Problem 6.1 (c)

This is an implementation of Count Sort using lists. The time complexity is (n + k)

```
L = array of k empty lists // O(k)
for j in range (n):
        L[key(A[j])].append(A[j]) // O(1)
out_array = [ ]
for i in range (k)
        out_array.extend(L[i]) // O(L[i])
```

Problem 6.1 (e)

As Bucket Sort is an algorithm that takes advantage when the input is distributed over a range and also by creating buckets; the worst-case scenario is when all elements are placed in a same bucket (*i.e.: <1223, 1224, 1225, 1226, 1227>*). The sorting would then be done by the algorithm used to sort each bucket, by the time complexity of ***$O(n^2)$*** (if we are using insertion sort which is most often used).

For an upper bound on the cost of sorting all the buckets the time complexity is ***$\Theta(n)$*** for all inputs. For a lower bound the time complexity will be ***$\Omega(n^2)$.*** (Considering the sum below which represents the cost of sorting all buckets.)

$$\sum_{i=1}^{n} c|Bi|^2$$

Problem 6.1 (f)

Sort by Euclidean distance:

euclidean_distance (x1, x2) // function to compute the distance

- store the values in a vector // say that you are using c++
- sort the values // by any sorting algorithm
- print the 2D points by the corresponding sorted values // of euclidean distance

Euclidean distance computation:

```
distance = 0;
for d <– 1 to N // Where N in this case is the dimension = 2
        distance = distance + (x1[d] – x2[d])^2
return sqrt(distance)
```

Problem 6.2 (b)

The time complexity for Radix Sort is between N and NW (where W is the number of digits and N the length of array). The space complexity is N + WR (where R si the radix size, 10 in this case). Therefore the overall time complexity for Radix Sort is O(nlog r) (where r is the range i.e.: [0..r]).

Problem 6.2 (c)

radix_sort (A, d) // where d is the highest digit

for j <- 1 to d

do : // Using counting sort 3 times

- subtract all numbers by 1; the range is now 0 to n^3
- as the range is now from 0 to n^3, implement counting sort three more times (to get the linear time) as done in the last problem 6.2(a).
- after the elements are sorted, we add 1 to all numbers to get the original values