"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

pre: b\[0..\text{b.length}\] ?
post: b \[0..\text{b.length}\] sorted

inv: b\[0..i\] is sorted
or: b\[0..i-1\] is processed

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?

\[
\begin{array}{llllll}
0 & & i & & b.\text{length} \\
\text{inv:} & b & \text{sorted} & ? \\
\text{e.g.} & b & 2 & 5 & 5 & 5 & 7 & 3 & ? \\
& b & 2 & 3 & 5 & 5 & 5 & 7 & ? \\
\end{array}
\]
What to do in each iteration?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>i</td>
<td>b.length</td>
</tr>
</tbody>
</table>

**inv:**

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

**e.g.**

<table>
<thead>
<tr>
<th>()</th>
<th>()</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Loop body**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

This will take time proportional to the number of swaps needed

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]

    Present algorithm like this
}

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
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]
    int k = i;
    while (k > 0 && b[k] < b[k-1]) {
        Swap b[k] and b[k-1];
        k = k–1;
    }
}

invariant P:  b[0..i] is sorted
except that b[k] may be < b[k-1]

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

example

start?
stop?
progress?
maintain
invariant?
Insertion Sort

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 0; i < b.length; i= i+1) {
    // Push b[i] down to its sorted
    // position in b[0..i]}

Pushing b[i] down can take i swaps.
Worst case takes
1 + 2 + 3 + … n-1 = (n-1)*n/2
swaps.

Let n = b.length

• Worst-case: O(n^2)
  (reverse-sorted input)
• Best-case: O(n)
  (sorted input)
• Expected case: O(n^2)
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>
</tbody>
</table>
SelectionSort

pre: b
post: b sorted

inv: b sorted, \(<= b[i..] \geq b[0..i-1]\)

Additional term in invariant

Keep invariant true while making progress?

e.g.: b

Increasing i by 1 keeps inv true only if b[i] is min of b[i..]
SelectionSort

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 $O(n^2)$

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

Each iteration, swap min value of this section into b[i]
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 = i+1) {
    int m = index of min of b[i..];
    Swap b[i] and b[m];
}

Here, swapping b[i] with the minimum of b[i..], 3, changes the relative position of the two 8’s.
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>
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 Algol 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, Hoare's, Grieses, 1980s
Partition algorithm of quicksort

pre:

$$\begin{array}{c|c}
h & h+1 \\
\hline
x & ? \\
k
\end{array}$$

post:

$$\begin{array}{c|c|c}
<= x & x & >= x \\
\end{array}$$

x is called the pivot

Swap array values around until b[h..k] looks like this:
Partition algorithm of quicksort

Partition algorithm of quicksort

pivot

partition

j

Not yet sorted

these can be in any order

these can be in any order

The 20 could be in the other partition
Partition algorithm

\[ h \quad h+1 \quad k \]

\text{pre:} \quad b \quad x \quad ?

\[ h \quad j \quad k \]

\text{post:} \quad b \quad \leq x \quad x \quad \geq x

Combine pre and post to get an invariant

\[ h \quad j \quad t \quad k \]

\[ b \quad \leq x \quad x \quad ? \quad \geq x \]

\text{invariant needs at least 4 sections}
Partition algorithm

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

Terminate when \( j = t \), so the “?” segment is empty, so diagram looks like result diagram.

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

/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;  // Base case!

    int j = partition(b, h, k);
    // We know b[h..j−1] <= b[j] <= b[j+1..k]
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}

\[
\begin{array}{c|c|c}
  h & j & k \\
  \hline
  \leq x & x & \geq x \\
\end{array}
\]
Worst case quicksort: pivot always smallest value

/** 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);
}

Depth of recursion: $O(n)$

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

$O(n^2)$
Best case quicksort: pivot always middle value

0 \leq x_0 \leq x_1 \leq x_2 \leq x_0 \leq x_2 \leq x_0

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 \( n \)

```java
/** 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);
    // We know b[h..j-1] <= b[j] <= b[j+1..k]
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

Time complexity
- Worst-case: \( O(n^2) \)
- Average-case: \( O(n \log n) \)

Worst-case space: ?
What’s depth of recursion?

Worst-case space: \( O(n) \)!
--depth of recursion can be \( n \)

Can rewrite it to have space \( O(\log n) \)
Show this at end of lecture if we have time
Partition. Key issue. How to choose pivot

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)!

Choosing pivot

Ideal pivot: the median, since it splits array in half

But computing the median is \(O(n)\), quite complicated
## 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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

sorted                    sorted
h                     t                        k

merged,   sorted
h                                              k
/* 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) {
    
}
Merge two adjacent sorted segments

Place all values in c[0..] and b[t+1..k] into b[h..k] in sorted order

0 t

\[\begin{array}{cccc}
4 & 7 & 7 & 8 & 9 \\
\end{array}\]

\[\begin{array}{ccc}
h & t & k \\
\end{array}\]

\[\begin{array}{ccc}
h & t & k \\
3 & 4 & 4 & 7 & 7 & 7 & 8 & 8 & 9 \\
\end{array}\]
Merge two adjacent sorted segments

Place all values in c[0..] and b[t+1..k] into b[h..k] in sorted order
Merge two adjacent sorted segments

Step 1. Move 3 from $b[t+1]$ to $b[h]$
Merge two adjacent sorted segments

Step 2. Move 4 from c[0] to b[h+1]
Merge two adjacent sorted segments

Step 3. Move 4 from $b[t+2]$ to $b[h+2]$
Merge two adjacent sorted segments

<table>
<thead>
<tr>
<th>c</th>
<th>?</th>
<th>?</th>
<th>7</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Step 4. Move 7 from c[1] to b[h+3]

Invariant:

<table>
<thead>
<tr>
<th>c</th>
<th>moved</th>
<th>still to move</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>In place, sorted</td>
<td>still to move</td>
</tr>
</tbody>
</table>

if i < c.length and h < u,

- b[u-1] ≤ c[i]

if v ≤ k and h < u,

- b[u-1] ≤ b[v]
Merge two adjacent sorted segments

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

pre: c \(0 .. t-h\) b \(h .. t\) c.length

post: b \(h .. k\) x and y, sorted

invariant: c \(0 .. i\) b \(h .. v \) tail of y

head of x and head of y, sorted

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

/** Sort b[h..k] */

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

```java
/** 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);
}
```

One processes the array then recurses.
One recurses then processes the array.
* 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

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!
QuickSort with logarithmic space

/** 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
    }
}
QuickSort with logarithmic space

/** 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; }
    }
}