Algorithms and Data Structures with Java

By Andira Putri

From Microsoft's edX course: https://www.edx.org/course/algorithms-and-data-structures

Part One: Basic Algorithms

- **Human vs. Computer Thinking: Introduction to Algorithms and Data Structures**

Say you go to the grocery store for cereal and milk. A human and computer will go about this task using different algorithms. An **algorithm** is a set of steps or rules that solves a specific problem. Going back to the grocery store... A human's "algorithm" would be to walk into the store, go to the cereal aisle and choose any box, head to the dairy aisle and choose a milk, and then check out.

A computer's algorithm will be different because it is not capable of randomness and improvisation like humans are. To illustrate, a human could walk into a store and decide what cereal they want once they see the options in the aisle. A computer is not capable of making that decision- without knowing what cereal they want before entering the store, they would just walk in and be completely lost. It is necessary for the computer to know *beforehand* what steps to take; it is not capable of the "in-between" steps, i.e. picking out a cereal on the spot.

Now, let's consider the milk. The dairy aisle has many offerings, but I just want the store brand gallon of 2% milk. A human would naturally just pick up this specific item without having to go through all the other things in the dairy section. However, a computer would have to go through the yogurts and heavy creams, and once it reached the milk offerings, it would have to distinguish between size, cost, fat content, etc. It's important to organize the dairy offerings such that a computer can complete the task in the most efficient way possible, which brings us to **data structures**. Data structures are the ways data is organized in a computer's memory, much like how milk is organized within a dairy aisle. If my milk data is more organized and structured, I would have specific sections for 2% and skim milk. I know I don't want skim milk, so I would just focus on the 2% milk selections and choose from there, making the selection process much easier on the computer. When you have data that is well-organized, algorithms are much more efficient.

- **Selection Sort Algorithm**

Imagine you have 5 cards numbered 1, 3, 5, 9, 10. They are randomly assorted from left to right. Your task is to organize them in ascending order. Let's do it here:

(5, 9, 1, 10, 3)

Looking at the entire set, I first find the minimum which is 1. I move it to the left to create an ascending order. Originally, there was a 5 in the very left. We call the 5 a *temporary value*, and we swap its position to where the 1 initially was. This is called a swap.

1, **(9, 5, 10, 3)**

Card with value 1 is done, so we can ignore it. Now, we look at the smaller subset of cards in blue: 9, 5, 10, and 3. We choose the minimum of this smaller subset, 3, and move it to the left of the subset. We swap the 9 and 3.

1, 3, **(5, 10, 9)**

Next, we consider the smaller subset: 5, 10, 9. Here, 5 is the minimum, and it's already the left-most number within the subset. We don't have to do anything!

1, 3, 5, **(10, 9)**

Finally, we consider the last and smallest subset: 10, 9. Here, 9 is the minimum, so we move it to the left and swap its location with 10.

1, 3, 5, 9, 10

The cards are now in ascending order. This process is the **selection sort algorithm.** In Java, we would implement this task using the following for-loop code:

```
package sortSelection;
public class array {
      public static void main(String[] args) {
      int[] lst = {5,9,1,10,3}; //this is my set of cards
      int min = 0; //starting index
      int n = lst.length; //a guide for how many iterations will take place

          for (int i = 0; i < n; i++) {
              min = i;
              for(int j = i + 1;j<n;j++)
              {
                  if(lst[j] < lst[min]) { min = j;}
              }
              int temp = lst[i];
              lst[i] = lst[min];
              lst[min] = temp;
              System.out.println(lst[i]);
          }
      }
}
```

```
//i++ means adding 1 to i after each iteration, the first for loop goes through n-1
iterations. It divides the cards into subsets

//the second for loop accounts for moving the minimum of each subset and swapping
cards with a temporary object
```

- **Linear Search**

My deck of cards are flipped so that I can't see the numbers. Say that I want to choose a 10 within my randomly arranged stack. The linear search algorithm entails going from left to right flipping each card until I get a 10. Blue means the card is showing the number.

5, 9, 1, 10, 3 --> **5,** 9, 1, 10, 3 --> **5, 9,** 1, 10, 3 --> **5, 9, 1**, 10, 3 --> **5, 9, 1, 10**, 3

The code for this algorithm is:

```
public class LinearSearch {

	public static void main(String[] args) {

		int[] lst = {5,9,1,10,3}; //this is my deck of cards
		int v = 1; //this is the card I would like to find
		int n = lst.length; //how many iterations should take place is n-1
		for (int i = 0; i < n; i++) {
			if (lst[i] == v) {
				System.out.println("Found! It is at " + i);
				return;
			}
		}
		System.out.println("The element is not in the array.");
		return;
		//for loop goes through each card (turns it number-side up) until my
desired card is found.
		//otherwise, it says that the card is not in my set.
	}
}
```

- **Bubble Search (and I guess also Bubble Sort...)**

Bubble search is kind of like selection sort, but not quite. Say that I have this list: {3, 2, 1, 4}, and I would like to order them in ascending order.

Here is a diagram of the algorithm. The blue cells are the numbers we are considering for swapping. Rows represent the steps in the algorithm.

3	2	1	4
2	3	1	4
2	1	3	4
2	1	3	4
1	2	3	4

I start with the two numbers in the left. If the number on the left is greater than the right, I swap them. In row 1, 3 > 2 and I swap them in the next row.

Now, the first column is done. We move on the the two numbers starting in the next column. Here, 3 > 1 so I swap them in the next row.

The first and second columns are done. Now, we start with the two numbers starting with the third column. 3 < 4, so we do not need to do anything.

Our list now stands at: {2, 1, 3, 4}. However, it's not sorted the way we would want. We perform the bubble search algorithm again, and we start with the two numbers in the left. 2 > 1, so we swap them in the next step.

Finally, we look at the next pair of numbers starting in the second column. 2 < 3, so we do not do anything. Our list is finished! It's like following a staircase pattern like you would echelon form in linear algebra.

Here is the code to perform the bubble search algorithm:

```java
import java.util.Arrays;
public class bubbleSearch {
	public static void main(String[] args) {
		int[] lst = {3,2,1,4};
		int n = lst.length;
		boolean swapped;
		do
		{
			swapped = false; //initialization; assumes list is not ordered?
			for (int i = 0; i < n-1; i++) { //For n-1 iterations
				if (lst[i] > lst[i+1]) {
				int temp = lst[i];
				lst[i] = lst[i+1];
				lst[i+1] = temp; //Changes positions if element i > i+1
				swapped = true;
				}
			}
		} while (swapped == true);
```

```
                System.out.println(Arrays.toString(lst));
        }
}
}
```

Part Two: Algorithmic Analysis

- **Merge Sort**

Merge Sort involves splitting a set of numbers at the midpoint until they are in the smallest possible groups, ordering them within the group, and then merging all sets back together. For example:

```
6 5 3 1 8 7 2 4

6 5 3 1     8 7 2 4

6 5   3 1   8 7   2 4

5 6   1 3   7 8   2 4
```

Now, we order and merge the two left pairs and the two right pairs.

```
1 3 5 6     2 4 7 8
```

Merge again by comparing the left and right lists.

```
1 2 3 4 5 6 7 8
```

The code to run the merge sort algorithm is:

```
Static  int[] mergeSort(int[] lst) {

int[] lst = {6,5,3,1,8,7,2,4}
int n = lst.length;
    int[] left;
    int[] right;

    // create space for left and right subarrays
    if (n % 2 == 0) {
        left = new int[n/2];
        right = new int[n/2];
    }
    else {
        left = new int[n/2];
        right = new int[n/2+1];
    }

    // fill up left and right subarrays
    for (int i = 0; i < n; i++) {
        if (i < n/2) {
            left[i] = lst[i];
        }
        else {
            right[i-n/2] = lst[i];
        }
    }

    // recursively split and merge
    left = mergeSort(left);
    right = mergeSort(right);

    // merge
    return merge(left, right);
}
// the function for merging two sorted arrays
static int[] merge(int[] left, int[] right) {
    // create space for the merged array
    int[] result = new int[left.length+right.length];

    // running indices
    int i = 0;
    int j = 0;
    int index = 0;

    // add until one subarray is deplete
    while (i < left.length && j < right.length) {
        if (left[i] < right[j]) {
            result[index++] = left[i++];
        {
        else {
            result[index++] = right[j++];
        }
    }

    // add every leftover element from the subarray
    while (i < left.length) {
        result[index++] = left[i++];
    }

    // only one of these two while loops will be executed
    while (j < right.length) {
        result[index++] = right[j++];
    }

    return result;
}
```

Given the complexity of this algorithm, it is better to perform bubble search in most cases.

- **Binary Search**

This is an efficient algorithm, and it's even faster than linear search. However, the starting list must already be sorted. Our list is {1,2,4,**5**,7,8,9}.
We select the midpoint, 5. We are searching for 2. We see if the midpoint equals the number we are looking for, and 5 does not equal 2. We know 2 < 5, so we eliminate the numbers greater than or equal to 5.

Our list now becomes {1,**2**,4}. 2 is now the midpoint, and it is exactly the number we are looking for. Now, say we are not searching for 2, and we are looking for 4. With the list {1,**2**,4}, the algorithm says the midpoint 2 does not equal 4, so it eliminates the numbers less than or equal to 2. We are left with {**4**}, which is exactly what we need.

The code for the binary search algorithm is:

```
static boolean binarySearch(int v, int[] lst, int low, int high) {

int[] lst = {1,2,4,5,7,8,9};
if (low > high) {
        System.out.println("not found");
        return false;
    }

    int middle = (low+high)/2;

    if (v == lst[middle]) {
        System.out.println("found! It is at " + middle);
        return true;
    }
    else if (v > lst[middle]) {
        return binarySearch(v, lst, middle+1, high);
    }
    else {
        return binarySearch(v, lst, low, middle-1);
    }
}
```

- **Big O Notation**

Big O notation describes the efficiency and performance of an algorithm. As input size increases, what are the effects on the algorithm's running time and space requirements? To study Big O notation, we assess the bubble sort and merge sort algorithms.

The bubble sort pseudocode is:

```
repeat until no swaps
    for i from 0 to n-2
        if i'th and i+1'th elements are out of order
            swap them
```

After the first complete loop, it is guaranteed the largest element will be in the last position of the array. After the second complete loop, the second largest will be immediately before the position of the first largest element. For this algorithm, we need to run it for “n” loops, and in every loop, there are “n-1” comparisons. Therefore, there are n*(n-1) steps, or n^2 - n. As n goes to infinity, the “-n” term becomes irrelevant, so we refer to this as order n^2 and notate as $O(n^2)$.

Now, we consider the merge sort algorithm. The pseudo code is:

```
if the size of array < 2: // already sorted
    return array
else
    sort left subarray
    sort right subarray
    combine them
```

Initially, we have n arrays of size 1 (after the split). There are three *steps* to merge sort: splitting, comparing, and combining. There are n/2 combinings, so at this point, there are 3n/2 steps. After this, we look at arrays of size 2. We merge n/2 arrays of size 2 into n/4 arrays of size 4. Combining two arrays of size 2 requires 6 steps (both arrays need to be split, compared, and combined), and there are n/4 combinings. Now, there are 6n/4 == 3n/2 steps. This pattern repeats, and it’s easy to see that each level requires 3n/2 steps which is linear to n, so we have O(n) steps.

Taking this into account, we look at *levels*. At every level, we cut each array by half until we reach size 1. Mathematically, the amount times necessary until we get to the size 1 arrays is $\log_2(n)$, and this is in order O(log(n)).

Since merge sort has O(n) steps and O(log(n)) levels, the algorithm is in the order O(nlog(n)). This outperforms the bubble sort algorithm, $O(n^2)$, since nlog(n) grows much slower as n approaches infinity.

Even though merge sort beats bubble sort in terms of efficiency, merge sort does take more space. At each level, we split arrays that add up to size n, but since the algorithm considers only one level at a time, it is not necessary to give space for more than an n-sized array. Therefore, merge sort is in order of O(n) space. As for bubble sort, we didn’t need to use extra space at all, so it is in order of O(1) space, meaning constant.

Part Three: Basic Data Structures

- **Linked Lists**

Before going into linked lists, we review arrays. When you define arrays, your computer will save adjacent with the same size, for easy access to the data. With an array of size N=1,000,000, the computer reserves 1,000,000 spaces in memory. If the first space is at 300, the last space will be 1,000,300. If we want to locate the 500,000th element in the array, we add it to the location of the first space (so it will be the address off 500,300). Arrays are efficient, O(2), but we cannot resize, add, or delete the array without making a copy of the array, taking O(N) steps. This is why we have linked lists (though it will not give you access like arrays).

Linked lists consist of multiple nodes. Let’s start with this simple node with two parts.

First **Node**

Pointer | Data |
| Pointer |

The data could be mixed data or multiple variables, and there’s a pointer or reference (in this case, First). The only way to access the node is by knowing its address.

First **Node**

Pointer ---> | Data |
| Pointer |

We take the address of the node and put it in the First pointer.

The code to implement a linked list is:

```
// create a new linked list
LinkedList<Integer> lst = new LinkedList<Integer>();

// add elements
lst.add(15); // [15]
lst.add(19); // [15,19]
lst.add(21); // [15,19,21]
lst.addFirst(13); // [13,15,19,21]
lst.addLast(24); // [13,15,19,21,24]
System.out.println(lst);

// remove elements
```

```
lst.removeFirst(); // [15,19,21,24]
lst.removeLast(); // [15,19,21]
lst.remove(19); // [15,21]
System.out.println(lst);
```

- **Stack**

A stack is a special type of linked list. This structure is good for the fast access of the top element. There are three essential stack operations: push, pop, peek. Pushing pushes a new element to the top of the stack; popping removes and returns the top element from the stack. Peeking returns the top element without removing it from the stack.

```
// create a new stack
Stack stack = new Stack();
stack.push(new Integer(3)); // [3]
stack.push(new Integer(5)); // [3,5]
System.out.println(stack.peek()); // prints 5 but does NOT remove it
stack.push(new Integer(6)); // [3,5,6]
int x = stack.pop();
System.out.println(x); // prints 6 and removes it
System.out.println(stack); // prints [3,5]
```

- **Queue**

Queue is very similar to a stack, but it's a first-in, first-out situation. Think about standing in line at a cafeteria. The first person to stand in line is the first person to purchase their food and leave. The first element to enter a queue is the first element removed. There are three main operations. Enqueue adds an element to the end of a list, dequeue removes the first element, and peek returns the first element WITHOUT removing it.

```
Queue queue = new LinkedList();
queue.add(2); // [2]
queue.add(4); // [2,4]
System.out.println(queue.peek()); // prints 2
queue.add(5); // [2,4,5]
queue.poll();
System.out.println(queue); // prints [4,5]
```