CS 2134

Catalog Description:
Abstract data types and the implementation and use of standard data structures. Fundamental algorithms and the basics of algorithm analysis. Grade of C- or better required of undergraduate computer science and computer engineering majors.

Recommended Textbook:

Mark Allen Weiss, Data Structures and Algorithm Analysis in C++
Fourth Edition
Published by Pearson, 2014
ISBN-13: 9780132847377
ISBN-10: 013284737X

Source code from the textbook: http://users.cis.fiu.edu/~weiss/dsaa_c++4/code/

homework assignments	12%
quizzes (based on homework assignment topics)	8%
extra credit question (you must explain your code to me)	1%
4 exams (20% each)	80%
recitation participation, and other evidence of engagement (e.g. answering questions on Piazza, …)	1%

Course Work and Grading

Although the homework makes up a relatively small percentage of the final grade, it is a key component to mastering the course material. Experience has shown that you will not do well on the exams if you have not done the homework.

Attendance at exams is mandatory. Make-up exams will only be given in the case of a emergency, such as illness, which must be documented, e.g. with a doctor's note. In such cases, you **must** notify me as early as possible, preferably **before** the exam is given. If you miss an exam without a valid excuse, you will receive a grade of zero for that exam.

Homework assignments, announcements, and the occasional helpful hint will be posted on MyPoly. You are responsible for being aware of any information posted there, so you should check it regularly.

Policy on Collaboration

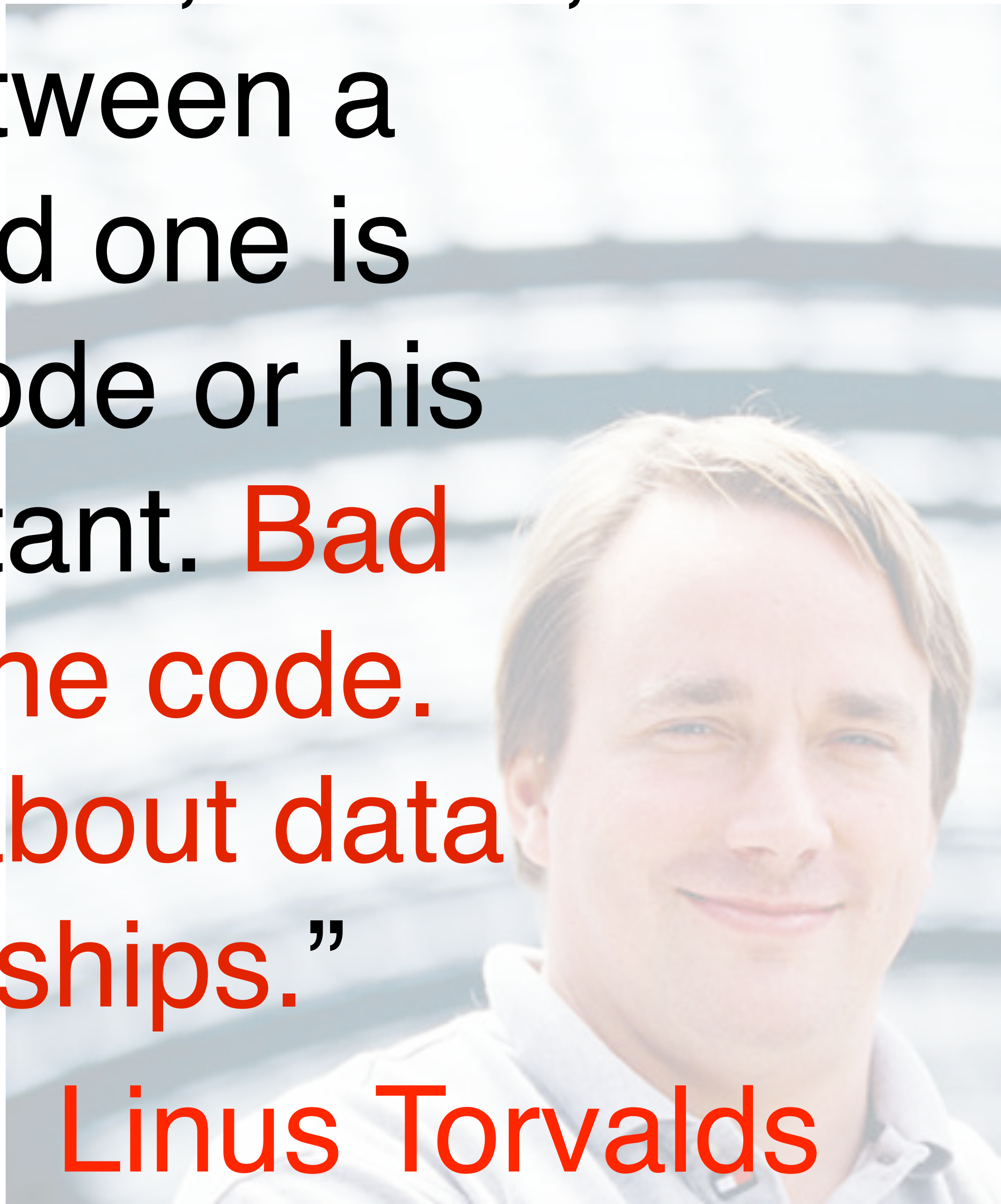
Cheating will not be tolerated. Absolutely no communication with other students is permitted on quizzes or exams.

You are encouraged to discuss general concepts, key concepts, review class notes, draw pictures, etc. with other members of the class. However, you must *write up the solutions alone*, and *write your programs on your own*. No copying. No cut and paste. No listening while someone dictates a solution. No looking at someone else's solution. No writing down a group answer, etc. All work turned in must be written in your own words. (See http://engineering.nyu.edu/academics/code-of-conduct/academic-misconduct) If you discuss the material with other students, you **must** fully understand the work you submit. If you are not sure whether you are crossing the line between general discussion and inappropriate collaboration, please ask me. **If you discuss the material with anyone else, you must list all collaborators.**

If you allow your work to be copied, you are cheating.

Cheating may not only result in a zero grade for the assignment/quiz/exam and the CS department being informed, potentially you will receive a zero for the entire course, and possible additional actions at my discretion including involving the CS department and the administration.

“In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful […] I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”



Linus Torvalds

Algorithms needed for:

queue

- Simulations: waiting times for customers at a bank, traffic analysis

vector

- Sorting: checks by check number, subroutine in graphical programs, phonebook, student records, ...

hash table

- Compilers: symbol table, parsing, implementing function calls

linked list

priority queue

- Graph Searching: shortest driving distance, social relations, fewest connections in a network, cheapest airline route, robot navigation

tree

- Word Processing: text compression, undo command, postscript printing, spell checker

stack

- Internet: web searching, back button on Web browser, marking visited pages
- ...

We won't discuss most of these applications

Goals of Course

- Concept of Data Abstraction
- Toolbox of useful abstract data types and their implementations (data structures)
- Toolbox of useful algorithms
- Basics of analyzing efficiency of algorithms
- Some more C++
- More advanced programming techniques
  - more on recursion, dynamic data structures, STL, ...

Concept of Data Abstraction

– Classes in Object Oriented languages group data (member variables) with operations to manipulate the data (member functions)
– Abstract Data Types
    - Abstract description of the operations provided and the relationships among them
    - Different implementations are possible for the same ADT
    - Separation of concerns between data type implementation and use
– OO languages developed to support ADTs

Toolbox of fundamental data structures

- vectors
- lists
- stacks
- queues
- sets and maps
- binary trees
- priority queues
- graphs

Toolbox of fundamental algorithms

–sorting and searching
–parsing and evaluation of expressions
–graph algorithms

Words With Friends

aardvark
aardvarks
aardwolf
aardwolves
aargh
aarrgh
aarrghh
aas
aasvogel
aasvogels
ab
aba
abaca
abacas
abaci
aback
abacterial
abacus
abacuses
abaft
abaka
abakas
abalone
abalones
abamp
abampere
abamperes
abamps
abandon
abandoned
abandoner
abandoners
abandoning
abandonment
abandonments
abandons
abapical
abas
abase
abased
abasedly
abasement
abasements
abaser
abasers
abases
abash
abashed
abashes
abashing
abashment
abashments
abasia
abasias

Suppose we want to know if

- nopar is allowed

http://www.merriam-webster.com/dictionary/nonpar

Definition of NONPAR

: being a bank that has not agreed to pay all checks drawn on it at par and so cannot join the par clearance system of the Federal Reserve system

- rebode is allowed

How do we estimate the number of steps an algorithm takes?

Linear Search

```
int index = -1;
for( i = 0; i < n; i++ )
   if( item == a[i] )
      {
            index = i;
            break;
      }
```

a

aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
aardvarks
aardwolf
aardwolves
aargh
aarrgh
aarrghh
aas
aasvogel
aasvogels
ab
aba
abaca
abacas
abaci
aback

How we will count

Each +, -, *, /, =, … takes 1 time step

A subroutine takes 1 time step for the call
plus the time for the subroutine to run

We have as much memory as we need.

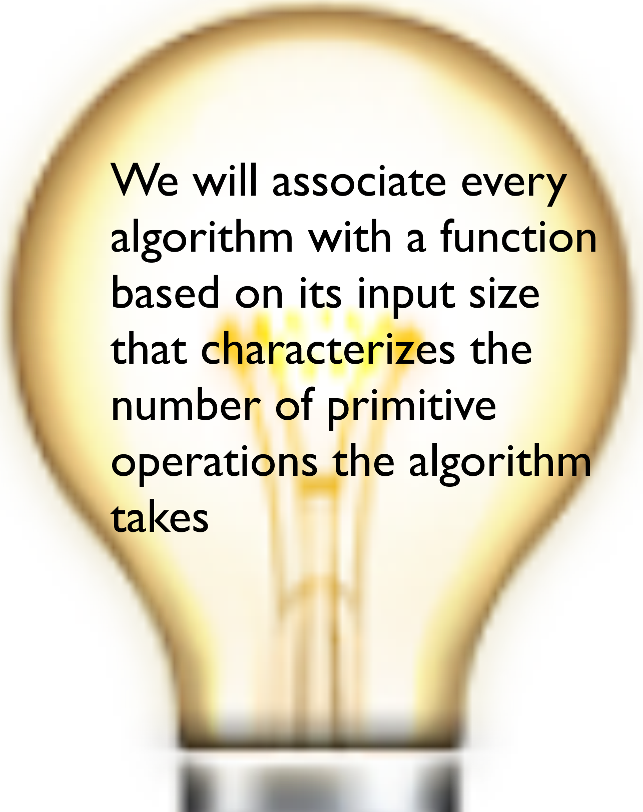
If a word isn't found, how many steps are performed by the code snippet? Let n be the number of words stored in a vector, a.

Uniform cost model - every operation we assume has a constant cost.

Linear Search

```
int index = -1;
for( i = 0; i < n; i++ )
   if( item == a[i] )
      {
         index = i;
         break;
      }
```

	n=50	n=100	n=200	n=400
	1	1	1	1
	1, 51, 50	1, 101, 100	1, 201, 200	1, 401, 400
	50	100	200	400
Total	153	303	[illegible]	1203



We will associate every algorithm with a function based on its input size that characterizes the number of primitive operations the algorithm takes

Ideally, every operation we count would correspond to a single hardware instruction, but in reality, some of the operations we are counting would correspond to a small number of operations, but the additional accuracy would not give more insight.

Could we have looked for the words using a different algorithm?

Two ways to find an item in a sorted vector

Linear Search

```
int index = -1;
for( i = 0; i < n; i++ )
   if( item == a[i] )
       {
            index = i;
            break;
       }
```

Binary Search

```
index = -1;
int high = n;
int low = 0;
while (high >= low)
{
 int mid = (high + low)/2;
 if( item == a[mid])
 {
        index = mid;
        break;
 }
 if ( a[mid] > item)
        high = mid - 1;
       else
       low = mid + 1;
}
```

Which is a more effective algorithm?

- Correctness?
- Time?
- Space?

Factors affecting Running Time of a Program

- size of input, n
- particular input of a given size
  - worst input
  - "average" input
  - best input (not usually interesting)
- details of environment: processor speed, number of registers, access time for memory, … We usually ignore these when analyzing the *algorithm*

Particular Input

- Find position of first occurrence of x in array a:

```
for (i=0; i<n; i++)
  if (a[i]==x) break;
```

- Worst case: x in last slot or not present
- “Average” case: x in middle
  - more precisely find average number of steps over all possible positions of x
  - Doing this carefully depends on input distribution

How many steps are performed by the code snippet?

Uniform Cost Model
Only counting operations (not actual machine code, or the difference in time to compute the different instructions)

```
for (i=0;i<n;i++)
   for(j=0; j<n; j++)
      sum += 1;
```

	n=50	n=100	n=200
	1 + 51 + 50	1 + 101 + 100	1 + 201 + 200
	50(1 + 51 + 50)	100(1 + 101 + 100)	200(1 + 201 + 200)
	50 * 50	100 * 100	200 * 200
Total	7702	30402	120802

This is a lot of work. We will talk about a simpler way to see how the run time changes when the size of the input changes. I won't ask you to do this level of detail.
I want you to have an intuitive understanding

This was too much work!!!

How do we (roughly and somewhat quickly) estimate the running time?

A rough estimate (off by a small multiplicative constant) of the running time of a loop is usually the running time of the statements inside the loop (including tests) times the number of iterations.

Rough upper bound on # of instructions?

```
for (i=0; i<n; i++)
    sum += 1;
```

# steps $1 + (n+1) + n + n$

$< 4*n$ for $n > 2$

```
for (i=0;i<n; i++)
   for(j=0; j<n; j++)
      sum += 1;
```

# steps
$1 + (n+1) + n$
$+ n(1+ (n+1) + n)$
$+ n*n$

$< 4*n*n$ for $n > 10$

Not tight! What could be a tighter number?

Rough upper bound on # of instructions?

```
for (i=0; i < n; i++)
  for(j=0; j < n; j++){
      sum += 1;
      cout << sum;
  }
```

# steps
1 + (n+1) + n
+ n (1+ (n+1) + n)
+ n*n
+ n*n
< 5*n*n
for n > 10

```
for (i=0; i<n; i++)
  for(j=i; j<i+4; j++)
      sum += 1;
```

# steps
1 + (n+1) + n
+ n(1+ (4+1) + 4)
+ n*4
< 17*n
for n > 10

How many items are added to sum?

```
for (i=1; i<n; i*=2)
   sum += a[i];
```

n	#
10	4
100	7
1000	10
10000	14

Rough upper bound on # of instructions?

# steps is roughly 1 + (log(n)+1) + log(n) + log(n)

< 4*log(n) for n > 2

How many items are added to sum?

```
for (i=0;i<n;i++)
   for(j=i; j<n; j++)
       sum += a[i]*a[j]
```

n	#
10	
100	
1000	
10000	

(inner loop starting at j=i cuts number of steps by about a factor of two)

How many items are added to sum?

```
for (i=0;i<n;i++)
  for(j=i; j<n; j++)
     sum += 1
```

If n = 10 then

i : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

times 1 is added 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

If n = 100 then

i : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., 97, 98, 99

times 1 is added 100, 99, 98, 97, 96 ,95, 94 ,93 ,92 , 91, ... , 3 , 2 , 1

For a general n

i : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., n-3, n-2, n-1

times 1 is added n n-1 n-2 , ... , 3 , 2 , 1

Gauss' Series Summation

$$1 + 2 + 3 + 4 + 5 = (1 + 2 + 3 + 4 + 5 + 5 + 4 + 3 + 2 + 1)/2$$
$$= (6 + 6 + 6 + 6 + 6)/2$$
$$= 5\,(5+1)/2$$

$$1 + 2 + 3 + \cdots + n-1 + n =$$
$$(1 + 2 + 3 + \cdots + n-1 + n + n + n-1 + n-2 + \cdots + 2 + 1)/2 = n\,(n+1)/2$$

How many items are added to sum?

```
for (i=0;i<n;i++)
   for(j=i; j<n; j++)
      sum += 1
```

n	#
10	55
100	5050
1000	500500
10000	50005000

(inner loop starting at j=i cuts number of steps by about a factor of two)

Rough upper bound on # of instructions?

# steps roughly 1 + (n+1) + n
n + [n(n+1)/2 + n] + n(n+1)/2
+ n(n+1)/2
< c*n*n for n > ? and c = ?

What is the value of k after this code is executed?

```
k=0;
j=0;
  while(j<n)
  {
    for (i=0; i<n*n;i++)
    {
       k++;  ←
    }
    j = j+2;
  }
```

n	k
10	500
100	500000
1000	500000000
10000	500000000000

If n is even, the outer loop is executed n/2 times,

The inner loop is executed n^2 times

So, the line of code "`k++;`" is executed $n^3/2$ times

Rough upper bound on # of instructions?

Suppose we have two algorithms, one runs in time $10n^2$ and the other in time $\frac{n^3}{6}$.

Which is faster?

Is $10n^2 < \frac{n^3}{6}$?

…

Only if $n > 60$

Suppose we have two algorithms, one runs in time $6n+14$ and the other in time $\frac{n^2}{2}$.

Which is faster?

Is $6n + 14 < \frac{n^2}{2}$?

…

Only if $n > 14$

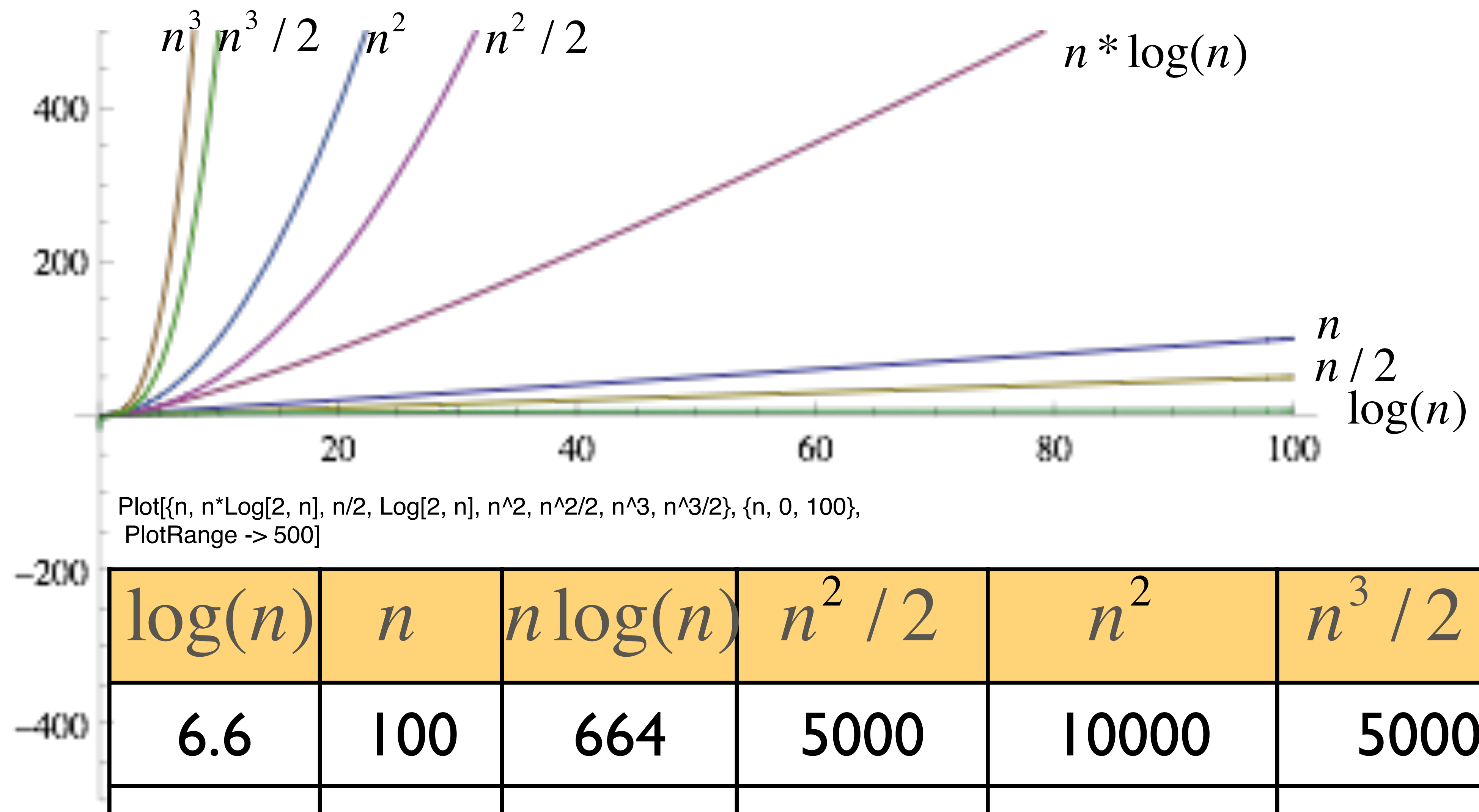
Suppose we have two algorithms; one runs in time $n^2 + 4n$ and the other in time $\frac{n^3}{2}$.

Which is faster?

Is $n^2 + 4n < \frac{n^3}{2}$?

...

Only if $n > 4$



```
Plot[{n, n*Log[2, n], n/2, Log[2, n], n^2, n^2/2, n^3, n^3/2}, {n, 0, 100},
 PlotRange -> 500]
```

$\log(n)$	n	$n\log(n)$	$n^2/2$	n^2	$n^3/2$	n^3
6.6	100	664	5000	10000	50000	1000000
7.6	200	1529	20000	40000	4000000	8000000
8.2	300	2469	45000	90000	13500000	27000000
8.6	400	3458	80000	160000	32000000	64000000
...	...	...	...	...	...	...
9.96	1000	9966	500000	1000000	500000000	1000000000

Computational Complexity

“A mathematical characterization of the difficulty of a mathematical problem which describes the resources required by a computing machine to solve the problem...”
from http://www.yourdictionary.com/computational-complexity

Algorithmic Analysis

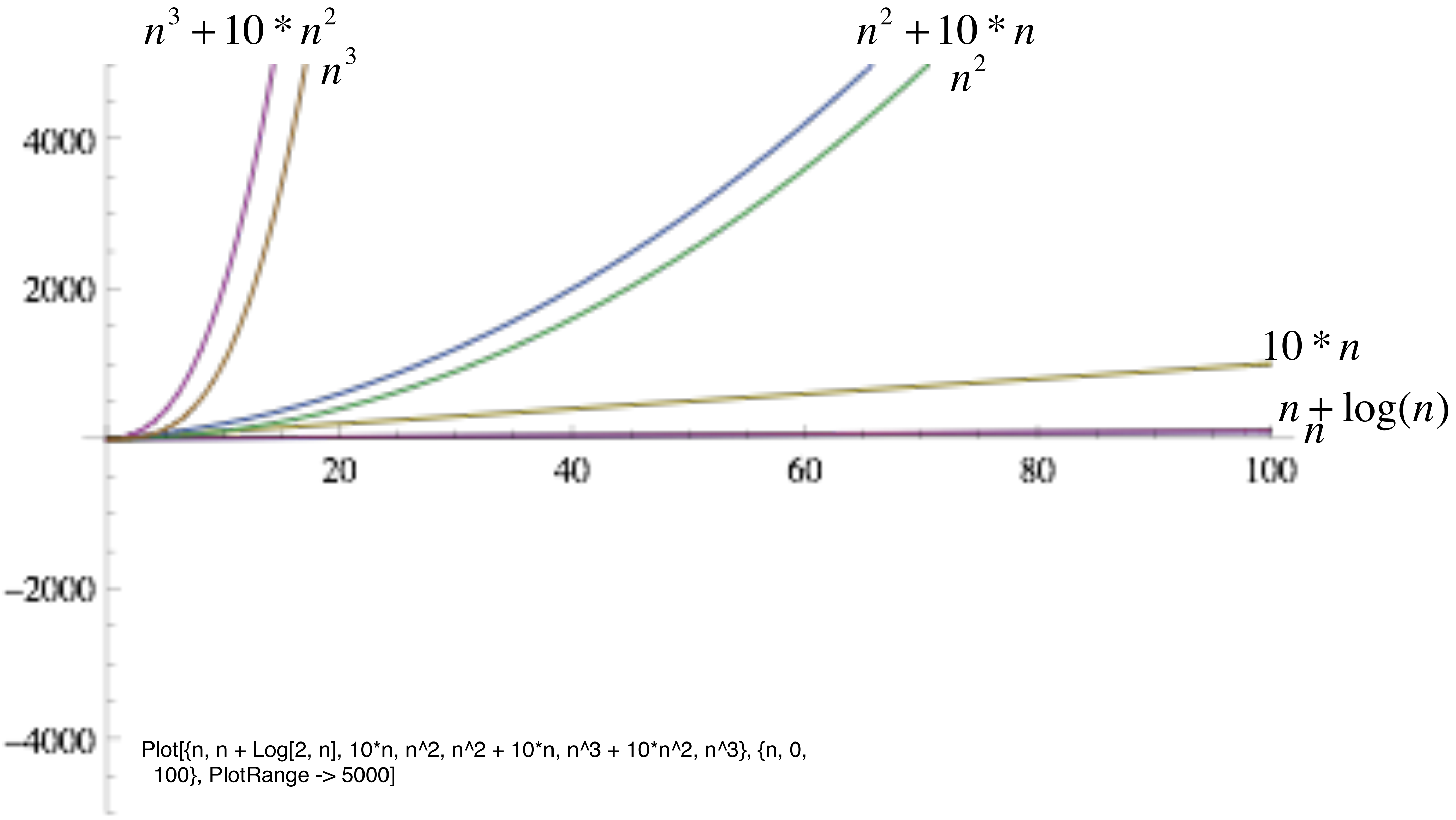
“In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, Big-omega notation and Big-theta notation are used to this end. For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in O(log(n)), colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.”
from https://en.wikipedia.org/wiki/Analysis_of_algorithms

Asymptotic Complexity

A ballpark estimate.

Intro to Big-Oh

- Consider running time vs input size
- We're interested in general behavior, not exact details
  - ignore lower order terms
  - ignore multiplicative constant
- Example: $f(n)=1 + ... + n = n(n+1)/2$ is $O(n^2)$
- graph has "same shape"



```
Plot[{n, n + Log[2, n], 10*n, n^2, n^2 + 10*n, n^3 + 10*n^2, n^3}, {n, 0,
  100}, PlotRange -> 5000]
```

Big-Oh notation, O(g(n))
(gives the "big" picture)

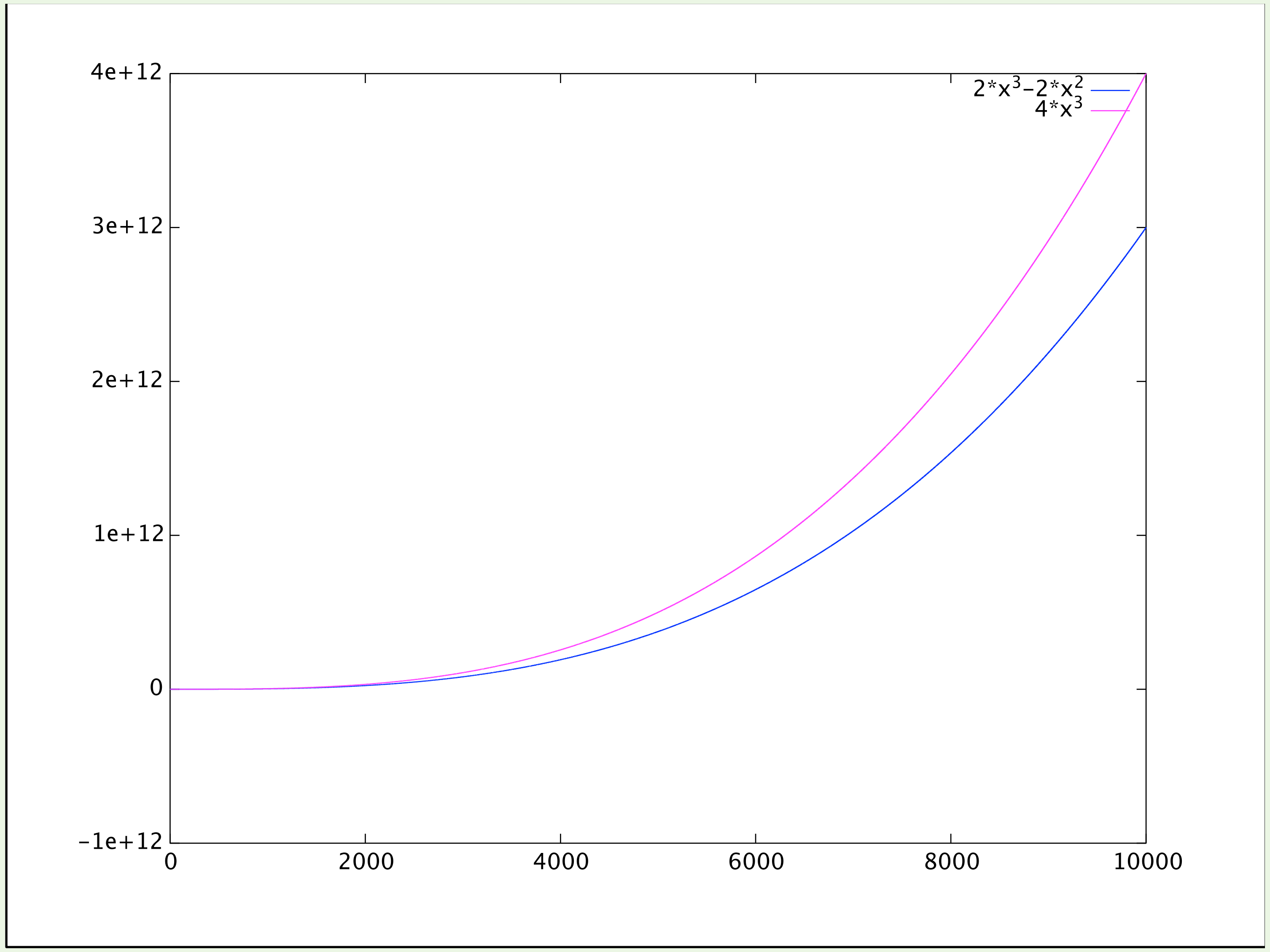
- c g(n) is an upper bound on growth rate, for some constant c
- platform-independent,
  not affected by choice of computer, compiler, operating system, etc.
- g(n) is written as simply as possible where the constants and low order term are omitted
- if we say a program runs is O(g(n)) time, we mean it takes at most c*g(n) time
  - O(n) - running time is some constant times n
  - $O(n^2)$ - running time is some constant times n^2
  - …

Growth of Functions

This is about "large" input

if n is large,

$$2n^3 - 4n^2 < 4n^3 \qquad 2n^3 - 4n^2 = O\left(n^3\right)$$



Some examples

$$n^2 + \cancel{5} = O(n^2)$$

$$\cancel{10}n + \cancel{7}n + \cancel{10} = O(n)$$

$$n(n - \cancel{1})(n - \cancel{2}) + \cancel{7}n + \cancel{21} = O(n^3)$$

$$n^3 - \cancel{3}n + \cancel{2} + \cancel{7}n = O(n^3)$$

$$\cancel{10}n^2 + (n - \cancel{1})(n - \cancel{2}) + \cancel{7}n + \cancel{99} = O(n^2)$$

$$\frac{\cancel{(3}n + \cancel{4})(\cancel{4n}/\cancel{2} + n^2)\cancel{11}}{9\cancel{0}00} = O(n^3)$$

Formal Definition

Definition: (Big-Oh) $T(n)$ is $O(g(n))$ if there are positive c and n_0 such that $T(n) <= cg(n)$ when $n >= n_0$.

... while $3n$ is $O(\cancel{2}n)$ do not write this!

$3n$ is $O(n)$

$3n + 10n^2$ is $O(\cancel{n+n^2})$ do not write this!

$3n + 10n^2$ is $O(n^2)$

$6n + 20$ is $O(n^{\cancel{2}})$ do not write this!

$6n + 20$ is $O(n)$

$3n + 10n^2$ is $O(n^2)$
Why?
$3n + 10n^2 <= 11n^2$ when $n >= 3$

$6n + 20$ is $O(n)$
Why?
$6n + 20 <= 10n$ when $n >= 5$

n*log(n) is NOT O(n)
Why?
n*log(n) > c*n when c < log(n)
and 2^c < n

Common code fragments and their O()'s

```
for (i=0; i <=n; i++)
      simple-stmt
```

$O(n)$

```
for (i=0; i <n; i++)
   for (j=0; j<n; j++)
      simple-stmt
```

$O(n^2)$

```
for (i=0; i <n; i++)
   for (j=i+1; j<n; j++)
      simple-stmt
```

$O(n^2)$

```
for (i=0; i <n; i++)
     simple-stmt1;
for (j=0; j<n; j++)
      simple-stmt2
```

$O(n)$

```
for (i=0; i <n; i++)
    for (j=i+1; j<n; j++)
         for (k=j+1;k<n;k++)
                simple-stmt
```

$O(n^3)$

```
for (i=1; i <=n; i*=2)
        simple-stmt
```

$O(\log n)$

```
for (i=n; i >=1; i=i/2)
        simple-stmt
```

$O(\log n)$

```
for (i=0; i <n^2; i++)
    simple-stmt1;
for (j=0; j<n; j++)
     simple-stmt2
```

$O(n^2)$

```
for (i=0; i <n^2; i++)
   for (j=i+1; j<n^2; j++)
        for (k=j+1;k<n^2;k++)
             simple-stmt
```

$O(n^6)$

Some Important Big-Oh's

• $O(1)$	constant	assignment statement invoking a function
• $O(\log n)$	logarithmic	binary search inserting into a red-black tree
• $O(n)$	linear	searching in an unordered list inserting a new item into a sorted vector
• $O(n \log n)$	linearithmic	mergesort
• $O(n^2)$	quadratic	insertion sort two embedded loops
• $O(n^3)$	cubic	maximum subsequence sum three embedded loops
• $O(2^n)$	exponential	finding all subsets of a set of size n. Exponential is often used in a more generic sense
• $O(n!)$	factorial	all permutations of a string of size n

Big Theta, θ

- More precise statements can be made using Big Theta, rather than Big-Oh
- f(n) is Big-Theta(g(n)) if f is O(g) and g is O(f)
  - $n^2/2 + n/2$ is Theta(n^2)
  - $n^2/2 + n/2$ is O(n^3) True....but we want the more precise O(n^2)
  - $n^2/2 + n/2$ is NOT Theta(n^3)
- For some reason, data structures text books often use big-Oh when they could use big-Theta

Examples of Running Times

- Find element k of array A: $O(1)$
- Find kth element in a list: $O(n)$
- Find smallest element in array: $O(n)$
- Find smallest element in sorted array: $O(1)$
- Insertion sort: $O(n^2)$
- Merge sort: $O(n \log n)$

Comments

- For the same problem (e.g. sorting) there are different algorithms, some of which are substantially better than others
- Selection of right data structure and/or preprocessing may allow substantial improvement in running time (but may have a one-time preprocessing cost).