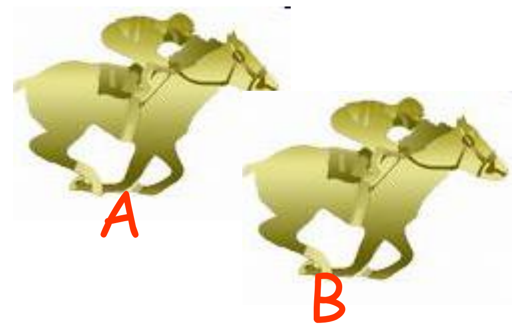
Lecture #10

- Big-O aka "How Fast Is that Algorithm?"
- Sorting Algorithms, part I

How Fast is That Algorithm?

Sometimes we want to ask "How fast is that algorithm?"

Or: "Which algorithm is faster, A or B?"



Question: How can you measure the "speed" of an algorithm?

How Fast is That Algorithm?

Right! We could measure the time it takes for an algorithm to run.

But that has flaws! What are they?

Carey: "My algorithm finished in 31 seconds."

Cedric: "Mine finished in 30 seconds, it's better!"

Carey: "Not so fast! How fast is your PC? Mine is 1GHZ."

Cedric: "Err… Mine is 3GHZ."

Carey: "Aha – so my algorithm is really almost 3x faster!"

Cedric: "Sigh. Carey's right again."

How Fast is That Algorithm?

Ok, so simply measuring the run-time of an algorithm isn't really all that useful.

What if instead we measure an algorithm based on:

how many computer instructions it takes to solve a problem of a given size

Carey: "My algorithm took 370 million instructions to sort 1,000 numbers."

Cedric: "Dude – you SUCK! Mine only took only 5 million instructions, it's better!"

Carey: "Not so fast grasshopper! Mine might be slower on 1,000 numbers, but what if we sort 1 million numbers?"

Cedric: "Hmm. I don't know – I haven't tried."

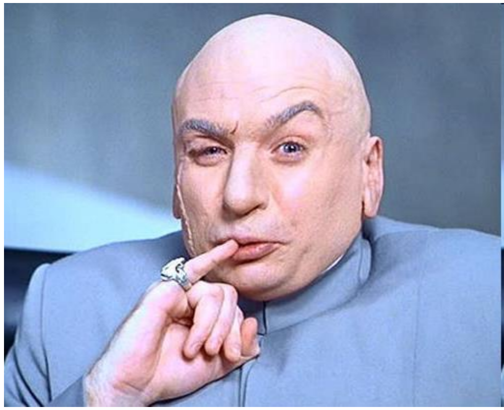
How Fast is That Algorithm?

So just rating an algorithm based on how many steps it takes on a particular set of data doesn't tell us much.

An algorithm might look efficient when applied to a small amount of data (e.g., 1,000 numbers)

But really "blow chunks" when applied to a lot of data (e.g. 1 billion numbers)

We'd like to understand how our algorithm performs under all circumstances!



How Fast is That Algorithm?

Hmmm. What else could we do?

Right! What if we specify

the number of instructions used by an algorithm as a function of the size of the input data.

"I'm trying to sort N numbers."
"Algorithm A takes $5 \cdot N^2$ instructions to do that."
"Algorithm B takes $37{,}000 \cdot N$ instructions to do that."

Now we can predict which algorithm will be faster for any value of N!

How Fast is That Algorithm?

"I'm trying to sort N numbers."
"Algorithm A takes $5 \cdot N^2$ instructions to do that."
"Algorithm B takes $37{,}000 \cdot N$ instructions to do that."

Ok, what if we're sorting 1,000 numbers:

"Algorithm A takes 5M instructions." 

"Algorithm B takes 37M instructions."

Ok, what if we're sorting 10,000 numbers:

"Algorithm A takes 500M instructions."

"Algorithm B takes 370M instructions." 

Ok, what if we're sorting 1 million numbers:

"Algorithm A takes 5 trillion instructions."

"Algorithm B takes 37 billion instructions." 

How Fast is That Algorithm?

"I'm trying to sort N numbers."
"Algorithm A takes $5 \cdot N^2$ instructions to do that."
"Algorithm B takes $37{,}000 \cdot N$ instructions to do that."

Cool! When we measure this way, we get two benefits:

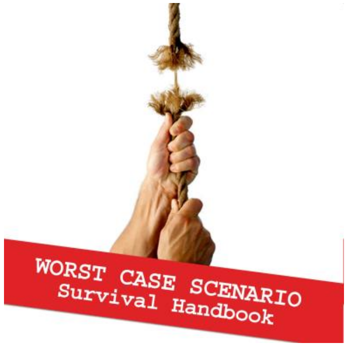
1. We can compare two algorithms for a given sized input.
2. We can predict the performance of those algorithms when they are applied to less or more data.

This is the idea behind the "Big-O" concept used in Computer Science.

No, not Oprah. Let's learn the details!

Big-O: The Concept

The Big-O approach measures an algorithm by the gross number of steps that it requires to process an input of size N in the WORST CASE scenario.



We could be specific and say:
"Algorithm X requires $5N^2+3N+20$ steps to process N items."

But with Big-O, we ignore the coefficients and lower-order terms of the expression…

So we'd say:
"The Big-O of Algorithm X is N^2."

While less specific, this still gives us an overall impression of an algorithm's worst-case efficiency.

Big-O: The Concept

Big-O Idea: Use simple functions like $\log(n)$, n, n^2, $n \log(n)$, n^3, etc. to convey how many operations an algorithm must perform to process n items in the worst case.

This is pronounced:
"oh of n squared"

"That sorting algorithm is $O(n^2)$, so to sort $n=1000$ items it requires roughly 1 million operations."

"That sorting algorithm is $O(n \cdot \log_2 n)$, so to sort $n=1000$ items requires roughly 10,000 operations."

This allows us to easily compare two different algorithms:

"Algorithm A is $O(n^2)$, which is much slower than algorithm B which is $O(n \cdot \log_2 n)$."

Big-O

So how do we compute the Big-O of a function?

First, we need to determine the number of operations an algorithm performs. Let's call this f(n).

By *operations*, we mean any of the following:

1. Accessing an item (e.g. an item in an array)
2. Evaluating a mathematical expression
3. Traversing a single link in a linked list, etc…

Let's see how to evaluate the number of operations for a simple example…

Big-O – How to Compute f(n)

```
int arr[n][n];

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

Compute f(n), the # of critical operations, that this algorithm performs?

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

$f(n) = 3n^2 + 3n + 1$

1. Our algorithm initializes the value of i once.
2. Our algorithm performs n comparisons between i and n.
3. Our algorithm increments the variable *i*, n times.
4. Our algorithm initializes the value of j, n different times.
5. Our algorithm performs n^2 comparisons between j and n.
6. Our algorithm increments the variable j, n^2 times.
7. Our algorithm sets arr[i][j]'s value n^2 times.

Now that we have f(n), we can compute our algorithm's Big-O.

Big-O – The Complete Approach

Here are the steps to compute the Big-O of an algorithm:

1. Determine how many steps $f(n)$ an algorithm requires to solve a problem, in terms of the number of items n.
2. Keep the most significant term of that function and throw away the rest. For example:
   a. $f(n) = 3n^2+3n+1$ becomes $f(n) = 3n^2$
   b. $f(n) = 2n \log(n) + 3n$ becomes $f(n) = 2n \log(n)$
3. Now remove any constant multiplier from the function:
   a. $f(n) = 3n^2$ becomes $f(n) = n^2$
   b. $f(n) = 2n \log(n)$ becomes $f(n) = n \log(n)$
4. This gives you your "big oh":
   a. $f(n) = 3n^2+3n+1$ is therefore $O(n^2)$
   b. $f(n) = 2n \log(n) + 3n$ is therefore $O(n \log(n))$

Big-O Simplification

Actually, if you think about it, there's no need to compute the exact f(n) of an algorithm…

Why? Because we end up throwing away all of the lower-order terms and coefficients anyway!

All you need to do is focus on the most frequently occurring operation(s) to save time!

```
int arr[n][n];

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

$f(n) = n^2$

Our algorithm is $O(n^2)$.

Big-O

So if I say: "This algorithm is $O(n^2)$."

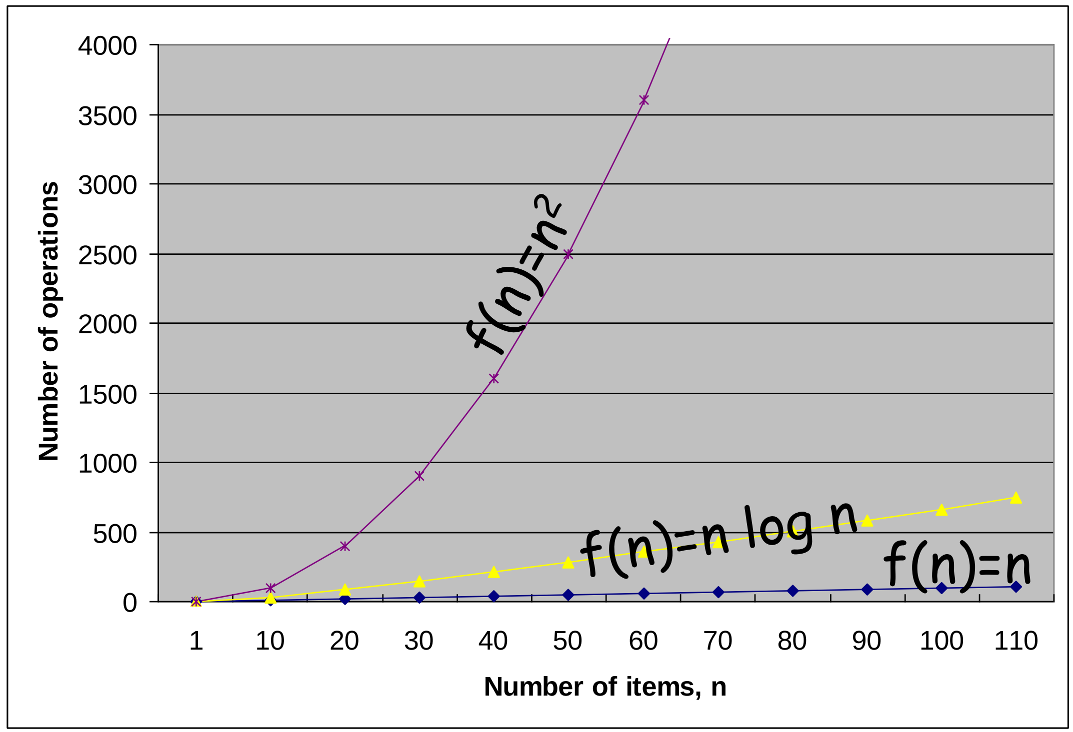
I really mean: "To process n items, this algorithm requires roughly n^2 operations."

By using only the most significant term (e.g. n^2 from $2n^2+3n+1$)

We can quickly obtain a rough approximation

of how many steps our algorithm will take

to process n items.



Big-O Complexity

$\log_2 n$	N	$n\log_2 n$	n^2	n^3	2^n
3	10	30	100	1000	1000
6	100	600	10,000	1,000,000	10^{30}
9	1,000	9,000	1,000,000	1,000,000,000	10^{301}
13	10,000	130,000	100,000,000	10^{12}	WOW!
16	100,000	1,600,000	10^{10}	10^{15}	WOW!
19	1,000,000	19,000,000	10^{12}	10^{18}	WOW!

What if you wanted to use an $O(n^3)$ algorithm to sort a million numbers? Your algorithm would require roughly 1,000,000,000,000,000,000 steps!

But an $O(n \log_2(n))$ algorithm would use only 19,000,000 steps!

Big-O Complexity

1,000,000,000,000,000,000 vs 19,000,000!

GREAT PROGRAMMERS know that the choice of algorithm makes all the difference in the world.

NOT-SO-GREAT programmers think that you can tweak a poor algorithm to make it better!

Say you improve an $O(n^3)$ algorithm from $f(n) = 5n^3$ steps to $f(n) = 1.5n^3$ steps. For n=1,000,000, that reduces the number of steps from 5,000,000,000,000,000,000 to 1,500,000,000,000,000,000.

(Big deal… so it'll take 1.5 years to run instead of 5 years)

However, if you can find an algorithm that's $O(n \cdot \log n)$ steps, say $f(n) = 10 \cdot n \log n$, you can solve the problem in 190,000,000 steps.

(Which will take just a few seconds or less on a modern PC!)

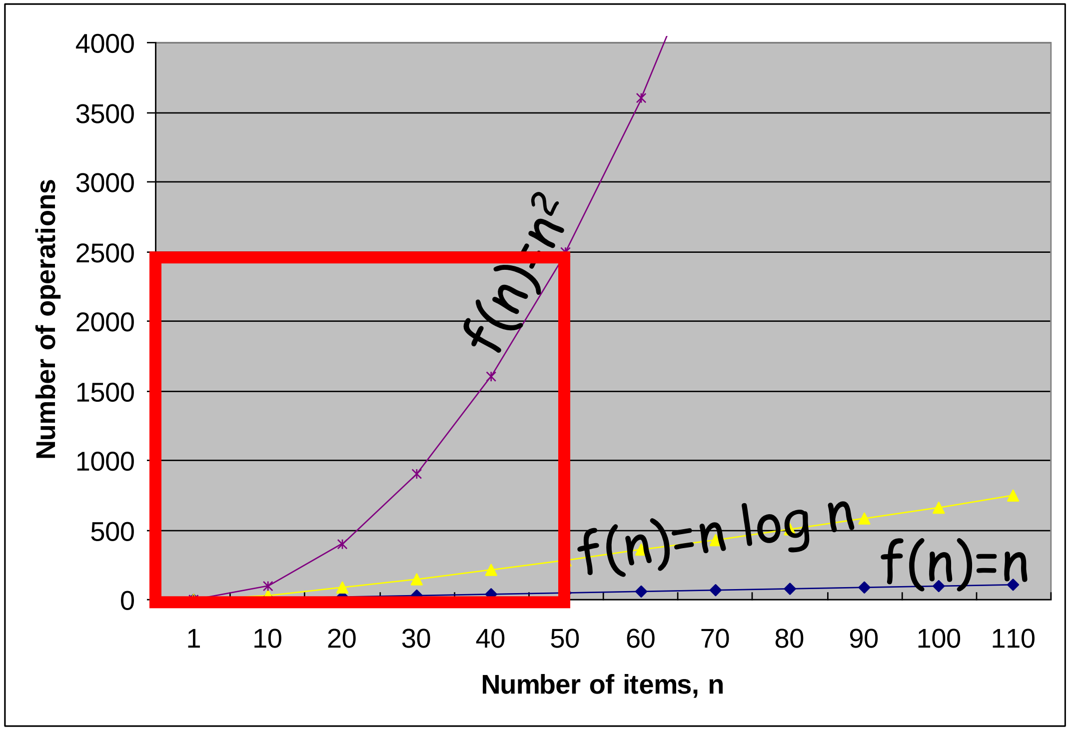
Don't be a Big-O NUT!

When you're writing a program that operates on a large number of items, evaluating Big-O is key. It can mean the difference between a usable program and an unusable one.

But what if you have a small number of items, e.g. n<50?

In this case, all of your algorithms require only a small number of steps.

In such situations (when you know n is small), forget which Big-O is better and choose the easiest-to-program algorithm. It'll save you lots of headaches.



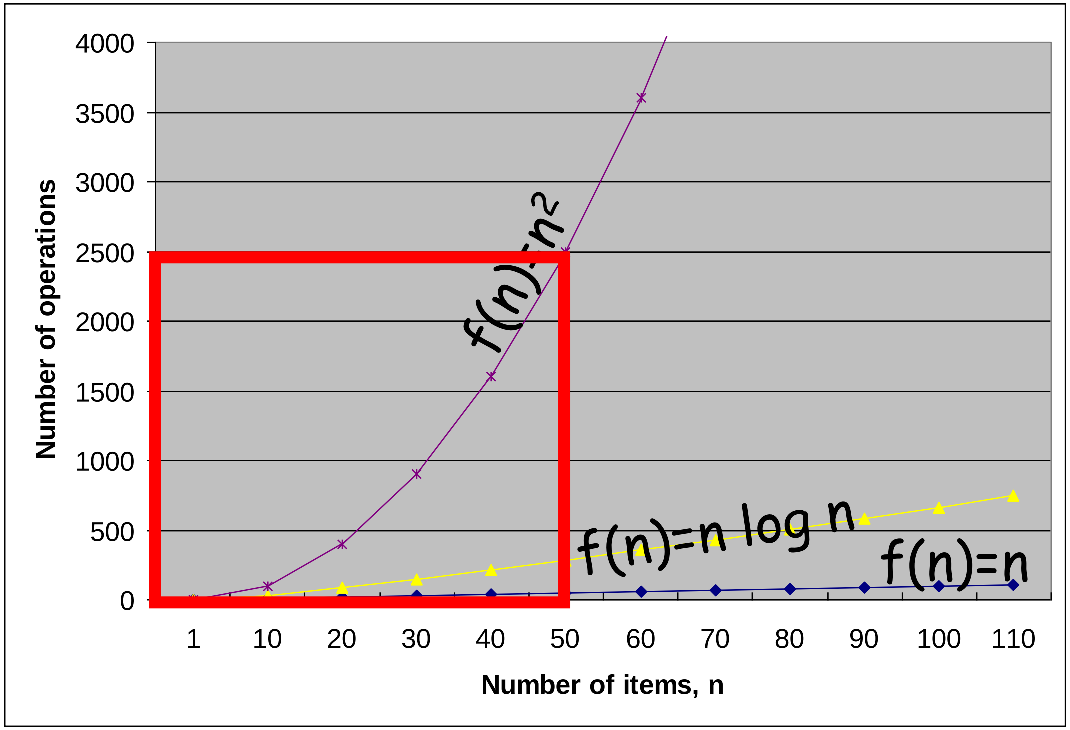
Don't be a Big-O NUT!

When you're writing a program that operates on a large number of items, evaluating Big-O is key. It can mean the difference between a usable program and an unusable one.

But what if you have a small number of items, e.g. n<50?

In this case, all of your algorithms require only a small number of steps.

In such situations (when you know n is small), forget which Big-O is better and choose the easiest-to-program algorithm. It'll save you lots of headaches.



Find the Big-O Challenge, Part 1

```
for ( int i = 0; i < n; i+=2 )
   sum++;

for ( int i = 0; i < q; i++ )
   for ( int j = 0; j < q; j++ )
      sum++;

for ( int i = 0; i < n; i++ )
    for ( int j = 0; j < n*n; j++ )
        sum++;

 k = n;
 while (k > 1)
 {
     sum++;
     k = k/2;
 }
```

```
for (int i=0 ; i < n ; i++)
{
    int k = n;
    while (k > 1)
    {
        sum++;
        k = k/2;
    }
}
```

```
void foo( )
{
    int i, sum = 0;

    for (i=0 ; i < n*n ; i++)
        sum += i;

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

Find the Big-O Challenge, Part 1

(Solutions in the slide notes below)

```
for ( int i = 0; i < n; i+=2 )
   sum++;

for ( int i = 0; i < q; i++ )
   for ( int j = 0; j < q; j++ )
      sum++;

for ( int i = 0; i < n; i++ )
    for ( int j = 0; j < n*n; j++ )
        sum++;

 k = n;
 while (k > 1)
 {
     sum++;
     k = k/2;
 }
```

```
for (int i=0 ; i < n ; i++)
{
    int k = n;
    while (k > 1)
    {
        sum++;
        k = k/2;
    }
}

void foo( )
{
    int i, sum = 0;

    for (i=0 ; i < n*n ; i++)
        sum += i;

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

Find the Big-O Challenge, Part 1a

```
int searchArray(int arr[], int n, int forValue)
{
    for ( int i = 0; i < n; i++ )
    {
        if (arr[i] == forValue)
            return i;
    }

    return -1;  // not found
}

void addItemToEndOfArray(int arr[], int &n, int addMe)
{
   arr[n] = addMe;
   n = n + 1;
}
```

Big-O… my

```
void mystery(int n)
{
   for ( int q = 0; q < n; q++ )
   {
      for (int x = 0 ; x < q ; x++)
      {
           cout << "Waahoo!";
      }
   }
}
```

Sometimes you'll run into an algorithm that isn't so clear-cut. For example, what's the Big-O of mystery?

It's clear that the outer loop runs n times, but what about the inner loop?

When q = 0, the inner loop runs 0 iterations.
When q = 1, the inner loop runs 1 iteration.
When q = 2, the inner loop runs 2 iterations.
…
When q = n-1, the inner loop runs n-1 iterations.

So what's the Big-O?

$$f(n) = \frac{n^2}{2} - \frac{n}{2}$$

$$O(n^2)$$

So the cout statement will run a total of:

0 times + 1 time + 2 times + 3 times + … + n-1 times

And if you recall a clever trick, this is equal to: $\frac{n \cdot (n-1)}{2}$

Big-O: Such Ugly Math! ☺

But we're not Math geeks… We're CS geeks! So here's a way to address these situations without formulas!

Step 1:

Locate all loops that don't run for a fixed number of iterations and determine the maximum number of iterations each loop could run for.

Step 2:

Turn these loops into loops with a fixed number of iterations, using their maximum possible iteration count.

Step 3:

Finally, do your Big-O analysis.

```
func1(int n)
{
   for ( int i = 0; i < n; i++ )
      for (int j=0; j  <  i  ;j++)
         cout << j;
}
```

n

$O(n^2)$

```
int main( )
{
   for ( int x = 0; x < n; x++ )
      for (int j=0; j < x*x  ; j++)
         cout << "Burp!";
}
```

n*n

$O(n^3)$

Find the Big-O Challenge, Part 2

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

```
for ( int i = 0; i < q*q; i++ )
   for ( int j = 0; j < i; j++ )
      sum++;
```

```
for ( int i = 0; i < n; i++ )
   for ( int j = 0; j < i*i; j++ )
      for ( int k = 0; k < j; k++)
         sum++;
```

```
for ( int i = 0; i < p; i++ )
   for ( int j = 0; j < i*i; j++ )
      for ( int k = 0; k < i; k++)
         sum++;
```

```
for ( int i = 0; i < n; i++ )
{
    Circ arr[n];

    arr[i].setRadius(i);
}
```

```
for (int i=0 ; i < n ; i++)
{
    int k = i;
    while (k > 1)
     {
        sum++;
        k = k/2;
     }
}
```

Big-O for Multi-input Algorithms

What's the Big-O of the following algorithms?

```
int mysteryAlg1(int arr1[],int size1,
                int arr2[], int size2)
{
  int i,j,count = 0;

  for (i=0;i<size1;i++)
    for (j=0;j<size2;j++)
      count += arr1[i]*arr2[j];

  return(count);
}
```

O( )

```
int mysteryAlg2(int arr1[],int size1,
                int arr2[], int size2)
{
  int i,count = 0;

  for (i=0;i<size1;i++)
   count += arr1[i];

  for (i=0;i<size2;i++)
   count += arr2[i];

  return(count);
}
```

O( )

In these examples, we have two *independent* input data sets of sizes size1 and size2.

Therefore, when we compute their Big-O, we must to take into account both independent sizes.

Find the Big-O Challenge, Part 3

```
void bar( int n, int q )
{
    for (int i=0 ; i < n*n ; i++)
    {
        for (int j = 0; j < q; j++)
            cout << "I love CS!";
    }
}
```

```
void burp( int n )
{
    for (int i=0 ; i < n ; i++)
        cout << "Muahahaha!";

    for (int i=0 ; i < n*n ; i++)
        cout << "Vomit!";
}
```

```
void bletch( int n, int q )
{
    for (int i=0 ; i < n ; i++)
        cout << "Muahahaha!";

    for (int i=0 ; i < q*q ; i++)
        cout << "Vomit!";
}
```

```
void barf( int n, int q )
{
    for (int i=0 ; i < n ; i++)
    {
        if (i == n/2)
        {
            for ( int k = 0; k < q; k++ )
                cout << "Muahahaha!";
        }
        else
            cout << "Burp!";
    }
}
```

The STL and Big-O

Remember the STL - stacks, queues, sets, vectors, lists and maps?

Well, these classes use algorithms to get things done…

And these algorithms have Big-Os too!

```
void inDict(set<string> & d, string w)
{
   if ( d.find(w) == d.end() )
       cout << w << " isn't in dictionary!";
}

void otherFunc(vector<int> & vec)
{
   vec.push_back(42);

   vec.erase( vec.begin() );

}
```

For example, if we want to search for a word in a set that contains n words (a dictionary), it requires $O(\log_2(n))$ steps!

But if we want to add a value to the end of a vector holding n items, it takes just one step, so it's $O(1)$!

And if we want to delete the 1st value from a vector containing n items, it takes a whopping n steps, making it $O(n)$!

and Big-O

Well, to search a set of n items for a single value requires $\log_2(n)$ steps…

And we repeat this search operation D different times…

It's important to understand the Big-O of each operation (e.g. push_back, erase) for each STL class (e.g., list, vector)…

… because without knowing the Big-Os of the STL classes,

we can't compute the Big-O of code that uses the STL classes!

For example…

If we write a loop of our own that runs D times…

And each iteration of our loop searches for an item in a set holding n items…

Then what's the Big-O of our loop???

Then the Big-O of our whole loop would be $O(D * \log_2(n))$.

```
void inDict(set<string> &d, string w)
{
    if ( d.find(w) == d.end() )
        cout << w << " isn't in dictionary!";
}

void spellCheck( set<int> &dict,
                 string doc[], int D )
{
    for (int i=0; i < D ; i++)
        inDict( dict, doc[i] );
}
```

the Big-O of algorithms that use STL

Ok, our vector contains q values...

And as such, our loop runs q times...

What is the Big-O of the loop in terms of q?

Let's look at another example to see how this works...

```
void printNums( vector<int>  & v )
{
   int q = v.size();
   for ( int i = 0; i < q ; i++ )
  {
      int a = v[0];            // get 1st item
      cout << a;               // print it out
      v.erase( v.begin() );  // erase 1st item
      v.push_back(a);         // add it to end
  }
}
```

And each time our loop runs, we:

1. Access an item: the cost of accessing an item in a vector is O(1) - so we'll remember that!

2. Erase the first item in the vector: the Big-O of erasing the first item in a vector with q items is O(q).

3. Add the item to the end of the vector: the Big-O of adding an item to the end of a vector is O(1).

The STL vector	
Insert at the top/middle:	O(n)
Insert an the end:	O(1)
Delete an item from top/middle:	O(n)
Delete an item from the end:	O(1)
Access an item:	O(1)
Finding an item:	O(n)

Total steps performed during our loop:

$$q * (1 + q + 1)$$

So our total Big-O is:

$$O(q^2)$$

STL and Big Oh Cheat Sheet

When describing the Big-O of each operation (e.g. insert) on a container (e.g., a vector) below, we assume that the container holds n items when the operation is performed.

Name: list
Purpose: Linked list
Usage: list<int> x; x.push_back(5);
Inserting an item (top, middle*, or bottom): $O(1)$
Deleting an item (top, middle*, or bottom): $O(1)$
Accessing an item (top or bottom): $O(1)$
Accessing an item (middle): $O(n)$
Finding an item: $O(n)$
*But to get to the middle, you may have to first iterate through X items, at cost $O(x)$

Name: vector
Purpose: A resizable array
Usage: vector<int> v; v.push_back(42);
Inserting an item (top, or middle): $O(n)$
Inserting an item (bottom): $O(1)$
Deleting an item (top, or middle): $O(n)$
Deleting an item (bottom): $O(1)$
Accessing an item (top, middle, or bottom): $O(1)$
Finding an item: $O(n)$

Name: set
Purpose: Maintains a set of unique items
Usage: set<string> s; s.insert("Ack!");
Inserting a new item: $O(\log_2 n)$
Finding an item: $O(\log_2 n)$
Deleting an item: $O(\log_2 n)$

Name: map
Purpose: Maps one item to another
Usage: map<int,string> m; m[10] = "Bill";
Inserting a new item: $O(\log_2 n)$
Finding an item: $O(\log_2 n)$
Deleting an item: $O(\log_2 n)$

Name: queue and stack
Purpose: Classic stack/queue
Usage: queue<long> q; q.push(5);
Inserting a new item: $O(1)$
Popping an item: $O(1)$
Examining the top: $O(1)$

If instead of holding n items, a container holds p items, then just replace "n" with "p" when you do your analysis.

Computing the Big-O of Algorithms that use STL

When evaluating STL-based algorithms, first determine the maximum # of items each container could possibly hold..

Then do your Big-O analysis under the assumption that each container always holds exactly this number of items.

Ok, I'll assume that our set always has q items in it. That means that each time we insert an item into our set it takes $\log_2(q)$ steps.

Ok. Well, after the loop finishes, our set will hold q values…

This is the maximum # of values it can hold.

```
int main( )
{
    set< int > nums;
    for (int i=0; i < q ; i ++)
      nums.insert( i );
}
```

So if our loop runs a total of q iterations… and each iteration we insert at a cost of $\log_2(q)$ into our set… Then our total cost is $q * \log_2(q)$

And that's the correct answer!

The STL set	
Inserting a new item:	$O(\log_2 n)$
Finding an item:	$O(\log_2 n)$
Deleting an item:	$O(\log_2 n)$

Find the Big-O Challenge, Part 4

See if you can figure out the Big-O of these functions which use the STL.

```
// Assume s starts out empty
void addItems(set<int>  &s, int n)
{
   for (int i=0; i < n*n; i++)
      s.insert(i);
}
```

Hints:

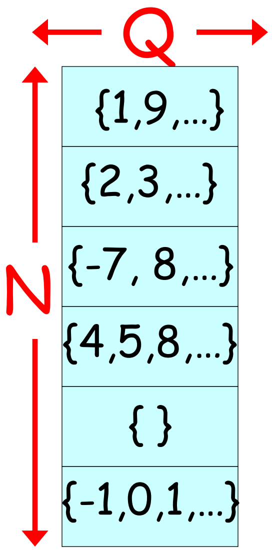
The STL vector

Insert at the top/middle:	O(n)
Insert an the end:	O(1)
Delete an item from top/middle:	O(n)
Delete an item from the end:	O(1)
Access an item:	O(1)
Finding an item:	O(n)

The STL set

Inserting a new item:	$O(\log_2 n)$
Finding an item:	$O(\log_2 n)$
Deleting an item:	$O(\log_2 n)$

Find the Big-O Challenge, Part 5



I have a vector of sets of ints:

`vector< set<int> >   v;`

You may assume vector v has N total sets in it.
You may assume that each set has an avg of Q items.

Questions:

What is the Big-O of determining if the first set, v[0], contains the value 7?

What is the Big-O of determining if any set in v has a value of 7?

What is the Big-O of determining the number of even values in all of v?

What is the Big-O of finding the first set with a value of 7 and then counting the number of even values in that set?

The STL set	
Inserting a new item:	$O(\log_2 n)$
Finding an item:	$O(\log_2 n)$
Deleting an item:	$O(\log_2 n)$

Pre-Sorting Intermission

(Brought to you by last year's students who claimed this lecture was too boring)

Sorting!

Sorting is the process of ordering a bunch of items based on one or more rules, subject to one or more constraints…

Items - what are we sorting, and how many are there?

- Strings, numbers, student records, C++ objects (e.g., Circles, Robots)
- Thousands, millions or trillions?

Rules - how do we order them?

- Ascending ↗ vs. Descending ↘ order
- Based on Circle radius? Student GPA?
- Based on multiple criteria, e.g.:
  by last name, then first name

Constraints?

- Are the items in RAM or on disk?
- Is the data in an array or a linked list?

Carey's 2 Rules of Sorting

Rule #1:

Don't choose a sorting algorithm until you understand the requirements of your problem.

Rule #2:

Always choose the simplest sorting algorithm possible that meets your requirements.

The Selection Sort

- Look at all N books, select the shortest book
- Swap this with the first book
- Look at the remaining N-1 books, and select the shortest
- Swap this book with the second book
- Look at the remaining N-2 books, and select the shortest
- Swap this book with the third book and so on…

So, is our sort efficient?

If we have N books, how many steps does it take to sort them?

Let's assume a step is any time we either swap a book or point our finger at a book.

The Selection Sort- Speed

- Look at all N books, select the shortest book

N steps

- Swap this with the first book

1 step

- Look at the remaining N-1 books, and select the shortest

N-1 steps

- Swap this book with the second book

1 step

- Look at the remaining N-2 books, and select the shortest

N-2 steps

- Swap this book with the third book, and so on…

1 step

So this comes to:

N swap steps

PLUS

N + N-1 + N-2 + … + 2 + 1

steps to find the smallest item

So Selection Sort is

$O(N^2)$

Or, for N books, you need roughly N^2 steps to sort them.

(It's considered pretty slow)

Selection Sort - Better or Worse?

Are there any kinds of input data where Selection Sort is either more or less efficient?

For example, what if all of the books are mostly in order before our sort starts?

```
void selectSort(shelf of N books)
{
  for i = 1 to N
  {
     find the smallest book
       between slots i and N

     swap this smallest book
       with book i;
  }
}
```

No! Selection sort takes just as many steps either way!

The Se

Selection Sort Questions

Can Selection Sort be applied easily to sort items within a linked list?

Is Selection Sort "stable" or "unstable"?

And here's the C++ source code to sort a bunch of numbers...

```
void selectionSort(int A[], int n)
{
  for (int i = 0; i < n; i++)
  {
    int minIndex = i;
    for (int j = i+1; j < n; j++)
    {
      if (A[j] < A[minIndex])
        minIndex = j;
    }
    swap(A[i], A[minIndex]);
  }
}
```

For each of the n array elements...

Locate the smallest item in the array between the i^{th} slot and slot n-1.

Swap the smallest item found with slot A[i].

What's a Stable Sort?

Imagine that N old people line up to buy laxatives at a drugstore.

And the drugstore wants to sort them and serve them based on urgency.

The drugstore needs to pick a sort algorithm to re-order the guests.
They can choose between a "stable" sort or an "unstable" sort.

An "unstable" sorting algorithm re-orders the items without taking into account their initial ordering.

A "stable" sorting algorithm does take into account the initial ordering when sorting, maintaining the order of similar-valued items.

As you solve problems (in class or at work) you should choose your sort depending on whether stability is important.
If you forget the concept, just remember the laxatives! ☺

People in line	Unstable Sort Results	Stable Sort Results
Ebeneezer - 8 days	Steve – 8 days	Ebeneezer - 8 days
Carey – 5 days	Vicki – 8 days	Steve – 8 days
David – 2 days	Ebeneezer - 8 days	Vicki – 8 days
Michael – 4 days	Andrea – 5 days	Carey – 5 days
Steve – 8 days	Carey – 5 days	Andrea – 5 days
Vicki – 8 days	Michael – 4 days	Michael – 4 days
Andrea – 5 days	David – 2 days	David – 2 days

The Se

And here's the C++ source code to sort a bunch of numbers…

Selection Sort Questions

Can Selection Sort be applied easily to sort items within a linked list?

Is Selection Sort "stable" or "unstable"?

When might you use Selection Sort?

```
void selectionSort(int A[], int n)
{
  for (int i = 0; i < n; i++)
  {
    int minIndex = i;
    for (int j = i+1; j < n; j++)
    {
      if (A[j] < A[minIndex])
        minIndex = j;
    }
    swap(A[i], A[minIndex]);
  }
}
```

Here's a hint – consider this array:

10	10	1

When Selection Sort finds the 1, it swaps it with the first 10.

Then our array ends up like this:

1	10	10

Sorting Intermission

(Brought to you by last year's students who claimed this lecture was too boring)

The Insertion Sort

Well, we couldn't just teach you one sort, right?

Let's learn another!

The insertion sort is probably the most common way…

to sort playing cards!

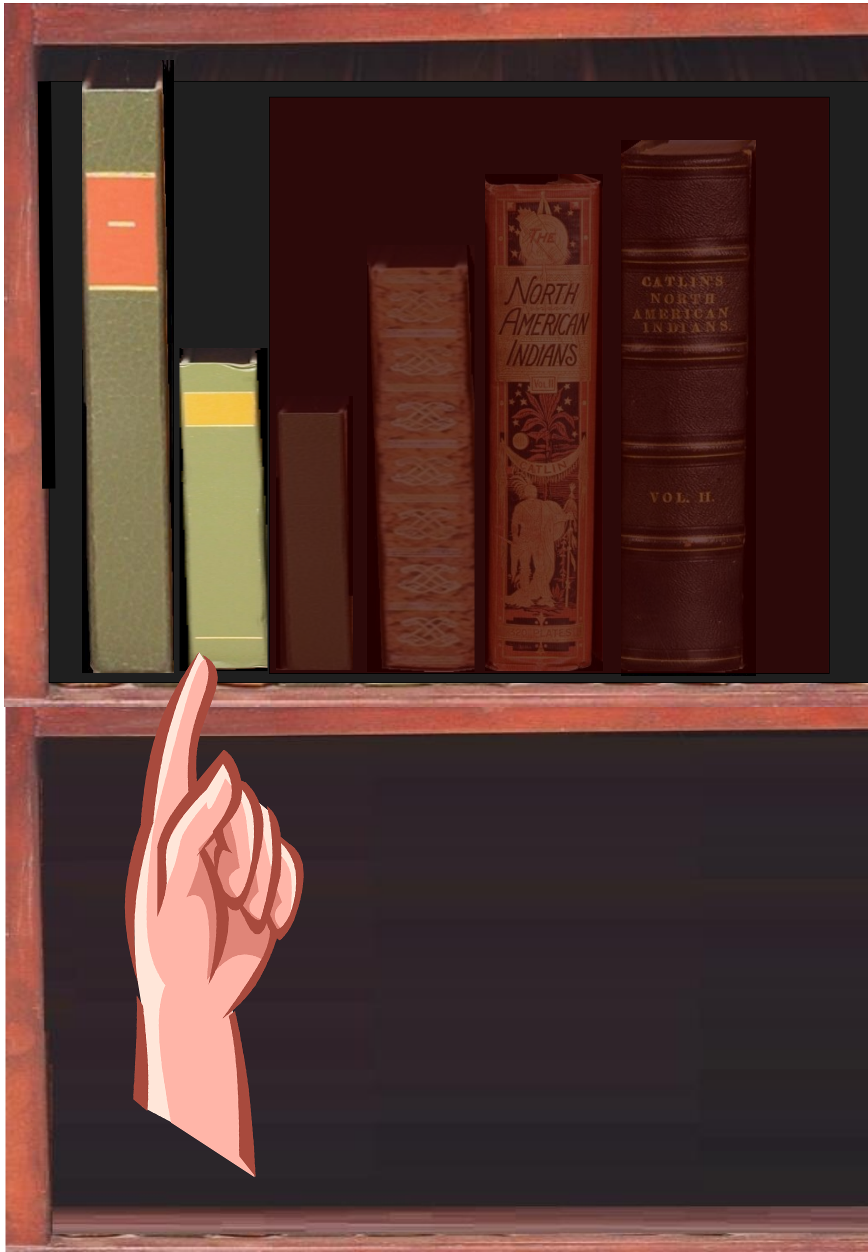
(But I'll still explain the sort with library books)

The Insertion Sort

Let's focus on the first two books – ignore the rest.

- If the last book in this set is in the wrong order
  - Remove it from the shelf
  - Shift the book before it to the right
  - Insert our book into the proper slot

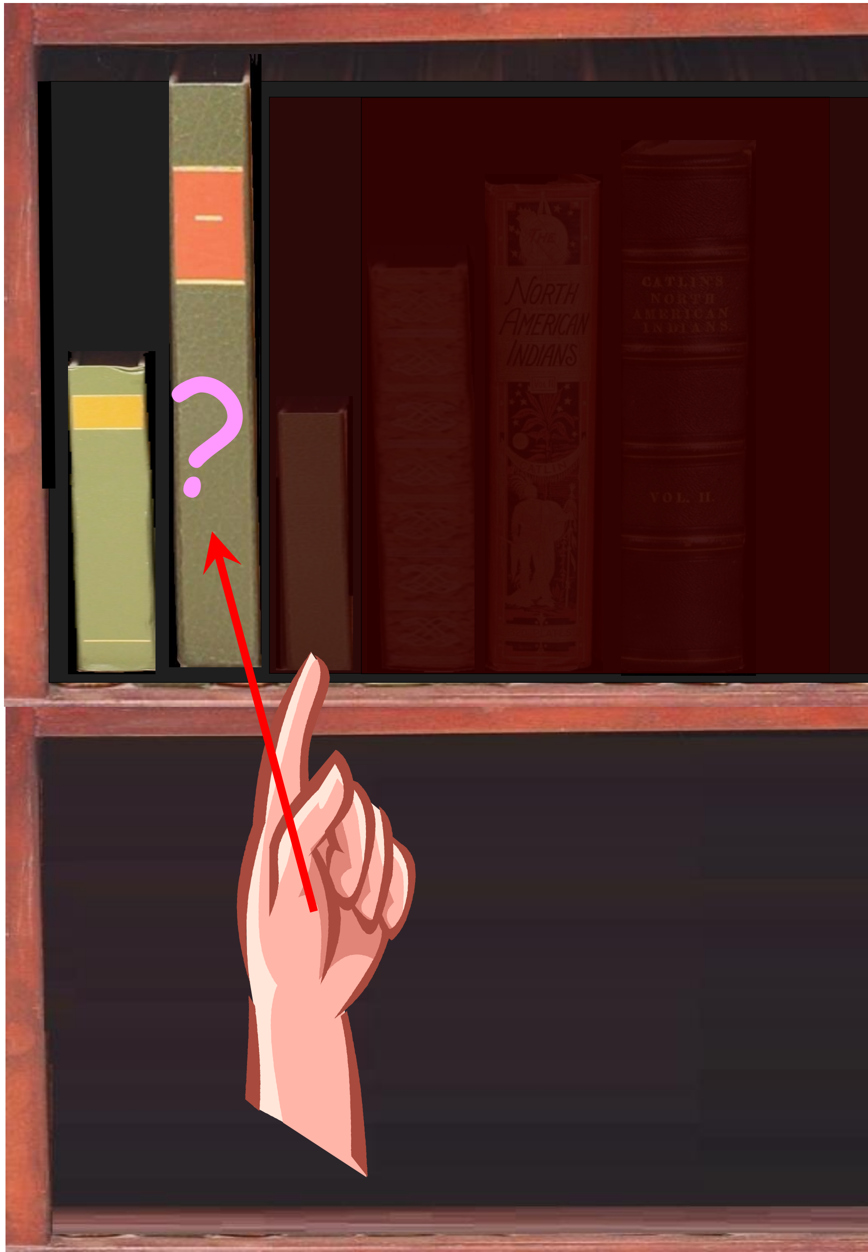
Great! Now our first two books are in sorted order (ignoring the others)



The Insertion Sort

Ok, now focus on the first three books - ignore the rest.

- If the last book in this set is in the wrong order
  - Remove it from the shelf
  - Shift the books before it to the right, as necessary
  - Insert our book into the proper slot



Great! Now our first three books are in sorted order (ignoring the others)

The Insertion Sort

Ok, now focus on the first four books – ignore the rest.

- If the last book in this set is in the wrong order
  - Remove it from the shelf
  - Shift the books before it to the right, as necessary
  - Insert our book into the proper slot

Great! Now our first four books are in sorted order!

We just keep repeating this process until the entire shelf is sorted!

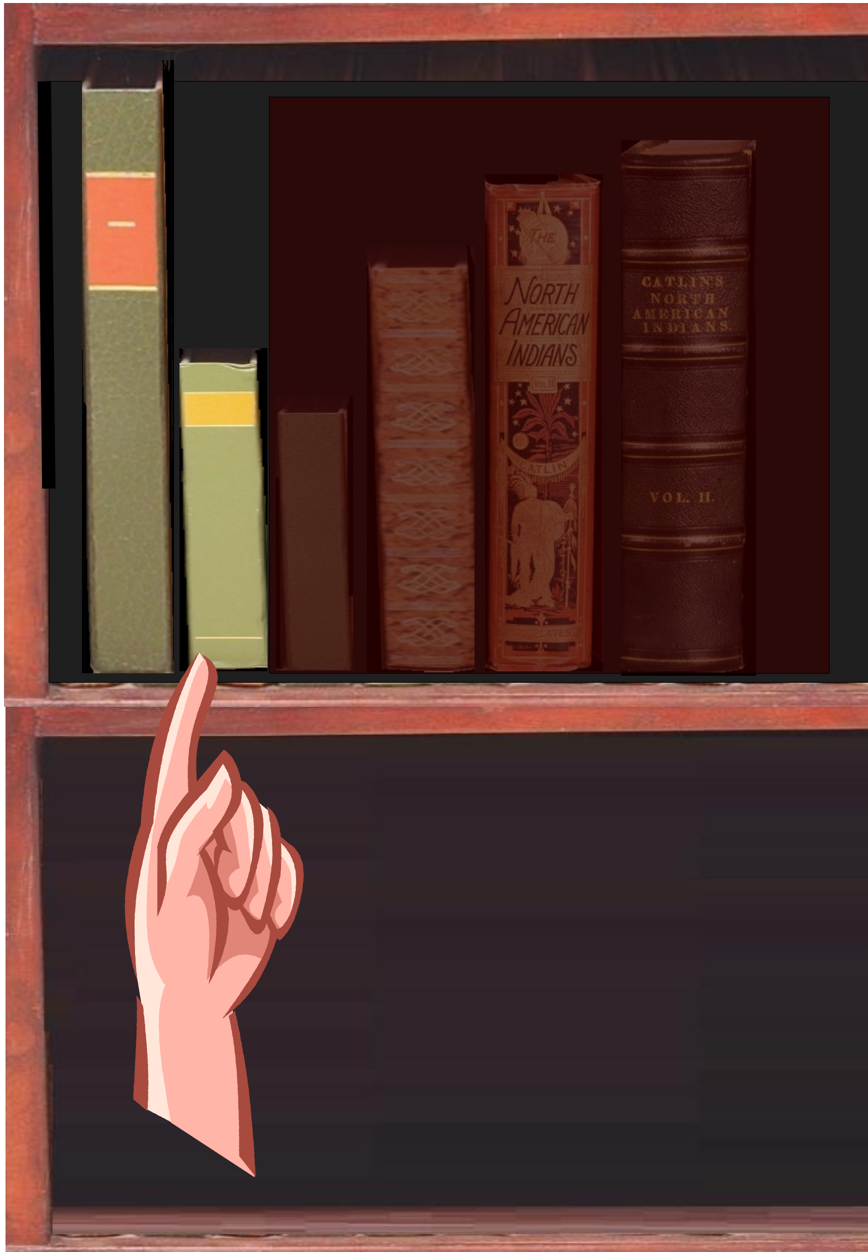
The Insertion Sort

So what's the complete algorithm?

Start with set size s = 2

While there are still books to sort:

- Focus on the first s books
- If the last book in this set is in the wrong order
    - Remove it from the shelf
    - Shift the books before it to the right, as necessary
    - Insert our book into the proper slot
- s = s + 1



The Insertion Sort - Speed

So what's the Big-O of our Insertion Sort?

During each round of the algorithm we consider a larger set of books.

During the first round, we may need to shift up to one book to find the right spot.

The Insertion Sort - Speed

So what's the Big-O of our Insertion Sort?

During each round of the algorithm we consider a larger set of books.

During the first round, we may need to shift up to one book to find the right spot.

During the second round, we may need to shift up to two books to find the right spot.

2 shifts →

The Insertion Sort - Speed

So what's the Big-O of our Insertion Sort?

During each round of the algorithm we consider a larger set of books.

N-1 shifts →

During the first round, we may need to shift up to one book to find the right spot.

During the second round, we may need to shift up to two books to find the right spot.

...

During the last round, we may need to shift up to N-1 books to find the right spot.

1 step in round 1
+ 2 steps in round 2
+ ...
+ N-1 steps in last rnd

= roughly N^2 steps

Thus, Insertion Sort is $O(N^2)$, and is generally quite slow!

Insertion Sort - Better or Worse?

Are there any kinds of input data where Insertion Sort is either more or less efficient?

Any ideas?

Right! If all books are in the proper order...

then Insertion Sort never needs to do any shifting!

In this case, it just takes roughly ~N steps to sort the array! O(N)

Conversely, a perfectly mis-ordered set of books is the worst case.

Since every round requires the maximum shifts!

The Insertio

And here's the C++ version w
sorts an array in ascending o

Insertion Sort Questions

Can Insertion Sort be applied easily to sort items within a linked list?
Is Insertion Sort a "stable" sort?
When might you use Insertion Sort?

```
void insertionSort(int A[], int n)
{
  for(int s = 2; s <= n; s++)
  {
     int sortMe = A[ s - 1 ];

     int i = s - 2;
     while (i >= 0 && sortMe < A[i])
     {
         A[i+1] = A[i];
         --i;
     }

    A[i+1] = sortMe;
  }
}
```

Focus on successively larger prefixes of the array. Start with the first s=2 elements, then the first s=3 elements...

Make a copy of the last val in the current set – this opens up a slot in the array for us to shift items!

Shift the values in the focus region right until we find the proper slot for sortMe.

Store the sortMe value into the vacated slot.

Sorting Intermission

(Brought to you by last year's students who claimed this lecture was too boring)

Everyone loves to make fun of the:

Sort

But it's actually quite simple… And sometimes simple is good!

Ok, what's the algorithm?

Start at the top element of your array

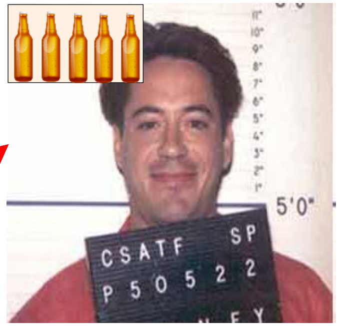
Compare the first two elements: A[0] and A[1]
If they're out of order, then swap them

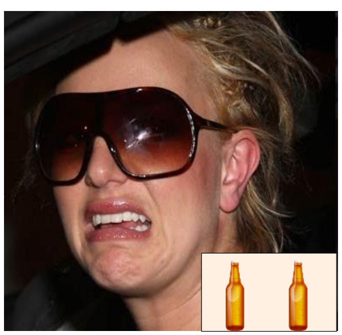
Then advance one element in your array
Compare these two elements: A[1] and A[2]
If they're out of order, swap them
…
Repeat this process until you hit the end of the array

When you hit the end, if you made at least one swap, then repeat the whole process again!

Bubble Sort In Action! Hic!

?





SWAP?

Let's see how bubble sort works with some of your favorite celebrities…

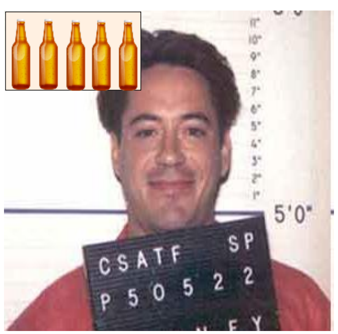
…to sort them based on the number of times they've been in… *rehab!*

Compare the first two elements
If they're out of order, then swap them

Compare the next two elements: A[1] and A[2]
If they're out of order, swap them

Compare the final two elements: A[2] and A[3]
If they're out of order, swap them

When you hit the end, if you made at least one swap, then repeat the whole process again!



Bubble Sort In Action! Hic!

SWAP?

Let's see how bubble sort works with some of your favorite celebrities…

…to sort them based on the number of times they've been in… *rehab!*

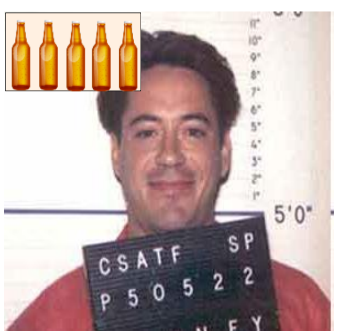
Compare the first two elements
If they're out of order, then swap them

Compare the next two elements: A[1] and A[2]
If they're out of order, swap them

Compare the final two elements: A[2] and A[3]
If they're out of order, swap them

When you hit the end, if you made at least one swap, then repeat the whole process again!

Bubble Sort In Action! Hic!



SWAP?

Let's see how bubble sort works with some of your favorite celebrities…

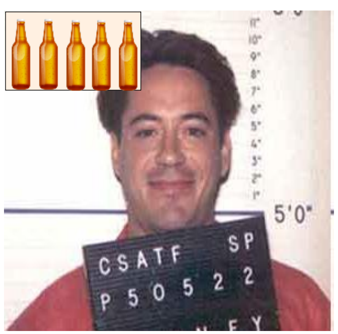
…to sort them based on the number of times they've been in… *rehab!*

Compare the first two elements
If they're out of order, then swap them

Compare the next two elements: A[1] and A[2]
If they're out of order, swap them

Compare the final two elements: A[2] and A[3]
If they're out of order, swap them

When you hit the end, if you made at least one swap, then repeat the whole process again!



Bubble Sort In Action! Hic!

SWAP?

Let's see how bubble sort works with some of your favorite celebrities...

...to sort them based on the number of times they've been in... *rehab!*

Compare the first two elements
If they're out of order, then swap them

Compare the next two elements: A[1] and A[2]
If they're out of order, swap them

Compare the final two elements: A[2] and A[3]
If they're out of order, swap them

When you hit the end, if you made at least one swap, then repeat the whole process again!

And we're done!

Bubble Sort Speed

Ok, so Bubble Sort has a bad wrap.

Question: How fast is it?

Just like Insertion Sort, Bubble Sort is really efficient on pre-sorted arrays and linked lists!

During each pass, we compare every element with its successor (and possibly swap each).

That requires about N steps.

If we did even one swap, we need to repeat the whole process again.

What's the worst case? How many times might we have to repeat the process?

Hint? Ok!

Right! We might have to repeat this entire process N times.

N passes of N "bubbles" = N^2

Ok, so Bubble Sort is $O(N^2)$...

But can it ever run faster in certain cases?

The Bubble Sort

```
void bubbleSort(int Arr[], int n)
{
  bool atLeastOneSwap;

  do
  {
    atLeastOneSwap = false;
    for (int j = 0; j < (n-1); j++)
    {
      if (Arr[j] > Arr[j + 1])
      {
        Swap(Arr[j],Arr[j+1]);
        atLeastOneSwap = true;
      }
    }
  }
  while (atLeastOneSwap == true);
}
```

Bubble Sort Questions

Can Bubble Sort be applied easily to sort items within a linked list?
Is Bubble Sort a "stable" sort?
Is Bubble Sort ever faster than $O(n^2)$?
When might you use Bubble Sort?

Start by assuming that we won't do any swaps

Compare each element with its neighbor and swap them if they're out-of-order.

Don't forget-we swapped!

If we swapped at least once, then start back at the top and repeat the whole process.

Sorting Intermission

(Brought to you by last year's students who claimed this lecture was too boring)

Sorting Challenge

Consider the following array of integers:

By one round, I mean one full trip through the sort's while/for loop.

2	5	9	14	7	3

which has been sorted by one round of either selection sort, insertion sort or bubble sort.

Which of these sorts could NOT have been used on this array? Why?

selectionSort

For each of the N books
    Find the smallest book between slots i and N
    Swap this smallest book with book i

insertionSort

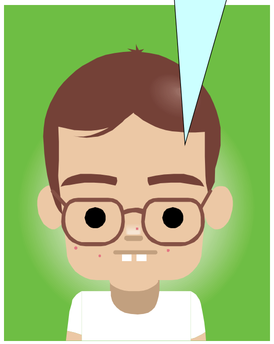
s = 2
While books need sorting:
    Focus on the first s books
    If the last book in set is in the wrong order THEN
        A. Remove it from shelf
        B. Shift the books to the right as required
        C. Insert our book into the proper slot
    s = s + 1

bubbleSort

while the shelf isn't sorted
    repeatedly swap adjacent books if they're out of order

The  sort

Shellsort is based on an underlying procedure called h-sorting. Let's learn h-sorting first...



The method for h-sorting an array is simple:

Pick a value of h

For each element in the array:

- If A[i] and A[i+h] are out of order then
  - Swap the two elements

If you swapped any elements during the last pass, then repeat the entire process again (same h value).

The Shellsort: h-sorting

Pick a value of h
For each element in the array:
- If A[i] and A[i+h] are out of order
  - Swap the two elements

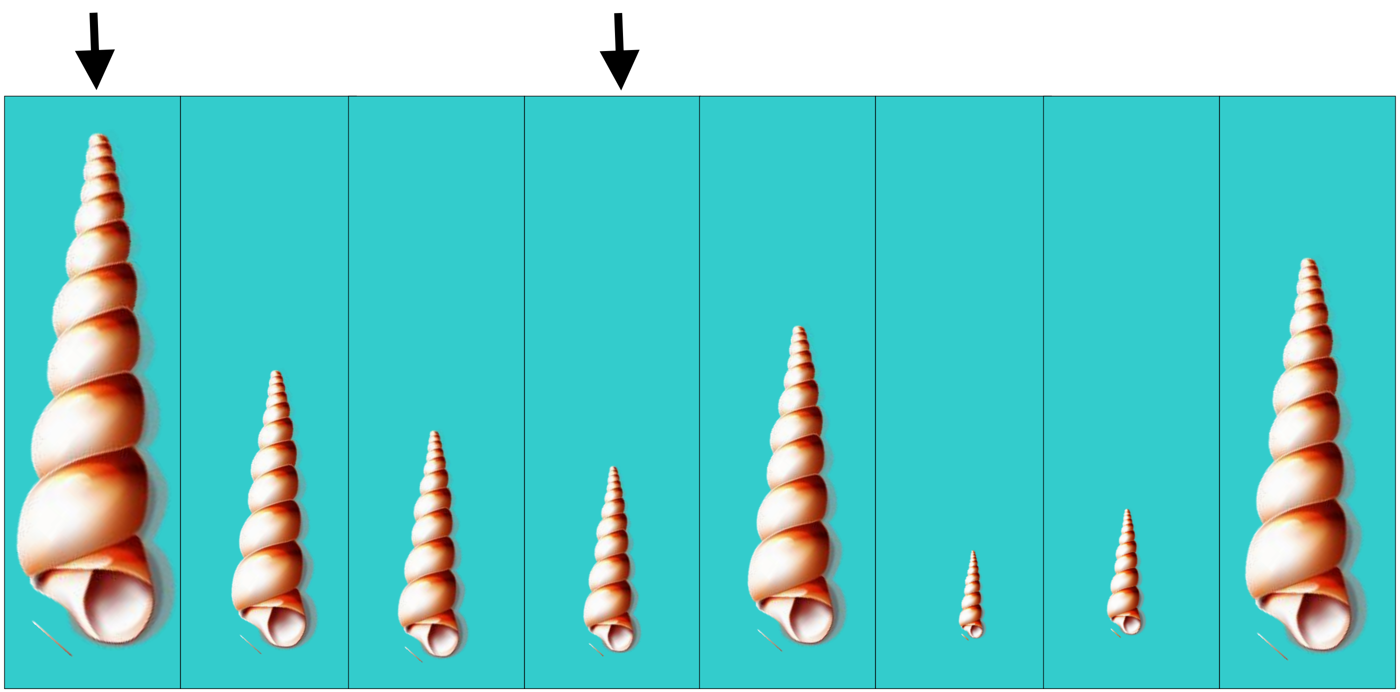
If you swapped any elements,
repeat the entire process again.

Let's 3-sort this array so the shells are ascending.
e.g., h=3

i

i+3

SWAP?



The Shellsort: h-sorting

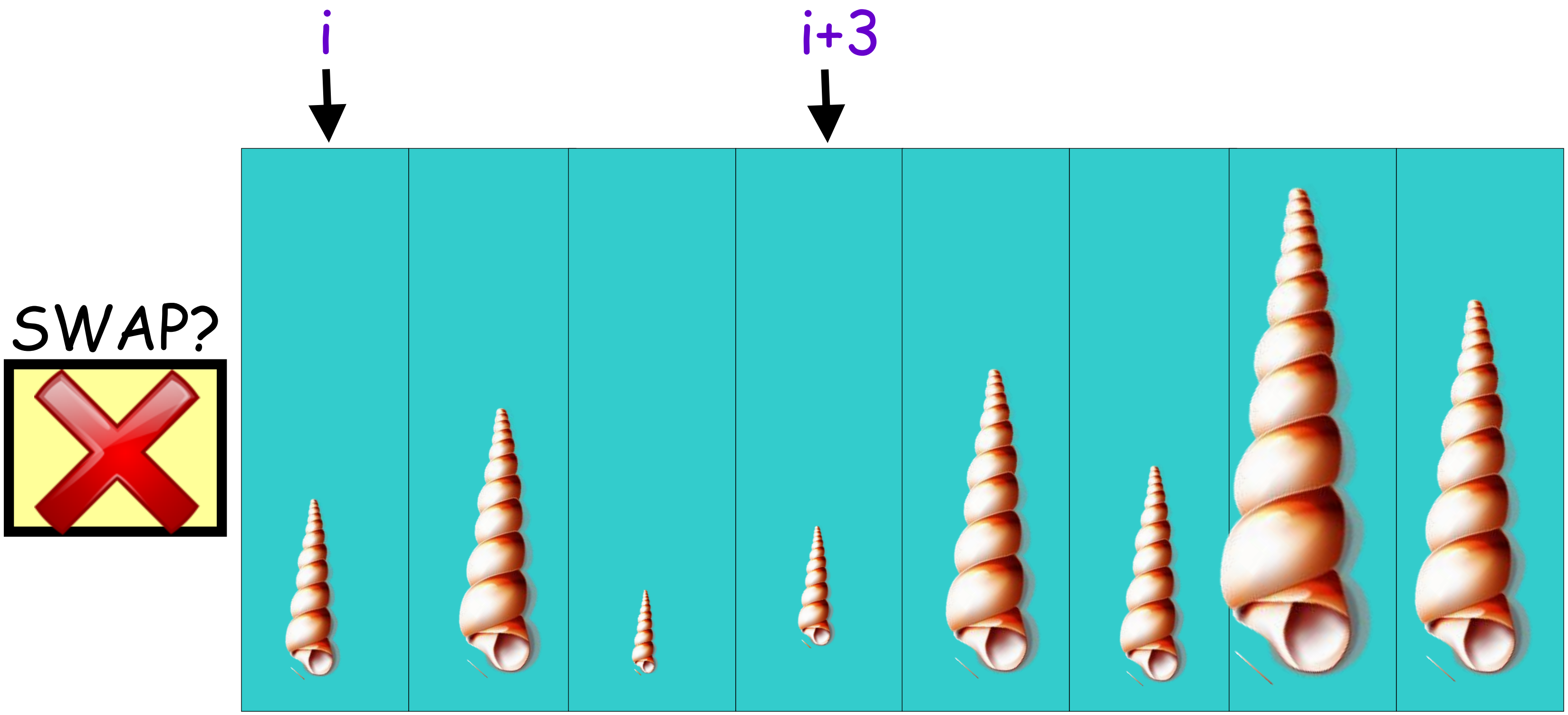
Pick a value of h

For each element in the array:

- If A[i] and A[i+h] are out of order
    - Swap the two elements

If you swapped any elements, repeat the entire process again.

Let's 3-sort this array so the shells are ascending. e.g., h=3



The ... -sorting

Pick ...
For ...

I...
r...

...et's 3-sort this array so
...he shells are ascending.
e.g., h=3

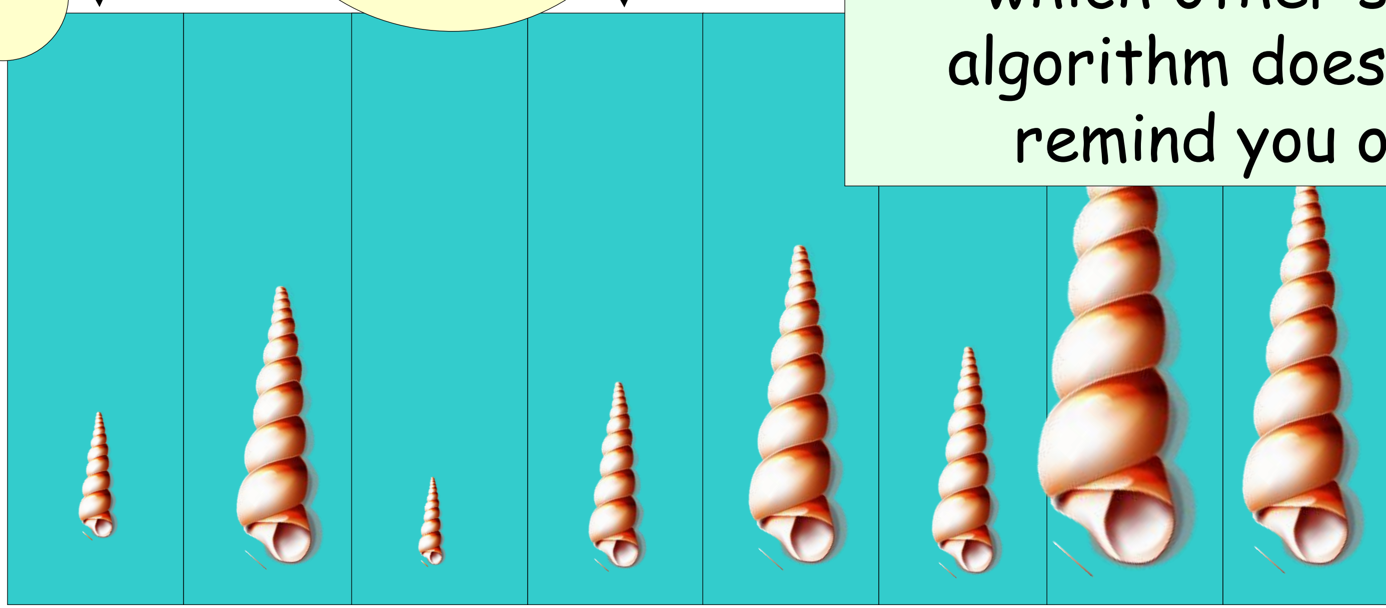
This time we had no swaps!
Our array is now 3-sorted!

This means that every element
is smaller than
the element 3 items later
in the array.

Of course, it's not completely
sorted yet, just 3-sorted!

Question:
If you 1-sort an array,
which other sort
algorithm does this
remind you of?

SWAP?



The Shellsort

It's required to always end with h=1!
e.g. 5, 3, 1 or
10, 7, 4, 1

The overall Shellsort works as foll

Step 1:

Select a sequence of *decreasing* h-values, *ending with an h-value of 1*: e.g. 8,4,2,1.

Step 2:

First completely 8-sort the array…
Then completely 4-sort the array…
Then completely 2-sort the array…
Finally, completely **bubble sort** the array…

and the array's now fully sorted!

Each h-sort more correctly sorts the array, making the process simpler each iteration.

Shell Sort Questions

Can Shell Sort be applied easily to sort items within a linked list?

Is Shell Sort a "stable" sort?

What's the Big-O of Shell Sort?

When might you use Shell Sort?

The Shellsort

Let's do an example on the board:

Shellsort the following array using h values of: 3, 2, and 1.

9	5	2	14	3	7

Sorting Challenge

Given the following numbers, show what they would look like after one, two and three outer-loop iterations of selection sort, insertion sort and bubble sort:

9	5	2	14	3	7

<u>selectionSort</u>

For each of the N books
  Find the smallest book between slots i and N
  Swap this smallest book with book i

<u>insertionSort</u>

s = 2
While books need sorting:
  Focus on the first s books
  If the last book in set is in the wrong order THEN
    A. Remove it from shelf
    B. Shift the books to the right as required
    C. Insert our book into the proper slot
  s = s + 1

<u>bubbleSort</u>

while the shelf isn't sorted
  repeatedly swap adjacent books if they're out of order