Ronald Chatelier

1. Write Pseudocode for the following

   a. Write pseudocode to determine if there are more odd or even numbers in a list of integers. It should take a list of integer numbers as input and return either “odd” or “even”. For example, if given the list [1,2,3,4,5] as input your pseudocode should return “odd”

```
procedure oddEven (int arr[])
    int even = 0
    int odd = 0
    // results of odd numbers in list
    for (int i = 0; i < arr.length; i++)
            if (arr[i] % 2 == 0)
    odd++
    else
    even++
    // returns “odd” for the number of integers if odd greater than even
    if (odd > even)
    return odd
else
    return even
```

   b. Cosine similarity (cos sim) is a commonly used similarity metric that measures how similar two vectors (arrays) of numbers are. The equation for cosine similarity is given by

$$sim(\boldsymbol{A}, \boldsymbol{B}) = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{||\boldsymbol{A}|| * ||\boldsymbol{B}||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

   where A_i and B_i are components of vector **A** and **B** respectively. Write pseudocode to calculate cosine similarity of two vectors **A** and **B**. it should take two arrays of numbers, **A** and **B**, as arguments and return a single value, their cosine similarity.

```
// results of sigma A and B
// numerator portion of fraction
aTimesB = 0
for (i = 0 to length - 1) {
aTimesB += (a[i] * b[i])
}
```

```
// results of sigma A^2
// denominator portion of fraction
aTwo = 0
bTwo = 0
for  (i = 0 to length - 1) {
aTwo += (a[i] * a[i])
bTwo += (b[i] * b[i])
}
// finding the cosine similarity between A and B
aTwo = sqrt(aTwo)
bTwo = sqrt(bTwo)
results = aTimesB | aTwo * bTwo
return results
```

2. Big-O analysis (Worst Case Time Complexity)

a. Find the worst case runtime (big-O notation) for the following pseudo code which returns true if an integer n is prime, false if it is not prime.

```
 procedure isPrime(n)
// O (1)
if n <= 1
// O (1)
return false
    // O (1)
    else if n <= 3
    // O (1)
    return true
     // O (1)
    else
     // O ( sqrt(n))
     for i in 2 to sqrt(n)
    // O (1)
     if n % i == 0
              // O (1)
               return false
    // O (1)
     return true
```

The for loop would be iterating sqrt(n) times since n is not the prime, which makes the complexity O (sqrt(n)).

(O (sqrt(n)) * O (1)) + O (1) + O (1) = O (sqrt(n))

Worst case runtime: O (sqrt(n))

b. Find the worst case runtime (big-O notation) for the SelectionSort algorithm. This algorithm sorts an array A of length n.

```
proceedure SelectionSort(array a, length(A) = n)
// O (n) time complexity
for i in 0 to n - 2
        // O (1) time complexity
        maxIndex = i
        // O (n) time complexity
        for j in (i + 1) to (n - 1)
                // O (1) time complexity
                if a[j] > A[maxIndex]
                // O (1) time complexity
                maxIndex = j
                // O (1) time complexity
                swap(A[i], A[maxIndex])
```

Both for loops are iterating by O (n) times, which makes the complexity O (n^2).

O (n) * O (n) = O (n^2)

Worst case runtime: O (n^2)

c. Find the worst case runtime (big-O notation) for BogoSort, pseudocode is given below:

```
procedure BogoSort(array A)
        while not isInOrder(A):
                // O (n!) time complexity
                shuffle(A)
```

shuffle() randomly reorders A. isInOrder() returns true if the list is in order, false otherwise. It has the following pseudocode:

```
// O (n) time complexity
procedure isInOrder(array A, length(A) = n)
        for i in 0 to n-1
                if A[i] > A[i+1]
                        return false
```

With shuffle(A) having it randomly reorders A, it would make the complexity O(n!). As

the numbers are checked to be in order, it will make the complexity O (n).

O (n!) * O (n) = O (n! * n)

Worst case runtime: O (n! * n)

3. Find the average runtime complexity of binary search

procedure binary search (x: integer, $a_1,a_2,\ldots, a_n$: increasing integers)
$i := 1$ {i is the left endpoint of interval}
$j := n$ {j is right endpoint of interval}
while $i < j$
// O (n/2) time complexity
$m := \lfloor (i + j)/2 \rfloor$
if $x > a_m$ then $i :=$ m + 1
else $j :=$ m
if $x = a_i$ **then** *location* := i
else *location* := 0
return *location*

With the while loop of i < j, it divides by 2 causing the size of the array to become halved until it ends up with 1. This will make the average runtime complexity O (log n) as the array is being run to the maximum of times for the integers.

Average runtime complexity: O (log n)

4. Greedy strategy

a. Does using the greedy algorithm for making change of n cents give a correct solution if a nickel was worth 6 cents instead of 5? (we have 1, 6, 10 and 25 cent coin denominations).

No, since the greedy algorithm will have an issue with the 1, 6, 15, and 25 coin denominations. It will start off with the biggest cent, 25, go down to 10, skip 6, and stop at the lowest cent, 1. This will end up having to use 2 nickles since the nickel is with 6 cents.

b. We have n unique items. For i = 1,2,3,…,n, each item has a weight $w_i > 0$ and value $v_i > 0$. Write a greedy algorithm to find the maximum value of items that can be carried in a knapsack with a maximum weight capacity of W. An item may be broken into pieces and only a fraction put into the knapsack. It should take an array of weights, an array of item values and the weight limit of the knapsack as input. The array of weights and items are ordered such that w_i corresponds to the same item as v_i.

```
procedure knapsackCapacity (w [1 … n], v [1 ... n], W)

for i = 1 to n
     compute (v(i) / W(i))
for i = 1 to n
// comparing the weight capacity
if (w > 0 and W(i) <= w)
          // weight of item subtracted
          W = w - W(i)
          finalWeight = finalWeight + v(i)
 else
 if (w >= 0)
     finalWeight = finalWeight + (W/w(i)) + v(i)
```