Sorting

When you rearrange data and put it into a certain order, you are **sorting** the data. You can sort data alphabetically, numerically, and in other ways. Often you need to sort data before you use searching algorithms to find a particular piece of data.

There are many different algorithms that can be used to sort data. A few popular ones are listed below:

- Selection Sort
- Bubble Sort
- Merge Sort
- Quick Sort
- Insertion Sort
- Heap Sort

For CSE 214, we will be studying the first four.

Selection Sort

The name of Selection Sort comes from the idea of selecting the smallest element from those elements not yet sorted. The smallest element is then swapped with the first unsorted element.

Here is the basic process of sorting an n-element array, A, using selection sort:

1. Find the smallest element from A[0]…A[n]
2. Swap that smallest element with A[0]
3. Find the smallest element from A[1]…A[n]
4. Swap that smallest element with A[1]
5. Find the smallest element from A[2]…A[n]
6. Swap that smallest element with A[2]

.

.

.

Continue this process until the last element in the array.

Selection Sort Example

Here we are sorting an array containing the following numbers:

8, 27, 33, 2, 20, 12, 19, 5

In the following figure:

- The already sorted part is shown in *italics*
- The first unsorted element is shown underlined
- The minimum element of the unsorted part is shown in **bold**

8, 27, 33, **2**, 20, 12, 19, 5

2, 27, 33, 8, 20, 12, 19, **5**

2, 5, 33, **8**, 20, 12, 19, 27

2, 5, 8, 33, 20, **12**, 19, 27

2, 5, 8, 12, 20, 33, **19**, 27

2, 5, 8, 12, 19, 33, **20**, 27

2, 5, 8, 12, 19, 20, 33, **27**

2, 5, 8, 12, 19, 20, 27, **33**

2, 5, 8, 12, 19, 20, 27, 33 Resulting sorted array

Implementing Selection Sort in Java

How can we write a Java method that will implement the selection sort algorithm which is explained above?

Assume we have a recursive `minIndex(int[] a, int left, int right)` method which returns the index of the minimum element of an array within the specified indices. Then we can implement the selection sort algorithm in the following way:

```
public static int[] selectionSort(int[] a)

{
    for(int i = 0; i < a.length;  i++)
    {
      int min_ind = minIndex(a, i, a.length);

      // swap a[i] with a[min_ind]
      int temp = a[i];
      a[i] = a[min_ind];
      a[min_ind] = temp;
    }

    return a;
}
```

As an exercise, implement the recursive `minIndex` method. As another exercise, change the code above so that you get rid of the recursive `minIndex` method, providing an iterative solution.

Bubble Sort

The idea of bubble sort is to compare two adjacent elements. If they are not in the right order, switch them. Do this comparing and switching (if necessary) until the end of the array is reached. Repeat this process from the beginning of the array n times.

Bubble Sort Example

Here we want to sort an array containing [8, 5, 1]. The following figure shows how we can sort this array using bubble sort. The elements in consideration are shown in **bold.**

8, 5, 1	Switch 8 and 5
5, **8, 1**	Switch 8 and 1

5, 1, 8	Reached end start again.
5, 1, 8	Switch 5 and 1
1, **5, 8**	No Switch for 5 and 8
1, 5, 8	Reached end start again.
1, 5, 8	No switch for 1, 5
1, **5, 8**	No switch for 5, 8
1, 5, 8	Reached end. But do not start again since this is the nth iteration of same process

Implementing Bubble Sort in Java

```
public static int[] bubbleSort(int[] a)
{
      for(int i = 0; i < a.length;  i++)
      {
            for(int j = 0; j < a.length;  j++)
            {
                if(a[j] > a[j+1])
                {
                  // then swap these two
                  int temp = a[j];
                  a[j] = a[j+1];
                  a[j+1] = temp;
                }
            }
      }
      return a;
}
```

Merge Sort

Merge sort is a *divide and conquer* sorting technique. Divide and conquer algorithms are essentially more complicated recursive algorithms, so there is little new information here:

The divide phase repeatedly divides the problem into smaller sub-problems until the problem is small enough to solve.

The conquer step solves the simpler sub-problems and reconstructs a solution to the overall problem.

In merge sort we assume we know:

- how to sort an array containing the smallest possible number of values (ie, only one value), and
- how to merge two sorted lists.

The idea behind merge sort to divide the problem in half, recursively, until there is one item to sort, and then merge the sorted items when returning from the recursion.

Merge Sort Example

Generation of Subproblems:

42, 7, 22, 2, 10 | 9, 15, 4, 33, 26

42, 7, 22 | 2, 10   9, 15, 4 | 33, 26

42, 7 | 22   2 | 10   9, 15 | 4   33 | 26

42 | 7   22   2   10   9 | 15   4   33   26

42   7   9   15

Result of Merge Operations (the topmost result is the last one returned, i.e. start tracing from the bottom of the pyramid):

2, 4, 7, 9, 10, 15, 22, 26, 33, 42

2, 7, 10, 22, 42   4, 9, 15, 26, 33

7, 22, 47   2, 10   4, 9, 15   26, 33

7, 47   22   2   10   9, 15   4   33   26

42   7   9   15

Implementing Merge Sort in Java

```
public static int[] mergeSort(int[] a, int left, int right)
{
      if(left < right)
      {
         int mid = (left + right) / 2;

         // Sort the first half
         int[] left =  mergeSort(a, left, mid);

         // Sort second half
         int[] right = mergeSort(a, mid+1, right);

         return merge(left, right);

      }

      int[] arr = new int[1];
```

```
        arr[0] = a[left];
        return arr;
}
```

Then, how should `merge` method look like?

```
public static int[] merge (int[] a, int[] b)

{
    // create a new array which will be result of this merge operation
    int[] result = new int[a.length + b.length];

    int index = 0;
    int aIndex = 0;
    int bIndex = 0;

    while(aIndex < a.length && bIndex < b.length)
    {
       if(a[aIndex] <= b[bIndex])
       {
          // then, a[aIndex] is less then all elements in b
          // since b is already sorted

          result[index] = a[aIndex];

          index++;
          aIndex++;
       }
       else
       {
          // then, b[bIndex] is less then all elements in a
          // since a is also already sorted

          result[index] = b[bIndex];

          index++;
          bIndex++;
       }
    }

    if(aIndex < a.length)
    {
       while(aIndex < a.length)
       {
          result[index] = a[aIndex];
          aIndex++;
          index++;
       }
    }

    if(bIndex < b.length)
    {
       while(bIndex < b.length)
       {
          result[index] = b[bIndex];
          bIndex++;
          index++;
       }
    }

    return result;
}
```

Quick Sort

Another efficient sorting algorithm is Quick Sort which, like Merge Sort, uses a divide-and-conquer approach to sort the elements of an array. In order to sort the elements of an array A in increasing order from position left through position right, Quick Sort performs the following steps:

- Partition the array so that all elements in the range A[left]…A[p-1] are smaller than A[p] and A[p] is smaller than any element in the range A[p+1]…A[right]
- Sort the first partition by recursively applying Quick Sort on the elements A[left]…A[p-1]
- Sort the last partition by recursively applying Quick Sort on the elements A[p+1]…A[right]

If you have a method that performs partition, then you would easily implement this recursive method. As an exercise think about how can you implement the approach explained above by assuming you have this partition method.

Partition

Partitioning is the most complex part of quick sort. The simplest thing is to use the first value in the array, A[0], or in general A[left], as the pivot. After the partitioning, all values to the left of the pivot are <= pivot and all values to the right are > pivot.

For example, consider

	l												r
k	0	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	**8**	2	5	13	4	19	12	6	3	11	10	7	9

where the value of a[l], namely 8, is chosen as pivot. Then the partition function moves along the array from the left until it finds a value > pivot. Next it moves from the right, passing values > pivot and stops when it finds a value <= pivot. This is done by the following piece of code:

```
i = l; j = r + 1;
do ++i;
while( a[i] <= pivot && i <= r );
do --j;
while( a[j] > pivot );
```

Then if the lhs value is to the left of the rhs value, they are swapped. Variable i keeps track of the current position on moving from left to right and j does the same for moving from right to left. You then get

	l			i								j	r
a[k]	8	2	5	13	4	19	12	6	3	11	10	7	9

a[i] and a[j] are swapped to give

	l			i								j	r
a[k]	8	2	5	7	4	19	12	6	3	11	10	13	9

This process is repeated until i >= j. At the end,

	l						j	i					r
a[k]	8	2	5	7	4	3	6	12	19	11	10	13	9

When j passes i, the partitioning is finished. At this stage the partition code breaks out of the main while loop. All a[k] <= pivot where k <= j. All that we do next is swap the pivot with a[j] to get

	l						j	i					r
a[k]	6	2	5	7	4	3	8	12	19	11	10	13	9

Exercise: how can this partition method be implemented?