1 Sorting

We focus this week on sorting. We'll cover three slow/simple sorting methods and three fast/complex sorting methods.

1. Read Chapter 8 (Selection Sort, Insertion Sort, and Shell Sort) and Chapter 9 (Merge Sort, Quick Sort, and Radix Sort).
    - Don't worry too much about the coding, especially for the more complex sorts.
    - Focus on the general approach of each sort. You should be able to explain how a sort works and trace it by hand.
2. Read these notes that highlight key points from the chapters.
3. Watch the instructor-created videos that go along with this week's reading: http://tinyurl.com/Week08Sorting
    - There are additional video resources linked on each sorting page.
    - There are *a lot* of sorting videos out there, but I link to those I think are among the better videos. If you find better videos, please share them on the discussion board!
    - You can also view the videos with closed-captioning through this link: https://www.3cmediasolutions.org/f/3daa8bfc9717266055ff1117ca2918a12f339b9c. Use the "cc' button on the lower right of the video player to enable captions. Captioning was completed through the Distance Education Captioning and Transcription Grant.

1 Sorting

1.1 Selection Sort

The *selection sort* searches for either minimums or maximums and then *swaps* that value with the appropriate place on the list.

The general idea is:

- Find the smallest value, swap it to put it in the first position.
- Find the second smallest value, swap it to put it in the second position.
- Find the third smallest value, swap it to put it in the third position.
- And so on.

This is why it's called *selection* sort- you are continually selecting the nth smallest value and putting that element in the nth place (by swapping with whatever is currently in that place).

One key thing to note about selection sort is that it uses a **swap** to put the nth smallest value in the nth place. It does **not** use a shift. This is a common mistake made when tracing selection sort.

For example, to sort [10,14,12,-11,13] using maximums (instead of minimums):

1. Find the max of all x_0 to x_{n-1}, where n=5 in this case. This is the value 14.
2. Swap 14 at the location for the largest i value (i=4). In this case, that is the value 13
3. We now have: [10,13,12,-11,14]
4. Now that 14 is in the correct location, find the largest of x_0 to x_3. This is the value 13.
5. Swap 13 at the location for second largest value. In this case, that is the value -11.
6. We now have [10,-11,12,13,14]
7. Now find the maximum of x_0 to x_2. This is the value 12. This value is already in the correct position so we do nothing.
8. The last iteration finds the max of x_0 to x1. This is the value 10.
9. Swap 10 into the correct location.
10. We now have [-11,10,12,13,14] as the final result.

The selection sort algorithm using maximums is:

```
for i=n-1 down to 1

    find the maximum of x[0] to x[i]

    swap x[i] with the maximum
```

Selection sort can be implemented iteratively or recursively. With either implementation, finding the maximum requires an additional loop through the elements. Thus, selection sort is O(n^2).

Here are some additional video resources for selection sort:

- http://www.youtube.com/watch?v=MZ-ZeQnUL1Q
- http://www.youtube.com/watch?v=6kg9Dx72pzs

1 Sorting

1.2 Insertion Sort

The *insertion sort* considers sorted subsets and places elements in their proper place.

The general idea is:

- Consider the first item to be a sorted sublist (of size one).
- Insert the second item into this sorted sublist, *shifting* the first item as needed to make room for the new element.
- You now have a sorted sublist of size two.
- Insert the third item into this sorted sublist, shifting the first and second items as needed to make room for the new element.
- And so on.

This is why it's called *insertion* sort- you are continually inserting an element into a sorted sublist.

As stated on the previous page, selection sort does not involve shifting- it only involves swapping two elements at any given time. Insertion sort **does** involves shifting to make room for the newly inserted elements.

Note: There is **no** algorithm that combines these approaches (finds the minimum and then shifts it into place). This would essentially be the *worst of both worlds!*Selection finds the min and swaps. Insertion shifts elements into their proper place. Don't combine these approaches!

For example, to sort [4,2,0,5,1,6]:

1. Start by leaving the first value in place since a single-valued list is already sorted.
2. Next, consider the second value (2).
3. Compare this value to the predecessor (4). If the unsorted value is smaller, shift this value and compare with the next predecessor. Stop when either a predecessor is smaller or we reach the beginning of the list. Then store the 2.
4. We now have: [2,4] and the rest of the list as is: [0,5,1,6]
5. Next examine the third value (0).
6. Compare 0 with 4. 0 is smaller so compare 0 with 2. 0 is smaller. Try to compare to the next value but we have reached the beginning of the list, so store the 0.
7. We now have: [0,2,4] and the rest of the list as is: [5,1,6]
8. Next examine the fourth value (5).
9. Compare 5 with 4. 4 is smaller so we leave the 5 in place.
10. We now have: [0,2,4,5] and the rest of the list as is: [1,6]
11. Next examine the fifth value (1).
12. Compare 1 with 5, 1 is smaller. Compare 1 with 4, 1 is smaller. Compare 1 with 2, 2 is smaller. Compare 1 with 0. 0 is smaller so insert 1 here.
13. We now have: [0,1,2,4,5] and the rest of the list as is: [6]
14. Next example the sixth value (6).
15. Compare 6 with 5. 5 is smaller so we leave the 6 in place.
16. We now have [0,1,2,4,5,6] as the final result.

The insertion sort algorithm is:

```
for(i=1 to n-1)

    toInsert = x[i]

    j = i

    while(j>0 and toInsert < x[j-1])

        x[j] = x[j-1]

        j--
        x[j] = toInsert
```

Insertion sort can be implemented iteratively or recursively. Either way, the nested loops result in O(n^2).

The recursive version of insertion sort will be important later in the course. This sort is useful for linked lists because the "next item" to be added to the partially sorted part of the list is the head of "rest of the list," and the head is the only node we have available in a linked list.

Here are additional resources for insertion sort:

- https://www.youtube.com/watch?v=c4BRHC7kTaQ
- https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort

1 Sorting

1.3 Shell Sort

The *Shell sort* is related to the logic for insertion sort, but instead of swapping adjacent items, the algorithm uses a gap >1 of equally-spaced indices making a larger distance for items that need to move. A gap of n/2 is is typically used to start then halved repeatedly until it is decreased to 1.

Read the book's example and review the illustration beginning in Figure 8-11. In the book's example, n is 13, so the first gap is 13/2 (or 6).

As an example, to sort: 8, 3, 19, 7, 12, 6, 2, 1, 5

1. In this list, n is 9 so we start with a gap of 9/2 = 4. That means we sort the elements that are 4 units apart from each other and ignore the other values in between. (Note that I'm just omitting the values from display to make clear what values are being sorted. In practice, all values are always present in the array!)

8, -, -, -, 12, -, -, -, 5	becomes	5, -, -, -, 8, -, -, -, 12
-, 3, -, -, -, 6, -, -, -	becomes	-, 3, -, -, -, 6, -, -, -
-, -, 19, -, -, -, 2, -, -	becomes	-, -, 2, -, -, -, 19, -, -
-, -, -, 7, -, -, -, 1, -	becomes	-, -, -, 1, -, -, -, 7, -

After this step, we have: 5, 3, 2, 1, 8, 6, 19, 7, 12

1. Next, we cut the gap in half. So now gap = 4/2 = 2. That means we sort the elements that are 2 units apart from each other and ignore the other values in between:

5, -, 2, -, 8, -, 19, -, 12   becomes   2, -, 5, -, 8, -, 12, -, 19

-, 3, -, 1, -, 6, -, 7, -   becomes   -, 1, -, 3, -, 6, -, 7, -

After this step, we have: 2, 1, 5, 3, 8, 6, 12, 7, 19

1. Finally, we cut the gap in half one last time. So now gap = 2/1 = 1. That means we perform one final sort of all numbers. In practice, we would use an insertion sort here. And notice that each element does not move more than two places to the left. So it's a pretty efficient insertion sort.

    1, 2, 3, 5, 6, 7, 8, 12, 19

Here are video resources for Shell sort:

- http://www.youtube.com/watch?v=IlRyO9dXsYE
- http://www.youtube.com/watch?v=qzXAVXddcPU

1 Sorting

1.4 Summarizing the "Slow" Sorts

Review the efficiency table in Figure 8-15. You'll notice that all of the three sorts just described are inefficient. They are relatively easy to understand and to implement, but they are inefficient for large lists. If you are sorting smaller lists, or even larger lists that only need to be sorted a single time, these are reasonable algorithms to use. And if you have linked nodes as an implementation, recursive insertion sort is a good choice.

However, if you have large lists that need to be sorted repeatedly, none of these approaches are good ones. You'll need to use a more efficient type of sort.

1 Sorting

1.5 Merge Sort

The *merge sort* divides a list into smaller sublists, sorts the sublists, then merges the sublists back together. Merge sort is typically defined recursively.

The key step in merge sort is the merge. For the first step, the dividing, you can always just divide down to lists of size one, which are then sorted. You could also divide down to lists that are reasonably small (perhaps 10 elements or less), and then sort those lists using one of the simple sorts we just discussed. So the divide step is somewhat trivial. It's merging the elements back together that is key.

The pseudocode for merging is below. Assume you are given two ararys a (sizeA) and b (sizeB) that are already sorted. The merge step will create an array c (sizeC = sizeA + sizeB).

```
i=0, j=0, k=0

while(i < sizeA and j < sizeB) // there are more elements in both
lists

    if(a[i] < b[j]) // the current element in a is smaller, so put it
    in c and advance to the next element in a

        c[k] = a[i]

        i++

    else // the current element in b is smaller, so put it in c and
    advance to the next element in b

        c[k] = b[j]

        j++

    k++

while(i < sizeA) // we've put in all the elements from b, but there
are still elements left in a to move into c

    c[k] = a[i]

    i++

    k++

while(j < sizeB) // we've put in all the elements from a, but there
are still elements left in b to move into c

    c[k] = b[j]

    j++

    k++
```

Figure 9-3 traces through a merge sort, showing the splitting stages until each sublist contains a single element. Since a list of one element is already sorted, we then repeatedly call the merge algorithm to combine one-element lists into two-element lists. Then we merge these sorted two-element lists into four-element lists, and so on.

Notice that the merge sort only works this neatly for lists of sizes that are powers of two (2, 4, 8, 16, etc.). When list lengths are not powers of 2, then some merge steps are not needed.

For example, given the list [18,22,33,27,14]:

1. First we split down to one-element lists: [18] [22] [33] [27] [14]
2. Then we merge to two-element lists: [18,22] [27,33] and [14] stays a one-element list
3. Now we merge two-element lists into four-element lists when possible: [18,22,27,33] and [14] stays a one-element list
4. The final merge combines the two lists: [14,18,22,27,33]

Here is a video resource for merge sort:

- http://www.youtube.com/watch?v=GCae1WNvnZM

The efficiency of the merge sort is O(n logn).

The Collections.sort method uses a version of merge sort. Here are implementation notes from the API:

> This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.
>
> The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.
>
> The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

1 Sorting

1.6 Quicksort

The *quicksort* algorithm was developed by Tony Hoare, who won a prize for designing the fastest sorting method. The general approach is to consider some index k such that all elements to the left of index k are smaller than the value at k and all elements to the right of index k are larger than the value at k. This would means that k is in the correct location of the sorted list. If this property was true for all values of k, then the list is sorted! This property can be written as:

data[i] <= data[k] for 1<=i<=k

and

data[k] <= data[j] for k<=j<=n

Quicksort repeatedly utilizes this property on sublists until the list is sorted. Once this property is established for the value k in a range data[low] to data[hi], then recursively call quick sort on data[low] to data[k-1] and also on data[k+1] to data[hi]. This is a *divide and conquer* strategy with the basic idea:

1. First choose a pivot point (or partition value).
    - Some choose the median of the first three values, others choose the median of the first, last, and middle values.
    - The question of choosing the *best* pivot point is nontrivial. In fact, it is often a subject of PhD dissertations!
    - For our purposes, we will choose the first value in the list as our pivot.
2. Then *partition* the list such that everything to the left is smaller than the pivot and everything to the right is bigger than the pivot.
    - Be sure to complete the scan from the right **before** the scan from the left.
    - Scan from the right looking for a value that we need to move (a value smaller than the pivot). Stop when we find one.
    - Scan from the left looking for a value that we need to move (a value larger than the pivot). Stop when we find one.
    - Swap these values.
    - Repeat until the scans cross.
    - Once the scans cross, swap the pivot with the value from the right-side scan.
    - The pivot is now in the correct position.
3. Repeat recursively on the left-of-pivot list and the right-of-pivot list.

For example, given the list: [14,15,13,29,22,17,10,11,30]

1. First choose your pivot. We will choose the first value in the list: 14
2. We eventually want 14 to be in the middle and all values to the left to be less than 14 and all values to the right to be larger than 14
3. First we scan **from the right** looking for a value that we would need to move (a value that is less than 14). The first one we find is 11.
4. Now we can from the left to look for a value that needs to move (a value that is greater than 14). The first one we find is 15.
5. Next we swap these two values so we have: [14,11,13,29,22,17,10,15,30]

6. We continue to scan from the right looking for a value less than 14. The next one we find is 10.
7. We continue to scan from the left looking for a value that is greater than 14. The next one we find is 29.
8. Swap these values to get: [14,11,13,10,22,17,29,15,30]
9. Again we scan from the right looking for a value less than 14. The first one we find is the 10.
10. We scan from the left looking for a value greater than 14. The first one we find is the 22.
11. But wait! At this point the scans from the left and right have crossed each other. When this happens, we swap the **pivot** (14) with the value we found from the right-side scan (10).
12. So we now have: [10,11,13,14,22,17,29,15,30]
13. Notice that now 14 is in the correct location- all numbers to the left are less than 14 and all numbers to the right are greater than 14.
14. We now recursively call quick sort on the two sublists: [10,11,13] [14] [22,17,29,15,30]
15. First, the left sublist: [10,11,13]. For this list, 10 is the pivot.
    1. We scan from the right looking for a value less than 10. We hit 10, the pivot itself. This means we do not have to scan from the left since there are no values less than the pivot.
    2. So we partition into two sublists: [10] [11,13]
    3. 10 is now in the correct place
    4. We recursively call quick sort on [11,13] with a similar result.
16. We are now halfway there with: [10] [11] [13] [14] [22,17,29,15,30]
17. Second, let's go back to the original right sublist: [22,17,29,15,30]. For this sublist, 22 is the pivot.
    1. First we scan from the right for a value less than 22 and stop at 15.
    2. We scan from the left for a value greater than 22 and stop at 29.
    3. We swap these values to get: [22,17,15,29,30]
    4. Then we scan from the right for a value less than 22 and get 15.
    5. We scan from the left for a value greater than 22 and get 29.
    6. But here the scans have met so we want to swap the pivot with 15 to get: [15,17,22,29,30]
    7. The pivot is in the correct place and we recursively call quick sort with the sublists: [15,17] and [29,30]
    8. Quick sort will leave both of these unchanged
18. We have our final result: [10,11,13,14,15,17,22,29,30]

Here is a video resource for quick sort:

- http://www.youtube.com/watch?v=8hHWpuAPBHo
- https://www.youtube.com/watch?v=mN5ib1XasSA

The Arrays.sort method uses a version of Quicksort. From the API:

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log👎) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

1 Sorting

1.7 Radix Sort

The *radix sort* comes from back when punch cards were sorted in machines. This algorithm treats data like strings of the same length and uses *buckets* to do the sort.

For example, given the list: [7239,123,307,3,1389,25]

1. First make all values the same length by padding with leading zeros: [7239,0123,0307,0003,1389,0025]
2. Now we group into 10 buckets (for the digits 0-9) and group based on the **ones** digit (being sure to keep our data in order from left to right *within* a bucket) :
    1. Bucket 0: none
    2. Bucket 1: none
    3. Bucket 2: none
    4. Bucket 3: 0123, 0003 // 0123 was left of 0003 in the original array, and so it remains to the left in the bucket
    5. Bucket 4: none
    6. Bucket 5: 0025
    7. Bucket 6: none
    8. Bucket 7: 0307
    9. Bucket 8: none
    10. Bucket 9: 7239, 1389
3. We reorder the list based on this grouping: [0123,0003,0025,0307,7239,1389]
4. Next, we group into buckets for the **tens** digit (keeping our data in order from left to right based on this new ordering):
    1. Bucket 0: 0003, 0307,
    2. Bucket 1: none
    3. Bucket 2: 0123, 0025
    4. Bucket 3: 7239
    5. Bucket 4: none
    6. Bucket 5: none
    7. Bucket 6: none
    8. Bucket 7: none
    9. Bucket 8: 1389
    10. Bucket 9: none
5. We reorder the list based on this grouping: [0003,0307,0123,0025,7239,1389]
6. Next, we group into buckets for the **hundreds** digit (keeping our data in order from left to right based on this new ordering):

        1. Bucket 0: 0003, 0025
        2. Bucket 1: 0123
        3. Bucket 2: 7239
        4. Bucket 3: 0307, 1389
        5. Bucket 4: none
        6. Bucket 5: none
        7. Bucket 6: none
        8. Bucket 7: none
        9. Bucket 8: none
        10. Bucket 9: none
7. We reorder the list based on this grouping: [0003,0025,0123,7239,0307,1389]
8. Finally, we group into buckets for the **thousands** digit (keeping our data in order from left to right based on this new ordering):
    1. Bucket 0: 0003, 0025, 0123, 0307
    2. Bucket 1: 1389
    3. Bucket 2: none
    4. Bucket 3: none
    5. Bucket 4: none
    6. Bucket 5: none
    7. Bucket 6: none
    8. Bucket 7: 7239
    9. Bucket 8: none
    10. Bucket 9: none
9. The final reordering based on these buckets is our final result: [3, 25, 123, 307, 1389, 7239]

Here is a video resource for radix sort:

- http://www.youtube.com/watch?v=50_TCQGjNJc3
- https://www.youtube.com/watch?v=xuU-DS_5Z4g

1 Sorting

1.8 Summarizing the "Fast" Sorts

Review Figure 9-11 that summarizes the efficiency of the slow and fast sorting algorithms.

You can see from this table that you have to know a lot about your data and how it will be sorted in order to choose the appropriate sort.

For example, radix sort is fastest, but what if you are sorting Student objects? You can't use radix sort on these!

Some of the slower sorts will work just fine if you always have smaller data sets or if your data is known to be in a *best case* scenario. For example, if you had a large data set that you know was *nearly* sorted and that only needed to be sorted a single time before being used, you might choose insertion sort over one of the more complex efficient methods.

Knowing your data and how it will be used is key!

1 Sorting

1.9 Example Traces of the Sorts

Below is a sample trace of each kind of sort on a single data set. I recommend you trying the trace yourself and then comparing your answers to the answers below.

Dataset: 15, 7, 8, 5, 1, 11, 2, 4

Selection Sort

In the trace below, the smallest value at each pass is highlighted in red. That value will be **swapped** into its proper place the next pass through (shown in green).

[15, 7, 8, 5, 1, 11, 2, 4]

[1, 7, 8, 5, 15, 11, 2, 4]

[1, 2, 8, 5, 15, 11, 7, 4]

[1, 2, 4, 5, 15, 11, 7, 8]

[1, 2, 4, 5, 15, 11, 7, 8]

[1, 2, 4, 5, 7, 11, 15, 8]

[1, 2, 4, 5, 7, 8, 15, 11]

[1, 2, 4, 5, 7, 8, 11, 15]

Insertion Sort

In the trace below, the sorted subarray is shown in green.

[15, 7, 8, 5, 1, 11, 2, 4]

[7, 15, 8, 5, 1, 11, 2, 4]

[7, 8, 15, 5, 1, 11, 2, 4]

[5, 7, 8, 15, 1, 11, 2, 4]

[1, 5, 7, 8, 15, 11, 2, 4]

[1, 5, 7, 8, 11, 15, 2, 4]

[1, 2, 5, 7, 8, 11, 15, 4]

[1, 2, 4, 5, 7, 8, 11, 15]

Shell Sort

gap is 4

[15, 7, 8, 5, 1, 11, 2, 4]

[15, -, -, -, 1, -, -, -] becomes [1, -, -, -, 15, -, -, -]

[-, 7, -, -, -, 11, -, -] becomes [-, 7, -, -, -, 11, -, -]

[-, -, 8, -, -, -, 2, -] becomes [-, -, 2, -, -, -, 8, -]

[-, -, -, 5, -, -, -, 4] becomes [-, -, -, 4, -, -, -, 5]

so for gap = 4, array becomes:

[1, 7, 2, 4, 15, 11, 8, 5]

gap is 2

[1, -, 2, -, 15, -, 8, -] becomes [1, -, 2, -, 8, -, 15, -]

[-, 7, -, 4, -, 11, -, 5] becomes [-, 4, -, 5, -, 7, -, 11]

so for gap = 2, array becomes:

[1, 4, 2, 5, 8, 7, 15, 11]

gap is 1, array becomes:

[1, 2, 4, 5, 7, 8, 11, 15]

Merge Sort

[15, 7, 8, 5, 1, 11, 2, 4]

[15, 7, 8, 5][1, 11, 2, 4]

[15, 7][8, 5][1, 11][2, 4]

[15] [7] [8] [5] [1] [11] [2] [4]

[7, 15] [5, 8] [1, 11] [2, 4]

[5, 7, 8, 15] [1, 2, 4, 11]

[1, 2, 4, 5, 7, 8, 11, 15]

Quicksort

The following is a trace of one partitioning step.

[15, 7, 8, 5, 1, 11, 2, 4]

partition(array, 0, 7)

mid = 3

[4, 7, 8, 5, 1, 11, 2, 15] // result of sortFirstMiddleLast (sorts elements at 0, 3, and 7)

[4, 7, 8, 2, 1, 11, 5, 15] // result of swapping mid (position 3, the 5) and last-1 (position 6, the 2)

[4, 1, 8, 2, 7, 11, 5, 15] // first swap: position 4 (the 1) and position 1 (the 7)

[4, 1, 2, 8, 7, 11, 5, 15] // second swap: position 3 (the 2) and position 2 (the 8)

[4, 1, 2, 5, 7, 11, 8, 15] // swap the pivot (the 5) into place

returned pivotIndex = 3 (everything to the left of position 3 is < 5 and everything to the right of position 3 is > 5)

The next two recursive calls of the algorithm will be:

quickSort(a, 0, 2)

quickSort(a, 4, 7)

Radix Sort

[15, 7, 8, 5, 1, 11, 2, 4]

First pad with leading 0s:

15 07 08 05 01 11 02 04

Then sort into buckets based on ones digit:

Bucket 0: 01 11

Bucket 1:

Bucket 2: 02

Bucket 3:

Bucket 4: 04

Bucket 5:15 05

Bucket 6:

Bucket 7: 07

Bucket 8: 08

Bucket 9:

Then fill the array back up in the order of the buckets:

01 11 02 04 15 05 07 08

Sort into buckets based on tens digit:

Bucket 0: 01 02 04 05 07 08

Bucket 1: 11 15

Bucket 2:

Bucket 3:

Bucket 4:

Bucket 5:

Bucket 6:

Bucket 7:

Bucket 8:

Bucket 9:

Then fill the array back up in the order of the buckets:

01 02 04 05 07 08 11 15

Remove the padded zeros:

1 2 4 5 7 8 11 15