"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

- Tonight!!!!
- Two Sessions:
  - You should know by now what room to take the final. Jenna emailed you.
- Bring your Cornell ID!!!
- We will grade this evening, and if everything works out well, you will receive an email in early morning from Gradescope telling you to look at your grade.
Recitation 5. next week:
Enums and Java Collections classes.
Nothing to prepare for it!

But get A3 done.
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)
Some Sorting Algorithms

- Insertion sort
- Selection sort
- Quick sort
- Merge sort
### InsertionSort

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pre:</td>
<td>(0)</td>
<td>(b)</td>
<td>(?)</td>
<td>(b)</td>
</tr>
<tr>
<td>post:</td>
<td>(0)</td>
<td>(b)</td>
<td>(\text{sorted})</td>
<td>(b)</td>
</tr>
</tbody>
</table>

**inv:**
- \(0\) \(\leq i \leq b\).length
- \(b[0..i-1]\) 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?

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv:</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sorted</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>e.g.</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 5 5 5 7</td>
<td>3</td>
<td>?</td>
</tr>
</tbody>
</table>

Loop body (inv true before and after)

- 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
// 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
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]
  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>k</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

2 5 3 5 5 7 ?

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

\[1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2}\]

swaps.

Let \(n = b.length\)

- Worst-case: \(O(n^2)\) (reverse-sorted input)
- Best-case: \(O(n)\) (sorted input)
- Expected case: \(O(n^2)\)
# 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>Merge Sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We’ll talk about stability later
SelectionSort

**pre:** $b \quad ? \quad b.length$

**post:** $b \quad \text{sorted} \quad b.length$

**inv:** $b \quad \text{sorted, } \leq b[i..] \quad \geq b[0..i-1]$

Additional term in invariant

Keep invariant true while making progress?

**e.g.:**

$0 \quad i \quad b.length$

$\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 9 & 9 & 9 & 7 & 8 & 6 & 9 \\
\end{array}$

Increasing $i$ by 1 keeps inv true only if $b[i]$ is min of $b[i..]$
Another common way for people to sort cards

Runtime with \( n = b.\text{length} \)
- Worst-case \( O(n^2) \)
- Best-case \( O(n^2) \)
- Expected-case \( O(n^2) \)

//sort b[], an array of int
// 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]
## 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge sort</td>
<td></td>
<td></td>
<td></td>
<td></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, Hoares, Grieses, 1980s
Partition algorithm of quicksort

Swap array values around until $b[h..k]$ looks like this:

<table>
<thead>
<tr>
<th>h</th>
<th>h+1</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

pre:

post:

$\leq x$  $x$  $\geq x$

x is called the pivot
Partition algorithm of quicksort

| 20 | 31 | 24 | 19 | 45 | 56 | 4 | 20 | 5 | 72 | 14 | 99 |

pivot

| 19 | 4 | 5 | 14 | 20 | 31 | 24 | 45 | 56 | 20 | 72 | 99 |

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

Not yet sorted
Partition algorithm

pre: \[ b \begin{array}{c|c|c} h & x & ? \\ \hline h & j & k \end{array} \]

post: \[ b \begin{array}{c|c|c} <= x & x & >= x \\ \hline \end{array} \]

Combine pre and post to get an invariant

\[ b \begin{array}{c|c|c|c|c|c} h & j & t & k \\ \hline <= x & x & ? & >= x \end{array} \]

invariant needs at least 4 sections
Partition algorithm

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

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

\( j = h; t = k; \)

while (\( j < t \)) {
    if (\( b[j+1] \leq b[j] \)) {
        Swap \( b[j+1] \) and \( b[j] \);  \( j = j+1; \)
    } else {
        Swap \( b[j+1] \) and \( b[t] \);  \( t = t-1; \)
    }
}

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]. */

gpublic 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);
}

h     j     k
| <= x | x | >= x |
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*n)
**Best case quicksort: pivot always middle value**

<table>
<thead>
<tr>
<th>0</th>
<th>j</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\leq x_0)</td>
<td>(x_0)</td>
<td>(\geq x_0)</td>
</tr>
</tbody>
</table>

Depth 0. 1 segment of size \(\sim n\) to partition.

| \(\leq x_1\) | \(x_1\) | \(\geq x_1\) | \(x_0\) | \(\leq x_2\) | \(x_2\) | \(\geq x_2\) |

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

Depth 3. 4 segments of size \(\sim n/4\) 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

/** 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*n)
Average-case: O(n log n)

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

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></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];
}
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) {
}

\[
\begin{array}{cccc}
  h & t & k \\
  4 & 7 & 7 & 8 & 9 & 3 & 4 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccc}
  h & t & k \\
  \text{sorted} & \text{sorted} \\
\end{array}
\]

\[
\begin{array}{cccc}
  h & t & k \\
  3 & 4 & 4 & 7 & 7 & 7 & 8 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccc}
  h & k \\
  \text{merged, sorted} \\
\end{array}
\]
Merge two adjacent sorted segments

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

```plaintext
pre:   c [0..t-h] x b [h..t] ? y k
post: b [h..k] x and y, sorted
```

```
invariant: c [0..c.length] | head of x | tail of x
```

```
b [h..u] ? v [u..k] tail of y
```

head of x and head of y, sorted
int i = 0;
int u = h;
int 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++;
    }
}

pre: c_{\text{sorted}} \quad b_{\text{?}} \quad c_{\text{sorted}}$

post: b_{\text{sorted}}$

inv: 0 \quad i \quad c.\text{length}$

\begin{align*}
\text{c} & & \text{sorted} & & \text{sorted} \\
\text{b} & & \text{sorted} & & \text{?} & & \text{sorted} \\
\end{align*}
Mergesort

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

<table>
<thead>
<tr>
<th>h</th>
<th>t</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td></td>
<td>sorted</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>merged, sorted</td>
<td></td>
</tr>
</tbody>
</table>
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.
### 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>$O(n \log n)$</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 $O(\log n)$
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