Announcements

- **Lab 5 Today**
  - Submit partners!
  - Challenging, but shorter and a partner lab – more time for exam prep!
- **Mid-term exam is Wednesday, March 14**
  - During your normal lab session
  - You’ll have approximately 1 hour & 45 minutes (if you come on time!)
  - Closed-book: Covers Chapters 1-7 & 9, handouts, and all topics up through Sorting
  - A “sample” mid-term **and** study sheet will be available online
Last Time

- Basic Sorting Summary
- Comparator interfaces for flexible sorting
- More Efficient Sorting Algorithms
  - MergeSort
Today

• Sorting Wrap-Up (Merge and Quick)
• Linear Structures
  • The Linear Interface (LIFO & FIFO)
  • The AbstractLinear and AbstractStack classes
• Stack Implementations
  • StackArray, StackVector, StackList,
• Stack applications
  • Expression Evaluation
  • PostScript: Page Description & Programming
  • Mazerunning (Depth-First-Search)
Merge Sort

- A **divide and conquer** algorithm
- Merge sort works as follows:
  - **Base case:**
    - If the list is of length 0 or 1, then it is already sorted. Return the sorted list.
  - Divide the unsorted list into two sublists of about half the size of original list.
  - **Recursive call:**
    - Sort each sublist by re-applying merge sort.
  - Merge the two sublists back into one sorted list.
Merge Sort

• [8 14 29 1 17 39 16 9]
• [8 14 29 1] [17 39 16 9] split
• [8 14] [29 1] [17 39] [16 9] split
• [8] [14] [29] [1] [17] [39] [16] [9] split
• [8 14] [1 29] [17 39] [9 16] merge
• [1 8 14 29] [9 16 17 39] merge
• [1 8 9 14 16 17 29 39] merge

Transylvanian Merge Sort Folk Dance
Merge Sort

• How would we implement it?

• Pseudocode:

```c
//recursively mergesorts A[from..To] “in place”
void recMergeSortHelper(A[], int from, int to)
    if (from < to)
        // find midpoint
        mid = (from + to)/2
        //sort each half
        recMergeSortHelper(A, from, mid)
        recMergeSortHelper(A, mid+1, to)
        // merge sorted lists
        merge(A, from, to)
```

But `merge` hides a number of important details…. 7
Merge Sort

• How would we implement it?
  • Review MergeSort.java
  • Note carefully how temp array is used to reduce copying
  • Make sure the data is in the correct array!

• Time Complexity?
  • Takes at most $2k$ comparisons to merge two lists of size $k$
  • Number of splits/merges for list of size $n$ is $\log n$
  • Claim: At most time $O(n \log n)$...We’ll see soon...

• Space Complexity?
  • $O(n)$?
  • “Clever” implementation “ping-pongs” between 2 arrays
    • Need an extra array, so really $O(2n)$!
    • But $O(2n) = O(n)$
Merge Sort = $O(n \log n)$

- $[8 \ 14 \ 29 \ 1 \ 17 \ 39 \ 16 \ 9]$
- $[8 \ 14 \ 29 \ 1] \ [17 \ 39 \ 16 \ 9] \ \text{split}$
- $[8 \ 14] \ [29 \ 1] \ [17 \ 39] \ [16 \ 9] \ \text{split}$
- $[8 \ 14] \ [1 \ 29] \ [17 \ 39] \ [9 \ 16] \ \text{merge}$
- $[1 \ 8 \ 14 \ 29] \ [9 \ 16 \ 17 \ 39] \ \text{merge}$
- $[1 \ 8 \ 9 \ 14 \ 16 \ 17 \ 29 \ 39] \ \text{merge}$

merge takes at most $n$ comparisons per line
Merge Sort

- Unlike Bubble, Insertion, and Selection sort, Merge sort is a divide and conquer algorithm
  - Bubble, Insertion, Selection sort: $O(n^2)$
  - Merge sort: $O(n \log n)$
- Are there any problems or limitations with Merge sort?
- Why would we ever use any other algorithm for sorting?
Problems with Merge Sort

• Need extra temporary array
  • If data set is large, this could be a problem
• Waste time copying values back and forth between original array and temporary array
• Can we avoid this?
Quick Sort

- Quick sort is designed to behave much like Merge sort, without requiring extra storage space.

<table>
<thead>
<tr>
<th>Merge Sort</th>
<th>Quick Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide list in half</td>
<td>Partition* list into 2 parts</td>
</tr>
<tr>
<td>Sort halves</td>
<td>Sort parts</td>
</tr>
<tr>
<td>Merge halves</td>
<td>Join* sorted parts</td>
</tr>
</tbody>
</table>
private static void mergeSortRecursive(Comparable data[],
        Comparable temp[], int low, int high) {
    int n = high - low + 1;
    int middle = low + n/2;

    if (n < 2) return; // already sorted

    // move lower half of data into temporary storage
    for (int i = low; i < middle; i++)
        temp[i] = data[i];

    // sort lower half of array
    mergeSortRecursive(temp, data, low, middle-1);

    // sort upper half of array
    mergeSortRecursive(data, temp, middle, high);

    // merge halves together
    merge(data, temp, low, middle, high);
}
Quick Sort

// pre: low <= high
// post: data[low..high] in ascending order
public static void quickSortRecursive(Comparable data[], int low, int high) {

    int pivot;

    // base case: low and high coincide
    if (low >= high) return;

    // step 1: split using pivot
    pivot = partition(data, low, high);
    // step 2: sort small
    quickSortRecursive(data, low, pivot-1);
    // step 3: sort large
    quickSortRecursive(data, pivot+1, high);
}
Partition

1. Put first element (pivot) into sorted position
2. When done, all to the left of pivot are smaller and all to the right are larger
3. Return index of pivot

Partition by Hungarian Folk Dance
**Partition**

```c
int partition(int data[], int left, int right) {
    while (true) {
        while (left < right && data[left] < data[right])
            right--;

        if (left < right)
            swap(data, left++, right);
        else
            return left;

        while (left < right && data[left] < data[right])
            left++;

        if (left < right)
            swap(data, left, right--);
        else
            return right;
    }
}
```
Figure 6.7
The partitioning of an array's values based on the (shaded) pivot value.

Snapshots depict the state of the data after the statements of the method.

Bailey pg 132
Complexity

- **Time:**
  - Partition is $O(n)$
  - If partition breaks list exactly in half, same as merge sort, so $O(n \log n)$
  - If data is already sorted, partition splits list into groups of 1 and $n-1$, so $O(n^2)$

- **Space:**
  - $O(n)$ (so is MergSort)
    - In fact, it’s $n + c$ compared to $2n + c$ for MergSort
Merge vs. Quick
Food for Thought…

• How to avoid picking a bad pivot value?
  • Pick median of 3 elements for pivot
    • Heuristic! No guarantees!

• Combine selection sort with quick sort
  • For small \( n \), selection sort is faster
  • Switch to selection sort when elements is \( \leq 7 \)
  • Switch to selection/insertion sort when the list is almost sorted (partitions are very unbalanced)
    • Heuristic! No guarantees!
# Sorting Wrapup

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>Worst: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Best: $O(n^2)$ as written, but can be “optimized” to $O(n)$</td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Worst: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td></td>
<td>Best: $O(n)$</td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>Worst = Best: $O(n^2)$</td>
<td>$O(n) : n + c$</td>
</tr>
<tr>
<td>Merge</td>
<td>Worst = Best: $O(n \log n)$</td>
<td>$O(n) : 2n + c$</td>
</tr>
<tr>
<td>Quick</td>
<td>Average = Best: $O(n \log n)$</td>
<td>$O(n) : n + c$</td>
</tr>
</tbody>
</table>