Linear Sorting Algorithms

Counting Sort: Efficient for small range and small sized data. Correct for only positive values.

```
Counting Sort (A [], N)
{
    K = getMax(A, N)    //get Maximum value of the array (Linear time) O(N)
    C[0 … K] = {0}       // create an array to store the frequency count of each element of the given array,
                            initialize the array with 0's

// get frequency count of each element of the given array
    for (i=1 to N)         // Linear time O(N)
        C[ A[i] ]++        //  get the value of A[i] and treat this value as an index of frequency count array
                           //   "C" and update that index value of "C" by 1

// now compute cumulative frequency.
   for (i=1 to K)          // O(K)
       C[i] = C[i] + C[i-1]

// R[1…N]                        // create a resultant array of size "N" to store the sorted data
   for (i=N downto 1)         // O(N) reverse loop to make it stable
       R[C[A[i]]] = A[i]        // store the data (treat A[i] as an index of "C" and C[A[i]] as an index of R)
       C[A[i]] = C[A[i]] – 1 // update the count of cumulative frequency array

// Copy the data back into the original array.
   for (i=1 to N)
       A[i] = R[i]
   delete C
   delete R
}
```

Overall time complexity: $\Theta(\boldsymbol{N} + \boldsymbol{K})$

K is the maximum value. If the range of data is small then time complexity will be linear but if the range of data is too large like input size is N e.g., (N=10) and the maximum value is (100) then the time complexity will be N^2. Similarly time complexity will be N^5 or N^7 or any higher order term depending upon the range of data.

Radix Sort:

Sort the data on the bases of digit positions. Start sorting the array from least significant digit and move towards most significant digit. Sort the data by using any stable sort algorithm. Since Counting sort is a stable sorting algorithm having Linear time complexity so we will use counting sort.

```
Radix Sort(A[], N)
{
    max = getMax(A, N)
// Number of iterations of the following loop will be equal to the number of digits of the max value.
    for (digit_position = 1; max / digit_position > 0; digit_position*=10)
        countingSort(A, N, digit_position)
}
Counting Sort(A[], N, digit_Position)
{
```

Since we are sorting the array on the bases of each digit, so the maximum range is 10 i.e., (0-9)

```
 C[0…9]  // array to store frequency

// frequency count
 for(i= 1 to N)
    C[ (A[i]/digit_position)%10 ] ++   // divide A[i] by digit position and then take mod with 10. Treat
                                       // the result as an index of C.
// cumulative frequency
  for(i=1 to 9)
     C[i] += C[i-1]

// R[1…N]  // resultant array to store the data
  for(i=N to 1)
     R[ C[ (A[i]/digit_position)%10) ] ] = A[i]
     C[ (A[i]/digit_position)%10] ] -= 1

// copy the data back into the original array
  for(i=1 to N)
     A[i] = R[i]
}
```

Time Complexity: $\Theta(d*(N+K))$

Since K = 10 which is constant so $\Theta(d*N)$ where ‘d’ represents the number of digits of the maximum value in the given array.

Bucket Sort

Data must be in the range 0 to 1 and uniform distribution of data will provide the ideal results.

Idea: Create an array of vectors equal to the size of original data. In other words, we will have buckets equal to the size of data where each bucket will maintain a vector. Data will be stored in these buckets. Each bucket will be sorted and then data of all the buckets will be merged into a resultant sorted array.

```
Bucket Sort(A[], N)
{
// Create an array of vectors.
     Vector<float> B [1…N]

// store the data in buckets.
    for (i=1 to N)
       b_index = N * A[i]       // Since values are in decimals so get the bucket index by N*A[i]
       B[i].push_back(A[i])   // each bucket of “B” is maintaining a vector

// sort the data of each bucket using insertion sort algorithm
   for (i=0 to N)
        sort(B[i])

// since all the buckets are sorted so store the sorted data back into the original array
   K = 1 //index to store data in original array
   for(i=1 to N)
       for(j = 1 to B[i].size())
            A[k++] = B[i][j]
}
```

Issues: if all the values are in the same bucket then time complexity will become O(N^2). Since uniform distribution of data is the ideal scenario so, in the worst case scenario the time complexity of bucket sort is O(N^2).