"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

- It's on Tuesday Evening (3/13)
- Two Sessions:
  - 5:30-7:00PM: netid aa..ks
  - 7:30-9:00PM: netid kt..zz
  - If you have a conflict with your assigned time but can make the other time, fill out conflict assignment on CMS BY TOMORROW
- Three Rooms:
  - We will email you Tuesday morning with your room
- Bring your Cornell ID!!!
Recitation 5: prelim review

Review Session: Sunday 3/11, 1-3pm in Kimball B11

Study guide on course website
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 about several different algorithms (and how to derive them)
Some Sorting Algorithms

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

pre: b 0 \(\mathbf{?}\) b.length

post: b 0 \(\mathbf{\text{sorted}}\) b.length

inv: b 0 \(\mathbf{\text{sorted}}\) i \(\mathbf{?}\) b.length

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

A loop that processes elements of an array in increasing order has this invariant
Each iteration, $i = i + 1$; How to keep $\text{inv}$ true?

<table>
<thead>
<tr>
<th></th>
<th>$i$</th>
<th>$\text{b.length}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{inv}$:</td>
<td>$\text{b}$</td>
<td>sorted</td>
</tr>
<tr>
<td>0</td>
<td>$i$</td>
<td>$\text{b.length}$</td>
</tr>
<tr>
<td>$\text{e.g.}$</td>
<td>$\text{b}$</td>
<td>2 5 5 5 7</td>
</tr>
<tr>
<td>0</td>
<td>$i$</td>
<td>$\text{b.length}$</td>
</tr>
<tr>
<td>$\text{b}$</td>
<td>2 3 5 5 5 7</td>
<td>?</td>
</tr>
</tbody>
</table>
**What to do in each iteration?**

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

Loop body (inv true before and after)

- 2 5 5 5 3 7 ?
- 2 5 5 3 5 7 ?
- 2 5 3 5 5 7 ?
- 2 3 5 5 5 7 ?

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

| 2 | 3 | 5 | 5 | 5 | 7 | ? |
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]
  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]
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]
}

Let n = b.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>Time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$ to $O(n^2)$</td>
<td>$O(1)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SelectionSort

pre: \[ b \]

post: \[ b \text{ sorted} \]

inv: \[ b \]

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

//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 minimum 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)

Each iteration, swap min value of this section into b[i]
## Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n) \text{ to } O(n^2)$</td>
<td>$O(1)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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) {
}

\[
\begin{array}{cccc|c}
  h & t & k & \hline 
  4 & 7 & 7 & 8 & 9 & 3 & 4 & 7 & 8 \\
  \hline 
  \text{sorted} & & \text{sorted} \\
\end{array}
\]

\[
\begin{array}{cccc|c}
  h & t & k & \hline 
  3 & 4 & 4 & 7 & 7 & 7 & 8 & 8 & 9 \\
  \hline 
  \text{merged, sorted} & & \text{sorted} \\
\end{array}
\]
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];
}

<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>

merged, sorted
Merge two adjacent sorted segments

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

pre:     c  x  b  ?  y  x, y are sorted

post: b  x and y, sorted

invariant: c  head of x  tail of x

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 \begin{array}{|c|c|c|c|}
\hline
0 & t-h & h & t \hline
\end{array}
\begin{array}{c}
sorted \end{array}
\begin{array}{c}
sorted \end{array}

post: b \begin{array}{|c|c|c|c|}
\hline
h & t & k \hline
\end{array}
\begin{array}{c}
sorted \end{array}

inv: \begin{array}{|c|c|c|c|}
\hline
0 & i & c.length \hline
\end{array}
\begin{array}{|c|c|}
sorted & sorted \end{array}

\begin{array}{|c|c|c|c|}
\hline
h & u & v & k \hline
\end{array}
\begin{array}{|c|}
sorted \end{array}
\begin{array}{|c|}
? \end{array}
\begin{array}{c}
sorted \end{array}
/** 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);
}
## Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$ to $O(n^2)$</td>
<td>$O(1)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log(n)$</td>
<td>$O(n)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick Sort</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).

83 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

**pre:**

<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>

x is called the pivot

**post:**

| <= x | x | >= x |

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

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

pivot 20 31 24 19 45 56 4 20 5 72 14 99 partition j
Partition algorithm

pre: $b \begin{array}{|c|c|c|} \hline h & x & ? \hline \end{array}$

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

Combine pre and post to get an invariant

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

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]. */

```java
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);
}
```

<table>
<thead>
<tr>
<th>h</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= x</td>
<td>x</td>
<td>&gt;= x</td>
</tr>
</tbody>
</table>
Worst case quicksort: pivot always smallest value

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

```java
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**

depth 0. 1 segment of size $\sim n$ to partition.

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: ?
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
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);
}

One processes the array then recurses.
One recurses then processes the array.
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 is $O(n)$, quite complicated
## Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$ to $O(n^2)$</td>
<td>$O(1)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log(n)$</td>
<td>$O(n)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n \log(n)$ to $O(n^2)$</td>
<td>$O(\log(n))$</td>
<td>No</td>
</tr>
</tbody>
</table>
Java.util.Arrays has a method Sort() 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 mergesort.

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; }
    }
}