CSE 373/548 – Algorithms

Instructor: Steven Skiena

Office: 1411 Computer Science Building

Phone: 516-632-9026

Email: skiena@cs.sunysb.edu

Office Hours: 1-2:30PM Tuesday-Thursday, and by appointment.

Course Time: 11:30AM-12:50PM **Place:** Studio A, ECC Building

Teaching Assistant: Daren Krebsbach

Office: 1307 Computer Science Building

Email: daren@sbcs.sunysb.edu

Office Hours: 3:00-4:30PM Monday-Wednesday, and by appointment.

Textbook: Cormen, Leiserson, Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

- **Undergraduate Grading:** Grades will be assigned based on the following formula, with cut-offs determined by my opinion of students on the boundary.

  Daily Problems - 5%

  Homework Assignment - 15%

  Midterm 1 - 25%

  Midterm 2 - 25%

  Final - 30%

- **Graduate Student Grading:** Grades will be assigned based on the following formula, with cut-offs determined by my opinion of students on the boundary.

  Homeworks - 20%

  Midterm 1 - 15%

  Midterm 2 - 15%

  Project - 25%

  Final - 25%

- **Homeworks:** There will be five homeworks over the course of the semester. The third assignment will be a program. As discussed below, all homeworks (except the third) can be done in pairs. On each homework assignment, only a subset of the problems will be graded.

- **Graduate Student Project:** This is your opportunity to study some aspect of the design and analysis of algorithms in depth. Suitable projects will be term papers, implementations, or original research. A list of possible topics will be distributed about two weeks into the semester, although you are encouraged to create your own. A brief proposal of what you intend to do must be submitted by mid-March. Each student will give a five minutes oral report on their project during the last week of class.

Rules of the Games:

1. This semester CSE 548 will be part of EngiNet, meaning that all lectures will be videotaped. Any EngiNet students should contact me by email or phone as soon as possible so I am aware of your existence. For local students, these tapes provide an opportunity to review lectures or enable you to attend lectures at an alternate time. A regularly scheduled screening of the previously lecture will be held in room Javits 108, time TBA. The tapes are also available for loan or viewing in the AV room in Javits Hall.

2. Combining CSE 373 and CSE 523 has proven very successful each time it has been taught. The grad and undergrad courses will graded on separate curves, and I will expect considerably more from the graduate students, in terms of the project and better performance.

3. I will lecture from slides, which are now more or less available on-line. I will also make copies of my slides available in the CS library after lecture. If there is sufficient demand, we may also make them available through BASIX or a print shop off campus.

4. The WWW page for the course is:

   http://www.cs.sunysb.edu/~ skiena/548/cse548.html

   All course handouts and notes are available there, along with the latest announcements. Please check it out.

5. The best way to learn the material is by solving problems. You are encouraged to work in pairs, for the best way to understand the subtleties of the homework problems is to argue about the answers. Each of you should look at all the problems independently, and not just divide the list in two parts each time. Don't be a leech and let your partner do all the work. Unless you learn how to solve problems, I *promise* that you will get burned on the exams and thus for your final grade.

6. The partner system relies upon a certain maturity among the students. If you don't have a partner, tell me and I will hook you up with one. If you are having trouble with your partner and want a divorce, tell me and I will set you up with a new one. I will act as a broker *but not* as a counselor. I

do not want to hear what a louse your old partner is, and you will get a dirty look from me when you demand a divorce regradless of who was at fault.

7. At the start of each class, I will work out one previously identified homework problem, emphasizing the thought process leading to the solution. To get the most benefit from this, you should try to work out the problem before lecture, I will collect your solutions for these daily problems at the beginning of each class.

8. Only one solution to the assignment per pair should be turned in, with the partners alternating who writes up the final solution. The scribe for each assignment will have to label themselves as such. Unless announced otherwise in class, any solution to a part of a homework problem which takes more than one side of a sheet of paper will not be graded. This is to save you the ordeal of trying to impress with volume instead of quality.

9. Because a primary goal of the course is to teach professionalism, any academic dishonesty will be viewed as evidence that this goal has not been achieved, and will be grounded for receiving a grade of F. (See CEAS Procedures and Guideline Governing Academic Dishonesty, 1/81.)

10. If you have any condition, such as a physical or mental disability, which will make it difficult for

you to carry out the work as I have outlined it or which requires extra time on examinations, please notify me in the first two weeks of the course so that we may make appropriate arrangements.

11. I understand that everyone gets into a time bind now and then, and that accidents and troubles befall even the most dedicated student. Thus every student will get one free extension on a homework for up to a week without a late penalty. You do not have to ask for this – just write that you are using your free extension when you turn it in. Don't waste this extension or feel obligated to use it, since you will get a very dirty look if try to get another one even with a good excuse.

12. Homework assignments will be due at the *beginning of class*. The penalty will be 20% per day.

13. I hope to establish as much personal contact with each of you as is possible in a class this size. Don't be afraid to stop by during office hours to ask questions or say hello. To facilitate interaction, every few weeks there will be 'Pizza with the Prof'. Outside my office will be a sheet for you to sign-up to join 5-10 other students from the class for a pizza lunch (on me). I look forward to getting to know you.

Tentative Schedule

subject	topics	reading
Preliminaries	Analyzing algorithms	1-32
”	Asymptotic notation	32-37
”	Recurrence relations	53-64
Sorting	Heapsort	140-150
”	Quicksort	153-167
”	Linear Sorting	172-182
Searching	Data structures	200-215
”	Binary search trees	244-245
”	Red-Black trees:insertion	262-272
“	Red-Black trees:deletion	272-277
”	Splay Trees/Amortized Analysis	
MIDTERM 1		
Comb. Search	Backtracking	
”	Elements of dynamic programming	301-314
”	Examples of dynamic programming	314-323
Graph Algorithms	Data structures for graphs	465-477
”	Breadth/depth-first search	477-483
”	Topological Sort/Connectivity	485-493
”	Minimum Spanning Trees	498-510
”	Single-source shortest paths	514-532
”	All-pairs shortest paths	550-563
MIDTERM 2		
Intractability	P and NP	916-928
”	NP-completeness	929-939
”	NP-completeness proofs	939-951
”	Further reductions	951-960
”	Approximiation algorithms	964-974
”	Set cover / knapsack heuristics	974-983
Semester Review		HW5 IN
Graduate Student Presentations		talks
FINAL EXAM		

What Is An Algorithm?

Algorithms are the ideas behind computer programs.

An algorithm is the thing which stays the same whether the program is in Pascal running on a Cray in New York or is in BASIC running on a Macintosh in Kathmandu!

To be interesting, an algorithm has to solve a general, specified problem. An algorithmic problem is specified by describing the set of instances it must work on and what desired properties the output must have.

Example: Sorting

Input: A sequence of N numbers $a_1 ... a_n$

Output: the permutation (reordering) of the input sequence such as $a_1 \leq a_2 \ldots \leq a_n$.

We seek algorithms which are *correct* and *efficient*.

Correctness

For any algorithm, we must prove that it *always* returns the desired output for all legal instances of the problem.

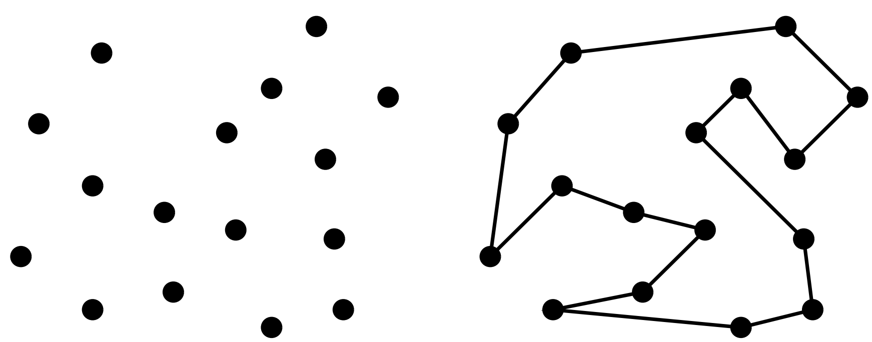
For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.

Correctness is Not Obvious!

The following problem arises often in manufacturing and transportation testing applications.

Suppose you have a robot arm equipped with a tool, say a soldering iron. To enable the robot arm to do a soldering job, we must construct an ordering of the contact points, so the robot visits (and solders) the first contact point, then visits the second point, third, and so forth until the job is done.

Since robots are expensive, we need to find the order which minimizes the time (ie. travel distance) it takes to assemble the circuit board.



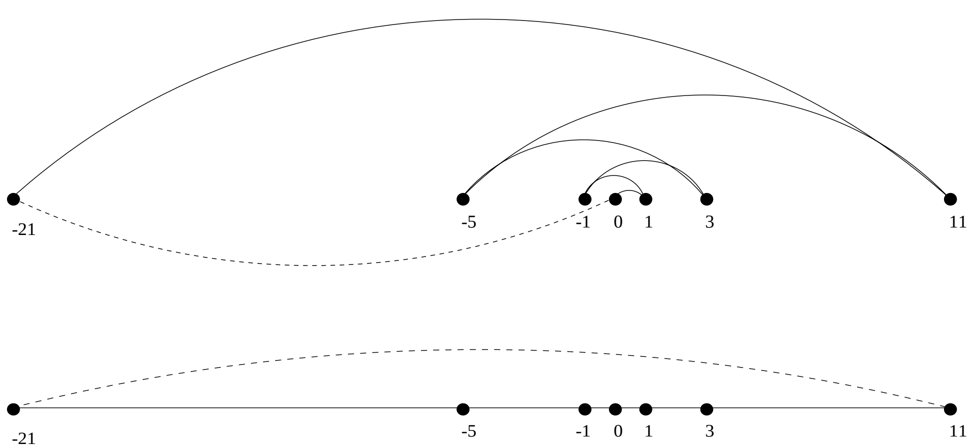
You are given the job to program the robot arm. Give me an algorithm to find the best tour!

Nearest Neighbor Tour

A very popular solution starts at some point p_0 and then walks to its nearest neighbor p_1 first, then repeats from p_1, etc. until done.

Pick and visit an initial point p_0
$p = p_0$
$i = 0$
While there are still unvisited points
    $i = i + 1$
    Let p_i be the closest unvisited point to p_{i-1}
    Visit p_i
Return to p_0 from p_i

This algorithm is simple to understand and implement and very efficient. However, it is **not correct!**



Always starting from the leftmost point or any other point will not fix the problem.

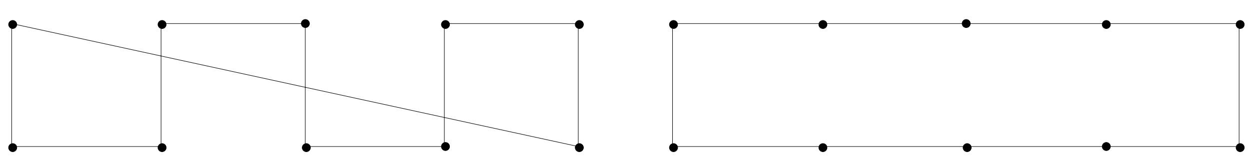
Closest Pair Tour

Always walking to the closest point is too restrictive, since that point might trap us into making moves we don't want.

Another idea would be to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch to be formed, until we have a single chain with all the points in it.

Let n be the number of points in the set
$d = \infty$
For $i = 1$ to $n - 1$ do
  For each pair of endpoints (x, y) of partial paths
    If $dist(x, y) \leq d$ then
      $x_m = x$, $y_m = y$, $d = dist(x, y)$
  Connect (x_m, y_m) by an edge
Connect the two endpoints by an edge.

Although it works correctly on the previous example, other data causes trouble:



This algorithm is **not correct**!

A Correct Algorithm

We could try all possible orderings of the points, then select the ordering which minimizes the total length:

$d = \infty$
For each of the $n!$ permutations Π_i of the n points
    If $(cost(\Pi_i) \leq d)$ then
        $d = cost(\Pi_i)$ and $P_{min} = \Pi_i$
Return P_{min}

Since all possible orderings are considered, we are guaranteed to end up with the shortest possible tour.

Because it trys all $n!$ permutations, it is extremely slow, much too slow to use when there are more than 10-20 points.

No efficient, correct algorithm exists for the *traveling salesman problem*, as we will see later.

Efficiency

"Why not just use a supercomputer?"

Supercomputers are for people too rich and too stupid to design efficient algorithms!

A faster algorithm running on a slower computer will *always* win for sufficiently large instances, as we shall see.

Usually, problems don't have to get that large before the faster algorithm wins.

Expressing Algorithms

We need some way to express the sequence of steps comprising an algorithm.

In order of increasing precision, we have English, pseudocode, and real programming languages. Unfortunately, ease of expression moves in the reverse order.

I prefer to describe the *ideas* of an algorithm in English, moving to pseudocode to clarify sufficiently tricky details of the algorithm.

The RAM Model

Algorithms are the *only* important, durable, and original part of computer science *because* they can be studied in a machine and language independent way.

The reason is that we will do all our design and analysis for the RAM model of computation:

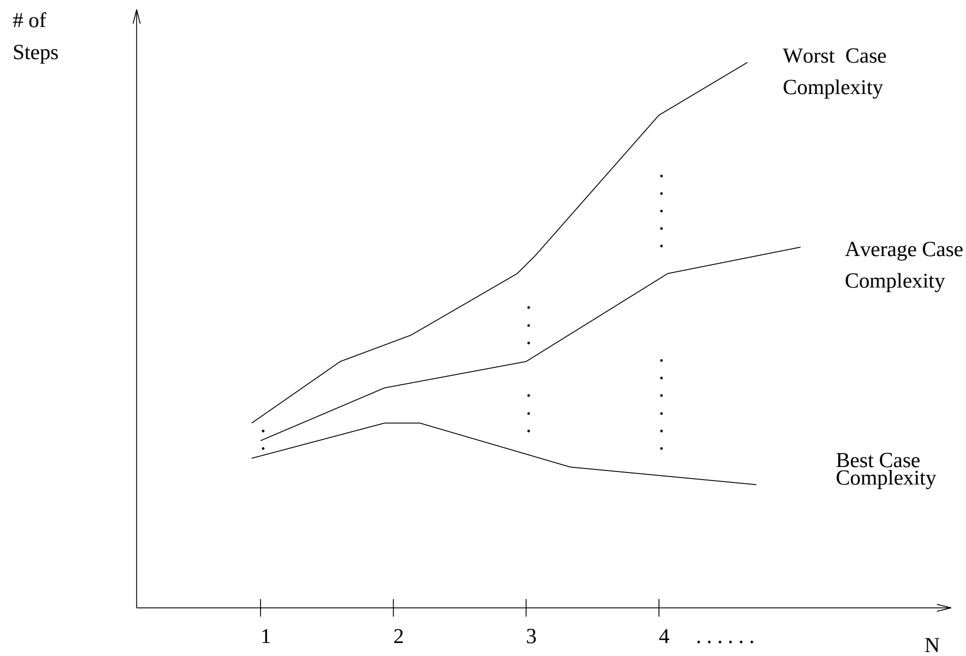
- Each "simple" operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are *not* simple operations, but depend upon the size of the data and the contents of a subroutine. We do not want "sort" to be a single step operation.
- Each memory access takes exactly 1 step.

We measure the run time of an algorithm by counting the number of steps.

This model is useful and accurate in the same sense as the flat-earth model (which *is* useful)!

Best, Worst, and Average-Case

The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.



The *best case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

The *average-case complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size n.

Each of these complexities defines a numerical function – time vs. size!

Insertion Sort

One way to sort an array of n elements is to start with a_n empty list, then successively insert new elements in the proper position:

$$a_1 \leq a_2 \leq \ldots \leq a_k \mid a_{k+1} \ldots a_n$$

At each stage, the inserted element leaves a sorted list, and after n insertions contains exactly the right elements. Thus the algorithm must be correct.

But how *efficient* is it?

Note that the run time changes with the permutation instance! (even for a fixed size problem)

How does insertion sort do on sorted permutations?

How about unsorted permutations?

Exact Analysis of Insertion Sort

Count the number of times each line of pseudocode will be executed.

Line	InsertionSort(A)	#Inst.	#Exec.
1	for j:=2 to len. of A do	c1	n
2	key:=A[j]	c2	n-1
3	/* put A[j] into A[1..j-1] */	c3=0	/
4	i:=j-1	c4	n-1
5	while $i > 0 \& A[1] > key$ do	c5	tj
6	A[i+1]:= A[i]	c6	
7	i := i-1	c7	
8	A[i+1]:=key	c8	n-1

The **for** statement is executed $(n-1)+1$ times (why?)

Within the **for** statement, ”key:=A[j]” is executed n-1 times.

Steps 5, 6, 7 are harder to count.

Let $t_j = 1+$ the number of elements that have to be slide right to insert the jth item.

Step 5 is executed $t_2 + t_3 + ... + t_n$ times.

Step 6 is $t_{2-1} + t_{3-1} + ... + t_{n-1}$.

Add up the executed instructions for all pseudocode lines to get the run-time of the algorithm:

$c_1 * n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1)$
$+ c_7 \sum_{j=2}^{n}(t_j - 1) + c_8$

What are the $t_j's$? They depend on the particular input.

Best Case

If it's already sorted, all t_j's are 1.

Hence, the best case time is

$$c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1) = Cn + D$$

where C and D are constants.

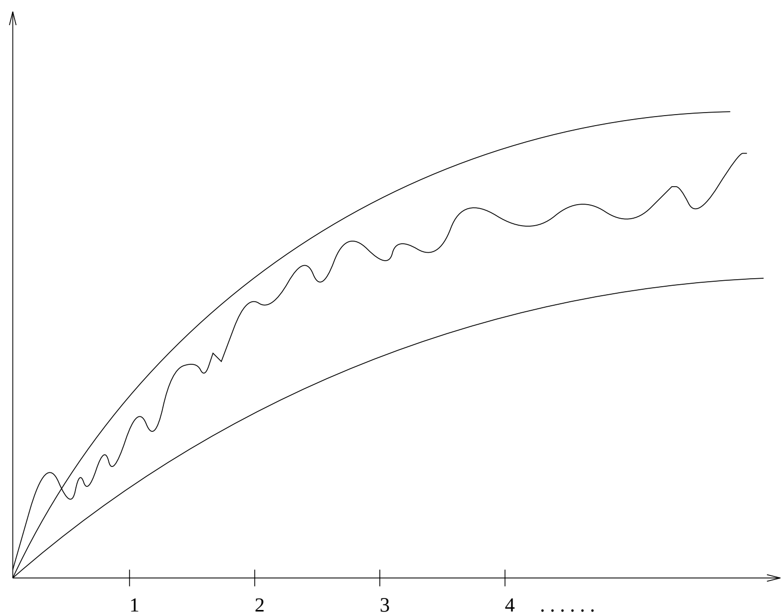
Worst Case

If the input is sorted in *descending* order, we will have to slide *all* of the already-sorted elements, so $t_j = j$, and step 5 is executed

$$\sum_{j=2}^{n} j = (n^2 + n)/2 - 1$$

Exact Analysis is Hard!

We have agreed that the best, worst, and average case complexity of an algorithm is a numerical function of the size of the instances.



However, it is difficult to work with exactly because it is typically very complicated!

Thus it is usually cleaner and easier to talk about *upper and lower bounds* of the function.

This is where the dreaded big O notation comes in!

Since running our algorithm on a machine which is twice as fast will effect the running times by a multiplicative constant of 2 - we are going to have to ignore constant factors anyway.

Names of Bounding Functions

Now that we have clearly defined the complexity functions we are talking about

- $g(n) = O(f(n))$ means $C \times f(n)$ is an *upper bound* on $g(n)$.

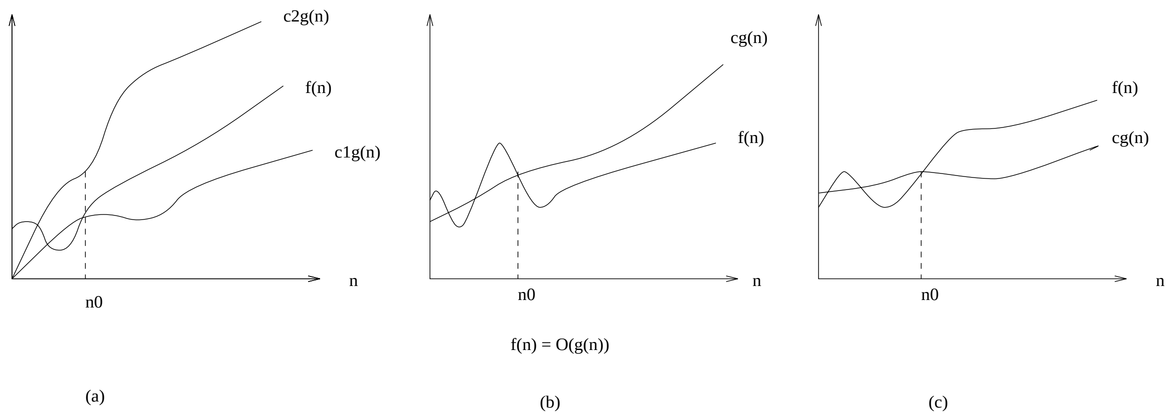
- $g(n) = \Omega(f(n))$ means $C \times f(n)$ is a *lower bound* on $g(n)$.

- $g(n) = \Theta(f(n))$ means $C_1 \times f(n)$ is an upper bound on $g(n)$ and $C_2 \times f(n)$ is a lower bound on $g(n)$.

Got it?

All of these definitions imply a constant n_0 *beyond which* they are satisfied. We do not care about small values of n.

O, Ω, and Θ



The value of n_0 shown is the minimum possible value; any greater value would also work.

(a) $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ inclusive.

(b) $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c \cdot g(n)$.

(c) $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $c \cdot g(n)$.

Asymptotic notation (O, Θ, Ω) are as well as we can practically deal with complexity functions.

What does all this mean?

$$
\begin{aligned}
3n^2 - 100n + 6 &= O(n^2) \; \textit{because} \; 3n^2 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &= O(n^3) \; \textit{because} \; .01n^3 > 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &\neq O(n) \; \textit{because} \; c \cdot n < 3n^2 \quad \textit{when} \quad n > c
\end{aligned}
$$

$$
\begin{aligned}
3n^2 - 100n + 6 &= \Omega(n^2) \; \textit{because} \; 2.99n^2 < 3n^2 - 100n + 6 \\
3n^2 - 100n + 6 &\neq \Omega(n^3) \; \textit{because} \; 3n^2 - 100n + 6 < n^3 \\
3n^2 - 100n + 6 &= \Omega(n) \; \textit{because} \; 10^{10^{10}} n < 3n^2 - 100 + 6
\end{aligned}
$$

$$
\begin{aligned}
3n^2 - 100n + 6 &= \Theta(n^2) \; \textit{because} \; O \quad \textit{and} \quad \Omega \\
3n^2 - 100n + 6 &\neq \Theta(n^3) \; \textit{because} \; O \quad \textit{only} \\
3n^2 - 100n + 6 &\neq \Theta(n) \; \textit{because} \; \Omega \quad \textit{only}
\end{aligned}
$$

Think of the equality as meaning *in the set of functions*.

Note that time complexity is every bit as well defined a function as $\sin(x)$ or you bank account as a function of time.

Testing Dominance

$f(n)$ dominates $g(n)$ if $\lim_{n\to\infty} g(n)/f(n) = 0$, which is the same as saying $g(n) = o(f(n))$.

Note the little-oh – it means "grows strictly slower than".

Knowing the dominance relation between common functions is important because we want algorithms whose time complexity is as low as possible in the hierarchy. If $f(n)$ dominates $g(n)$, f is much larger (ie. slower) than g.

- n^a dominates n^b if $a > b$ since
$$\lim_{n\to\infty} n^b/n^a = n^{b-a} \to 0$$
- $n^a + o(n^a)$ doesn't dominate n^a since
$$\lim_{n\to\infty} n^b/(n^a + o(n^a) \to 1$$

Complexity	10	20	30	40
n	0.00001 sec	0.00002 sec	0.00003 sec	0.00004 sec
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.016 sec
n^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec
n^5	0.1 sec	3.2 sec	24.3 sec	1.7 min
2^n	0.001 sec	1.0 sec	17.9 min	12.7 days
3^n	0.59 sec	58 min	6.5 years	3855 cent

Working with the Notation

Suppose $f(n) = O(n^2)$ and $g(n) = O(n^2)$.

What do we know about $g'(n) = f(n) + g(n)$? Adding the bounding constants shows $g'(n) = O(n^2)$.

What do we know about $g''(n) = f(n) - g(n)$? Since the bounding constants don't necessary cancel, $g''(n) = O(n^2)$

We know nothing about the lower bounds on $g' + g''$ because we know nothing about lower bounds on f, g.

Suppose $f(n) = \Omega(n^2)$ and $g(n) = \Omega(n^2)$.

What do we know about $g'(n) = f(n) + g(n)$? Adding the lower bounding constants shows $g'(n) = \Omega(n^2)$.

What do we know about $g''(n) = f(n) - g(n)$? We know nothing about the lower bound of this!

Problem 2.1-4:

(a) Is $2n + 1 = O(2n)$?

(b) Is $2^{2n} = O(2^n)$?

(a) Is $2n + 1 = O(2n)$?

Is $2n + 1 \leq c * 2n$?

Yes, if $c \geq 2$ for all n

(b) Is $2^{2n} = O(2^n)$?

Is $2^{2n} \leq c * 2n$?

note $2^{2n} = 2n * 2n$

Is $2n * 2n \leq c * 2n$?

Is $2n \leq c$?

No! Certainly for any constant c we can find an n such that this is not true.