CSCI 340

Analysis of Algorithms

Jon Lehuta

Northern Illinois University

August 22, 2019

Analysis of Algorithms - Outline

Analysis of Algorithms

- Introduction
- Big O notation
- Examples

Time and space

Analyzing an algorithm means:

- developing a formula for predicting how fast an algorithm is, based on the size of the input (time complexity), and/or
- developing a formula for predicting how much memory an algorithm requires, based on the size of the input (space complexity)

Usually *time* is our biggest concern

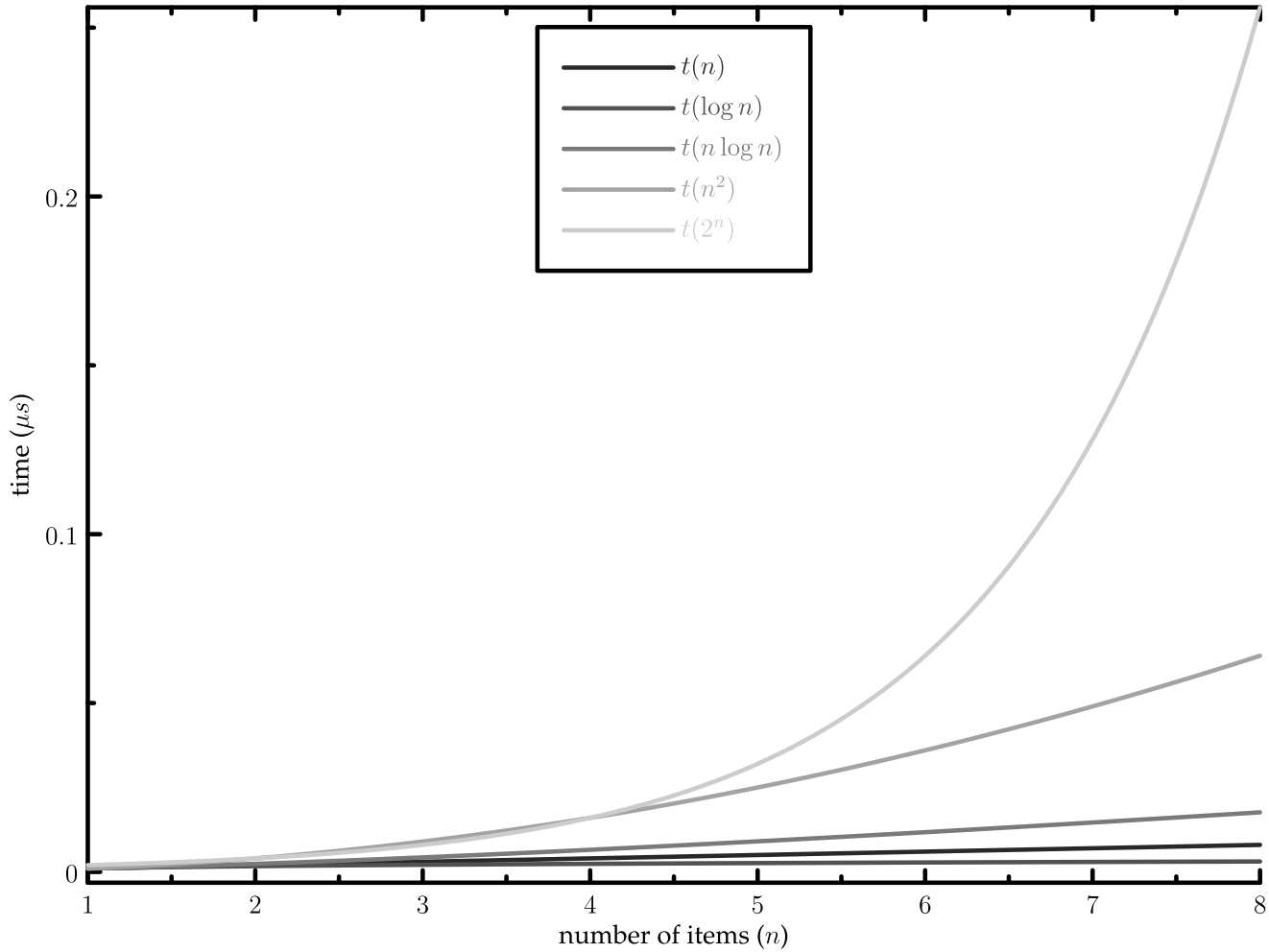
- We can buy more storage, but if a deadline is missed, it's a big deal.

How long to execute?

n	$t(n)$	$t(\log_2 n)$	$t(n \log_2 n)$	$t(n^2)$	$t(2^n)$
10	$0.01\mu s$	$0.003\mu s$	$0.033\mu s$	$0.1\mu s$	$1\mu s$
20	$0.02\mu s$	$0.004\mu s$	$0.086\mu s$	$0.4\mu s$	$1ms$
30	$0.03\mu s$	$0.005\mu s$	$0.147\mu s$	$0.9\mu s$	$1s$
40	$0.04\mu s$	$0.005\mu s$	$0.213\mu s$	$1.6\mu s$	18.3min
50	$0.05\mu s$	$0.006\mu s$	$0.282\mu s$	$2.5\mu s$	13 days
100	$0.10\mu s$	$0.007\mu s$	$0.664\mu s$	$10\mu s$	4×10^{13} years
1,000	$1.00\mu s$	$0.010\mu s$	$9.966\mu s$	$1ms$	
10,000	$10\mu s$	$0.013\mu s$	$130\mu s$	$100ms$	
100,000	$0.10ms$	$0.017\mu s$	$1.67ms$	$10s$	
1,000,000	$1ms$	$0.020\mu s$	$19.93ms$	16.7min	
10,000,000	$0.01s$	$0.023\mu s$	$0.23s$	1.16 days	

n is a number of items to process, and $t(x)$ is how much time it would take to process them all with an algorithm of complexity x, using a computer that does 1 *billion* operations per second.

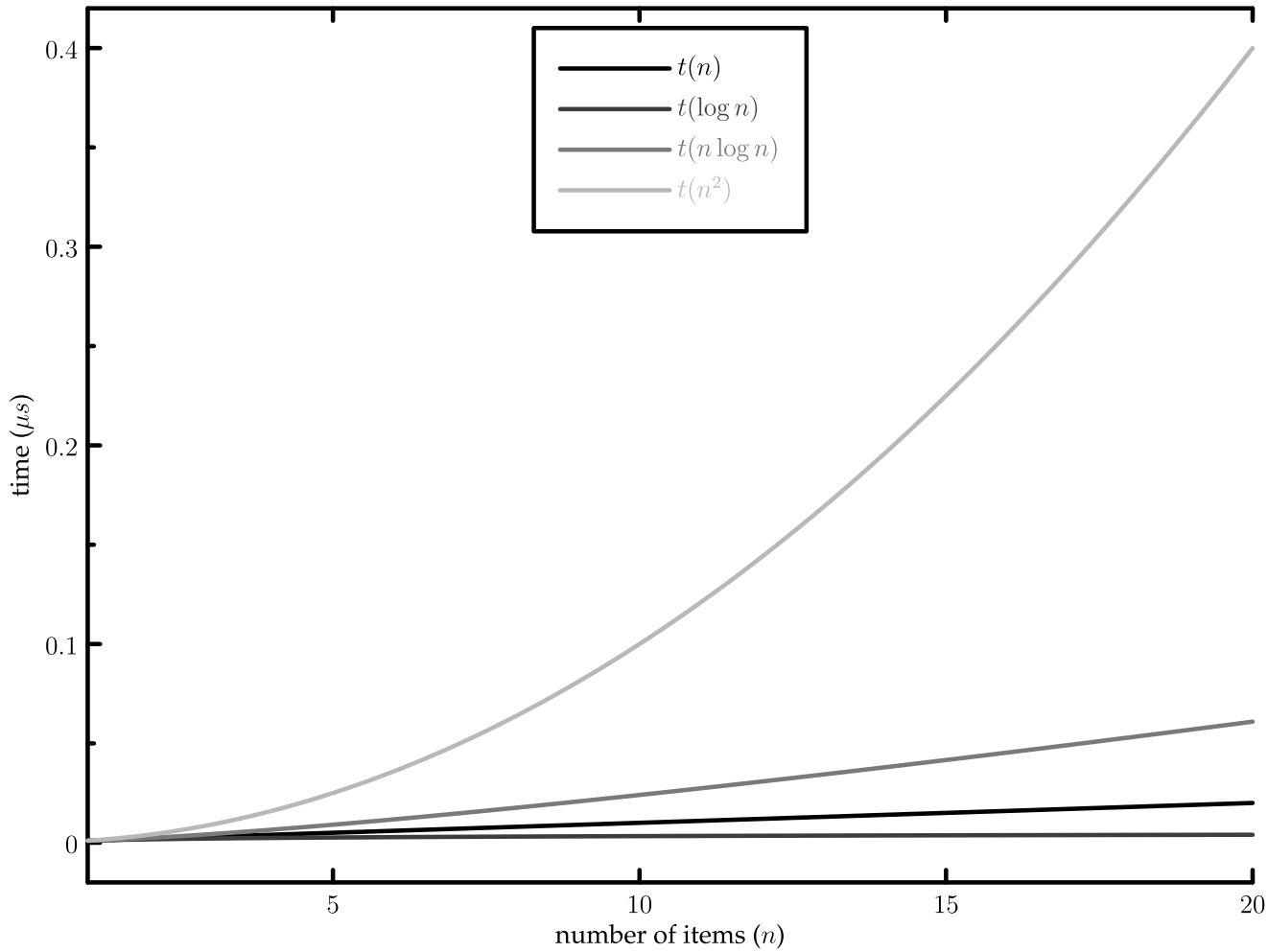
Graph: n up to 8



Notice that the 2^n is already taking way more time at $n = 8$. It only gets worse.

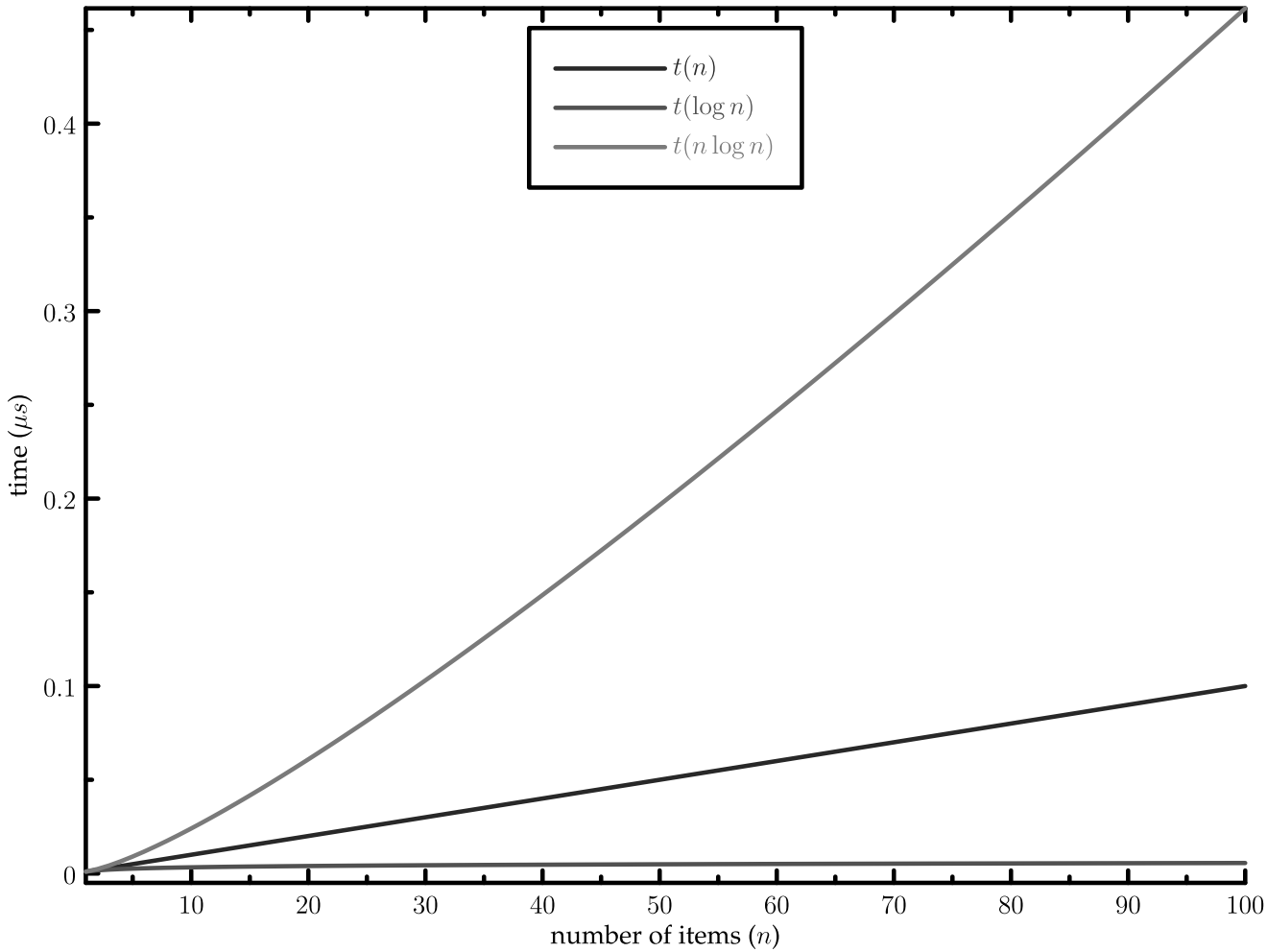
Northern Illinois University

Graph: n up to 20



2^n is not shown because it dwarfs all the others. Notice that n^2 is already starting to do the same.

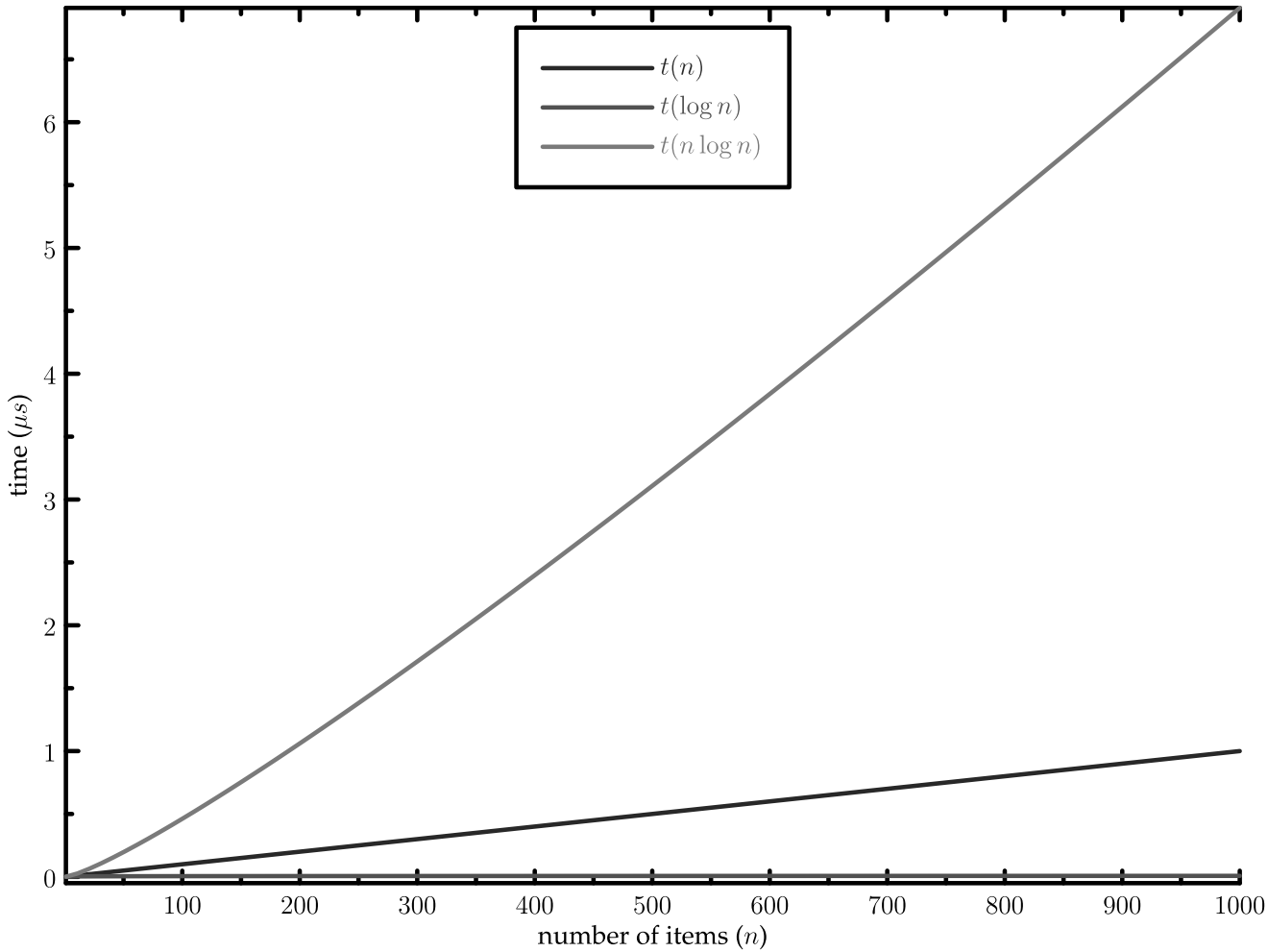
Graph: n up to 100



n^2 has been removed from graph for the same reason. Notice that n is growing much faster than the others.

Northern Illinois University

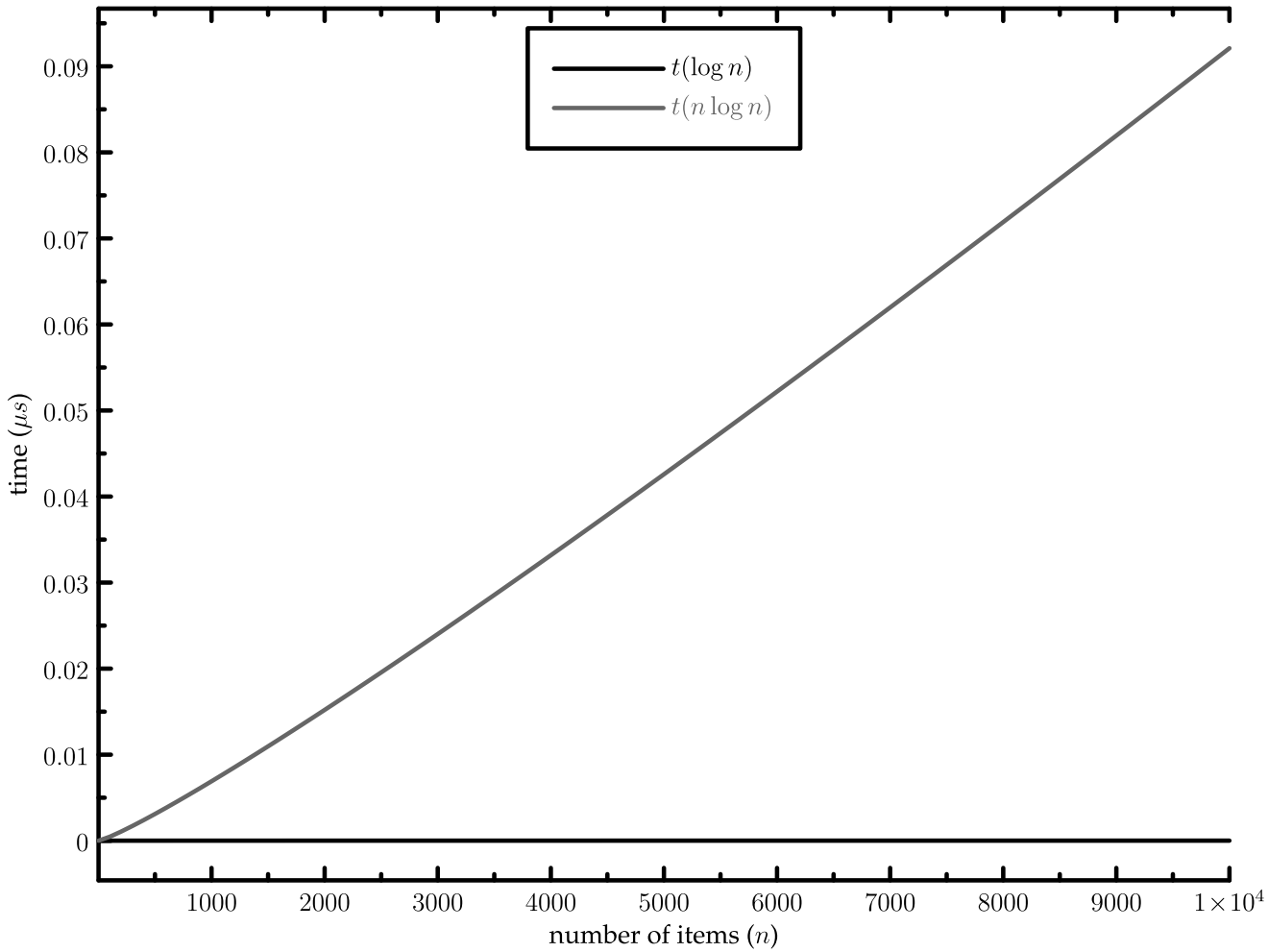
Graph: n up to 1000



As n increases, the difference between n and $n \log n$ grows.

Northern Illinois University

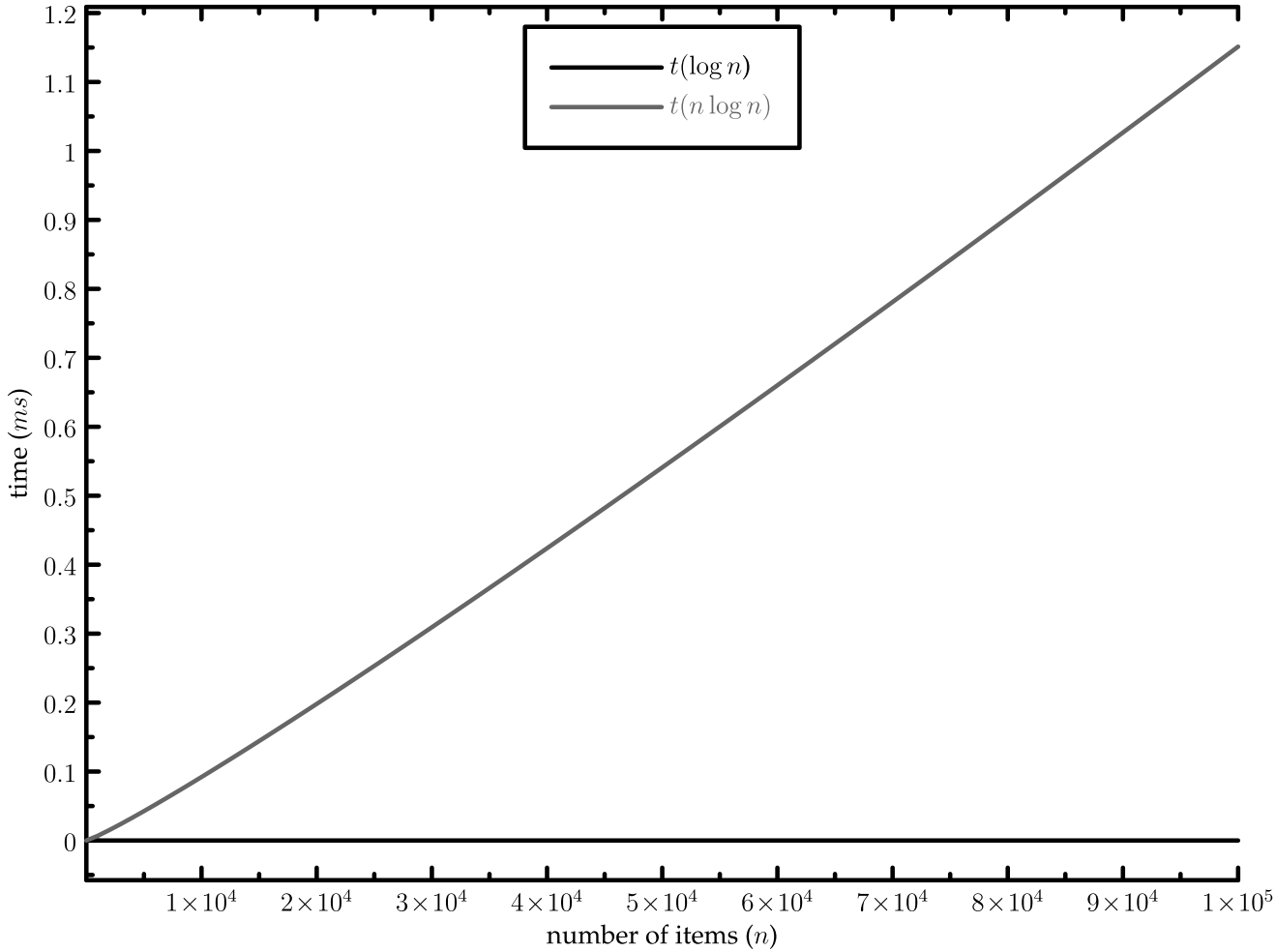
Graph: n up to 10000



$n \log n$ is taking way more time than $\log n$ now.

Northern Illinois University

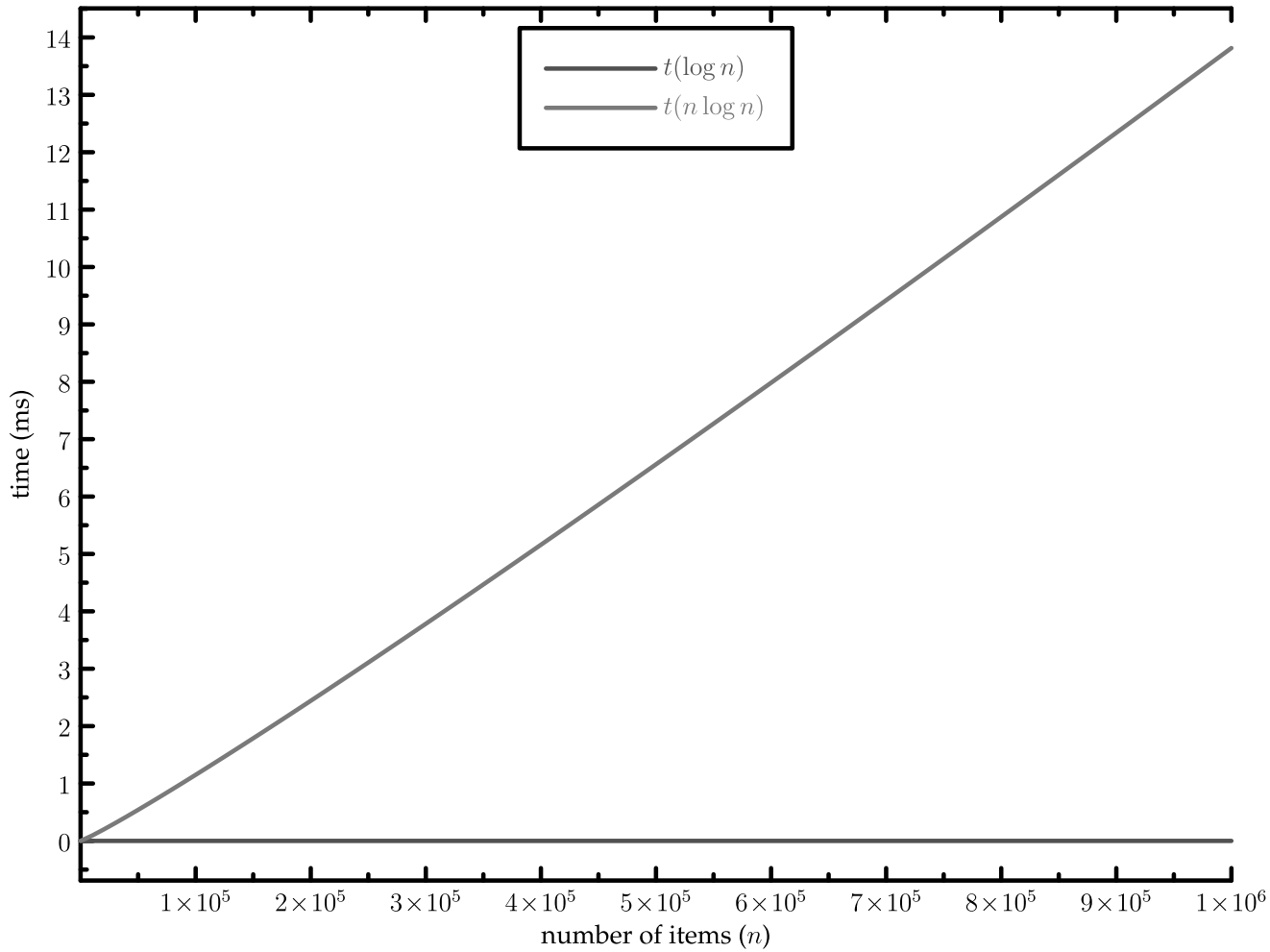
Graph: n up to 100000



At this point, $\log n$ is basically free compared to $n \log n$

Northern Illinois University

Graph: n up to 1000000



The time continues to go up for both, but we don't even notice $\log n$ anymore.

What does size of the input mean?

- If we are searching an array, the "size" of the input would be the number of elements in the array
- If we are merging two arrays, the "size" could be the total number of elements in both arrays
- If we are computing the nth Fibonacci number, or the nth factorial, the "size" is the number we are given, n.

We choose the "size" to be a parameter that has an effect on the time (or space) required. There doesn't always have to be just one answer, sometimes there are multiple parameters that matter, and we might want to look at the efficiencies of each of them separately.

Exact values

It is sometimes possible, in assembly language, to compute exact time and space requirements

- We can know exactly how many bytes and how many cycles each machine instruction requires.
- For a problem with a predictable sequence of steps (Fibonacci, factorial), we can determine how many instructions of each type are required

However, often the exact sequence of steps cannot be known in advance

- The steps required to sort an array depend on the contents of some data structure provided at runtime (which we cannot know in advance)

Higher-level languages

In a higher-level language, we have far less knowledge about exactly how long each operation will take.

- Which finishes faster, `x < 10` or `x <= 9`?
  - We don't know exactly what the compiler does with this
  - The compiler almost certainly optimizes the test anyway (replacing the slower version with the faster one)

Because of this, higher-level languages do not generally allow for an exact analysis

- Our timing analyses will use *major* simplifications
- Nevertheless, we can get some very useful results

Average, best, and worst cases

Usually we would like to find the average time to perform an algorithm

However,

- Sometimes the "average" isn't well defined
  - Example : Sorting an "average" array
    - Time typically depends on how out of order the array is
    - How out of order is the "average" unsorted array?
- Sometimes finding the average is too difficult

Often we have to be satisfied with finding the *worst case* (longest)

- Sometimes this is even what we want (i.e., for time-critical operations)

The best (fastest) case is seldom of interest

Constant time

Constant time means there is some constant k such that this operation always takes k units of time.

A statement takes constant time if:

- It does not include a loop
- It does not include calling a method whose time is unknown or is not a constant

If a statement involves a choice (`if` or `switch`) between several operations, each of which takes constant time, we consider the statement to take constant time

- This is consistent with worst-case analysis

Northern Illinois University

Example

Consider the following algorithm. (Assume that all variables are properly declared.)

```
//////////////////////////////////////////////////// Line // Ops //
cout << "Enter two numbers";                          //    1 //   1 //
cin >> num1 >> num2;                                  //    2 //   2 //
if(num1 >= num2)                                      //    3 //   1 //
  max = num1;                                         //    4 //   1 //
else                                                  //    5 //   - //
  max = num2;                                         //    6 //   1 //
cout << "The maximum number is: " << max << endl;     //    7 //   3 //
////////////////////////////////////////////////////////////////////////
```

Either Line 4 or Line 6 executes. Each is 1 operation. Therefore, the total number of operations executed is $1 + 2 + 1 + 1 + 3 = 8$. In this algorithm, the number of operations executed is *constant*.

Linear time

We may not be able to predict to the nanosecond how long a program will take, but do know *some* things about timing:

```
for(i=0, j=1; i<n; i++) {
  j = j * i; }
```

This loop takes time $kn + c$, for some constants k and c

- k How long it takes to go through the loop once (the time for `j = j * i`, plus loop overhead)
- n The number of times through the loop (we can use this as the "size" of the problem)
- c The time it takes to initialize the loop

The total time $kn + c$ is *linear* in n

Northern Illinois University

Example

```
//////////////////////////////////////////////////// Line // Ops //
sum = 0;                                             //    1 //   1 //
num = 10;                                            //    2 //   1 //
while(num != -1)                                     //    3 //   1 //
{                                                    //    4 //   - //
  sum = sum + num;                                   //    5 //   2 //
  num = num - 1;                                     //    6 //   2 //
}                                                    //    7 //   - //
cout << sum;                                         //    8 //   1 //
////////////////////////////////////////////////////////////////////////
```

If we were to change `num` to match our n, then the `while` loop executes n times:

- The loop body is 4 operations, and it happens n times.
- The loop check is 1 operation, and it happens $n + 1$ times.
- There are 4 operations done outside the loop.

$4n + n + 1 + 4 = 5n + 5$ operations

Example, Cont.

n	$5n + 5$
10	55
100	505
1,000	5,005
10,000	50,005

For very large values of n, the $5n$ term dominates term and the 5 term becomes negligible.

Constant time is (usually) better than linear time

Suppose we have two algorithms to solve a task:

- Algorithm A takes 5000 time units
- Algorithm B takes $100n$ time units

Which is better?

- Clearly, algorithm B is better if our problem size is small, $n < 50$
- Algorithm A is better for larger problems, with $n > 50$
- B is better on small problems that are quick anyway.
- A is better for large problems, where it will matter more.

We usually care most about very large problems

- But not always!

What about the constants?

An added constant, $f(n) + c$, becomes less and less important as n gets larger, assuming the degree of $f(n)$ is greater than 0 (i.e., not constant in n).

A constant multiplier, $k \cdot f(n)$, does not get less important, but...

- Improving k gives a linear speedup (cutting k in half cuts the time required in half)
- Improving k is usually accomplished by careful code optimization, not by better algorithms
- We are not that concerned with *only* linear speedups.

Bottom line: Forget the constants.

Simplifying the formulae

Throwing out the constants is one of two simplifications we do in analysis of algorithms

- By throwing out constants, we simplify $12n^2 + 35$ to just n^2

Our timing formula is a polynomial, and may have terms of various orders (constant, linear, quadratic, cubic, etc.)

- We usually discard all but the highest-order term
- We simplify $n^2 + 3n + 5$ to just n^2

Big O notation

When we have a polynomial that describes the time requirements of an algorithm, we simplify it by:

- Throwing out all but the highest-order term
- Throwing out any constant factors

If an algorithm takes $12n^3 + 4n^2 + 8n + 35$ time, we simplify this formula to just n^3

We say the algorithm requires $O(n^3)$ time

- We call this Big O notation

Can we justify Big O notation?

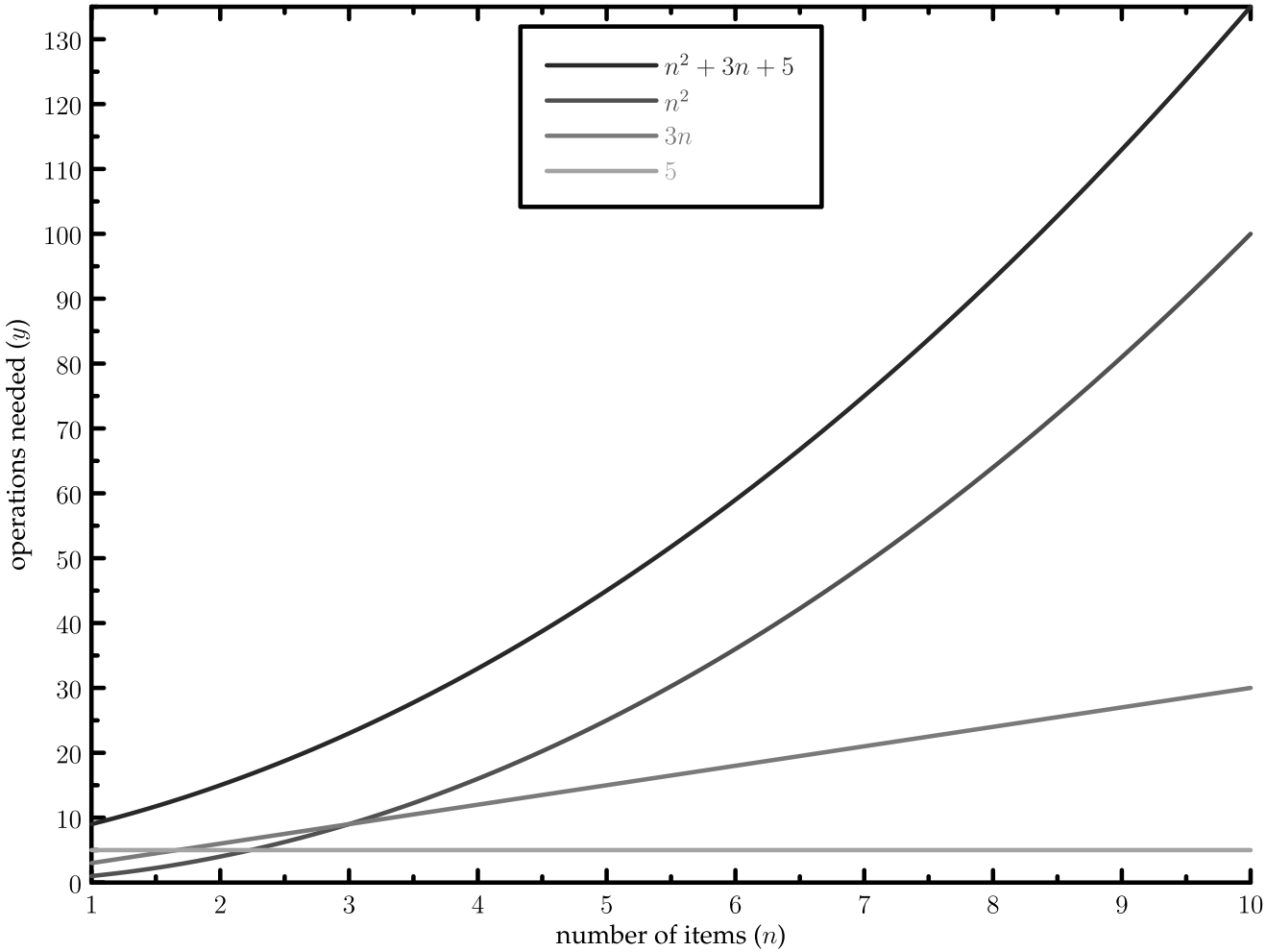
Big O notation is a *huge* simplification; can we justify it?

- It only makes sense for *large* problem sizes
- For sufficiently large problem sizes, the highest-order term overwhelms all the rest anyway.

Consider $R = x^2 + 3x + 5$ as x varies:

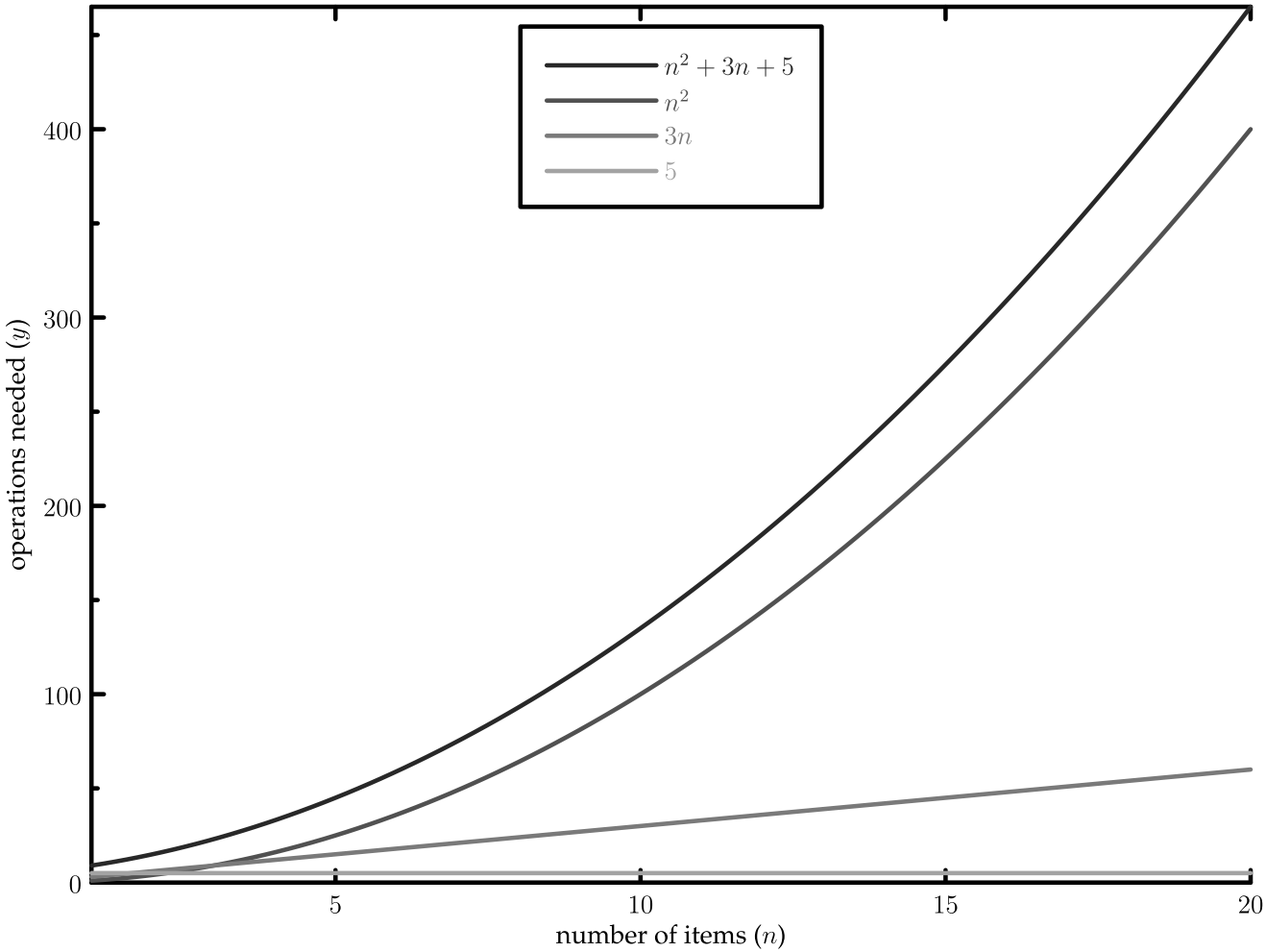
x	x^2	$3x$	5	$R = x^2 + 3x + 5$
0	0	0	5	5
10	100	30	5	135
100	10,000	300	5	10,305
1,000	1,000,000	3,000	5	1,003,005
10,000	100,000,000	30,000	5	100,030,005
100,000	10,000,000,000	300,000	5	10,000,300,005

Graph 1 <= n <= 10



$y = n^2 + 3n + 5$, for $n = 1..10$

Graph 1 <= n <= 20



$y = n^2 + 3n + 5$, for $n = 1..20$

Common time complexities

Big-O	Name
$O(1)$	constant time
$O(\log n)$	logarithmic time
$O(n)$	linear time
$O(n \log n)$	log linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(n^k)$	polynomial time
$O(2^n)$	exponential time

These are ordered from best (at the top) to worst, for large n. ($n \to \infty$)

NP-complete problem

Hard problem:

- Most problems discussed are efficient (poly time)
- An interesting set of hard problems: NP-complete.

Why interesting:

- Not known whether efficient algorithms exist for them.
- If exist for one, then exist for all.
- A small change may cause big change.

Why important:

- Arise surprisingly often in real world.
- Not waste time on trying to find an efficient algorithm to get best solution, instead find approximate or near-optimal solution.

Example: Traveling-salesman problem: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Example 1

Find the running time, worst time, complexity, or Big-O analysis for the following code

```
for(i=0; i<n; i++)      // loop 1
   for(j=0; j<n; j++)   // loop 2
       cin >> A[i][j]; // single operation takes k units of time
```

Loop 1 iterates over its body n times, which includes loop 2. Loop 2 iterates over its body of a single operation that takes k time to complete.

$$n(n(k)) = kn^2 \rightarrow O(n^2)$$

Example 2

```
for(i = 0; i <= n; i++)
  for(j = 0; j <= i; j++)
     A[i][j] = 0;
```

- First time through, inner loop does 1 iteration, then 2, then 3, etc. up to n. This is a simple arithmetic series, and:

$$k \sum_{i=1}^{n} i = \frac{k n(n-1)}{2} \rightarrow O(n^2)$$

Example 3

```
for(i = 0; i < n; i++)      // loop 1
{
  for(j = 0; j < n; j++)    // loop 2
    A[i][j] = j*2;
  for(k = 0; k <2* n; j++) // loop 3
    A[i][k] = k*3;
}
```

- Loop 1 surrounds both of the other loops, and does n iterations of its body.
- Loop 2 does n iterations with 1 operation each time.
- Loop 3 does $2n$ iterations with 1 operation each time.

$$n(n + 2n) = n(3n) = 3n^2 \rightarrow O(n^2)$$

Example 4

```
for(i = 0; i < n; i++)          // Loop 1
{
  for(j = 0; j < n; j++)        // Loop 2
    A[i][j]= i * j;
  for (k = 0; k < 2*n; k++)     // Loop 3
    for (m = 0; m < 2*n; m++) // Loop 4
      sum = sum + 1;
}
```

- Loop 1 contains all of the others and does n iterations of its body.
- Loop 2 does n iterations with 1 operation each time.
- Loop 3 does $2n$ iterations of its body, which contains loop 4.
- Loop 4 does $2n$ iterations with 1 operation each time.

$$n(n + 2n(2n(1))) = n(n + 4n^2) = 4n^3 + n^2 \rightarrow O(n^3)$$

Northern Illinois University

Example 5

```
int i, j, tofind, A[100], n = 100; // 1 assignment operation
for(j = 0; j < n; j++)             // Loop 1
  A[j] = j * 2;
i = 0;                             // 1 assignment operation
cin >> tofind;                     // 1 input operation
while(A[i]!=tofind && i<n) i++;    // Loop 2
if(i >= n) cout << "not found";    // constant in n, both
else       cout << "found";        //   branches same length
```

- There are 4 operations outside of the loop.
- Loop 1 iterates n times over its 1 operation body.
- Loop 2 iterates up to n times over its 1 operation body.

$$4 + n + n \text{ (worst case)} = 4 + 2n \rightarrow O(n)$$

$$4 + n + 1 \text{ (best case)} = 5 + n \rightarrow O(n)$$

Northern Illinois University

For more information

http://bigocheatsheet.com/