DEPARTMENT OF CYBER SECURITY

DAA Home Work 4	
SUBMITTED BY	211042 Noman Masood Khan A
SEMESTER	Fifth
SUBJECT	DAA
SUBMITTED TO	Sir Ammar Masood

8.2-1: illustrate the operation of COUNTING SORT on the array A = {6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2}

Step (a): for j = 1 to n
then do C[A[j]] < - C[A[j]] + 1

A array

6	0	2	0	1	3	4	6	1	3	2

C Array

2	2	2	2	1	0	2

Step (b):
C[i] = C[i] + C[i-1]

C` Array

2	4	6	8	9	9	11

Step (c):
B[C[A[j]]] = A[j]

B Array

					2					

					2		3			

			1		2		3			

..
..
..
..

0	**0**	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**6**	**6**

8.2-3: Suppose that we were to rewrite the for loop header in line 11 of the COUNTING-SORT

- Suppose we have this array : A = {6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2}.

In the Previous question, I have sorted an array with the counted sorting and now I am taking that sorting an example. I already have solved the C and C` Array in the previous so now I am

Came First

Came Second

A array

6	0	2	0	1	3	4	6	1	3	2

C Array

2	2	2	2	1	0	2

C` Array

2	4	6	8	9	9	11

Step (c): Now the last loop will run from the j = 1 down to 11

for j = 1 down to 11

B[C [A[j]]] = A[j]

C[A[j]] = C [A[j]] – 1

B Array

										6

This 0 was at the 2 positions at the array A

	0									6

	0				2					6

This 0 was at the 2 positions at the array A

0	0				2					6

This clearly shows that the value that was first is not coming at the first position; the algorithm is not stable.

0	0	1			2					6

0	0	1			2					6

0	0				2		3			6

.

.

.

0	0	1	1	2	2	3	3	4	5	6

- Clearly Showed that the algorithm sorts the elements very well but not stable.

The code for the increasing index with the elements being sorted and stable order.

8.2-5: Suppose that the array being sorted contains only integers in the range 0 to k and that there are no satellite data.

```
for j = n downto 1
    B[C [A[j]]] = A[j]
    C [A[j]] = C [A[j]] – 1          // to handle duplicate values
return B
```

```
COUNTING-SORT(A, n, k)
let B[1 : n] and C [0 : k] be new arrays
for i = 0 to k
C [i] = 0
for j = 1 to n
C [A[j]] = C [A[j]] + 1
// C [i] now contains the number of elements equal to i.
for i = 1 to k
C [i] = C [i] + C [i – 1]
// C [i] now contains the number of elements less than or equal to i.
// Copy A to B, starting from the end of A.
```

For j = n down to 1

value = C[A[j]]

swap = A[value]

A[value] = A[i]

A[i] = swap

Return A

8.2-3: illustrate the operation of RADIX-SORT on the following list of English words:

C	O	W
D	O	G
S	E	A
R	U	G
R	O	W
M	O	B
B	O	X
T	A	B
B	A	R
E	A	R
T	A	R
D	I	G
B	I	G
T	E	A
N	O	W
F	O	X

S	E	A
T	E	A
M	O	B
T	A	B
D	O	G
R	U	G
D	I	G
B	I	G
B	A	R
E	A	R
T	A	R
C	O	W
R	O	W
N	O	W
B	O	X
F	O	X

T	A	B
B	A	R
E	A	R
T	A	R
S	E	A
T	E	A
D	I	G
B	I	G
M	O	B
D	O	G
C	O	W
R	O	W
N	O	W
B	O	X
F	O	X
R	U	G

B	A	R
B	I	G
B	O	X
C	O	W
D	I	G
D	O	G
E	A	R
F	O	X
M	O	B
N	O	W
R	O	W
R	U	G
S	E	A
T	A	B
T	A	R
T	E	A

8.3-5: Show how to sort n integers in the range 0 to n^3 – 1 in O(n) time

- Using the Counting Sorting will take O(n^2) time. Likewise, using the Merge Sorting, Heap sorting and others will take O(nlogn) time or some other time which is not O(n).
- Radix Sort will be the Sort that will have the O(n) time.

The Running Time complexity of the Radix sort is O(d*(n + b)).
Where the d is the number of the digits; n is the integers and b is the base of the n.

- ➢ Prove following the following Steps:
- ✓ Since the numbers are in base n, the range of digits is 1 to n so k = n.
- ✓ The number of passes needed is 3 since n^2 = 100_n so d = 3.
- ✓ The running time of the Radix Sort is Θ(d*n + d*k) = Θ(3n + 3n) = Θ(n) ∈ O(n).

9.1-2: You want to find a number that is neither the minimum nor the maximum. The minimum number of steps for finding that.

```
N = 5
A = {5,4,10,21,30}
Max = Maximum(A,N);  // assigning the maximum value
Min = Minimum(A,N);   // assigning the minimum value

For (i=0; i<N; i++){
        If (A[i] == Max || A[i]==Min )  // if value is either minimum or maximum then go back
           continue;
        else
             notmaxmin = A[i];          // storing value which is neither max nor min.
              break;                    // breaking the Loop
}
```

Only 1 comparison is need in this function but the rest of the comparisons were done in the maximum and the minimum function.

9.2-2: Write an iterative version of RANDOMIZED-SELECT.

```
RANDOMIZED-SELECT(A, p, r, i)
  while p < r do
    q ← RANDOMIZED-PARTITION(A, p, r)
    k ← q – p +1

    if i ≤ k then
      r ← q
    else
      p ← q + 1
      i ← i – k

  return p
```

9.2-4: Argue that the expected running time of RANDOMIZED-SELECT does not depend on the order of the elements in its input array A[p : r].

Argument

The algorithm of the randomized-select does not depend on the order of the elements in input array. Therefore, the expected running time for the randomized-select is same for any permutation of the input array[p : r].

Prove By Induction

Base Case:

When the input array has only one element, the randomized-select will simply return one element regardless of its position in the array. The time complexity remains constant and not effected by the order of elements.

Inductive Step:

Assume that the claim is true for an Array [n-1].

For an input array of length n-1, RANDOMIZED-SELECT works as follows:

- Randomly selects a pivot element from the array.
- Partitions the array around the pivot, placing elements smaller than the pivot to its left and larger elements to its right.
- Recursively runs RANDOMIZED-SELECT one of the (left or right) array based on the pivot's position
- The expected running time of the partitioning is $\Theta(n)$ which does not depend on the order of the element.
- The expected running time of the recursive calls is T(max(i, k) - min(i, k).
- The total running time of the algorithm is $T(n) = \theta(n) + T(\max(i, k) - \min(i, k))$.

The induction step has shown us the time complexity for the any array of the length will have the same running time; this means that the running time of an algorithm does not depend on the order of the element.