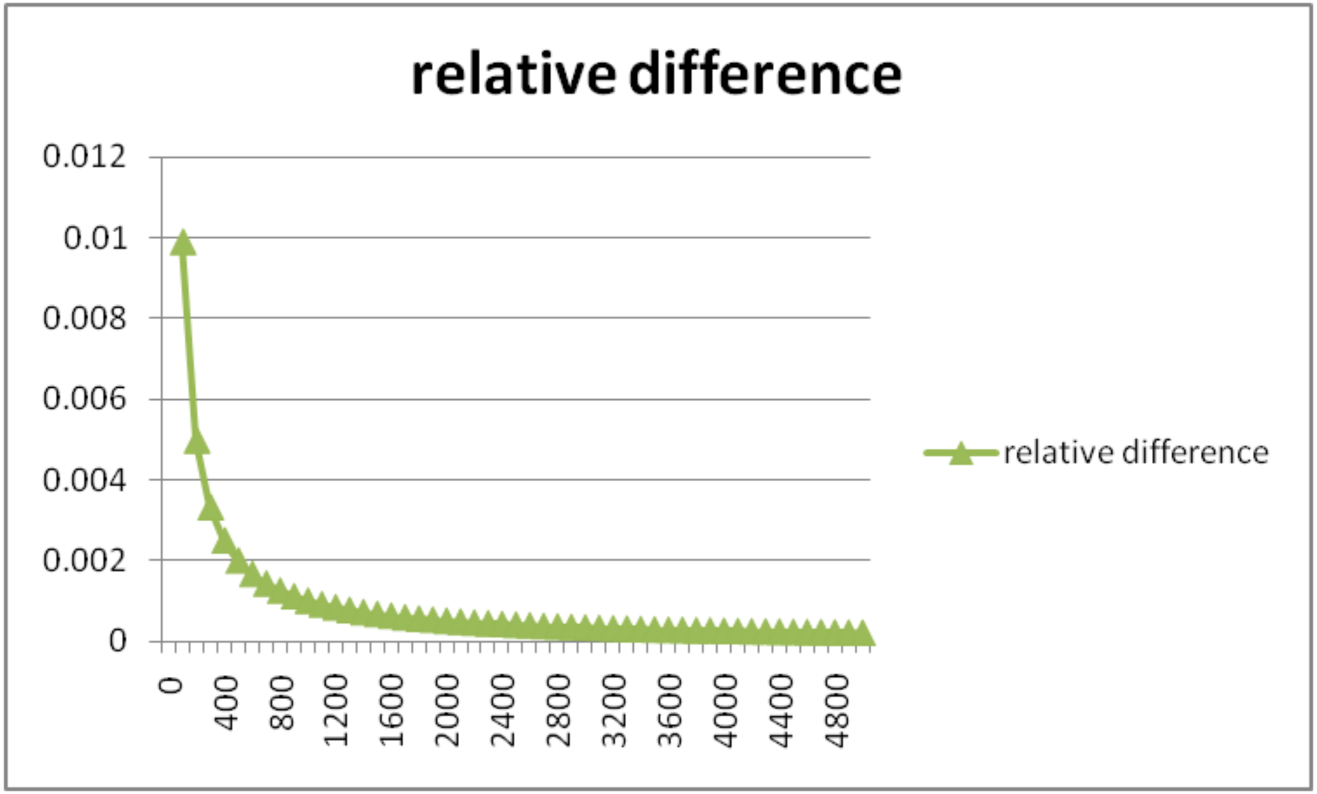
Lower-order terms don't matter

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



Lower-order terms don't matter

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



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

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
- assignment
- access an array index
- etc…

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

Consecutive statements	Sum of times
Conditionals	Time to test + 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)*
- $n^3 + 2^n + 3n$ = $O(2^n)$ EXPONENTIAL GROWTH!
- n **`log`** $(10n^2)$
  - n log(10)+ 2n log (n) = O(n log *n)*

Efficiency

- What does it mean for an algorithm to be *efficient*?
  - We care about *time* (and sometimes *space*)
- Is the following a good definition?
  - "An algorithm is efficient if, when implemented, it runs quickly on real input instances"

Gauging efficiency (performance)

- Why not just time the program?
  - Too much *variability*, not reliable or *portable*:
    - Hardware: processor(s), memory, etc.
    - OS, Java version,
    - Other programs running
    - Implementation dependent
  - Might change based on choice of input
    - May *miss* worst-case input
    - What happens when *n* doubles in size?
  - Often want to evaluate an *algorithm*, not an implementation

Comparing algorithms

When is one *algorithm* (not *implementation*) better than another?

We will focus on large inputs, everything is fast when n is small.

Answer is *independent* of CPU speed, programming language, coding tricks, etc. and is general and rigorous.

We usually care about worst-case running times

- Provides a guarantee
- Difficult to find a satisfactory alternative
  - What about average case?
    - Difficult to express full range of input
  - Could we use randomly-generated input?
    - May learn more about generator than algorithm

Example

2	**3**	**5**	**16**	**37**	**50**	**73**	**75**	**126**

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
    ???
}
```

Linear search

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   for(int i=0; i < arr.length; ++i)
      if(arr[i] == k)
         return true;
   return false;
}
```

Best case?

k is in arr[0]

c1 steps

= *O*(1)

Worst case?

k is not in arr

c2*(arr.length)

= *O*(arr.length)

Binary search

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   return help(arr,k,0,arr.length);
}
boolean help(int[]arr, int k, int lo, int hi) {
   int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
   if(lo==hi)        return false;
   if(arr[mid]==k)   return true;
   if(arr[mid]< k)   return help(arr,k,mid+1,hi);
   else              return help(arr,k,lo,mid);
}
```

Binary search

Best case:

c1 steps = $O(1)$

Worst case:

$T(n)$ = c2 + $T(n/2)$ where n is `hi-lo` and c2 is a constant

$O($`log` $n)$ where n is `array.length` (recurrance relation)

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   return help(arr,k,0,arr.length);
}
boolean help(int[]arr, int k, int lo, int hi) {
   int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
   if(lo==hi)        return false;
   if(arr[mid]==k)   return true;
   if(arr[mid]< k)   return help(arr,k,mid+1,hi);
   else              return help(arr,k,lo,mid);
}
```

Solving Recurrence Relations

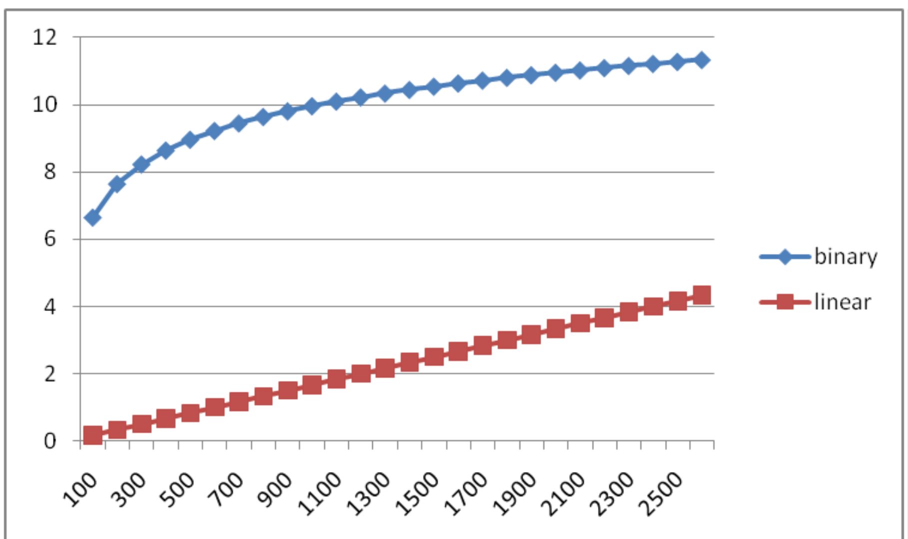
1. Determine the recurrence relation and the base case.
   - $T(n) = c2 + T(n/2)$ $T(1) = c1$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
   - $T(n) = c2 + c2 + T(n/4)$
     $= c2 + c2 + c2 + T(n/8)$
     $= \ldots$
     $= c2(k) + T(n/(2^k))$
3. Find a closed-form expression by setting *the number of expansions* to a value (e.g. 1) which reduces the problem to a base case
   - $n/(2^k) = 1$ means $n = 2^k$ means $k = \log_2 n$
   - So $T(n) = c2 \log_2 n + T(1)$
   - So $T(n) = c2 \log_2 n + c1$ (get to base case and do it)
   - So $T(n)$ is $O(\log n)$

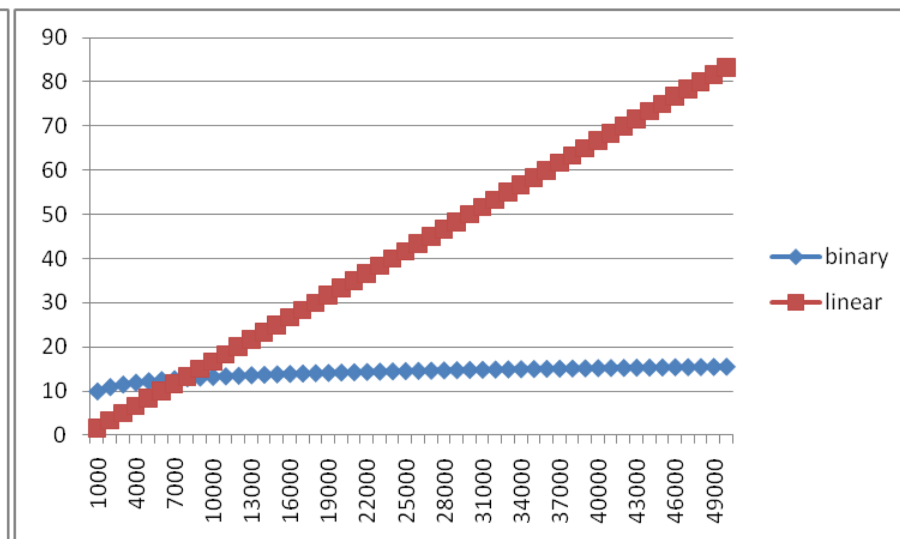
Ignoring constant factors

- So binary search is $O(\texttt{log}\ n)$ and linear is $O(n)$
  - But which is faster?
- Could depend on constant factors
  - How *many* assignments, additions, etc. for each n *(What is the constant c2?)*
    - *E.g.* $T(n) = 5{,}000{,}000n$ vs. $T(n) = 5n^2$
  - And could depend on overhead unrelated to n
    - *E.g.* $T(n) = 5{,}000{,}000 + \log n$ vs. $T(n) = 10 + n$
- But there exists some n_0 such that for all $n > n_0$ binary search wins

Example

- Let’s try to “help” linear search
  - Run it on a computer 100x as fast (say 2014 model vs. 1994)
  - Use a new compiler/language that is 3x as fast
  - Be a clever programmer to eliminate half the work
  - So doing each iteration is 600x as fast as in binary search





Big-O relates functions

O on a function $f(n)$ (for example n^2) means *the set of functions with asymptotic behavior* ***less than or equal to*** $f(n)$

So $(3n^2+17)$ **is in** $O(n^2)$

- $3n^2+17$ and n^2 have the same asymptotic behavior

Confusingly, we also say/write:

- $(3n^2+17)$ **is** $O(n^2)$
- $(3n^2+17) = O(n^2)$

We would never say $O(n^2) = (3n^2+17)$

Big-O, formally

Definition:

g(n) is in O(f(n)) if there exist positive constants c and n_0 such that

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Big-O, formally

Definition:

$g(n)$ is in $O(\ f(n)\ )$ if there exist positive constants c and n_0 such that

$g(n) \leq c\ f(n)$ for all $n \geq n_0$

- To show $g(n)$ is in $O(\ f(n)\ )$,
  - pick a c large enough to "cover the constant factors"
  - n_0 large enough to "cover the lower-order terms"
- Example:
  - Let $g(n) = 3n^2+17$ and $f(n) = n^2$

    What could we pick for c and n_0?

    $c = 5$ and $n_0 = 10$

    $(3*10^2)+17 \leq 5*10^2$ so $3n^2+17$ is $O(n^2)$

Example 1, using formal definition

- Let $g(n) = 1000n$ and $f(n) = n^2$
  - To prove $g(n)$ is in $O(f(n))$, find a valid c and n_0
  - The "cross-over point" is n=1000
    - $g(n) = 1000*1000$ and $f(n) = 1000^2$
  - So we can choose n_0=1000 and c=1
    - Many other possible choices, e.g., larger n_0 and/or c

> Definition: $g(n)$ is in $O(\ f(n)\ )$ if there exist positive constants c and n_0 such that
>
> $g(n) \leq c\, f(n)$ for all $n \geq n_0$

Example 2, using formal definition

- Let $g(n) = n^4$ and $f(n) = 2^n$
  - To prove $g(n)$ is in $O(f(n))$, find a valid c and n_0
  - We can choose n_0=20 and c=1
    - $g(n) = 20^4$ vs. $f(n) = 1*2^{20}$

Definition: $g(n)$ is in $O(\ f(n)\ )$ if there exist positive constants c and n_0 such that

$$g(n) \leq c\ f(n) \qquad \text{for all } n \geq n_0$$

What's with the c?

- The constant multiplier *c* is what allows functions that differ only in their largest coefficient to have the same asymptotic complexity
- Consider:

  $g(n) = 7n+5$

  $f(n) = n$

– These have the same asymptotic behavior (linear)
  - So $g(n)$ is in $O(f(n))$ even through $g(n)$ is always larger
  - The c allows us to provide a coefficient so that $g(n) \leq c\, f(n)$

– In this example:
  - To prove $g(n)$ is in $O(f(n))$, have $c = 12$, $n_0 = 1$

    $(7*1)+5 \leq 12*1$

What you can drop

- Eliminate coefficients because we don't have units anyway
  - $3n^2$ versus $5n^2$ doesn't mean anything when we have not specified the cost of constant-time operations

- Eliminate low-order terms because they have vanishingly small impact as n grows

- Do NOT ignore constants that are not multipliers
  - n^3 is not $O(n^2)$
  - 3^n is not $O(2^n)$

More Asymptotic Notation

- Upper bound: $O(\ f(n)\ )$ is the set of all functions asymptotically less than or equal to $f(n)$
  - $g(n)$ is in $O(\ f(n)\ )$ if there exist constants c and n_0 such that

    $g(n) \leq c\, f(n)$ for all $n \geq n_0$

- Lower bound: $\Omega(\ f(n)\ )$ is the set of all functions asymptotically greater than or equal to $f(n)$
  - $g(n)$ is in $\Omega(\ f(n)\ )$ if there exist constants c and n_0 such that

    $g(n) \geq c\, f(n)$ for all $n \geq n_0$

- Tight bound: $\theta(\ f(n)\ )$ is the set of all functions asymptotically equal to $f(n)$
  - $g(n)$ is in $\theta(\ f(n)\ )$ if **both** $g(n)$ is in $O(\ f(n)\ )$ **and**

    $g(n)$ is in $\Omega(\ f(n)\ )$

More Asymptotic Notation

- Upper bound: $O(\mathrm{f}(n))$

- Lower bound: $\Omega(\mathrm{f}(n))$

- Tight bound: $\theta(\mathrm{f}(n))$

Correct terms, in theory

A common error is to say $O(\ f(n)\ )$ when you mean $\theta(\ f(n)\ )$

- A linear algorithm is in both O(n) and $O(n^5)$
- Better to say it is $\theta(n)$
- That means that it is not, for example $O(\mathtt{log}\ n)$

Less common notation:

- "little-oh": intersection of "big-Oh" and *not* "big-Theta"
  - *For all c*, there exists an n_0 such that… ≤
  - Example: array sum is $o(n^2)$ but not $o(n)$
- "little-omega": intersection of "big-Omega" and *not* "big-Theta"
  - *For all c*, there exists an n_0 such that… ≥
  - Example: array sum is $\omega(\mathtt{log}\ n)$ but not $\omega(n)$

What we are analyzing

- We will give an *O* upper bound to the worst-case running time of an algorithm
- Example: binary-search algorithm
  - $O(\mathtt{log}\ n)$ in the worst-case
  - What is the best case?
  - The find-in-sorted-array ***problem*** is actually $\Omega(\mathtt{log}\ n)$ in the worst-case
    - *No* algorithm can do better
    - Why can't we find a O(f(n)) for a problem?
    - You can always create a slower algorithm

Other things to analyze

- Space instead of time
- Average case
  - If you assume something about the *probability distribution* of inputs
  - If you use randomization in the algorithm
    - Will see an example with sorting
  - With an *amortized guarantee*
    - Average time over any sequence of operations
    - Will discuss in a later lecture

Summary

Analysis can be about:

- The problem or the algorithm (usually algorithm)
- Time or space (usually time)
- Best-, worst-, or average-case (usually worst)
- Upper-, lower-, or tight-bound (usually upper or tight)

Big-O Caveats

- Asymptotic complexity focuses on behavior for large n

- You can be misled about trade-offs using it

- Example: $n^{1/10}$ vs. **`log`** n
  - Asymptotically $n^{1/10}$ grows more quickly
  - "Cross-over" point is around $5 * 10^{17}$
  - So for any smaller input, prefer $n^{1/10}$

- For *small* n, an algorithm with worse asymptotic complexity might be faster

Addendum: Timing vs. Big-O Summary

- Big-O
  - Examine the algorithm itself, not the implementation
  - Reason about performance as a function of *n*

- Timing
  - Compare implementations
  - Focus on data sets other than worst case
  - Determine what the constants actually are

Bubble Sort

```
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
0	n-1
1	n-2
2	n-3
…	…
n-2	1
n-1	0

Number of iterations
$0+1+2+3+..+(n-2)+(n-1)$
$= n(n-1)/2$

Each iteration takes c1

$O(n^2)$