# Course Description

**Weekly Overview**

This week focuses on sorting algorithms, a continued exploration of complexity analysis, and an introduction to recursion. Students will program three quadratic sorts and perform complexity analysis on them. They will also being a discussion of recursion by connecting it back to recursive mathematical formulas.

# Institutional Learning Outcomes

**Main Objectives**

* Code selection sort, bubble sort, and insertion sort.
* Performa complexity analysis on the three sorts.
* Understand the insertion sort can be used on streaming data, whereas the other two cannot.

# Understand recursion, both mathematic

# Discipline Specific Outcomes

# Student Readings

None

**Daily Outline**

Day 1: Exploring Sorting

Day 2: Selection Sort and Bubble Sort

Day 3: Insertion Sort

Day 4: Test

Day 5: Introduction to Recursion

**Included Resources**

Sorting Exploration Student Activity

Lecture Notes: Selection Sort and Bubble Sort

Homework Assignment: Modified Sorts

Lecture Notes: Modified Bubble Sort and Insertion Sort

Lecture Notes: Introduction to Recursion

Homework Assignment: Recursion Worksheet

**In-Class Activity: Sorting an Array**

Part I: Programming a sorting algorithm

Work with a partner to program the following method that leaves the parameter array in a sorted order from least to greatest.

//A is an array of ints

//Upon completion, A is sorted from least to greastest

//void return

public static void search(int[] A){

}

Part II: Analyzing the speed of the algorithm

How fast does your algorithm work? The “thing to count” is the number of comparisons made in your algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| **Length of the Array** | **Best Case** | **Worst Case** | **Average Case** |
| 10 |  |  |  |
| 50 |  |  |  |
| 100 |  |  |  |
| 10,000 |  |  |  |
| 1,000,000 |  |  |  |
| *n* |  |  |  |

There are approximately 450 million social security numbers that have been issues. If a computer can perform 85 million comparisons per second, how long will it take (worst, best, average) to sort the array of all 450 million numbers?

What is the order of magnitude of your algorithm (in the worst case)? Is it linear? Quadratic? Cubic? Exponential? Something different? Use your chart to help you figure this out.

Part III: Brainstorming

Is there anyway that you can think to speed up your algorithm?

**Lecture Notes: Selection Sort and Bubble Sort**

**Bell Work (5 minutes)**

What is the complexity (order of growth) for the following?

public static void foo(int[] A){

for(int i=0; i<A.length; i++)

for(int j=0; j<A.length; j++)

System.out.println(“\*”);

}

***Answer:*** Quadratic, or *n*2.

**Main Lecture: Part 1 – Selection Sort (15 minutes)**

The most obvious way to sort an array is to go through and find the smallest elements and place it in the correct spot. Then find the second smallest element, and place it in the second spot.

Make the point that we do not want to duplicate the array. Duplicating the array has two drawbacks. First, it requires addition time to copy the elements back into the original array. Second, and more detrimental, it requires additional storage space. This may not seem like a big deal, but imagine trying to sort terabytes of data. In fact, if a file takes up more than half of your disk space, sorting would be impossible if we insist on setting up a secondary array.

So, where do we put the element that is in the spot we are trying to fill? We swap it with the element we are moving. Illustrate the algorithm with an array:

[4, 6, 1, 8, 7, 5, 9, 10]

The smallest element is 1, so we swap it with the 4:

[1, 6, 4, 8, 7, 5, 9, 10]

The next smallest element is 3, so we swap it with the 6:

[1, 3, 4, 6, 7, 5, 9, 10]

The next element is 4, and it is already in its place. (We could say that we “swap the 4 with the 4.”)

The successive states of the array are:

[1, 3, 4, 6, 7, 5, 9, 10]

[1, 3, 4, 5, 7, 6, 9, 10]

[1, 3, 4, 5, 6, 7, 9, 10]

At this point, the array is sorted.

Because the code for these sorts can be confusing, it is helpful to start with the pseudo code so that students understand the “narrative” of the algorithm first.

public static void sort(int[] A){

for(int i=0; i<A.length; i++){

find the smallest value beginning at index i

switch the elements at index i and the index of the min value

}

}

Presenting the code this way also allows us to properly segment the code off into parts, making the main portions easy to understand. This sort of programming is also known as “top down programming.”

Turn the pseudo code into actual method calls:

public static void sort(int[] A){

for(int i=0; i<A.length; i++){

int min\_index = findMinIndex(A, i);

swap(A, i, min\_index);

}

}

The task now is to fill in the two methods. First, let’s write the findMinIndex method:

//returns the index of the smallest value of A from index s

//to the end of the array

public static int findMinIndex(int[] A, int start\_index){

min\_index = start\_index;

for(int i= start\_index; i<A.length; i++){

if(A[i] < A[min\_index]){

min\_index = i;

}

}

return min\_index;

}

We now need to write the swap method.

//Leaves A in the same state, except that

//the elements at the indices i and j are switched

public static void swap(int[] A, int i, int j){

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

This is Selection Sort.

Note to teacher: it is extremely helpful at this point to go through an example array on the board so that students can see how the algorithm works.

What is the complexity of this algorithm? In it’s segmented version, it might be hard to see, and it might even seem linear because the main sorting method is a single for loop. However, the “combined” version of the sort is:

public static void sort(int[] A){

for(int i=0; i<A.length; i++){

int min\_index = start\_index;

for(int j= start\_index; j<A.length; j++){

if(A[j] < A[min\_index]){

min\_index = j;

}

}

int temp = A[min\_index];

A[min\_index] = A[i];

A[i] = temp;

}

}

The first time through the array, we look at all elements of the array to find the smallest, or *n* elements. The second time through, we look at *n* – 1 elements, the third time through *n* – 2, and so on. Therefore, the total number of elements looked at is *n* + (*n* – 1) + (*n* – 2) + … + 3 + 2 + 1, or: . (Actually, the last time through, we would not have to look at the “1” element, but this will not effect the complexity analysis.)

From Calculus, we remember that . Therefore, this is an *n*2 algorithm.

Notice, that this is an *n*2 algorithm in the best and worst (and therefore average) case. No matter what the state of the array is, whether it is sorted, random, or even backwards sorted, we always look at the same number of elements. Therefore, this is *always* a quadratic algorithm.

**Main Lecture: Part 2 – Bubble Sort (30 minutes)**

There is a different way to sort, which involves only looking at adjacent elements. The idea is to start at the beginning and look at elements right next to each other, and switch them if they are out of order.

Go through an example of an array with students so that they see the way the algorithm is intended to work.

[8, 2, 5, 9, 3, 1, 4, 6, 7]

During the first pass:

[**2, 8**, 5, 9, 3, 1, 4, 6, 7]

[2, **5, 8**, 9, 3, 1, 4, 6, 7]

[2, 5, 8, 9, 3, 1, 4, 6, 7] //no swap

[2, 5, 8, **3, 9**, 1, 4, 6, 7]

[2, 5, 8, 3, **1, 9**, 4, 6, 7]

[2, 5, 8, 3, 1, **4, 9**, 6, 7]

[2, 5, 8, 3, 1, 4, **6, 9**, 7]

[2, 5, 8, 3, 1, 4, 6, **7, 9**]

This is the end of the first pass.

To start the code begins with a pass through the array:

for(int i= 0; i<A.length-1; i++){ //notice the “– 1” so that we do not

if(A[i] > A[i + 1]){ //step out of bounds

swap(A, i, i + 1);

}

}

What is true after this code? (Answer: the greatest element is in its place – it has “bubbled up to the top.”)

The advantage that this has over the prior code is that, while marching through the array, not only have we place one element where it needs to go, but we have partially sorted the rest of the array. If we found something out of order, we switch it.

Now, we make a second pass. But this time, we only need to go up to the penultimate index, because the first pass put the greatest element in its place.

for(int i= 0; i<A.length-2; i++){

if(A[i] > A[i + 1]){

swap(A, i, i + 1);

}

}

What is true after this code? (Answer: two elements are in place, the greatest two.)

Teacher note: go through the second pass with the previous array. It is helpful, too, to finish the sort on this array.

We continue to make passes, which effective changes the A.length-\_\_\_ every time. The complete code is:

public static void sort(int[] A){

for(int j=1; j<A.length-2; j++){

for(int i=0; i<A.length-j; i++){

if(A[i] > A[i + 1]){

swap(A, i, i + 1);

}

}

}

}

Talk students through the “A.length-2” in “for(int j=1; j<A.length-2; j++)”.

For whatever reason, a more common presentation of Bubble Sort is to use the outer loop as the “swapping loop”:

public static void bubbleSort(int A[])

{

for (int i = 0; i < A.length-1; i++)

for (int j = 0; j < A.length-i-1; j++)

if (A[j] > A[j+1]){

swap(A, j, j+1);

}

}

Nevertheless, the algorithm works exactly the same. What is the complexity analysis of this? Well, the same mathematics applies. The first time through, we look at every element, or *n* elements. The second time through we look at *n* – 1 elements, and so on. So, like Selection Sort, this is an *n*2 algorithm.

**Homework:** Complete the “Modified Sorts” worksheet.

**Homework: Modified Sorts**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Rewrite Insertion Sort, but first find the greatest element and put it in its place, then find the second greatest element and put it in its place, etc.
2. Rewrite Bubble Sort, but start from the “end” of the array instead of from the beginning and “bubble down.”

**Lecture Notes: Modified Bubble Sort and Insertion Sort**

**Bell Work (5 minutes)**

If Bubble Sort makes no swaps the first time through, what is true of the array?

***Answer:*** The array is already sorted.

**Main Lecture: Part 1 – Modifying Bubble Sort (25 minutes)**

You should have seen from the Bell Work that, if no swaps are made in Bubble Sort, then the array is already sorted. In fact, this is true at every pass. If no swaps are made, then the sorting is completed. Remember that Bubble Sort not only places an element in the correct spot with every pass (which Selection Sort also does), but it does some work “along the way” by making swaps. Selection Sort “forgets” about any work it did on a given pass. This makes Bubble Sort run faster on average (even though they are both quadratic algorithms), but also gives Bubble Sort the ability to “bail out” early if it discovers that its work is done.

*Give the class some time to think about modifying Bubble Sort (they can work in pairs) to incorporate the early bail out. The code should be written on the board once they have had a chance the think through it.*

public static void bubbleSort(int A[])

{

for(int i = 0; i < A.length-1; i++){

boolean swap\_made = false;

for(int j = 0; j < A.length-i-1; j++){

if(A[j] > A[j+1]){

swap(A, j, j+1);

swap\_made = true;

}

if(!swap\_made)

return;

}

}

}

What is the complexity analysis of this?

|  |  |  |
| --- | --- | --- |
| **Case** | **Conditions** | **Complexity** |
| Best | An already sorted array | *n* |
| Worst | An array in reverse order | *n*2 |
| Average | N/A | *n*2 |

*Teacher Note: Average cases are very difficult to determine and involve some heavy statistical analysis. It is okay to tell the students this.*

So it is the “best case” that benefits the most from the early bailout. However, the average speed of the algorithm, while still quadratic, speeds up quite a bit.

**Main Lecture: Part 2 – Insertion Sort (20 minutes)**

The first two sorts can only run on arrays that are already filled. In many applications, we get data in a “streaming” fashion, requiring what we call an “online algorithm.” There is a third sort that can be applied to this situation: Insertion Sort. The philosophy of this sort is to assume that the first part of the array has already been sorted, and then place the “new” element in its proper place. The “assumption” is not really an assumption, because we are going to run the algorithm so that this is a guarantee.

Suppose we have the array:

[8, 2, 5, 9, 3, 1, 4, 6, 7]

The first element is in place, so we go to the second element. We “bubble it down”, so to speak. Because it is out of order with respect to 8, we switch them:

[2, 8, 5, 9, 3, 1, 4, 6, 7]

The next element is the 5. We switch it with the 8 because it is out of order:

[2, 5, 8, 9, 3, 1, 4, 6, 7]

We do *not* switch it with the 2. Note at this point that the first three elements are in order. The next element is the 9. Because it is in its proper place with respect to the 8, there is nothing to do, so we move on to the 3.

The 3 bubbles down quite a bit:

[2, 5, 8, **3, 9**, 1, 4, 6, 7]

[2, 5, **3, 8**, 9, 1, 4, 6, 7]

[2, **3, 5**, 8, 9, 1, 4, 6, 7]

The bold numbers represent the swaps. We continue like this until we are through the entire array.

Teacher note: finish the sort “by hand” with the above array.

This algorithm has quite a bit in common with Bubble Sort. It still compares adjacent elements and swaps them in they are out of order. However, whereas Bubble Sort starts each pass at the beginning and bubble to the middle, Insertion Sort starts the passes in the middle and bubbles down to the beginning.

Also, Bubble Sort needed an explicit “bail out” placed in the algorithm. Insertion Sort has a sort-of built-in bail out. This is because Bubble Sort needs to bail out of making primary passes early, whereas Insertion Sort bails out of each pass early. This was the case when placing “9” in its proper place: it was already there, so there is nothing to do.

What is the code?

(Work through this with students.)

public static void insertionSort(int A[]){

   for(int i = 1; i < A.length; i++){

       int j = i-1;

while(j >= 0 && A[j] > A[j+1]){

           swap(A, j, j+1);

           j = j-1;

       }

   }

}

What is the complexity analysis of this? It is identical to Bubble Sort.

|  |  |  |
| --- | --- | --- |
| **Case** | **Conditions** | **Complexity** |
| Best | An already sorted array | *n* |
| Worst | An array in reverse order | *n*2 |
| Average | N/A | *n*2 |

Notice, however, that the length of the array could “change”. (Yes, I know, an array cannot change length, but if we see this as a stream of data coming in instead of an array with a static length, then we get the point.) A “new” element could be added to the array, and we would be able to continue the algorithm indefinitely. This is another way of saying that Insertion Sort “works” on an infinite array, whereas Bubble Sort does not (it could never finish its first pass).

**Homework:** None

**Lecture Notes: Introduction to Recursion**

**Bell Work (5 minutes)**

What are the next six terms in the sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, …

What is the name of the sequence?

***Answer:*** 55, 89, 144, 233, 377, 610; Fibonacci Sequence

**Main Lecture: Part 1 (25 minutes)**

Review with students arithmetic sequences, geometric sequences, and how to define them both explicitly and recursively.

|  |  |  |
| --- | --- | --- |
| **Sequence** | **Explicit** | **Recursive** |
| Arithmetic |  | is given, and . |
| Geometric |  | is given, and |

Work through the advantages and disadvantages of each. Explicit formulas are great for calculating things like the 1000th term. Recursive formulas are great if you know the previous element. Emphasize that the recursive formulas *must* have a base case, otherwise the sequence never gets off the ground.

Have students write the recursive formula for the Fibonacci Sequence (from the Warm Up), and emphasize that *two* base cases are necessary because of the double recursion in the definition.

**Main Lecture: Part 2 (20 minutes)**

Work with students to code two versions of the geometric sequence method: one explicit and one recsursive:

public static double geometric(double first\_term, double r, int term){

return first\_term \* Math.pow(r, term-1);

}

public static double geometric(double first\_term, double r, int term){

if(term == 0)

return first\_term;

return geometric(first\_term, r, term-1);

}

Quite a bit of discussion will need to happen for the recursive version. It is critical to emphasize the base case. Without it, the code gets caught in an infinite loop. In this case, it is okay for students to see that the explicit version is easier to program and understand.

Have students write the code for the Fibonacci method (both versions):

public static int Fib(int term){

int low = 0;

int high = 0;

for(int i=0; i<term-1; i++){

int temp = low + high;

low = high;

high = temp;

}

}

public static int Fib(int term){

if(term == 0 || term == 1)

return 1;

return Fib(n-1) + Fib(n-2);

}

Note in the recursive case the presence of 2 base cases and 2 recursive calls. Without *both* base cases, the method gets caught in an infinite loop.

In this case, students should see that the recursive call is easier to understand. This is probably because the Fibonacci Sequence is “recursive by nature.” As a note, there is an explicit formula known as Binet’s formula:



But one would be hard pressed to consider this “easier to understand.”

Next week we will be going through many more examples of recursive programming, so it is okay today to take the time to make sure that students understand what is happening.

**Homework:** Recursion Worksheet

**Homework: Recursion**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Write a recursive method to find the largest element in an array of numbers.

public static int max(int[] A){

}

1. Write a recursive method to print the elements in an array of characters.

public static void print(char[] B){

}