Homework 6 : Algorithms and Data Structures

Digdarshan Kunwar

March 2019

Problem 6.1

Sorting in Linear Time

(a) It is implemented in countingsort.py
$: make countingsort.

(b) It is implemented in bucketsort.py
$: make bucketsort.

(c) The pseudo code for **COUNT_IN_INTERVAL** is given below:

Algorithm 1 Pseudo code

```
procedure COUNT_IN_INTERVAL(A,a,b)
    Temp[max]
    for a=0 to max do
        Temp[a] ← 0
    for i = 0 to A.length do
        temp[A[i]]← temp[A[i]]+1
    for j = 1 to max do
        temp[j] ← temp[j]+temp[j-1]
    count ← Temp[b]-Temp[a-1]
    return count
```

(d) It is implemented in implement_word.py
$:make wordsort

(e) Here for the bucket sort the worst case complexity is when there are all the points in a single bucket and the third party sorting algorithm has to do all the work.
The worst case depends on the sorting algorithm used for sorting the buckets.
So at its worst case it has the time complexity of $O(n^2)$ if the sorting algorithm for the sorting the bucket is insertion sort.
Example:
Let the points be :

0.196	0.152	0.143	0.134	0.125	0.116

So we have the buckets as:

0	$\rightarrow$	/					
1	$\rightarrow$	0.196	0.152	0.143	0.134	0.125	0.116
2	$\rightarrow$	/					
3	$\rightarrow$	/					
4	$\rightarrow$	/					
5	$\rightarrow$	/					
6	$\rightarrow$	/					
7	$\rightarrow$	/					
8	$\rightarrow$	/					
9	$\rightarrow$	/					

(f) The pseudo code for Sorting Points is given below:

Algorithm 2 Pseudo code for Sorting Points

procedure E_DIS(A) ▷ Euclidean Distance from Origin
    distance$\leftarrow \sqrt{A_x^2 + A_y^2}$ ▷ $A_x\ andA_y$ are x and y coordinates of point A
    return distance

procedure POINTS_DISTANCE(A,B) ▷ Euclidean Distance between 2 points
    distance$\leftarrow \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$ ▷ A_x, A_y, B_x, B_y are x & y coordinates
    return distance

procedure BUCKETSORT(A)
    n = A.length
    Let B[0, . . . , n 1] be a new array
    for i=0 to n **do**
        A[i].distance=E_Dis(A[i])
    for i = 0 to n - 1 **do**
        B[i] $\leftarrow$ 0
    for i = 1 to n **do**
        B[nA[i].distance] $\leftarrow$ A[i].distance
    for i = 0 to n-1 **do**
        sort list B[i] using insertion sort
    concatenate the lists B[0], B[1], . . . , B[n 1]
    return B

`This is an alternative method :`

procedure BUBBLESORT(A) ▷ Procedure Declaration for Point Sort
    ▷ A is the array of Points
    for i = 0 to $A.length$ **do**
        swapflag=False
        for j= 0 to $A.length$-i-1 **do**
            if $E\_Dis(A[j]) > E\_Dis(A[j+1])$ **then**
                Swap (A[j] with A[j+1])
                swapflag=True
            if swapflag==False **then**
                break

Problem 6.2

Radix Sort

!! NOTE !!
Run file named radixsort.py

(a) It is implemented in radixsort.py.
$: make radixsort

(b) Here, we have a radix sort (MSD to LSD):
In this algorithm we start from the most significant bit in the number and carry on towards to the least significant bit in the number.The general algorithm makes buckets for the various digits in the first significant bit and then moves to the second making another bucket for the second most significant bit towards the LSB.

Asymptotic Time Complexity

Initially when we start with the loop of the number of the character/digits in the outer loop that is σ we have σ number of loops for the inner n numbers of numbers. Given that the input data is uniformly random distributed, then the expected running time of the MSD radix sort is O(n log(n)/log(d)). So in general we have for the asymptotic time complexity:
O(σ*n) $\implies$ O(n)

Asymptotic Storage Space

Here the storage space complexity is also **O(n)** this is because when there is formation of buckets for the elements in the array the number of buckets remain n.Although going recursively the number of buckets may increase by σ factor we have O(n*σ) given σ is constant then we have the space complexity as O(n+wr) where w is length of word and r is the size of radix.

(c) Here the implementation for the pseudo code of the algorithm which sorts n integers in the range 0 to $n^3 1$ in O(n) time.

Here the base of the numbers going inside the counting sort is n.This is done in order to have the O(n) time. If the base is 10 then if the n is large then we have large n cube so even though we use the radix sort we don't have the time complexity of 0(n).Thus changing the base for the counting sort will do the trick.
So now we have 0 to (n-1) for the auxiliary array in the counting sort algorithm.I haven't implemented counting sort for that particular base number as it is similar with few changes.

Algorithm 3 Pseudo code for the Algorithm

```
procedure RadixSort(A)                                  ▷ Procedure Declaration
    d=floor(lg_n(n^3 − 1)) +1                           ▷ lg has base n ,also d=3
    RSort(A,d)
procedure RSort(A,d)
    for i=d to 1 do
        ▷ Implementation for counting sort should be different and with the base for numbers n
        CountingSort_Variant(A,i)      ▷ Perform counting sort for i'th digit to sort the array elements
```