[dhutri's very helpful notes!]

[LSD Sort and Counting Sort — Content Review]

Identifying sorting algorithms trick (easiest -> hardest to identify)

1. MSD Radix
    a. First digits are sorted
        i. Assume that the first digit is 0 if a number is not the same length as the others
    b. If there's a big difference between the unsorted list and the first iteration
    c. Number of passes is the same as the number of digits
2. MergeSort
    a. Partially sorted subarrays whose length is in powers of 2
    b. Their length should only differ by one power of 2.
        i. So subarrays with lengths 1, 2, 3 are not valid
    c. If there's an odd number of elements, the last subarray should be 1
    d. Pair swapping / group swapping
3. LSD Radix
    a. First row *may* have all the ending digits sorted
    b. Next rows have the r-to-last digit sorted
    c. If there's a big difference between the unsorted list and the first iteration
    d. Number of passes is the same as the number of digits
4. HeapSort
    a. First columns is all the integers in reverse sorted order (if we ignore the iterations that reheapify everything)
    b. Bottom right triangle is sorted
    c. The first element of the heapify array gets swapped with the last element of the unsorted array
5. InsertionSort
    a. Smaller elements (ones at the front) are sorted
        i. Doesn't mean that the smallest element is at the front at first
    b. Start by comparing the first element
    c. List slowly becomes more sorted
    d. Elements at the end are unsorted. Last element stays at the end
6. SelectionSort
    a. Brings the minimum element to the front at first
    b. Looks for elements being swapped
7. QuickSort (hard if you don't know what the pivot is)
    a. Look for the pivot and see if *everything* to the left is smaller. Look to see if *everything* to the right is greater than the pivot
    b. Choose the new pivot as the first element of the unsorted array (to the left or right)

- *t*
- Unstable sorting algorithms cannot be the subroutines of stable sorting algorithms

Runtimes Explained

- MergeSort's runtime is Θ(nlog(n))
    - log(n) describes how the arrays are split
    - n because there are n elements so you're going to be doing n comparisons
- SelectionSort's runtime is $\Theta(n^2)$ because you're always going through the array to find the smallest element
- Insertion Sort
    - best case: Θ(n) no inversions because you don't need to do any swaps
    - Worst case: $\Theta(n^2)$ need to perform swaps for every element until the front
- Heap Sort
    - Expected: Θ(nlog(n)) each element takes log(n) time and there's n elements
        - Heapification takes log(n) time
    - best-case: Θ(n) when everything is an identical element because heapification would take constant time
- MSD radix
    - Best: Θ(N +R) no buckets are necessary (R - size of alphabet)
    - Worst: Θ(W(N +R)) buckets are necessary (W = width of the digit)

Kinda random facts

- Insertion sort is the best algorithm for when the list is nearly sorted or very short because insertion sort goes through the list the quickest if it is in order

HeapSort Steps

1. insert all elements into max-help. discard input array
2. create output array
3. repeat N times:
    a. delete max. Bubble down accordingly
    b. put max at the end of the unused part of the output array

QuickSort Steps Explained

1. Choose a pivot
2. Move the pivot to the end of the array to get it out of the way
3. Starting from the left of the array, choose the first item that is larger than the pivot
4. Starting from the right of the array, choose the first item that is smaller than the pivot
5. Swap the items in 3 and 4

    a. When you swap the items, the pointers for left and right don't change position. But the left item will point to the previous right item
6. Repeat the process with the same pivot
7. If the left and right items cross (left > right), then swap left with the pivot (which is at the end of my array)
8. You choose a new pivot.
    a. Choosing the new pivot would be a recursive call
    b. At this point, everything to the left of the pivot is smaller than the pivot. Everything right to the pivot is greater than the pivot
9. Do a recursive call on the left and right sides of the array from the pivot

```
int max = Integer.MIN_VALUE;
   for (int i = 0; i < k; i++) { //find max digit
       if (a[i] > max) {
           max = a[i];
       }
   }
   int temp = max;
   int maxDigit = 0;
   while (temp > 0) {
       maxDigit++;
       temp = temp/10;
   }

   while (maxDigit > 0) {
       digitSort(a, maxDigit, k);
       maxDigit--;
   }
}

private void digitSort(int[] a, int digit, int k) {
       // get the rightmost digit of each element. If it doesn't exist then the
value is 0
   digit = (int) Math.pow(10, digit - 1);

   for (int j = 1; j < k; j++) {
       int l = j - 1;
       while (l >= 0 && ((a[l] / digit) % 10) > ((a[l + 1] / digit) % 10)) {
           int temporary = a[l];
           a[l] = a[l + 1];
           a[l + 1] = temporary;
           l--;
       }
   }
```