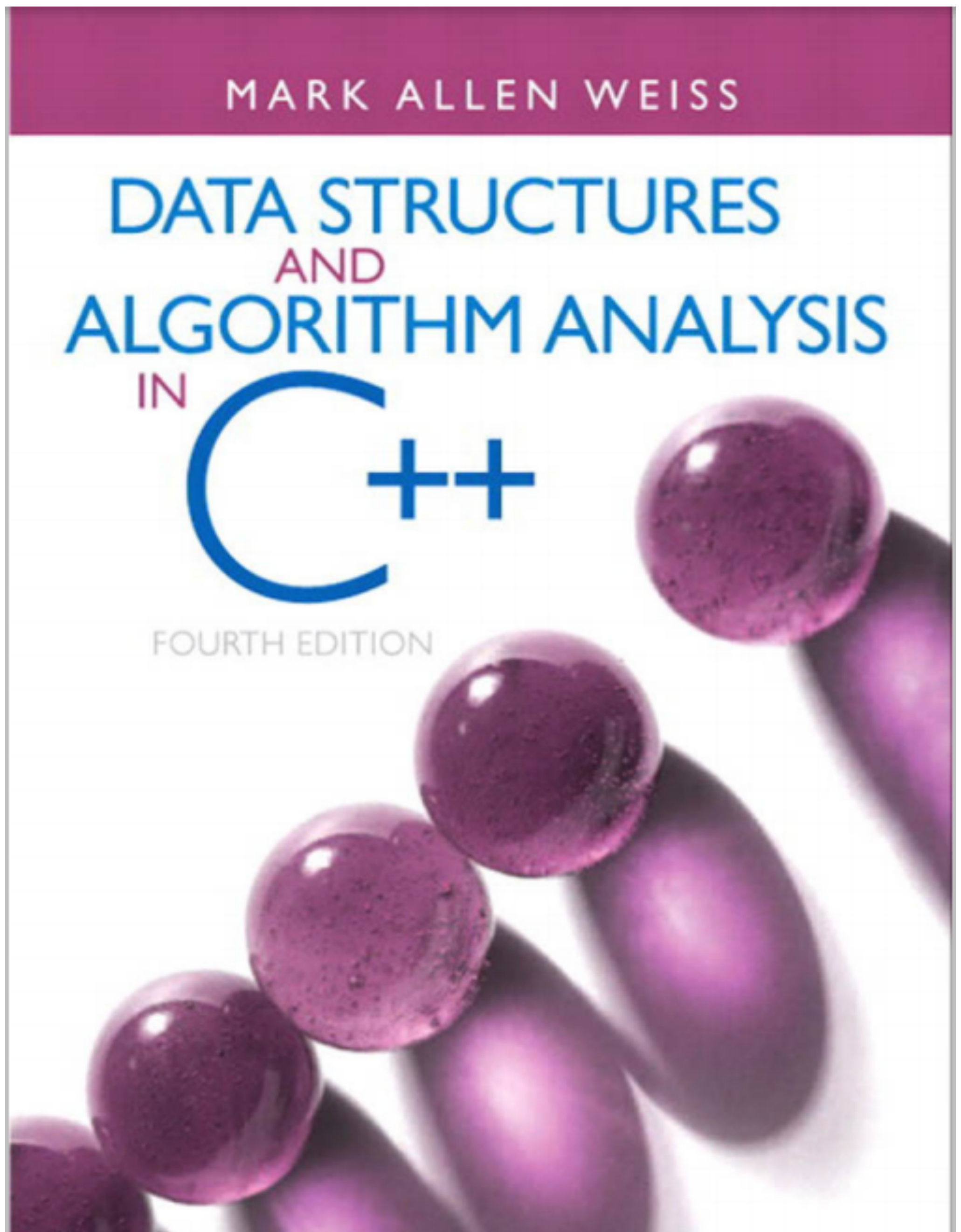
CS 2134

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/



APPROXIMATE SCHEDULE

Lecture #	Topic	Chapter or other resources
1	Algorithmic Analysis	2
2	"	"
3	C++	1.1-1.2.4, 1.4-1.6
4	C++	"
5	The STL	3.3-3.4
6	STL cont.	
7	Recursion	1.3
8	Sorting	7.1-7.3, 7.6
9	Exam 1	
10	Sorting cont.	7.7
15	Linked Lists	3.1-3.3, 3.5
16	Linked Lists	"
11	Stacks & Queues	3.6-3.7
12	Compilers	
13	Compilers	3.6.3
14	Exam 2	
17	Trees + Binary Search Trees	4.1-4.3
18	Binary Search Trees cont.	"
19	Red-Black Trees	12.2
20	Hash Tables	5.1-5.4, 5.6
21	Exam 3	
22	Hash Tables Cont.	"
23	Graphs	9.1, 9.3.1
24	Priority Queues	6.1-6.3
25	Graphs cont.	9.3.2
26	TBD	
	Exam 4	

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.

I want you to pass the course AND for you to know the material

If you do poorly (i.e. your score is less than 70) on exam 1, 2 or 3, you may schedule a time within one week of receiving your exam grade to demonstrate to me that you understand the material on the exam. If you succeed in doing this and, you have done well on the quizzes, and have done well on the homework assignments, and have not violated the cheating policy then you will safeguard your final grade from falling below a **C-** for this course. Your score for the exam will not be changed. (A grade of C or higher is entirely contingent on your actual final average.)

If you are on the border between two grades, more weight will be given to your exams and quizzes than your homework assignments.

Policy on Collaboration

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

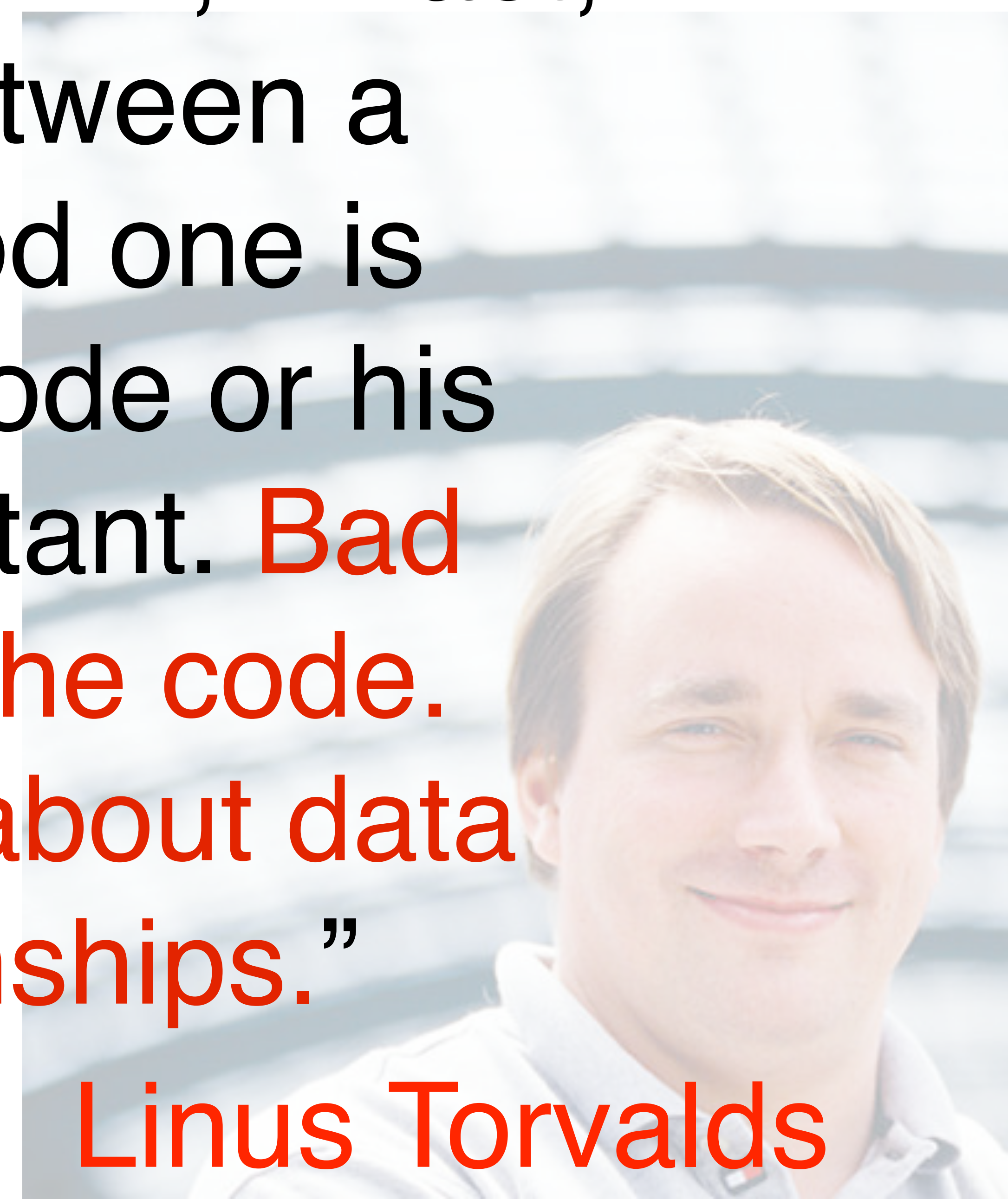
You are encouraged to discuss general concepts, key concepts, review class notes, draw pictures, etc. with other members of the class. However, you must *write up the solutions alone*, and *write your programs on your own*. No copying. No cut and paste. No listening while someone dictates a solution. No looking at someone else's solution. No discussing assignments in such detail that you duplicate part of someone solution in your answer. 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 allow your work to be copied, you are cheating. If you show someone else your answer, you are cheating.

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

Of course you may show your work to others., after ***all*** homework assignments have been submitted, (i.e. No abuse of the late policy.)

“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

Data Structures used in:

We won't discuss most of these applications

- 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
- ...

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

How could we compare algorithms?

algorithm |ˈalgəˌriT͟Həm|
noun

“a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer…”

Want a machine independent way of comparing algorithms…

Words With Friends

aals
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
abasing

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.

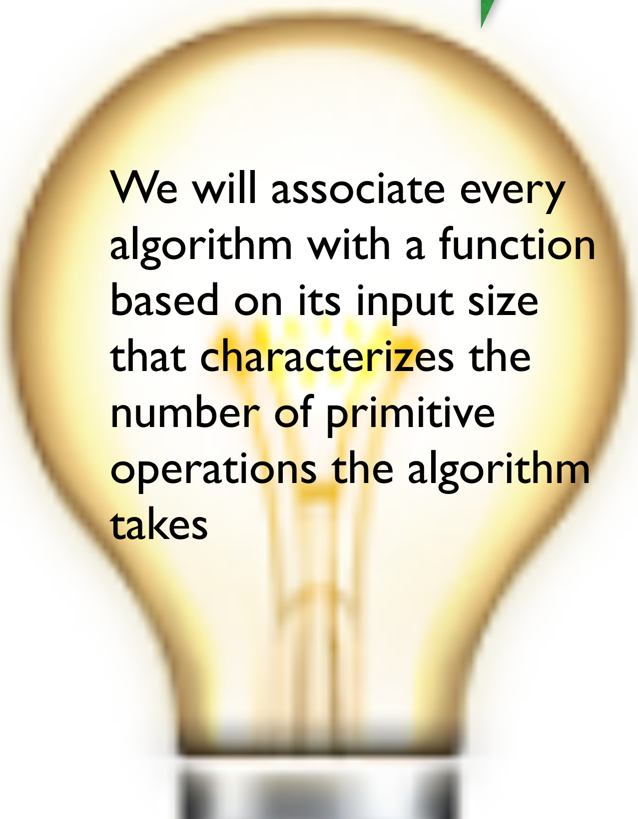
Uniform cost model - every operation we assume has a constant cost.

Linear Search

```
int in
for( i
   if( i
      {
         index = i;
         break;
      }
```

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
Want a function that takes as input the size of the input is proportional to the running time of the algorithm

	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



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?

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

```
for (i=0;i<n;i++)
   for(
      s
```

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

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

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 $n > 2$

$\log(n) = \log_2(n)$ in this class

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 $\dfrac{n^2}{2}$.

Which is faster?

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

...

Only if $n > 14$

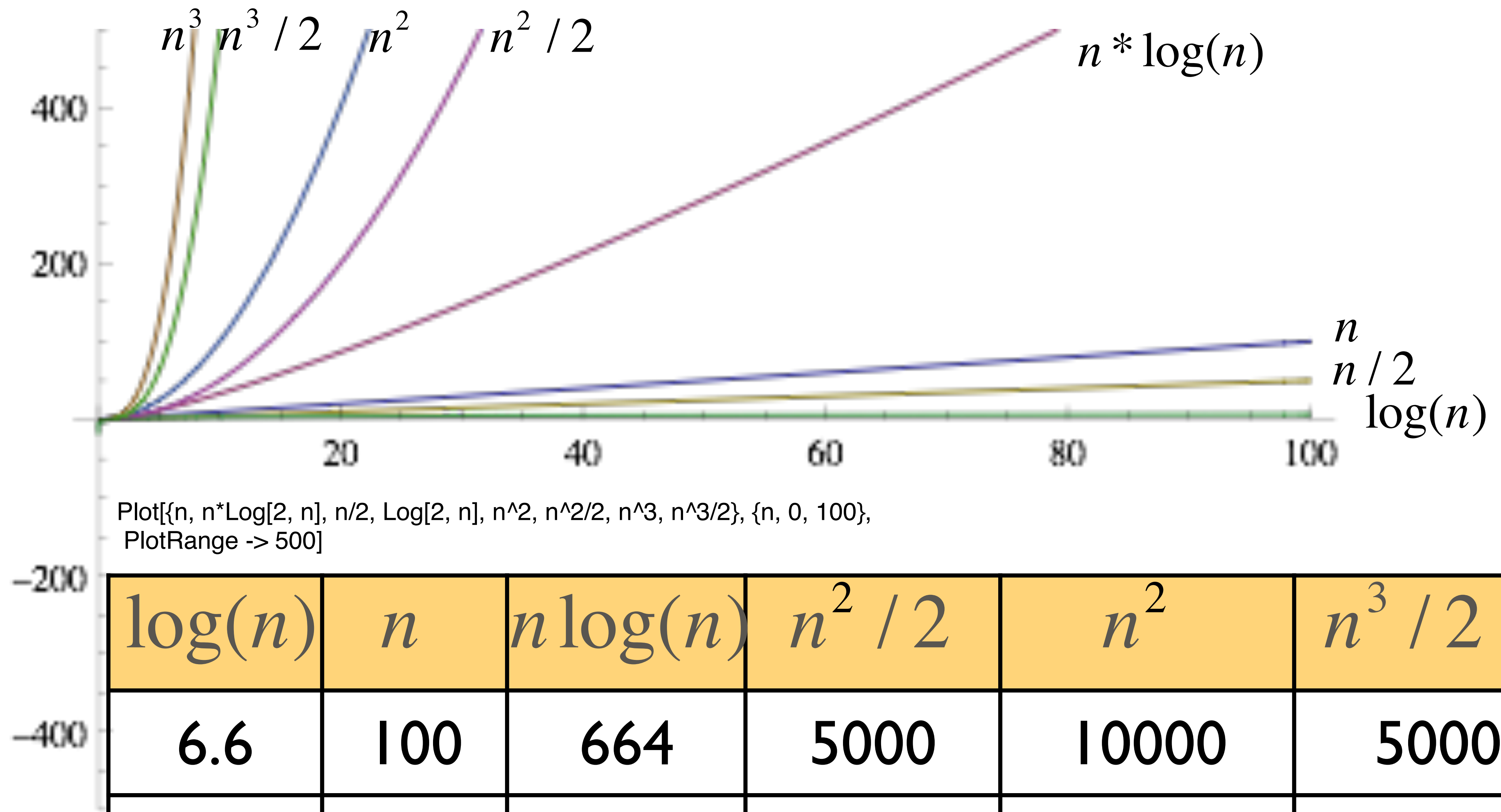
Suppose we have two algorithms; one runs in time $n^2 + 4n$ and the other in time $\frac{n^3}{2}$.

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

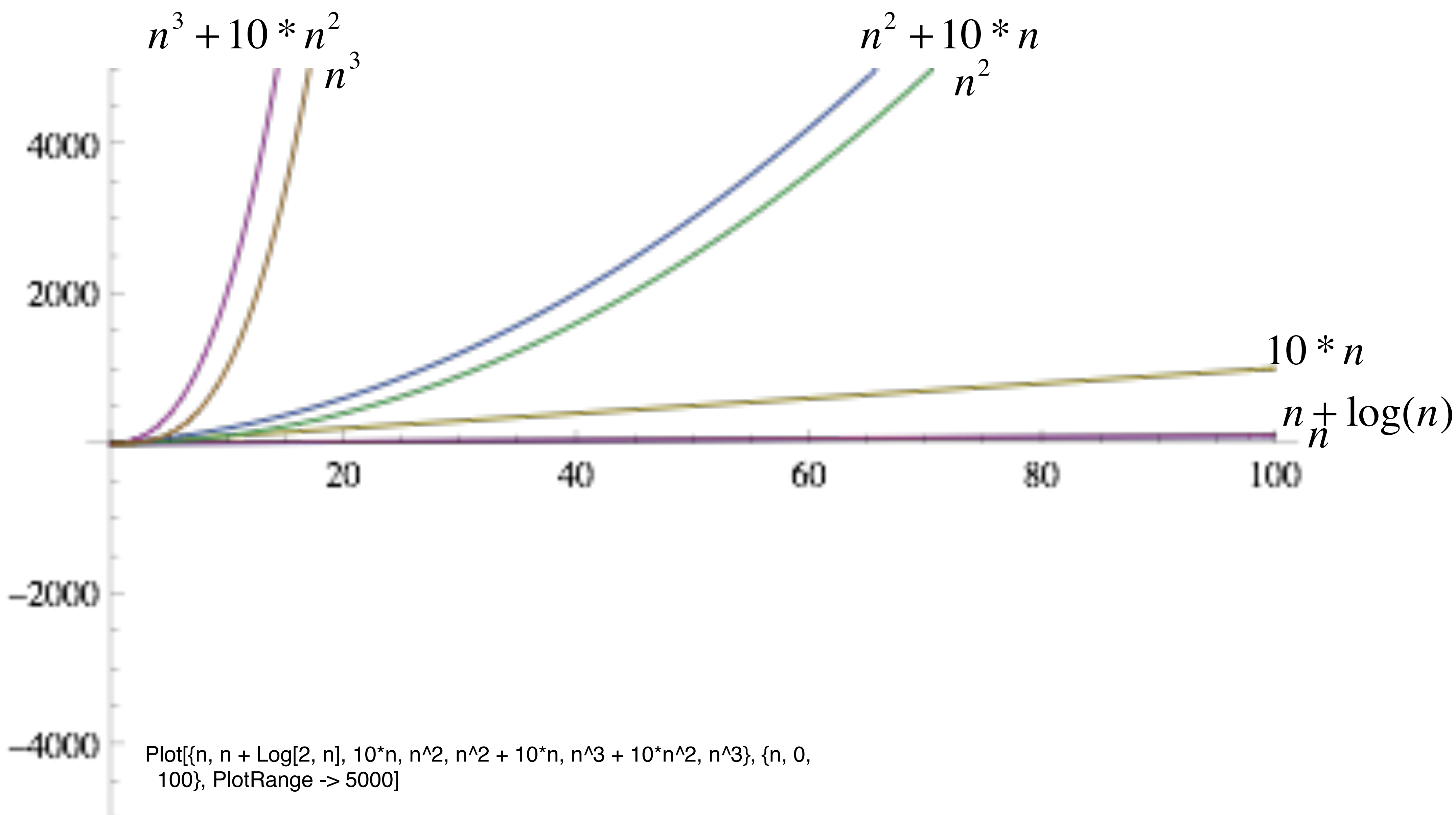
"In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, Big-omega notation and Big-theta notation are used to this end. For instance, binary search is said to run in 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: $f(n)=1 + \ldots + n = n(n+1)/2$ is $O(n^2)$
- graph has "same shape"



n^3 + 10 * n^2
n^3
n^2 + 10 * n
n^2
10 * n
n + log(n)
n
4000
2000
-2000
-4000
20
40
60
80
100
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]

Formal Definition

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

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

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

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
  - …

Some examples

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

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

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

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

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

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

To prove $T(n)$ is $O(g(n))$ we state witnesses c and n_0 that prove $T(n) <= c*g(n)$ if $n >= n_0$ using algebraic manipulation.

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

Why? Let $c = 11$ and $n_0 = 3$

If $n >= 3$ then $3n + 10n^2 <= 11n^2$
iff $3n <= n^2$ iff $3 <= n$

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

Why? Let $c = 10$ and $n_0 = 5$

If $n >= 5$ then $6n + 20 <= 10n$
iff $20 <= 4n$ iff $5 <= n$

$n*\log(n)$ 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

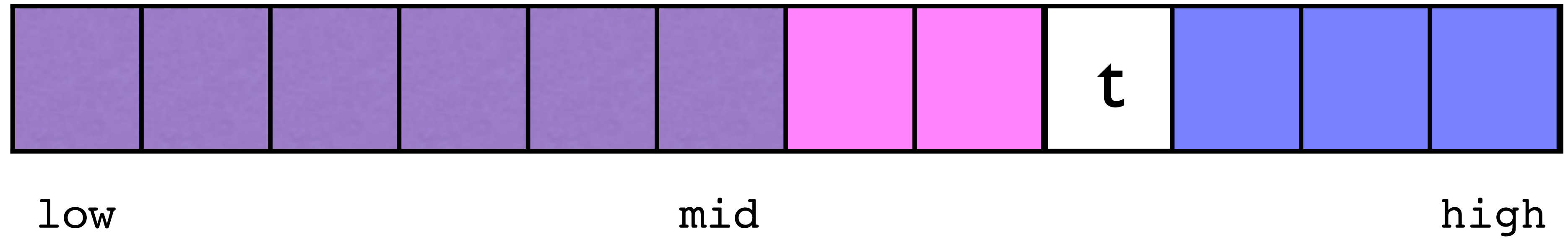
Comments

- For the same problem (e.g. sorting, searching) 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).

Static Searching (data doesn’t change)

If key x is in array a, return its position; else indicate that it’s not there.

- Recall Sequential search: O(n)
- If array is *sorted*, we can search much more efficiently by using Binary Search (our version is using 2-way comparisons)
- Analysis: Repeated Halving : O(log n)



Binary Search for x = s

```
template <class Comparable>
int binarySearch( const vector<Comparable> & a, const Comparable & x )
{
    int low = 0;
    int high = a.size( ) - 1;
    int mid;

    while( low < high )
    {
        mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else
            high = mid;
    }
    return ( low == high && a[ low ] == x ) ? low : NOT_FOUND;//NOT_FOUND = -1
}
```

Worst Case Running Time?

After the first comparison how many items remain? $n/2$

After the second comparison how many items remain? $(n/2)/2 = n/2^2$

After the third comparison how many items remain? $(n/2^2)/2 = n/2^3$

...

After the k^{th} comparison how many items remain? $n/2^k$

How many comparison till one item remains? $\lceil \log(n) \rceil$

Total number of comparisons? $\lceil \log(n) \rceil + 1$

Using Big-Oh notation: $O(\log(n))$

$\log(n) = \log_2(n)$ in this class

Worst Case for the number of comparisons of elements

# names	sequential search	Binary SearchB
512	512	9+1
1024	1024	10+1
2048	2048	11+1
4196	4196	12+1
8192	8192	13+1
16384	16384	14+1
32768	32768	15+1
65536	65536	16+1
...	...	...
16,777,216	16,777,216	24+1
...	...	...
268,435,456	268,435,456	28+1

Useful Facts about Logs

- $\log_b(a^c) = c \log_b a$
- $\log_b(a) = \log_c a / \log_c b$
- $\log_b n$ is $O(\log n)$

Notice, to change the base of the log you multiply by a constant!

- Log n grows very slowly:

n	log n
1	0
2	1
4	2
8	3
16	4
...	
1024	10
2^{20} (about 1 million)	20

The Base of the logarithm doesn't matter!

Big-Oh

$\log_4(n)$ is $O(\log(n))$

$\log_4(n^4)$ is $O(\log(n))$

$(\log_2(n))^3$ is $O(\log^3(n))$

Why?

Definition: (Big-Oh) $T(n)$ is $O(F(n))$ if there are positive c and n_0 such that $T(n) \leq cF(n)$ when $n \geq n_0$.

- Exponential function grows *very fast*; logarithmic function grows *very slowly*
- So programs with exponential running time run *very slowly*; programs with logarithmic running time run *very quickly*.
- Precise analysis usually involves floors or ceilings, but when we go to Big-Oh, it doesn’t really matter.

Repeated Halving and Doubling

- `for (i = n; i > 1; i = i/2)`   // log n iterations

- `for (i =1; i < n; i = i*2)`   // log n iterations

Algorithms based on repeated halving (or doubling) are generally very efficient

Estimating Running Times

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

n	steps	O(log(n))
64	6	
128	7	
256	8	
512	9	

```
for (i = 1; i < n; i = 4*i)
    sum += a[i]
```

n	steps	O(log(n))
64	3	
128	4	
256	4	
512	5	

```
for (i = n; i > 1; i /= 4)
    sum += a[i]
```

n	steps	O(log(n))
64	3	
128	4	
256	4	
512	5	

Bits in a Binary Number!

16-bit short integer represents the integers from
-32,768 to 32,767

65,636 integers (i.e. $2^{16} = 65,636$)

If you needed to store an integer in the range from 0 to 232222 how many bits would you need?

$\log(232223) = 17.8$ bits doesn't make sense!

$\lceil 17.8 \rceil = 18$ bits

Floors and Ceilings:

$$\lceil \log(2147483654) \rceil = \lceil 31.00000000403084 \rceil = 32$$

$$\lfloor 4.9 \rfloor = 4$$

$$\lfloor 4.01 \rfloor = 4$$

$$\lceil 4.9 \rceil = 5$$

$$\lceil 4.01 \rceil = 5$$

$$\lfloor 4 \rfloor = 4$$

$$\lceil 4 \rceil = 4$$

Estimating Run times!

If an item isn't found, how many operations are performed by the code snippet?

<u>Uniform cost model</u> - assume every simple statement 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	603	1203
	153/50	303/100	603/200	1203/400

as n gets large this ratio approaches 3

$T(n) = 2 + (n + 1) + n + n = 3n + 3$

$T(2n) = 2 + (2n + 1) + 2n + 2n = 3(2n) + 3$

For a linear time algorithm when we double the input size, the number of steps the algorithm takes is roughly twice as many c_1n+c_2 ☛ $c_1(2n) + c_2 \sim c_1(2n)$

For a linear time algorithm for and input size of n', the number of steps the algorithm takes is roughly $r = n'/n$ times as many c_1n+c_2 ☛ $c_1(n') + c_2 \sim c_1(rn)$

$T(n)/n = (c_1n+c_2)/n \sim c_1$

Counting the number of steps

<u>Uniform Cost Model</u>
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 += a[i]*a[j]
```

	n=50	n=100	n=200
	1 + 51 + 50	1 + 101 + 100	1 + 201 + 200
	50(1 + 51 + 50 + 50)	100(1 + 101 + 100 + 100)	200(1 + 201 + 200 + 200)
Total	7702	30402	120802
	7702/2500 = 3.0808	30402/10000 = 3.0402	120802/40000 = 3.02005

The number of steps for this code snippet:

$\mathbf{T(n) = 1 + (n + 1) + n + n(1 + (n + 1) + n) + n * n = 2 + 4n + 3n^2}$

doubling the size of n: $\mathbf{T(2n) = 2 + 4(2n) + 3(2n)^2 = 2 + 8n + 12n^2 \sim 4T(n)}$

for an input size n' where r = n'/n:

$$\mathbf{T(n') = 2 + 4(n') + 3(n')^2 = 2 + 4rn + 3r^2n^2 \sim r^2T(n)}$$

The number of steps for an algorithm which take $O(n^2)$ time:

$\mathbf{T(n) = c_1n^2 + c_2n + c_3}$

The largest term dominates as n gets large

The number of steps on an input size n' where r = n'/n:

$$\mathbf{T(n') = c_3 + c_2(n') + c_1(n')^2 = c_3 + c_2rn + c_1r^2n^2 \sim r^2T(n)}$$

$\mathbf{T(n)/n^2 = (c_1n^2 + c_2n + c_3)/n^2 \sim c_1}$

Estimating Run Times

T(n), the running time of the program
- Can represent the number of steps
- Can represent the time it takes a computer to execute the code
- ...

If the algorithm is O(n^2),
and it takes 0.0073 seconds when n = 2^7
How long should it take when n = 2^8?

Hmm ...

If $T(n) = c_1n^2 + c_2n + c_3$ is $O(n^2)$

$T(n) \sim c_1n^2$ when n is large

$T(2n) \sim c_1(2n)^2 = 2^2c_1n^2$

If $T(2^7) = 0.0073$

We approx $T(2^8) \sim 4*T(2^7) = 4*0.0073 = 0.0292$

Estimating Run Times

If the algorithm is $O(n^3)$,
and it takes 0.0073 seconds when $n = 2^7$
How long should it take when $n = 2^8$?

Hmm ...

If $T(n) = c_1n^3 + c_2n^2 + c_3n^2 + c_4$ is $O(n^3)$

$T(n) \sim c_1n^3$ when n is large

$T(2n) \sim c_1(2n)^3 = 2^3c_1n^3$

If $T(2^7) = 0.0073$

We approx $T(2^8) \sim 8 * T(2^7) = 8 * 0.0073 = 0.0584$

Estimating the Actual Running Time

- Program P has $O(n^k)$ running time and runs for time t_0 on input of size n_0
- How long will program take on input of size n_1?
- Compute $r = n_1/n_0$
- Program will take approximately $r^k * t_0$ on input of size n_1

Note: We get better estimates when we start with a larger number.

Estimating Run Times

For an $O(n^3)$ time algorithm where it takes 0.0073 seconds when $n = 2^7$. When $n = 2^8$ we estimated $T(2^8)$ is roughly 0.0584.

$T(2^9)$ is roughly 0.4672

$T(2^{10})$ is roughly 3.7376

$T(2^{11})$ is roughly 29.9008

$T(2^{12})$ is roughly 239.2064

	n^3
2^7	0.0073
2^8	0.05642
2^9	0.440588
2^{10}	3.50385
2^{11}	28.0077
2^{12}	233.944

$T(2^{18})=T(262144)$ is roughly $8589934592 * 0.0073 = 62706522.5216$

2.49234181522182 years

Estimating Run Times

If the algorithm is O(n^2),
and it takes 0.00024 seconds when n = 2^7
When n = 2^8 we estimated T(2^8) is roughly 0.00096.

T(2^9) is roughly 0.00384

T(2^{10}) is roughly 0.01536

T(2^{11}) is roughly 0.06144

T(2^{12}) is roughly 0.24576

T(2^{18})=T(262,144) is roughly
4194304* 0.00024 = 1006.63296

	n^2
2^7	0.00024
2^8	0.00091
2^9	0.00358
2^{10}	0.014356
2^{11}	0.058269
2^{12}	0.232687

Checking we have the correct asymptotic function!

$T(n^3)/n^3$

$T(2^7)/(2^7)^3 = 0.0073/2097152=0.0000000034809I$

$T(2^8)/(2^8)^3 = 0.05642/16777216=0.00000000336289$

$T(2^9)/(2^9)^3 = 0.440588/134217728=0.00000000032826$

$T(2^{10})/(2^{10})^3 = 3.50385/1073741824=0.000000003263$

What if we thought it was O(n^2)?

What if we thought it was O(n^4)?

	n^3
2^7	0.0073
2^8	0.05642
2^9	0.440588
2^{10}	3.50385
2^{11}	28.0077
2^{12}	233.944

$T(n^2)/n^2$

$T(2^7)/(2^7)^2 =$ 0.00024/16384=0.0000000146484

$T(2^8)/(2^8)^2 =$ 0.00091/65536=0.0000000138855

$T(2^9)/(2^9)^2 =$ 0.00358/262144=0.0000000136566

$T(2^{10})/(2^{10})^2 =$ 0.014356/1048576=0.0000000136944

	n^2
2^7	0.00024
2^8	0.00091
2^9	0.00358
2^{10}	0.014356
2^{11}	0.058269
2^{12}	0.232687

Running Times.... Which is O(n), O(n²),O(n³)?

	$O(n^3)$	$O(n^2)$	$O(n)$
n=10	0.000009	0.000004	0.000003
100	0.002580	0.000109	0.000006
1000	2.281013	0.010203	0.000031
10000	NA	1.2329	0.000317
100000	NA	135	0.003206

Max Contiguous Subsequence Problem

One problem
3 solutions!

* Ok, I really should have said we will discuss 3 of the many solutions possible to solve this problem.

Max Contiguous Subsequence Problem

- Given sequence A1,...,An of numbers find i and j such that Ai + ... + Aj is maximal.
- 1, 2, -4, 1, 2, -1, 4, -2, 1
- Max subsequence is 1,2,-1,4 whose sum is 6

When in doubt, use brute force.

-Ken Thompson, Bell Labs

Max Contiguous Subsequence Problem

{-2, 11, -4, 13}

	{}	0
	{-2}	-2
	{-2, 11}	9
	{-2, 11, -4}	5
	{-2, 11, -4, 13}	18
	{11}	11
	{11, -4}	7
Maximum	{11, -4, 13}	20
	{-4}	-4
	{-4, 13}	9
	{13}	13

Max Contiguous Subsequence Problem

Definition:

- Given sequence A1,...,An of numbers find i and j such that Ai + ... + Aj is maximal.
- A1, A2, A3, A4, A5, A6, A7, A8, A9
- 1, 2, -4, 1, 2, -1, 4, -2, 1
- Max subsequence is 1,2,-1,4 whose sum is 6

How did we know? Computed all subsequences...

Sum(1,9)=1+2-4+1+2-1+4-2+1=4
Sum(1,8)=1+2-4+1+2-1+4-2=3
Sum(1,7)=1+2-4+1+2-1+4=5
⋮
Sum(2,9)=1+2-4+1+2-1+4-2+1=3
⋮
Sum(5,7)=2-1+4=5

Till we tried everything!

Naive Algorithm

For each pair i, j compute the sum of the elements from i to j,
Keeping track of maxsofar.
For each pair (i,j):

```
    thissum =0;
    for (k=i; k <=j; k++) thissum += a[k];
       ...
       if (thissum > maxsofar) maxsofar = thissum;
```

1 2 -4 1 2 -1 4 -2 2
e.g., when (i,j) = (1,4), sum = 2 - 4 +1 +2 = 1

Triply nested loop: $O(n^3)$

```
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
        for( int j = i; j < a.size( ); ++j )
        {
            int thisSum = 0;

            for( int k = i; k <= j; ++k )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum   = thisSum;
        }

    return maxSum;
}
```

For each pair i, j
compute the sum of the elements from i to j,
keeping track of the largest one you found, maxsofar.

a	1	2	-4	2	-1	4	-2	2	-5	9	12	-3	7	11

First Algorithm

```
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
      for( int j = i; j < a.size( ); ++j )
      {
          int thisSum = 0;

          for( int k = i; k <= j; ++k )
              thisSum += a[ k ];

          if( thisSum > maxSum )
              maxSum   = thisSum;
      }

    return maxSum;
}
```

$O(1)$

$O(n)$

$O(1)$

$O(n^2)$

$O(n^3)$

Triply nested loop: O(n³)

$$O\left(n^3\right) \text{ time !}$$

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 \text{ iterations}$$

$$\#\,\text{ordered triplets } (i, j, k) \qquad 1 \le i \le k \le j \le n$$

$$n(n+1)(n+2)/6 \qquad = (n^3 + 3n^2 + 2)/6$$

$$(n^3 + \cancel{3n^2} + \cancel{2})/\cancel{6} \qquad = O(n^3)$$

Is there another way?

1, 2, -4, 1, 2, -1, 4, -2, 1

How can we do this faster? Notice! Sum(i,j+1)=Sum(i,j)+A(j+1)!

Sum(1,1)=1

Sum(1,2)=1+2=3

Sum(1,3)=1+2-4=-1

Sum(1,2)=Sum(1,1)+2=3

Sum(1,3)=Sum(1,2)-4=-1

⋮

Sum(3,3)=-4

Sum(3,4)=Sum(3,3)+1=-4+1=-3

Sum(3,5)=Sum(3,4)+2=-3+2=-1

⋮

Sum(3,9)=Sum(3,8)+1=0+1=1

Faster Algorithm

- Using Sum(i, j+1) = Sum(i,j) + A[j+1]; remember Sum(i,j) instead of recomputing it.
- Doubly nested for loop: $O(n^2)$

Max Contiguous Subsequence Problem

{-2, 11, -4, 13}

	{}	0
	{-2}	-2
	{-2, 11}	(-2)+11=9
	{-2, 11, -4}	(9)-4= 5
	{-2, 11, -4, 13}	(5)+13=18
	{11}	11
	{11, -4}	(11)-4=7
Maximum	{11, -4, 13}	(7)+13=20
	{-4}	-4
	{-4, 13}	(-4)+13=9
	{13}	13

Algorithm

```
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

For each pair i,j
compute the sum of the elements from i to j,
keeping track of the sum(i,j-1) and the largest o
you found, maxsofar.

a	1	2	-4	2	-1	4	-2	2	-5	9	12	-3	7	11

i

Algorithm $O\left(n^2\right)$ time !

```
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

$O(1)$

$O(1)$

$O(1)$

$O\left(n^2\right)$

$O\left(n^2\right)$ time !

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 \text{ iterations}$$

$$n(n+1)/2 = (n^2+n)/2$$

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

Any other improvements?

- 1, 2, -4, 1, 2, -1, 4, -2, 1

Should we ever check a subsequence starting with A3?

The moral

Any substring that has a maximal value does not contain a prefix which is negative!

and any positive prefix is better than no prefix!

Largest Sum Substring Algorithm

1, 2, -4, 1, 2, -1, 4, -2, 1

	Thissum=0	Maxsum=0
1	Thissum=1	Maxsum=1
1+2=3	Thissum=3	Maxsum=3
3-4=-1	Thissum=0	Maxsum=3
0+1=1	Thissum=1	Maxsum=3
1+2=3	Thissum=3	Maxsum=3
3-1=2	Thissum=2	Maxsum=3
2+4=6	Thissum=6	Maxsum=6
6-2=4	Thissum=4	Maxsum=6
4+1=5	Thissum=5	Maxsum=6

```cpp
template <class Comparable>
Comparable maxSubsequenceSum4( const vector<Comparable> & a )
{
    int n = a.size( );
    Comparable thisSum = 0;
    Comparable maxSum = 0;

    for( int i = 0, j = 0; j < n; j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
        {
            maxSum = thisSum;
        }
        else if( thisSum < 0 )
        {
            i = j + 1;
            thisSum = 0;
        }
    }
    return maxSum;
}
```

Algorithm

For every j
keeping track of the largest subsequence ending at j-1, and keep track of the largest subsequence (with no restrictions), maxsofar.

a	1	2	-4	2	-1	4	-2	2	-5	9	12	-3	7	11

Algorithm $O(n)$ time !

```
template <class Comparable>
Comparable maxSubsequenceSum4( const vector<Comparable> & a )
{
    int n = a.size( );
    Comparable thisSum = 0;
    Comparable maxSum = 0;

    for( int i = 0, j = 0; j < n; j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
        {
            maxSum = thisSum;
        }
        else if( thisSum < 0 )
        {
            i = j + 1;
            thisSum = 0;
        }
    }
    return maxSum;
}
```

$O(1)$ $O(n)$

$$O(n) \text{ time !}$$

$$\sum_{j=1}^{n} 1 \text{ iterations}$$

$$1 \le j \le n$$

$$n = O(n)$$