CSC317 Fall 2022 Myles Greene-Beaupre
Instructor: Odelia Schwartz
Assignment 1

August 30, 2022 (due Sept 6 2022 by midnight, on Blackboard). Note: Some of the questions are based on the Cormen textbook.

1. **Come up with a real-world problem in which only the best solution will do (example, it's important for this problem to find a solution that only gives the exact right answer). Then come up with a real-world problem in which a solution that is approximately the best is good enough. Note this question does not require explanation of how to solve your problem; just specifying example problems.**

If your job is to file sensitive documents into their proper positions in a filing cabinet, there is only one exact right answer, which is making sure every document is put in its proper category. If there is a document pertaining to government nuclear weapons filed into the "Department of Agriculture" category, then your solution to the problem is incorrect because there is a document out of place.

On the other hand, if you are throwing a baseball to your friend, you don't need to execute an exactly perfect throw that will follow the perfect path to your friend's catcher's mitt. It doesn't have to have the perfect angle, or the perfect velocity, or follow the same path through the air every time. You just need to get it in the general vicinity of your friend and he will move his hand accordingly in order to catch it, so an exact solution is not required to play catch.

2. **On the first birthday, we light one candle; on the second birthday 2 candles; and on the 120th birthday 120 candles. How many candles do we light total in all 120 birthdays? *Use the proper summation formula* we learned in class.**

$$\sum_{j=1}^{n} j = 1 + 2 + 3 + \dots n = (n(n + 1)) / 2$$

In this example, n = 120 to account for every birthday from age 1 to 120:

$$\sum_{j=1}^{120} j = 1 + 2 + 3 + \dots 120 = (120(120 + 1)) / 2 = (120 * 121) / 2 = 14520 / 2 = \mathbf{7260\ candles}$$

3. **Write an algorithm in Pseudocode to compute the sum of an array of n numbers. Write down the loop invariant that will make your algorithm correct. Then *prove correctness of the loop invariant by initialization, maintenance, and termination*.**

```
function sum(arr) {
    var sofar = arr[0];
    for(int i = 1; i < arr.length; i++) {
        sofar += arr[i];
    }
    return sofar;
}
```

The loop invariant is that the variable "sofar" is the sum of all elements of the array that the loop has gone through so far. To prove correctness, we first start with **initialization**:

```
var sofar = arr[0];
```

This line initializes our sofar variable to the first element of the array so that when the loop starts at i = 1, the variable sofar is the sum of all elements from arr[0] to arr[i-1]. Since i = 1, arr[i-1] = arr[1-1] = arr[0], so when the loop starts sofar should be the sum of all elements from arr[0] to arr[0]. The only element from arr[0] to arr[0] is arr[0], so the variable sofar is set equal to arr[0] and this ensures our loop invariant before the loop starts.

For our **maintenance** step, because of our initialization step we can assume that before the loop starts that sofar is the sum of all elements from arr[0] to arr[i-1]. Now we just have to ensure that after an iteration of the loop that this is still true. Inside the loop we are doing this:

```
sofar += arr[i];
```

So when i = 1, the value of sofar is increased by the value of arr[1]. We want to ensure that after this one iteration of the loop, that sofar is the sum of all elements from arr[0] to arr[i]. After a single run around the loop, i = 2. Before we do sofar += arr[i] for a second time, can we verify that sofar is still the sum so far? If i = 2, sofar should be the sum from arr[0] to arr[i-1] = arr[2-1] = arr[1]. The elements of the array from arr[0] to arr[1] are the values arr[0] and arr[1]. The initialization step set sofar = arr[0], and the first run around the loop added the value arr[1] to sofar, so this verifies that when i = 2 on the second run around the loop, sofar is still the sum so far because sofar = arr[0] + arr[1]. This is true generically, so at any point in the loop we can say when i = k, we know that sofar = arr[0] + … + arr[k] (in other words, sofar is the sum of all elements from arr[0] to arr[k-1]).

Finally, for the **termination** step, when the loop terminates we know that i = arr.length because we specified that our loop should run so long as i < arr.length. If we have n elements in our array, then i = arr.length = n. In other words, i is equal to the amount of elements in our array when the loop terminates. The variable sofar should be the sum of all elements from arr[0] to arr[i-1]. We can verify that this is true because before the loop ran sofar = arr[0] (initialization step), and the loop only ran so long as i < n, so on the last iteration of the loop i = n-1. Therefore, when the loop terminates, sofar = sum of all elements from 0 to n-1 (our loop invariant is still true), and we know we have gone through the entire array because we're using zero based indexing and the first element of the array is arr[0] so the last element is arr[n-1] where n is the length of the array.

4. **Consider sorting n numbers stored in array A *from largest to smallest* (in reverse order). For example, if A=[7,2,6,8] then your algorithm will return [8,7,6,2]. Use the following approach similar to what is known as selection sort: first find the largest element of A and exchange it with the element in A[1]. Then find the second largest element of A, and exchange it with A[2]. Continue in this manner up to finding the n-1 largest element of A and exchanging it with A[n-**1].

(a) **Write pseudocode for this algorithm.**

```
function maxIndex(arr, start) {
    var sofar = start;
    for(var i = start+1; i < arr.length; i++) {
        if(arr[i] > arr[sofar]) {
            sofar = i;
        }
    }
    return sofar;
}

function reverseSort(arr) {
    for(var i = 0; i < arr.length; i++) {
        var tmp = arr[i];
        var maximum = maxIndex(arr, i);
        if(maximum != i) {
            arr[i] = arr[maximum];
            arr[maximum] = tmp;
        }
    }
    return arr;
}
```

(b) **Demonstrate your pseudocode on the array [7,2,6,8].**

reverseSort([7, 2, 6, 8]) is called. A loop starts to iterate over every element of the array starting at i = 0 and ending when i >= arr.length. Since we have 4 elements, arr.length = 4, so the contents of the loop will execute for i = 0, i = 1, i = 2, and i = 3. The loop executes as follows:

<u>i = 0</u>
Variable tmp is set to 7 because arr[0] = 7. Variable maximum is set to 3 because the maxIndex function returns the index of the biggest number in the array starting at parameter "start," which is i = 0 so the entire array is considered. Finally, since maximum != i, arr[0] is set equal to the value of arr[3], and arr[3] is set equal to tmp = 7. The effect of this is that the values at arr[0] and arr[3] are swapped, and the array now looks like this:

[8, 2, 6, 7]

<u>i = 1</u>
tmp is set equal to 2 because arr[1] = 2, and maximum is set equal to 3 because maxIndex(arr, 1) = 3 since 7 is the biggest number in the array if you only consider the numbers to the right of the first element (start = i = 1). The values at arr[1] and arr[3] are swapped since maximum != i, and the array looks like this:

[8, 7, 6, 2]

<u>i = 2</u>
tmp is set equal to 6 because arr[2] = 6, and maximum is set equal to 2 because the biggest number to the right of 7 is 6 which has index 2 in the array. Now, since maximum is equal to i, no swap is performed and there is no change to the array since 6 is already in its correct position:

[8, 7, 6, 2]

<u>i = 3</u>
tmp is set equal to 2 because arr[3] = 2, and maximum is set equal to 3 because the biggest number to the right of 6 is 2 which has index 3. Again, maximum is equal to i, so no swap is performed and the array is unchanged since 2 is already in its correct position:

[8, 7, 6, 2]

The array has been looped through and the result is an array sorted from greatest to least value.

(c) **What is the run time for this algorithm when A is sorted from smallest to largest (e.g., worst case)? What is the run time when A is sorted from largest to smallest (e.g., best case or already sorted)? Explain your answer and if/how it differs from Insertion Sort.**

Insertion sort has a best case complexity of O(n) when the array it is given is already sorted and no swaps are required, but O(n^2) when it is in reverse sorted order because you have to go through n elements from left to right and then go back through it from right to left to ensure that arr[0] is swapped with arr[n-1], arr[1] is swapped with arr[n-2], etc. You need n swaps, which means O(n^2) time complexity.

In the case of my reverse sort algorithm, no matter if you have the best case or the worst case, you need to go through n elements in the reverseSort() function and for each of these elements the maxIndex() function is called, which pretty much has to go through n elements too (it goes through n elements, then n-1, then n-2, etc.), so I would say that the worst case and the best case are both O(n^2). So it is a worse algorithm than insertion sort because insertion sort at least has a best case of O(n).