BASIC SORTING, PART 2
administrivia...
-assignment 3 is due tonight at midnight

-assignment 4 is out
  - requires pair programming
  - due next Wednesday

-short homework for lab on Friday

-Ryan will have office hours at 6:30 tonight
How many hours did you spend on assignment 3?

A) <5
B) 5-10
C) 10-15
D) 15-20
E) >20
last time...
selection sort
the simplest sorting algorithm
Find (ie. select) the smallest item in the unsorted portion of the array and move to the end of the sorted portion of the array.
selection sort

1) find the minimum item in the unsorted part of the array
2) swap it with the first item in the unsorted part of the array
3) repeat steps 1 and 2 to sort the remainder of the array

what does this look like?
what is the relationship of the best and worst case complexity?

1) same
2) different

what is the relationship of the best and worst case complexity?
void selectionSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        min = i;
        for(int j=i+1; j < arr.length; j++)
            if (arr[j] < arr[min])
                min = j;

        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}
insertion sort

good for small $N$
Take the first item in the unsorted portion of the array and **insert** it into the sorted portion of the array.
insertion sort

1) the first array item in the unsorted array is the sorted portion of the array

2) take the second item and insert it in the sorted portion

3) repeat steps 1 and 2 to sort the remainder of the array

what does this look like?
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}

1) same
2) different

what is the relationship of the best and worst case complexity?
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}

what is the best-case complexity?

1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N^2)
6) O(N^3)
void insertionSort(int[] arr)
{
    for(int i=1; i < arr.length; i++)
    {
        index = arr[i];
        j = i;
        while(j>0 && arr[j-1]>index)
        {
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = index;
    }
}

Determining average & worst-case requires a measure of unsortedness.
unsortededness

- inversion: a pair of array items that are out of order

| 45 | -3 | 9 | 76 | 11 | -8 | 0 |

how many inversions are there?

-sorting efficiency depends on how many inversions are removed per step
insertion sort complexity

each swap to the left removes one inversion…

…we must visit each item at least once ($N$)…

…and we must undo $I$ inversions

```
45 -3 9 76 11 -8 0
```

**swap** removes one inversion

insertion sort is $O(N+I)$

How do we figure out what $I$ is?
worst case scenario...

-what are the number of inversions in the worst case?
  - what IS the worst case?
  - when every unique pair is inverted...

- how many unique pairs are there?
  -(hint: remember Gauss’ trick!)

\[(N+1) \times \frac{N}{2} = \frac{N^2 + N}{2}\]
What is the worst-case complexity of insertion sort?

1) $O(1)$
2) $O(\log N)$
3) $O(N)$
4) $O(N \log N)$
5) $O(N^2)$
6) $O(N^3)$
insertion sort is $O(N+I)$

what is the average-case complexity of insertion sort?

1) $O(1)$
2) $O(\log N)$
3) $O(N)$
4) $O(N \log N)$
5) $O(N^2)$
6) $O(N^3)$
recap...
selection vs insertion

worst: $O(N^2)$  $O(N^2)$
average: $O(N^2)$  $O(N^2)$
best: $O(N^2)$  $O(N)$
selection vs insertion

<table>
<thead>
<tr>
<th></th>
<th>worst:</th>
<th>average:</th>
<th>best:</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>insertion</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

which one performs better in practice?

A) selection
B) insertion
summary

- an inversion is a pair of items that are out of order
  - a sorted array has 0 inversions
  - an average (and worst) array has \( \sim N^2 \) inversions

-thus, we must undo \( N^2 \) inversions

-to do better than \( O(N^2) \) we must remove more than 1 inversion per step
  -(insertion sort only removes 1 inversion per step!)
what we want...

- a sorting algorithm that has subquadratic complexity

-swapping adjacent items removes exactly 1 inversion

\[
\begin{array}{ccccccccc}
 45 & -3 & 9 & 76 & 11 & -8 & 0 \\
\end{array}
\]

swap removes 1 inversion

- what if we consider swapping nonadjacent pairs?

\[
\begin{array}{ccccccccc}
 45 & -3 & 9 & 76 & 11 & -8 & 0 \\
\end{array}
\]

swap removes 7 inversions

- removes inversions not involved with the swap
today...
-shellsort

-bubble sort
shellsort
the simplest subquadratic sorting algorithm
Divide the array (smartly) into subarrays. Do insertion sort on the subarrays. Repeat.

* Take the first item in the unsorted portion of the array and **insert** it into the sorted portion of the array.
1) set the gap size to $N/2$

2) consider the subarrays with elements at gap size from each other

3) do insertion sort on each of the subarrays

4) divide the gap size by 2

5) repeat steps 2 — 4 until the is gap size is $<1$
1) set the gap size to \( \frac{N}{2} \)

2) consider the subarrays with elements at gap size from each other

3) do insertion sort on each of the subarrays

4) divide the gap size by 2

5) repeat steps 2 — 4 until the gap size is <1
How can we describe insertion sort with respect to shellsort?
-each $x$-sort (for a gap $x$) is performing an insertion sort on $x$ independent subarrays

-also known as the *diminishing gap sort*

-Shell originally suggested gaps $N/2, N/4, N/8, \ldots, 1$
  -gap sequences in which consecutive gaps share no common factors have been shown to perform better
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
            {
                arr[j+gap] = arr[j];
            }        
            arr[j+gap] = val;
        }
    }
}
void shellSort(int[] arr) {
    for (gap = arr.length/2; gap > 0; gap /= 2) {
        for (i = gap; i < arr.length; i++) {
            val = arr[i];
            for (j = i-gap; j >= 0 && arr[j] > val; j -= gap) {
                arr[j+gap] = arr[j];
            }
            arr[j+gap] = val;
        }
    }
}
void shellSort(int[] arr) {
    for (gap = arr.length / 2; gap > 0; gap /= 2) {
        for (i = gap; i < arr.length; i++) {
            val = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > val; j -= gap) {
                arr[j + gap] = arr[j];
            }
            arr[j + gap] = val;
        }
    }
}

diminishing gap sequence

until the insertion position is found, shift sorted items
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
            {
                arr[j+gap] = arr[j];
                arr[j+gap] = val;
            }
        }
    }
}
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}
what is the complexity of shellsort?

```java
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];

            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
            {
                arr[j+gap] = arr[j];
                arr[j+gap] = val;
            }
        }
    }
}
```

until the insertion position is found, shift sorted items
diminishing gap sequence
item to be inserted
insert item
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}

**what is the best-case complexity of shellsort?**
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}

what is the best-case complexity of shellsort?
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}

**what is the best-case complexity of shellsort?**
shellsort complexity

- worst case: $O(N^2)$ with Shell’s gaps, $O(N^{3/2})$ with better gaps

- average case: $O(N^{3/2})$ with Shell’s gaps, $O(N^{5/4})$ with better gaps

- proofs of these bounds are complicated
  - the $O(N^{5/4})$ bound is based on simulations only!

- insertion sort performs better the more sorted the array
  - remember, approaches $O(N)$ for a sorted array!
shellsort complexity

- still, $O(N^{5/4})$ is an encouraging bound for the average case
- for moderate $N$, this is better than $O(N \log N)$ algorithms
- around $N=100K$, $O(N \log N)$ wins
- best sorting algorithms are $O(N \log N)$
  - $\log N$ suggests repeated dividing by 2
  - “divide and conquer”
shell sort complexity

- still, $O(N^{5/4})$ is an encouraging bound for the average case
- for moderate $N$, this is better than $O(N \log N)$ algorithms
- around $N=100K$, $O(N \log N)$ wins
- best sorting algorithms are $O(N \log N)$
  - $\log N$ suggests repeated dividing by 2
  - “divide and conquer”

what algorithm do we know of that is $\log N$?
shellsort complexity

- still, $O(N^{5/4})$ is an encouraging bound for the average case
- for moderate $N$, this is better than $O(N \log N)$ algorithms
- around $N=100K$, $O(N \log N)$ wins
- best sorting algorithms are $O(N \log N)$
  - $\log N$ suggests repeated dividing by 2
  - “divide and conquer”

what algorithm do we know of that is $\log N$?
what does this imply about the “conquer” step?
bubble sort
the (usually) most inefficient sorting algorithm
Compare each pair of adjacent items and swap them if necessary. Repeat.
bubble sort

1) for each item, compare it to its next neighbor and swap if necessary

2) repeat step 1 until sorted

what does this look like?
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
            }
    }
}
void bubbleSort(int[] arr) {
    for(int i=0; i < arr.length-1; i++) {
        for(int j=0; j < arr.length-2; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
            }
    }
}
void bubbleSort(int[] arr) {
    for(int i=0; i < arr.length-1; i++) {
        for(int j=0; j < arr.length-2; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
            }
        }
    }
}
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
            }
    }
}
void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - 2; j++)
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = arr[j];
            }
    }
}

what is the complexity?

1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N²)
6) O(N³)
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
            }
    }
}

do we need to do every one of the outer-loop iterations?
void bubbleSort(int[] arr)
{
    swapped = false;
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
                swapped = true;
            }
        if (!swapped) return;
    }
    if (!swapped) return;
}
void bubbleSort(int[] arr) {
    
    swapped = false;
    for(int i=0; i < arr.length-1; i++) {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
                swapped = true;
            }
        if (!swapped) return;
    }
    if (!swapped) return;
}

what is the best-case complexity?

1) O(1)  
2) O(log N)  
3) O(N)  
4) O(N log N)  
5) O(N^2)  
6) O(N^3)
rabbits and turtles

-large numbers move “quickly” towards end, getting carried along by swaps

-small numbers move “slowly” towards beginning, only moving one place each iteration

-efforts to remove turtles:
  -cocktail sort
    -goes from beginning to end, then end to beginning
  -comb sort
    -using a diminishing gap for comparing pairs of items
next time…
- **reading**
  - chapters 7 & 8.5 - 8.8

- **homework**
  - assignment 3 due today
  - assignment 4 is out