1 Merge Sort

- Divide and conquer – split input in half until the input is just a single item
- Merge each subsequence into a sorted sequence – simply keep a pointer at the beginning of each subsequence, choosing the current minimum of the two, placing it at the end of the merged sequence
- Since each “level” of recursion will examine (merge) $n$ items, and there are $\log n$ levels of recursion, merge sort runs in $O(n \log n)$

2 Quick Sort

- Another divide and conquer, although with quick sort, we don’t always divide the sequence in half – we choose a pivot around which we divide
- Idea: select a pivot value, move everything less than it to its left and everything greater than it to its right, then quick sort the left and right sides, recursing until the input is one item or zero items
- If we choose good (or good enough) pivots, the running time will be just like merge sort – $O(n \log n)$ – but if we choose a bad pivot (the first element if the input is already sorted), then quick sort exhibits its worst-case behavior, which is $O(n^2)$

3 Quick Sort on Arrays

- In place sort – we need to rearrange input array around pivot before we recurse
- For each recursive call, have an upper and lower index bound and sort only the items within those bounds
- Select a pivot value, swap it with the upper bound
- Start one pointer at the lower index bound and one at the upper bound, and begin moving them inwards – as soon as the lower index pointer finds a value greater than or equal to the pivot, and the upper one finds a value less than or equal to the pivot, the values are swapped, and the process is continued until the two pointers meet
- Once they cross over (once the lower pointer is pointing to an index greater than the higher one), we don’t swap values, swap the pivot with the “lower pointer” which is now at an index higher than the “higher pointer”, and recurse on the two sublists

4 Choosing a good pivot

- Choosing the first item or last item will cause quick sort to run slowly on presorted (ascending or descending) arrays
- Choosing the middle item works sometimes
• Choosing the median of the first, middle and last works better
• Choosing a random element is guaranteed to give \(O(n \ lg \ n)\) performance
• Choosing the median of three random elements is even better
• With good pivot selection, quick sort is faster than merge sort (smaller constant factor)

5 Announcements

• TA Review session – Sunday, July 28, 7PM in 306 Soda
• Reading: Data Structures (Into Java) – 8.7, 8.8

6 Merge sort (implementation)

```java
static List mergeSort(List A) {
    if (A == null || A.next == null)
        return A;
    /* Split the List */
    List temp = A, temp2x = A;
    if (temp2x.next.next != null) {
        while (temp2x != null && temp2x.next != null) {
            temp = temp.next;
            temp2x = temp2x.next.next;
        }
    }
    List firstList = A;
    List secondList = temp.next;
    temp.next = null;

    firstList = mergeSort(firstList);
    secondList = mergeSort(secondList);

    return merge(firstList, secondList);
}

static List merge(List a, List b) {
    if (a == null && b == null) {
        return null;
    } else if (a == null) {
        return new List (b.data, merge(null, b.next));
    } else if (b == null || a.data < b.data) {
        return new List (a.data, merge(a.next, b));
    } else {
        return new List (b.data, merge(a, b.next));
    }
}
```
// Modified from JRS
static void quicksort(int[] a, int low, int high) {
    if (low < high) {
        int pivotIndex = (new Random()).nextInt(high-low+1)+low;
        int pivot = a[pivotIndex];
        a[pivotIndex] = a[high];
        a[high] = pivot;

        int i = low - 1;
        int j = high;
        do {
            do { i++; } while (a[i] < pivot);
            do { j--; } while (a[j] > pivot) && (j > low));
            if (i < j) {
                swap(a, i, j);
            }
        } while (i < j);

        a[high] = a[i];
        a[i] = pivot;
        quicksort(a, low, i - 1);
        quicksort(a, i + 1, high);
    }
}