Problem 7.1
For code implementation, see algorithms.h, for problem solutions, see main.cpp
a) Counting sort:

```
template <class T, class Iterator> void InsertionSort(Iterator start, Iterator end){
	Iterator cursor;
	T key;
	for (Iterator outerloop = start+1; outerloop < end; outerloop++){
		key = *outerloop;
		cursor = outerloop - 1;

		// shift the elements ofthe sub-array if they are greater than key
		while (cursor != start && *cursor > key){
			*(cursor + 1) = *cursor;
			cursor = cursor - 1;
		}
		// decouple this so non-random access iterators can also do this
		if (cursor == start && *cursor > key){
			*(cursor + 1) = *cursor;
			*cursor = key;
		}
		else
		*(cursor + 1) = key;//insert element where it belongs in sub-array
	}
}
```

b)
Bucket sort:

```
/**
 * @brief uses bucketsort on a positive valued container using 2 bidirectional iterators.
 * @tparam T :class, don't be dumb and use something that can't have floating point
numbers.
 * @tparam Iterator : BIDIRECTIONAL ITERATOR of Class T
 * @param begin :beginning of the iterator
 * @param end :end of the iterator
 */
template <class T, class Iterator> void BucketSort(Iterator begin, Iterator end){
	using namespace std;
	int size = 0;
	T largest = *begin;
	for (Iterator i = begin+1; i != end; i++) {
		size++;
		if (*i > largest)
			largest = *i;
	}
	//divides the numbers to make them from 0 to 1
	if (largest > 1) {
		for (Iterator i = begin; i != end; i++) {
			*i /= largest;
		}
	} else
		largest = 1;
	/*the total number of buckets is of size [size], our "partitions" go up to size,
```

```
         * since all of the values are normalized up to 1
         */

        typename std::vector<T> vector_arr[size+1];
        for (auto i = begin; i != end; i++){
                int vector_arr_index = *i * size;
                vector_arr[vector_arr_index].push_back(*i);
        }
        for (int i =0; i < size; i++){
                InsertionSort<T, typename std::vector<T>::iterator>(vector_arr[i].begin(),
vector_arr[i].end());
        }


        for (int i = 0; i < size; i++){
                for(int j = 0; j < vector_arr[i].size(); j++) {
                        *begin = vector_arr[i][j] * largest;
                        begin++;
                }
        }
        //last case: being inclusive of the one because why not :)
        for (T elem : vector_arr[size]) {
                *begin = elem * largest;
                begin++;
        }

}
```

Let

c)

Procedure f(A,k,a,b)

$B[k] \leftarrow$ array of size k with all 0

for i = 0 up to A.size exclusive do

$B[A[i]] \leftarrow B[A[i]] + 1$

for i = 1 to A.size exclusive do

$B[A[i]] \leftarrow B[A[i]] + B[A[i-1]]$

return B[b] - B[a]

d)

```
        template <class Iterator> void Radix_String(Iterator begin, Iterator end) {
        using namespace std;
        int k = (*begin).size();
        int radix  = 128;
        for (Iterator i = begin + 1; i != end; i++) {
                if ((*i).size() > k)
                        k = (*i).size();
```

```
        }
        for (int i = k-1 ; i >= 0; i--) {
                KeyIteratorCountingSort<std::string, Iterator, StringClass<Iterator> >(begin,
end, radix, StringClass<Iterator>(i));

        }
}


template <class T, class Iterator, class Key = StdClass<T, Iterator> > void
KeyIteratorCountingSort(Iterator begin, Iterator end, int range, Key key = Key() ) {
        using namespace std;
        int size  = std::distance(begin, end);
        T final_arr[size];

        int aux_array[range + 1];
        //set all values to zero
        for (int i = 0; i < range + 1; i++) {
                aux_array[i] = 0;
        }

        //if our count of the number found in the auxiliary array is increased by 1
        for (Iterator i = begin; i != end; i++) {
                aux_array[key(i)] += 1;
        }
        //print_arr<T>(aux_array, range + 1);
        for (int i = 1; i <= range; i++) {
                aux_array[i] += aux_array[i - 1];
        }

        for (Iterator i = end - 1; i != begin; --i) {
                final_arr[aux_array[key(i)] - 1] = *i;
                aux_array[key(i)] -= 1;
        }
        final_arr[aux_array[key(begin)]-1] = *begin;
        aux_array[key(begin)] -= 1;


        for (int i = 0; i < size; i++, begin++) {
                *begin = final_arr[i];
        }
}

template <class Iterator> class StringClass{
private:
        int index;
public:
        StringClass(int i) : index (i) {};
        inline int operator() (Iterator iterator){
                if ((*iterator).size() < index)
                        return 0;
                return (*iterator)[index];
```

```
        }
};
```

e) The worst case time complexity for the implementation of bucketSort shown in class is that all of the items end up in the same bucket, for example, take the array <0.99, 0.98, 0.97, 9.96, 0.95, 0.94, 0.93, 0.93, 0.92, 0.91, 0.90>. All of these elements would be placed inside the same "bucket" of 0.9. After this, the worst-case complexity is determined by **the algorithm used to sort the buckets.** In this case, the worst case time complexity of insertion sort is $O(n^2)$ for an array in decreasing order, which means that the worst time-complexity of bucket sort for this implementation is $O(n^2)$.

f)

Data Type Point (X, Y, distanceToOrigin)

Procedure Distance (Point P1, P2)

return $\sqrt{(P1.X - P2.X)^2 + (P1.Y - P2.Y)^2}$

Point origin = (0,0, 0)

Procedure Euclid(Point A[ ])

let n = A.size
let B[n+1] be a new array of empty lists
for i = 0 to n-1
A[i].distanceToOrigin ← distance(A[i], origin)
insert A[i] in B[floor(n * A[i].distance^2)]
for i = 0 to n
insertion sort(B[i])
Let L be an empty list
for i = 0 to n
for j = 0 to j.size -1
insert B[i][j] into L

PROBLEM 7.2
(a)

```
void OGRadix(std::vector<int>& arr){
                using namespace std;
                int radix  = 10;

                std::vector<int> iterate_thing;
                int greatest = arr[0];
                iterate_thing.push_back(arr[0]);
                for (auto i = arr.begin() +1; i != arr.end(); i++) {
                        if (*i > greatest)
                                greatest = *i;
                        iterate_thing.push_back(*i);
```

```
            }
            int exp = (log10(greatest));
            int exponential =  pow(10, exp);
            //cout<<"exponential is: "<<exponential<<endl;
            arr.clear();
            GhettoBucketSort(iterate_thing, exponential, arr);
        }
```

```
void GhettoBucketSort(std::vector<int>& arr, int exponent, std::vector<int>& out){
        using namespace std;
        if (exponent == 0){
                return;
        }
        vector<int> b[10];
        for (int i = 0; i < arr.size(); i++){
                int b_index = (arr[i] / exponent) % 10;
                b[b_index].push_back(arr[i]);
        }
        for (int i = 0; i < 10; i++){
                if (b[i].size() > 1)
                        GhettoBucketSort(b[i], exponent / 10, out);
        }
//PUSHES EXACTLY N sorted elements to the out vector
        arr.clear();
        for (int i = 0; i < 10; i++){
                for (int j = 0; j < b[i].size(); j++){
                        out.push_back(b[i][j]);
                }
        }
}
```

(b)
Average case:
My implementation of Radix Sort may be a somewhat bizarre in how it operates, so it is difficult to write it in one equation. Firstly, it recursively divides the “arrays” (vectors in this case) into new buckets based on the most significant digit. When either the size of the bucket is one, or the division exponent is zero (which means that the items in the bucket are identical), the function call clears the array that was used in the previous GhettoBucketSort call, and pushes its values **directly to the output vector**. This avoids useless recalculations while popping the function call stack, as the vector up the stack is cleared, resulting in the innermost for loop of section 2 not being executed redundantly.

In other words, assuming a range from 0 to k, the algorithm will first recursively divide n by 10 $log_{10}k$ times, after which it will perform multiple operations equating to n (process of pushing sorted elements from each sorted bucket to the final array) **once**, after which only the **outermost** for loop of section 2 will run for each sorted bucket, resulting in the following derivation:
$T(n) = n \times 10 \times log_{10}k + O(n)$

$T(n) = \Theta(n \times b \log_b k)$, where b = base used.

Notice that this algorithm is **NOT** of complexity $\Theta(nlogn)$.

Memory complexity

The amount of vectors allocated for each iteration of RadixSort is 10, this is done a total of $10 \times \log_{10} k$, or more generally, $b \log_b k$ times. If we consider this as a "recursion tree", we can see that each iteration of bucket creation of OGRadixSort will use exactly n elements. Knowing that the "height" oh the recursion tree is $\log_{10} k$, then the memory complexity is:

$M(n) = 10n \log_{10} k$, or, more generally:

$M(n) = \Theta(n \log k)$.

(C)

Procedure LinearTime(A)

let B be an array of size A

for i=0 to n-1

$B[i] \leftarrow$ Base n representation of $A[i]$

Radix sort B[i] using base n

//this results in a maximum of $log_n n^3 = 3$ digits for Radix Sort, and we can use counting sort

// to sort each digit in range n in $\Theta(n)$ time.

Sources:

geeksforgeeks.org/sort-n-numbers-range-0-n2-1-linear-time/

https://ita.skanev.com/08/04/04.html

https://en.wikipedia.org/wiki/Radix_sort