"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.

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

```
pre: b[0..i-1] is sorted
inv: b[0..i] is processed
or: b[0..i-1] is sorted
post: b[0..i] is sorted
```
Each iteration, \( i = i+1 \); How to keep \( \text{inv} \) true?

<table>
<thead>
<tr>
<th>( \text{inv} )</th>
<th>( b )</th>
<th>( \text{sorted} )</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_0 )</td>
<td>( i )</td>
<td>( b.\text{length} )</td>
<td>?</td>
</tr>
<tr>
<td>( b )</td>
<td>( 2 , 5 , 5 , 5 , 7 )</td>
<td>( 3 )</td>
<td>?</td>
</tr>
</tbody>
</table>

e.g. \( b_0 \), \( i \), \( b.\text{length} \)

<table>
<thead>
<tr>
<th>( b )</th>
<th>( \text{sorted} )</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_0 )</td>
<td>( i )</td>
<td>( b.\text{length} )</td>
</tr>
<tr>
<td>( b )</td>
<td>( 2 , 3 , 5 , 5 , 5 )</td>
<td>?</td>
</tr>
</tbody>
</table>

What to do in each iteration?

<table>
<thead>
<tr>
<th>( \text{inv} )</th>
<th>( b )</th>
<th>( \text{sorted} )</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_0 )</td>
<td>( i )</td>
<td>( b.\text{length} )</td>
<td>?</td>
</tr>
<tr>
<td>( b )</td>
<td>( 2 , 5 , 5 , 5 , 7 )</td>
<td>( 3 )</td>
<td>?</td>
</tr>
</tbody>
</table>

Loop body (inv true before and after)

<table>
<thead>
<tr>
<th>( b )</th>
<th>( i )</th>
<th>( b.\text{length} )</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_0 )</td>
<td>( i )</td>
<td>( b.\text{length} )</td>
<td>?</td>
</tr>
<tr>
<td>( b )</td>
<td>( 2 , 3 , 5 , 5 , 5 )</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

This will take time proportional to the number of swaps needed

<table>
<thead>
<tr>
<th>( b )</th>
<th>( \text{sorted} )</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_0 )</td>
<td>( i )</td>
<td>( b.\text{length} )</td>
</tr>
<tr>
<td>( b )</td>
<td>( 2 , 5 , 5 , 5 )</td>
<td>( 3 )</td>
</tr>
</tbody>
</table>

Insertion Sort

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

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

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

Let \( n = b.\text{length} \)

- Worst-case: \( O(n^2) \) (reverse-sorted input)
- Best-case: \( O(n) \) (sorted input)
- Expected case: \( O(n^2) \)

Pushing \( b[i] \) down can take \( i \) swaps.

Worst case takes

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

swaps.

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

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

inv: b sorted, <= b[i..] => b[0..i-1]  
Additional term in invariant

Keep invariant true while making progress?

e.g.: b

[1 2 3 4 5 6 9 9 9 7 8 6 9]

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 $O(n^2)$

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>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge 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.

Partition algorithm of quicksort

<table>
<thead>
<tr>
<th>pre:</th>
<th>post: &lt;= x x &gt;= x</th>
</tr>
</thead>
<tbody>
<tr>
<td>h h+1</td>
<td>k x is called the pivot</td>
</tr>
</tbody>
</table>

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

h j k
Partition algorithm of quicksort

\[
\begin{array}{cccccc}
20 & 31 & 24 & 19 & 45 & 56 \\
\end{array}
\]

\[
\begin{array}{cccccc}
31 & 24 & 19 & 45 & 56 & 20 \\
\end{array}
\]

pivot: 20

partition: 31, 24, 19, 45, 56, 20

Not yet sorted

The 20 could be in the other partition

Not yet sorted

these can be in any order

these can be in any order

Partition algorithm

\[
\begin{align*}
h & \leq x & x & \geq x \\
\end{align*}
\]

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

\[
/** \text{Sort } b[h..k]. */
\]

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

\[
\begin{align*}
h & \leq x & x & \geq x \\
\end{align*}
\]

Function does the partition algorithm and returns position \( j \) of pivot

Worst case quicksort: pivot always smallest value

\[
\begin{bmatrix}
\text{x0} & \geq \text{x0} \\
\text{x0} & \geq \text{x1} \\
\text{x0} & \geq \text{x2} \\
\end{bmatrix}
\]

partioning at depth 0

\text{Processing at depth } i: O(n^i)

\text{Depth of recursion: } O(n)

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

\text{Average time for Quicksort: } n \log n. \text{ Difficult calculation}

Best case quicksort: pivot always middle value

\[
\begin{bmatrix}
\text{0} & \leq \text{x0} & \geq \text{x0} \\
\text{x1} & \geq \text{x0} & \leq \text{x1} \\
\text{x1} & \leq \text{x2} & \geq \text{x2} \\
\end{bmatrix}
\]

depth 0. 1 segment of size ~n to partition.

Depth 2. 2 segments of size ~n/2 to partition.

Depth 3. 4 segments of size ~n/4 to partition.

\text{Total time: } O(n \log n). \text{ 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);  // Worst-case space: ?
  QS(b, j+1, k);  // What’s depth of recursion?
  --depth of recursion can be n
  Show this at end of lecture if we have time
}

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(n²)</td>
<td>O(n²)</td>
<td>O(1)</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection sort</td>
<td>O(n²)</td>
<td>O(n²)</td>
<td>O(1)</td>
<td>No</td>
</tr>
<tr>
<td>Quick sort</td>
<td>O(n log n)</td>
<td>O(n²)</td>
<td>O(n 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)

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)

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) {
  // Merge sorted c and b[t+1..k] into b[h..k]
  b[0..h-1] = head of x;  // x, y are sorted
  b[h..t] = y;
  b[t+1..k] = c;  // c.length
  h = head of x;  // head of x and head of y, sorted
  u = tail of x;
  v = tail of y;
  k = ...
}
Merge

```java
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++;
    }
}
```

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);
}
```

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);
}
```

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)

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