Last Time

• Sorting
  – Selection Sort
    • Good for a system where exchanges are expensive
  – Insertion Sort
    • Good for data that is already mostly sorted
Mergesort

• Basic idea: Split an array into two halves, sort them, and then merge them back into a single sorted array.
  – (How do we sort each half?)
Naïve Merge

......

......
public void merge(Comparable[] a, int lo, int mid, int hi) {
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    } //for
    for (int k = lo; k <= hi; k++) {
        if (i > mid) {
            a[k] = aux[j++];
        } else if (j > hi) {
            a[k] = aux[i++];
        } else if (less(aux[j], aux[i])) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        } //if-else
    } //for
} //merge
public void merge(Comparable[] a, int lo, int mid, int hi) {
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    } //for
    for (int k = lo; k <= hi; k++) {
        if (i > mid) {
            a[k] = aux[j++];
        } else if (j > hi) {
            a[k] = aux[i++];
        } else if (less(aux[j], aux[i])) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        } //if-else
    } //for
} //merge
Merge Visual

<table>
<thead>
<tr>
<th>k</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>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>copy</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>aux[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>E E G M R A C E R T</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>E E G M R A C E R T</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>E E G M R C E R T</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>E E G M R E R T</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>E G M R E R T</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>G M R E R T</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>G M R E R T</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>M R E R T</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>R E R T</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>R E R T</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>T E R T</td>
</tr>
</tbody>
</table>

merged result: A C E E E G M R R T
Merge

- Still uses extra space, but not a substantial amount more.
- We only need to allocate memory for aux once; then we can continually overwrite it as we work through the data.
- The merge function actually handles the sort! Now we just need to structure our merge() calls:
  - Top-down mergesort
  - Bottom-up mergesort
private static Comparable[] aux;

public static void sort(Comparable[] a) {
    aux = new Comparable[a.length];
    sort(a, 0, a.length-1);
} //sort

private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) {
        return;
    } //if
    int mid = lo + (hi - lo) / 2;
    sort(a, lo, mid);
    sort(a, mid+1, hi);
    merge(a, lo, mid, hi);
} //sort
Top-Down Mergesort Evaluation

• This is called a **divide-and-conquer** algorithm – it recursively breaks down the problem into 2+ subproblems of the same (or related) type, until they become simple enough to solve directly.

• Think of it as a proof by induction in code form – we can sort a simple array, and then we can sort a complex array from a previous pair of arrays.
Top-Down Mergesort Visual
Top-Down Mergesort Evaluation

• How many compares does this algorithm perform?
  – $N \times \log(N)$

• How many exchanges does this algorithm perform?
  – Well... we don’t actually do exchanges. Instead, we do array accesses.
  – Each merge uses $2N$ for the copy, $2N$ for the move back, and at most $2N$ for compares.
  – $6N \times \log(N)$
Top-Down Mergesort Improvements

• Now we can sort in $N \times \log(N)$ time – a substantial improvement over the $N^2$ time of Insertion and Selection Sorts.

• We know that Insertion Sort is efficient for small arrays, so if we switch to Insertion Sort once the problem is broken down below some threshold, we can improve Mergesort by 10-15%.
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 in partitioned.
  – This also seems like an improvement – instead of merging things that are far apart, let’s partially order the array first.
Quicksort

```java
public static void sort(Comparable a[]) {
    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>i</th>
<th>j</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>0</td>
<td>16</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>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>1</td>
<td>12</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>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>3</td>
<td>9</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</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>R</td>
<td>X</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>3</td>
<td>9</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</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>R</td>
<td>X</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>5</td>
<td>6</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>5</td>
<td>6</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>6</td>
<td>5</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>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>6</td>
<td>5</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>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>5</td>
<td></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>
</tr>
</tbody>
</table>

**Initial values:**
- The partition process starts with the initial values of the array.

**Scan left, scan right:**
- The values are scanned from both ends to find the pivot element.

**Exchange:**
- Elements are exchanged based on their order relative to the pivot.

**Result:**
- The final sorted array after the partitioning process.
**Quicksort Sort Visual**

<table>
<thead>
<tr>
<th>l0</th>
<th>j</th>
<th>hi</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></td>
<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></td>
<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>5</td>
<td>15</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>3</td>
<td>4</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>2</td>
<td>2</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>0</td>
<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>1</td>
<td>1</td>
<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>4</td>
<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>6</td>
<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>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>9</td>
<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>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<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>8</td>
<td>8</td>
<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>13</td>
<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>S</td>
<td>Q</td>
<td>R</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>12</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>11</td>
<td>11</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>10</td>
<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>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<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>
<tr>
<td>15</td>
<td>15</td>
<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>

**Initial values**
- Random shuffle

**Result**

```
ACEEIKLMOPQRSTUVWXYZ
```
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. $O(n \log n)$
  
  – **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 \( n \times \log(n) \) compares and \( 6n \times \log(n) \) array accesses.
  – Quicksort uses \( 2n \times \log(n) \) compares and \( \frac{1}{3}n \times \log(n) \) exchanges.
Any Questions?