Last Time

• Bottom-Up Mergesort
  – Same idea as Top-Down Merge Sort, except skipping the recursion

• The best sorting we can do (with comparisons)
Quicksort

```java
public static void sort(Comparable a[]) {
    StdRandom.shuffle(a);
    sort(a, 0, a.length-1);
} //sort1

private static void sort(Comparable a[], int lo, int hi) {
    if (hi <= lo) {
        return;
    } //if
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
} //sort
```
Quicksort

private static int partition(Comparable a[], int lo, int hi) {
    int i = lo, j = hi+1;
    Comparable v = a[lo];
    while (true) {
        while (less(a[++i], v)) {
            if (i == hi) {
                break;
            }
        } //if
    } //while
    while (less(v, a[--j])) {
        if (j == lo) {
            break;
        }
    } //if
    } //while
    if (i >= j) {
        break;
    } //if
    exch(a, i, j);
} //while
    exch(a, lo, j);
    return j;
} //sort
Quicksort Partition Visual

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td>a[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

initial values
0     16     K R A T E E L P U I M Q C X O S

scan left, scan right
1     12     K R A T E E L P U I M Q C X O S

exchange
1     12     K C A T E E L P U I M Q R X O S

scan left, scan right
3     9      K C A T E E L P U I M Q R X O S

exchange
3     9      K C A I E E L P U T M Q R X O S

scan left, scan right
5     6      K C A I E E L P U T M Q R X O S

exchange
5     6      K C A I E E L P U T M Q R X O S

scan left, scan right
6     5      K C A I E E L P U T M Q R X O S

final exchange
6     5      E C A I E E K L P U T M Q R X O S

result
5     6      E C A I E E K L P U T M Q R X O S
```
### Quicksort Sort Visual

#### Initial values

<table>
<thead>
<tr>
<th></th>
<th>j</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

#### Random shuffle

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q</td>
<td>U</td>
<td>I</td>
<td>C</td>
<td>K</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>Q</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>Q</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>14</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>15</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

#### No partition for subarrays of size 1

#### Result

|   | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |

02/12/2016
Quicksort vs. Mergesort

• In Mergesort, we always split the array in half (as best we could). In Quicksort, we split the array depending on input.
  – Makes sense that this would improve things – worry about what the input we’re sorting is rather than making it arbitrary.

• In Mergesort, we did our recursive calls before we touched the whole array in the same operation. In Quicksort, our recursive calls come after the whole array is partitioned.
  – This also seems like an improvement – instead of merging things that are far apart, let’s partially order the array first.
Why Shuffle the Input?

- Quicksort is a **randomized** algorithm.
  - After each `partition()` call, each subarray is in what is essentially a random order.
  - This random order turns out to be important in predicting the run time of Quicksort.
  - It then follows that we want to select keys randomly. We could either shuffle the array at the beginning, or we could pick a random key from the input instead of always picking the first key.
Quicksort Performance Characteristics

- Inner partition loop increments an index and compares an array entry against a fixed value. Mergesort and Shell Sort also do data movement in their inner loops.

- Quicksort doesn’t use many compares – the efficiency of the sort depends on how well the data is partitioned into subarrays, which hence depends on the choice of keys.
  
  - **Best case:** Each partitioning stage splits the array perfectly in half.  
    \[ C(N) = 2C\left(\frac{N}{2}\right) + N = O(n \times \log(n)) \]
  
  - **Worst case:** Each partitioning stage picks the worst possible key, so that every data item needs to be exchanged. *(what’s this complexity?)*
Quicksort Performance Characteristics

• Wait, so the best case of Quicksort is the average case of Mergesort. How is this better?
  – Mergesort used $n \log(n)$ compares and $6n \log(n)$ array accesses.
  – Quicksort uses $2n \log(n)$ compares and $\frac{1}{3}n \log(n)$ exchanges.
Quicksort Improvements

• Cutoff to Insertion Sort
• “Median-of-Three Partitioning” – Pick a few random items from the subarray, take the median, and use that as the pivot.
• “Entropy-Optimal Sorting” – In arrays with large numbers of duplicates, we’ll run into subarrays that don’t need to be sorted. Partition into three pieces – keys less than, keys greater than, and keys equal to the pivot.

• Quicksort is widely used today because it outperforms all other sorting algorithms in “practical applications.”
Any Questions?