"Organizing is what you do before you do something, so that when you do it, it is not all mixed up."

~ A. A. Milne

**Prelim 1: Tuesday, 12 March**

Visit exams page of course website. It tells you your assigned time to take it (5:30 or 7:30) and what to do if you have a conflict.

Anyone with any kind of a conflict must complete assignment P1 Conflict on the CMS by midnight of Wednesday, 6 March.

It is extremely important that this be done correctly and on time. We have to schedule room and proctors and know how many of each prelim (5:30 or 7:30) to print.

**Recitation next week**

Review for prelim!

**Why Sorting?**

- Sorting is useful
  - Database indexing
  - Operations research
  - Compression
- There are lots of ways to sort
  - There isn't one right answer
  - You need to be able to figure out the options and decide which one is right for your application.
  - Today, we'll learn several different algorithms (and how to develop them)

**We look at four sorting algorithms**

- Insertion sort
- Selection sort
- Quick sort
- Merge sort

**InsertionSort**

<table>
<thead>
<tr>
<th>pre: b</th>
<th>post: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sorted</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inv: b</th>
<th>or: b[0..i-1] is sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
</tbody>
</table>

A loop that processes elements of an array in increasing order has this invariant — just replace “sorted” by “processed”. 
Each iteration, i = i+1; How to keep inv true?

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sorted</td>
<td>?</td>
</tr>
</tbody>
</table>

E.g.:

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 7</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

Loop body (inv true before and after)

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 7</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

This will take time proportional to the number of swaps needed

What to do in each iteration?

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sorted</td>
<td>?</td>
</tr>
</tbody>
</table>

E.g.:

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 7</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

Push b[i] to its sorted position in b[0..i], then increase i

What to do in each iteration?

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sorted</td>
<td>?</td>
</tr>
</tbody>
</table>

E.g.:

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 7</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

Loop body (inv true before and after)

<table>
<thead>
<tr>
<th>i</th>
<th>b[0..i-1]</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 7</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

This will take time proportional to the number of swaps needed

Insertion Sort

```java
public void sort(int[] b) {
    for (int i = 0; i < b.length; i++) {
        // Push b[i] down to its sorted position in b[0..i]
    }
}
```

Note English statement in body.

**Abstraction.** Says what to do, not how.

This is the best way to present it. We expect you to present it this way when asked.

Later, can show how to implement that with an inner loop.

Insertion Sort is stable

```java
public void sort(int[] b) {
    for (int i = 0; i < b.length; i++) {
        // Push b[i] down to its sorted position in b[0..i]
    }
}
```

A sorting algorithm is stable if two equal values stay in the same relative position.

Initial: (3, 7, 2, 8, 7, 6)

Stably sorted: (2, 3, 6, 7, 7, 8)

Unstably sorted: (2, 3, 6, 7, 7, 8)

Insertion sort is stable

```
```
Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ave time</th>
<th>Worst-case time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
</tbody>
</table>

SelectionSort

```
// sort b[].
// inv: b[0..i-1] sorted AND
//      b[0..i-1] <= b[i..]
for (int i= 0; i < b.length; i++) {
    int m= index of min of b[i..];
    Swap b[i] and b[m];
}
```

Another common way for people to sort cards

Runtime with $n = b.length$

- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

SelectionSort - not stable

```
// sort b[].
// inv: b[0..i-1] sorted AND
//      b[0..i-1] <= b[i..]
for (int i= 0; i < b.length; i++) {
    int m= index of min of b[i..];
    Swap b[i] and b[m];
}
```

QuickSort

Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

84 years old. Developed Quicksort in 1958. But he could not explain it to his colleague, so he gave up on it.

Later, he saw a draft of the new language Algor 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. "Ah!,” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.
Dijkstra's, Hoares, Grieses, 1980s

Partition algorithm of quicksort

pre: \[ h \ h+1 \quad ? \quad k \]
x is called the pivot

Swap array values around until \([h..k]\) looks like this:

\[
\begin{array}{cccc}
 h & j & k \\
 x & x & x
\end{array}
\]

post: \[ \leq x \quad x \quad \geq x \]

-----

Partition algorithm of quicksort

\[
\begin{array}{cccc}
20 & 31 & 24 & 19 & 45 & 56 & 4 & 20 & 5 & 72 & 14 & 99 \\
\text{pivot} & \text{partition} & j
\end{array}
\]

Not yet sorted

these can be in any order

The 20 could be in the other partition

Not yet sorted

these can be in any order

-----

Partition algorithm

Initially, with \( j = h \) and \( t = k \), this diagram looks like the start diagram

\[
\begin{array}{cccc}
 h & j & t & k \\
 \leq x & x & \text{?} & \text{\geq x}
\end{array}
\]

Terminates when \( j = t \), so the \( \text{?} \) segment is empty, so diagram looks like result diagram

\[
\begin{array}{cccc}
 h & j & t & k \\
 \leq x & x & \text{?} & \text{\geq x}
\end{array}
\]

Takes linear time: \( O(k+1-h) \)

-----

QuickSort procedure

/** Sort \([h..k]\). */

\[
\begin{array}{l}
\text{public static void QS(int} b, \text{int} h, \text{int} k) \{ \\
\text{if (} b[h]..k] \text{ has < 2 elements) return; } \quad \text{Base case} \\
\text{int} j=\text{partition}(b, h, k); \\
\text{// We know } b[h..j-1] \leq b[j] \leq b[j+1..k] \\
\text{// Sort } b[h..j-1] \text{ and } b[j+1..k] \\
\text{QS}(b, h, j-1); \\
\text{QS}(b, j+1, k); \\
\} \\
\end{array}
\]

Function does the partition algorithm and returns position \( j \) of pivot
Worst case quicksort: pivot always smallest value

\[ \begin{array}{c|c|c|c} j & x_0 & x_1 & x_2 \\ \hline 0 & \geq x_0 & \geq x_1 & \geq x_2 \\ \end{array} \]

Partioning at depth 0:
\[ \begin{array}{c|c|c|c} 0 & \leq x_0 & x_0 & \geq x_0 \\ \hline j & \leq x_1 & \geq x_1 & \leq x_2 \geq x_2 \\ \end{array} \]

Partioning at depth 1:
\[ \begin{array}{c|c|c|c} x_0 & x_1 & x_2 & \geq x_2 \\ \hline j & \geq x_0 & \leq x_1 & \leq x_2 \geq x_2 \\ \end{array} \]

Partioning at depth 2:
\[ \begin{array}{c|c|c|c} x_0 & x_1 & x_2 & \geq x_2 \\ \hline j & \geq x_0 & \leq x_1 & \leq x_2 \geq x_2 \\ \end{array} \]

Depth of recursion: \( O(n) \)

Processing at depth \( i \): \( O(n-i) \)

\( O(n^2) \)

Best case quicksort: pivot always middle value

\[ \begin{array}{c|c|c|c} j & x_0 & x_1 & x_2 \\ \hline 0 & \leq x_0 & x_0 & \geq x_0 \\ \hline j & \leq x_1 & \geq x_1 & \leq x_2 \geq x_2 \\ \hline \end{array} \]

Partioning at depth 0. 1 segment of size \( n \) to partition.

Depth 1. 2 segments of size \( \leq n/2 \) to partition.

Depth 2. 4 segments of size \( \leq n/4 \) to partition.

Max depth: \( O(\log n) \). Time to partition on each level: \( O(n) \)

Total time: \( O(n \log n) \).

Average time for quicksort: \( n \log n \). Difficult calculation

QuickSort complexity to sort array of length \( n \)

Time complexity

Worst-case: \( O(n^2) \)

Average-case: \( O(n \log n) \)

```
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

Partition. Key issue. How to choose pivot

Choosing pivot

- Ideal pivot: the median, since it splits array in half
- But computing the median is \( O(n) \), quite complicated

Popular heuristics: Use

- first array value (not so good)
- middle array value (not so good)
- Choose a random element (not so good)
- median of first, middle, last, values (often used)

Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ave. time.</th>
<th>Worst-case time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
<td>( O(1) )</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection sort</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
<td>( O(1) )</td>
<td>No</td>
</tr>
<tr>
<td>Quick sort</td>
<td>( O(n \log n) )</td>
<td>( O(n^2) )</td>
<td>( O(\log n) )</td>
<td>No</td>
</tr>
<tr>
<td>Merge sort</td>
<td>( O(n \log n) )</td>
<td>( O(n^2) )</td>
<td>( O(\log n) )</td>
<td>No</td>
</tr>
</tbody>
</table>

* The first algorithm we developed takes space \( O(n) \) in the worst case, but it can be reduced to \( O(\log n) \)

Merge two adjacent sorted segments

```
/** Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted. */
public static merge(int[] b, int h, int t, int k) {
    Copy b[h..t] into a new array c;
    Merge c and b[t+1..k] into b[h..k];
}
```

Merge two adjacent sorted segments

```
/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted. */
public static merge(int[] b, int h, int t, int k) {
    Copy b[h..t] into a new array c;
    Merge c and b[t+1..k] into b[h..k];
}
```

```java
public static merge(int[] b, int h, int t, int k) {
    Copy b[h..t] into a new array c;
    Merge c and b[t+1..k] into b[h..k];
}
```
/* Merge two adjacent sorted segments. Precondition: b[h..t] and b[t+1..k] are sorted. */
public static merge(int[] b, int h, int t, int k) {
    b[h..k] = sorted, sorted
    c[0..] = Place all values in c[0..] and b[t+1..k] into b[h..k] in sorted order
    Step 1. Move 3 from b[t+1] to b[h]
    Step 2. Move 4 from c[0] to b[h+1]
    Step 3. Move 4 from b[t+2] to b[h+2]
Merge two adjacent sorted segments


Invariant: moved ?
to move

In place, sorted

Merge two adjacent sorted segments

// Merge c and b[t+1..k] into b[h..k]

pre: c x y, sorted
post: b h k

Invariant: c length

invariant: 0 head of x tail of x

head of x and head of y, sorted

Merge

i = 0; u = h; v = t+1;
while( i <= t){
if(v < k && b[v] < c[i]) {
b[u] = b[v];
u++; v++;
} else {
b[u] = c[i];
u++; i++;
}
}

Mergesort

/** Sort b[h..k] */
public static void mergesort(int[] b, int h, int k) {
if (size b[h..k] < 2) return;
t = (h+k)/2;
mergesort(b, h, t);
mergesort(b, t+1, k);
merge(b, h, t, k);
}

QuickSort versus MergeSort

/** Sort b[h..k] */
public static void QS(int[] b, int h, int k) {
if (k - h < 1) return;
int j = partition(b, h, k);
QS(b, h, j-1);
QS(b, j+1, k);
}

/** Sort b[h..k] */
public static void MS(int[] b, int h, int k) {
if (k - h < 1) return;
MS(b, h, (h+k)/2);
MS(b, (h+k)/2 + 1, k);
merge(b, h, (h+k)/2, k);
}

Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Ave time.</th>
<th>Worst-case time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>O(1)</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection sort</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>O(1)</td>
<td>No</td>
</tr>
<tr>
<td>Quick sort</td>
<td>O(n log n)</td>
<td>O(n^2)</td>
<td>O(n log n)</td>
<td>No</td>
</tr>
<tr>
<td>Merge sort</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* The first algorithm we developed takes space O(n) in the worst case, but it can be reduced to O(log n)
Sorting in Java

- Java.util.Arrays has a method sort(array)
  - implemented as a collection of overloaded methods
  - for primitives, sort is implemented with a version of quicksort
  - for Objects that implement Comparable, sort is implemented with timSort, a modified mergesort developed in 1993 by Tim Peters
- Tradeoff between speed/space and stability/performance guarantees

QuickSort with logarithmic space

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    int h1 = h; int k1 = k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        Reduce the size of b[h1..k1], keeping inv true
    }
}
```

Tradeoff between speed/space and stability/performance guarantees

QuickSort with logarithmic space

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    int h1 = h; int k1 = k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        int j = partition(b, h1, k1);
        // b[h1..j-1] <= b[j] <= b[j+1..k1]
        if (b[h1..j-1] smaller than b[j+1..k1])
            QS(b, h, j-1); h1= j+1;
        else
            QS(b, j+1, k1); k1= j-1;
    }
}
```

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively. We may show you this later. Not today!