**Question 2 (a)**

Since the pseudocode might be hard to understand due to a lot of indentation and such, I am attaching the code that I wrote to sort the array

*int* max = 0;

*int*[] copied = new *int*[array.length];

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

*int* num = array[i];

if(max < num) {

max = num;

}

copied[i] = num;

}

*int* maxNumDigits = (*int*) *Math*.log10(max) + 1;

*LinkedList<LinkedList<Integer>>* buckets = new *LinkedList<LinkedList<Integer>>*();

for(*int* p = 0; p < 10; p++) {

buckets.add(new *LinkedList<Integer>*());

}

for(*int* j = 1; j <= maxNumDigits; j++) {

for(*int* k = 0; k < array.length; k++) {

*int* number = copied[k];

*int* digit = (*int*) ((number % *Math*.pow(10, j)) / *Math*.pow(10,j-1));

buckets.get(digit).add(number);

}

*int*[] partiallySorted = new *int*[array.length];

*int* numAt = 0;

search:

for(*int* m = 0; m < 10; m++) {

while(!buckets.get(m).isEmpty()) {

partiallySorted[numAt] = buckets.get(m).pollFirst();

numAt++;

}

if(numAt == array.length) {

break search;

}

}

copied = partiallySorted;

}

return copied;

**Question 2 (b)**

The running time for the algorithm comes out as:

T(n) = 9n + 35 + ( ⎣ log10(max) ⎦ + 1) \* (12n + 176) ------- > O(n)

where ⎣ log10(max) ⎦ denotes the floor of log10(max). (Since we are dealing with numbers greater than 1, log will not be negative and so taking the floor is equivalent to casting the value to an integer)

The algorithm cannot be used to sort an **arbitrary** set of numbers because suppose that even though we have a small array (say n = 5 numbers) but the max of those numbers is say 10^64. This would significantly increase the number of steps that we would have to take and the algorithm would waste a lot of time doing extra computations just because we have one number that is super large. This makes the algorithm inefficient for an arbitrary set of integers. This algorithm would be good to use in a situation where the numbers are within some known range

**Question 4 (a)**

**Algorithm**: eccentricity(vertex u)

**Input**: a vertex u from the graph

**Output**: the eccentricity of u

q ← new Queue()

setVisited(u, true)

setDistance(u, 0)

q.enqueue(u)

eccentricity ← 0

while(!q.empty()) do

w ← q.deque()

eccentricity ← getDistance(w)

for all v ∈ getNeighbors(w) do

if (!getVisited(v)) then

setVisited(v, true)

setDistance(v, getDistance(w) + 1)

q.enqueue(v)

return eccentricity

**Question 4 (b)**

**Algorithm**: is2colorable(vertex u)

**Input**: a graph vertex u

**Output**: true if the graph to which u belongs is 2-colorable, and false otherwise

q ← new Queue()

setVisited(u, true)

setColor(u, 0)

q.enqueue(u)

while(!q.empty()) do

w ← q.deque()

for all v ∈ getNeighbors(w) do

if(!getVisited(v)) then

setVisited(v, true)

setColor(v, 1 – getColor(w))

q.enqueue(v)

else

if(getColor(v) == getColor(w)) then

return false

return true