Introduction

When we need to access data from a set, we are not only concerned with acquiring said data, but we are also concerned with how efficiently we acquire said data.

In my lower-level classes, I show students how to access data using linear search. This is the most basic way to find data in a set. We assume an unsorted array and we iterate through the entire array until we find the item we are looking for (or reach the end of the array in the case of a failed search).

The problem with linear search is that we must spend O(n) in the worst case for every search we do. Other problems, like the max/min value problems, also take linear time because we have to look at every element because the one we don't look at might have been larger/smaller than the previous values.

Because of this, I exposed them to basic sorting algorithms. As you probably know, sorting is when you arrange a data set in order. This order can be ascending (smallest to largest) or descending (largest to smallest.) The important thing is that we can now make certain assumptions about the data set that allow certain problems to be solvable in better than O(n) time.

Assuming an array sorted from smallest to largest, we can now solve the max/min value in constant time by looking at the first element or the last element of the array. This is an example of the benefits of sorting.

Basic Sorting (Selection Sort)

For the lower-level students, I show them “selection sort” since it is intuitive to understand, easy to implement, and decent performance for smaller data sets.

The following will explain how Selection Sort works:

```
{ 5, 1, 3, 8, 6, 2, 9 }  <- Data Set (List, Array, etc.)
  ^  ^
  i  j
```

In this first “iteration”, we start by checking the value at set[i] against set[j]. Assuming we are sorting from smallest to largest (ascending), we would check to see if set[i] > set[j] (you could also check if set[j] < set[i]). If so, we swap these values. To swap, we need a temporary swap variable to hold one of the values while we transfer the values to the opposite indexes. After the first check, we have the following:

```
{ 1, 5, 3, 8, 6, 2, 9 }  <- Data Set (List, Array, etc.)
  ^     ^
  i     j
```

As you see, the first two elements swapped, and j moved to the next element in the list. We continue to check if set[i] > set[j], and swap if that is the case. We leave i in its place until j gets to the end of the list and then i advances by one element and j starts right after it.

```
{ 1, 5, 3, 8, 6, 2, 9 }  <- Data Set (List, Array, etc.)
     ^  ^
     i  j <- swap set[i] and set[j] here!
```

```
{ 1, 3, 5, 8, 6, 2, 9 }  <- Data Set (List, Array, etc.)
     ^           ^
     i           j  <- Swap set[i] and set[j]!
```

Continue with this method of sorting until all possible combinations have been made between set[i] and set[j]. The list should then be sorted. This is a type of Selection Sort.

Implementing Selection Sort in Java

```java
/* Selection sort */
public static void selSort(int[] a){
    for(int j = 0; j < a.length-1; j++){
        for(int i = j+1; i < a.length; i++){
            if(a[j] > a[i]){
                int swap = a[j];
                a[j] = a[i];
                a[i] = swap;
            }
        }
    }
}
```

Issues with Selection Sort

As simple and straightforward as selection sort is, it is not considered an optimal sorting algorithm. The best possible general-purpose sorting algorithm is O(n log n). Selection Sort is O(n * n/2) which is less efficient. Therefore, if we are dealing with huge data sets and we need the absolute best performance guarantee possible, we need to write an algorithm that is optimal.

Optimal Sorting Algorithms

When we talk about optimal sorting algorithms, we mean algorithms that are guaranteed (worst case) to run in no worse than O(n log n) time. This limits our availability of sorting algorithms considerably.

We are left with two that fit this description:

- Merge Sort
- Heap Sort

But what about Quicksort?

Quicksort is an unstable sorting algorithm that runs faster, in the average case, than most other algorithms. It uses a “pivot” to accomplish the sort, but the performance of the sort is based on how the pivot relates to the data set being sorted. Despite this sort being the fastest “in practice”, it still has a worst case of $O(n^2)$ which means it is worse than Selection Sort if we are being pessimistic.

Because of the lack of an absolute worst-case guarantee, we exclude it from our list.

*Technically, there are other sorts that are O(n log n) such as Timsort and Cubesort but we are narrowing our focus to more common algorithms that are widely used. It can also be argued that insertion into a balanced binary search tree results in an O(n log n) sort but we are discussing the sorting of an array with an array as an output as well.

Heap Sort Implementation

Heap Sort, like Merge Sort, is a recursive sorting algorithm. It uses both a helper and a wrapper method. The helper method “buildHeap” handles arranging the array into a max heap. The wrapper method “heapSort" sets the stage for the sort and calls the recursive “buildHeap” method to get it going.

Here is the code, in Java, for the helper and the wrapper method:

```
/* Heap sort helper method */
private static void buildHeap(int i, int n, int a[])
{
    int max = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < n && a[left] > a[max])
        max = left;

    if (right < n && a[right] > a[max])
        max = right;

    if (i != max)
    {
        int swap = a[i];
        a[i] = a[max];
        a[max] = swap;

        // Recursive call
        buildHeap(max, n, a);
    }
}

/* Heap sort wrapper method */
public static void heapSort(int a[])
{
    final int n = a.length;

    // Build heap
    for (int i = n / 2 - 1; i >= 0; i--)
        buildHeap(i, n, a);

    for (int i=n-1; i>0; i--)
    {
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        // Call the helper method
        buildHeap(0, i, a);
    }
}
```

If you wanted to use this in your main method, here is an example:

```
public static void main(String[] args){
      int[] a = {4,7,5,6,3,1,2,9};
      heapSort(a);
      System.out.println(Arrays.toString(a));
}
```

Merge Sort Implementation

Since Merge Sort is part of your final project, I will leave it to you to research the topic of how it works and how to implement it in Java. Merge Sort is a recursive sorting algorithm so it is expected that you will implement it accordingly.

I will be doing a plagiarism checker to make sure you aren't just ripping your merge sort code from a site like "geeksforgeeks.org" or "Stack Overflow". Yes, I know about those sites. They have value if you want to learn the underlining concepts but do not simply rip code with no understanding of how it works. Otherwise, you are just wasting both of our time.

Note: MergeSort is NOT currently part of your final project. This was part of the lesson plan before I switched to doing the Gaming API for the final project. If you are interested in learning about Merge Sort, I recommend you research it and write it yourself. I may offer an extra credit assignment for those who wish to implement merge sort in a unique way.

The array is sorted optimally...now what?

Solving problems such as max/min value become trivial with a sorted array. Simply look at the first or last element of the array (depending on if it is ascending or descending order.)

What about search? We no longer need to use linear search since we can make assumptions on our data set since we know it is sorted.

We will look at binary search now to look for a given value inside of our array. We will assume our array is sorted from smallest to largest (ascending).

Binary Search is the idea that you have a list of **sorted** information (such as a sorted array of integers) and you need to know if a certain value is in the list. If so, it is generally helpful to know a true / false value (such as a boolean) or to know what the index is of the matching value (such as an integer). In our implementation, we will use a **sorted** integer array as a formal parameter and we return an integer value giving the index of the first matching value. If there are no values in our array that match the number parameter, we return a -1 value (since arrays cannot legally contain negative indexes.)

The "gotcha" of binary search is that it must be efficient. Unlike linear search which allows us to iterate through the array, from beginning to end, one element at a time, binary search must use no more than the $\log_2$ of n iterations. For example, if we have a list with 16 sorted elements in it, the $\log_2$ of 16 is 4 so we must solve this in 4 iterations. The way this is possible is because, since our array is sorted, we can select "pivot" points in the array. Let's look at an example:

```
{ 1, 2, 3, 4, 5, 6, 7, 8 } <- This is our sorted array
                              (Let's assume search number is 6)
```

We start our search with three points: start, pivot, and end points. Our initial start point is index 0. Our initial pivot point is length / 2 (in this case: index 4), and our end point is length – 1 (index 7). These are shown below:

```
{ 1, 2, 3, 4, 5, 6, 7, 8 } <- This is our sorted array
  ^           ^        ^
  S           P        E   (S = start; P = pivot; E = end)
```

Once we have determined start, pivot, and end points, we check whether number is greater than the pivot point. If so, we assign the start point as being equal to the old pivot point. Our new pivot point becomes half way (roughly) between the new start point and the end point. We can express this mathematically as : S = P; P = ((E – S) / 2) + S

If the number were smaller than the pivot, we would assign the end point to the pivot and make the new pivot half way between the start point and the new end point. We can express this mathematically as: E = P; P = ((E – S) / 2) + S

Back to our example, we would do it the first way since the number (6) is greater than the pivot (5). Our new arrangement would look as follows:

```
{ 1, 2, 3, 4, 5, 6, 7, 8 } <- This is our sorted array
              ^  ^     ^
              S  P     E   (S = start; P = pivot; E = end)
```

Since our pivot is equal to the number, we simply return the index of the pivot (index 5). We were able to accomplish this in 2 iterations. The $\log_2$ of 8 is 3 iterations so we actually did better than $\log_2$ of n. However, the $\log_2$ of n is listed as such because it is the worst case.

Implementing Binary Search

Below is the code for a binary search that returns an index of the matching value (or -1 if not found):

```
/* Binary Search */
 public static int binarySearch(int a[], int val)
    {
        int start = 0;
        int end = a.length - 1;
        while (start <= end) {
            int pivot = start + (end - start) / 2;

            if (a[pivot] == val)
                return pivot;

            if (a[pivot] < val)
                start = pivot + 1;

            else
                end = pivot - 1;
        }
        // Failed search
        return -1;
    }
```