### CS-590 B ALGORITHMS

### HOMEWORK - 2

## SUPREETH HASSAN RAVINDRA

PART -II

**IMPLEMENTATION OF RADIX SORT USING INSERTION SORT**

The objective is to implement an insertion sort algorithm to sort a given array of strings according to the character at position *d*. Using this insertion sort algorithm, we need to implement radix sort for an array of strings. The below given code snippet satisfies our objective of implementing Radix sort using Insertion sort.

Here, the function ‘insertion\_sort\_digit’ has the following parameters:

A: array of strings

A\_len: array of string lengths

l, r and d are the indexes in the array

In summary, the function has an outer for loop that iterates from ‘l+1’ to ‘r’. considering each string one at a time. For each string A[j], it extracts the character at position d and stores it in key. It also stores the length of the string in key\_len. The function then enters a while loop that compares the character at position d of key with the character at position d of the previous string A[i]. If key\_len is less than or equal to d (meaning the string is shorter than d), or if key\_len is greater than d and the character at position d in key is less than the character at position d in A[i], it swaps the strings and their lengths, effectively sorting them. The while loop continues until the characters at position d satisfy the sorting condition, and then it places key at the correct position in the array.

The ‘radix\_sort\_is’ function takes an array of strings A, an array of string lengths A\_len, and parameters n (number of strings) and m (maximum string length). It implements the radix sort algorithm by iterating through each character position from the least significant digit (LSB) to the most significant digit (MSB) in the strings. Inside the loop, it calls insertion\_sort\_digit to sort the array A based on the current character position d. The loop continues until it has sorted the strings based on all positions from LSB to MSB.

void insertion\_sort\_digit(char\*\* A, int\* A\_len, int l, int r, int d)

{

int i;

char\* key;

int key\_len;

for (int j = l + 1; j <= r; j++) {

key = A[j];

key\_len = A\_len[j];

i = j - 1;

while (i >= l && (key\_len <= d || (i >= 0 && key\_len > d && key[d] < A[i][d]))) {

A[i + 1] = A[i];

A\_len[i + 1] = A\_len[i];

i--;

}

A[i + 1] = key;

A\_len[i + 1] = key\_len;

}

}

void radix\_sort\_is(char\*\* A, int\* A\_len, int n, int m)

{

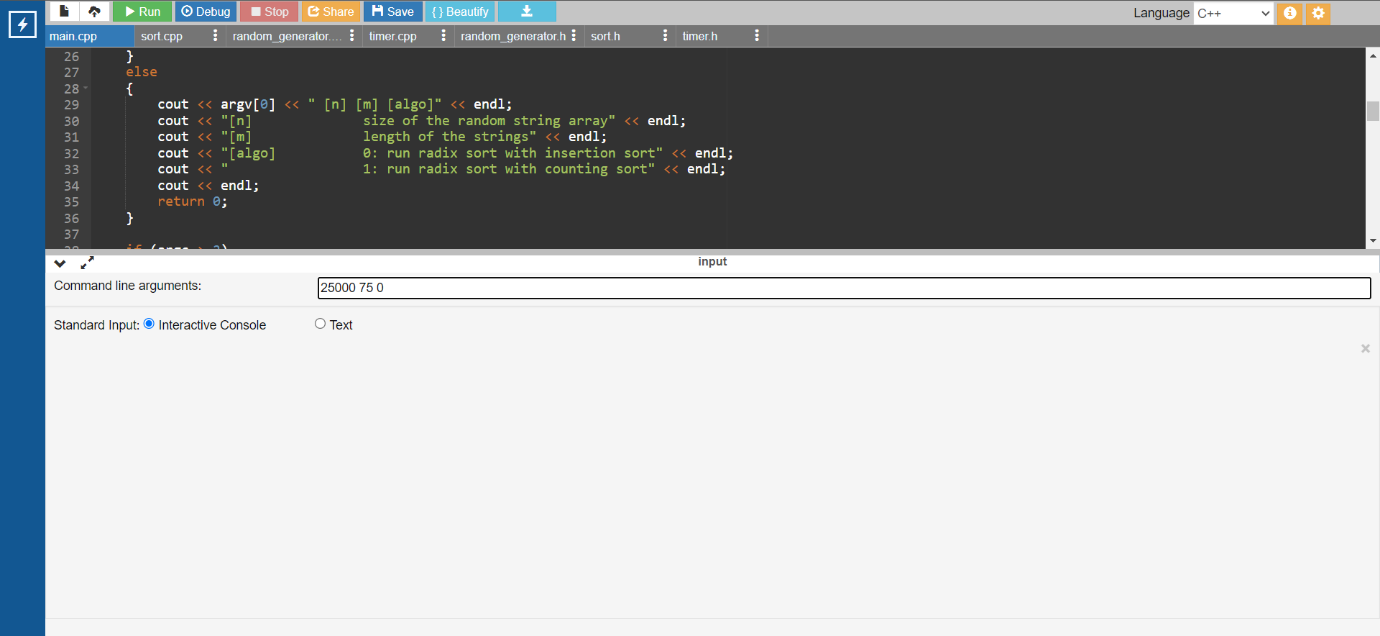
for (int d = m - 1; d >= 0; d--) {

insertion\_sort\_digit(A, A\_len, 0, n - 1, d);

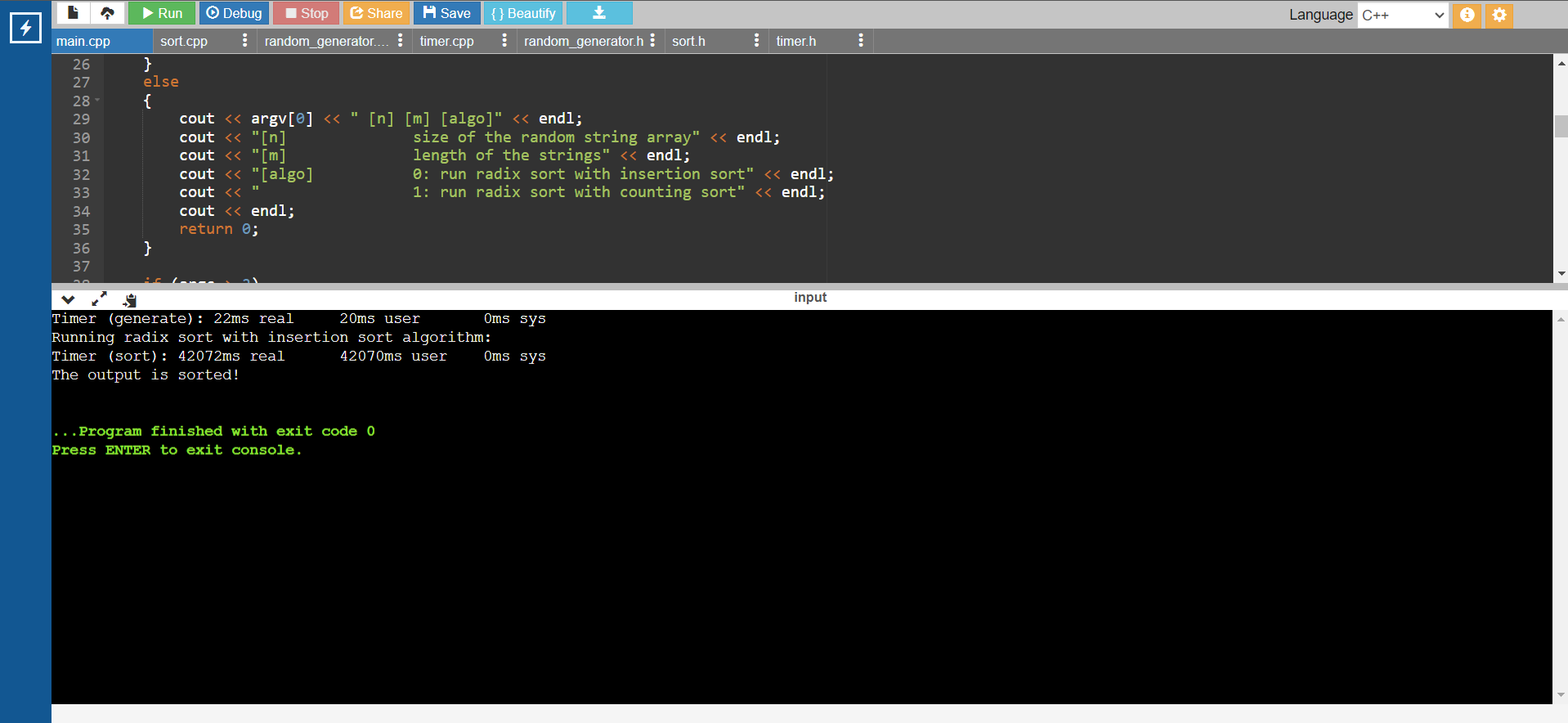
}

}

Now, the task given to me was to measure the runtime performance of this Radix sort for array size *n* = 5000, 10000, 25000, 50000, 100000 and the string length *m*= 10, 25, 50, 75. Then, repeating each test a number of times (usually at least 10 times) and compute the average running time for each combination of *m* and *n*.



**Fig (a) and (b)**



The above figures (a) and (b) show calculating the running time of the radix sort with Insertion sort algorithm with *n*=25000 and *m*=75 on an online C++ compiler https://www.onlinegdb.com/. Similarly, we calculate for every parameter value and below is the table of the average running time of the algorithm for each *m* and *n*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **m=10** | **m=25** | **m=50** | **m=75** |
| **n** | **RUNNING TIME T(N)** | | | |
| **5000** | 331 | 693.2 | 1733.62 | 1820.5 |
| **10000** | 1143.62 | 2845.3 | 5031 | 8754.8 |
| **25000** | 7462.2 | 14601.7 | 40849.6 | 42282.3 |
| **50000** | 28375.2 | 70422.1 | - | - |
| **100000** | - | - | - | - |

In the above table, all the running time is calculated in milli-seconds (ms). So, using this and creating a scatter plot for each case of *m* and *n*, I got 4 scatter plots. In the graphs, the Y axis is the Running time T(n) in milli-seconds (ms) and the X axis is the *n* values representing size of the random string array.

* CASE 1:

|  |  |
| --- | --- |
| n | m=10 |
| 5000 | 331 |
| 10000 | 1143.62 |
| 25000 | 7462.2 |
| 50000 | 28375.2 |
| 100000 | 298240.5 |

* CASE 2:

|  |  |
| --- | --- |
| n | **m=25** |
| 5000 | 693.2 |
| 10000 | 2845.3 |
| 25000 | 14601.7 |
| 50000 | 70422.1 |
| 100000 | **-** |

* CASE 3:

|  |  |
| --- | --- |
| n | **m=50** |
| 5000 | 1733.62 |
| 10000 | 5031 |
| 25000 | 40849.6 |
| 50000 | - |
| 100000 | - |

* CASE 4:

|  |  |
| --- | --- |
| **n** | **m=75** |
| **5000** | 1820.5 |
| **10000** | 8754.8 |
| **25000** | 42282.3 |
| **50000** | - |
| **100000** | - |

In conclusion, generalizing insights from the above 4 graphs, we can say*,* the runtime T(n) of the radix sort algorithm using insertion sort for the given value of the size of the array *n* and string length *m*, it increases drastically with increasing *m* and *n.* However, this version of Radix sort is not suitable for large input size as the algorithm takes significantly long time to sort the strings, which can be seen from the table as well as the graph above. Hence, we need to employ a better sorting algorithm like counting sort to implement Radix sort.

**IMPLEMENTATION OF RADIX SORT USING COUNTING SORT**

Now, considering the implementation of radix sort using counting sort to sort an array of strings. We have the below code.

The function ‘counting\_sort\_digit’ counts the occurrences of each character at position d in the input array A. It creates a count array C of size 256 (ASCII character range) to store the character counts. It calculates cumulative counts to determine the correct positions for each character. It builds the sorted array B and updates A and A\_len. Finally, it copies the sorted values back to A and A\_len.

The ‘radix\_sort\_cs’ function implements radix sort for strings by iterating from the most significant character position to the least significant character position (m - 1 to 0). It creates temporary arrays temp and temp\_len to store intermediate results. For each character position d, it calls counting\_sort\_digit to sort the strings. After each pass, it swaps the original array A with the temporary array temp.

void counting\_sort\_digit(char\*\* A, int\* A\_len, char\*\* B, int\* B\_len, int n, int d)

{

const int k = 256;

int C[k] = {0};

for (int i = 0; i < n; i++) {

if (A\_len[i] <= d) {

C[0]++;

} else {

C[A[i][d]]++;

}

}

for (int i = 1; i < k; i++) {

C[i] += C[i - 1];

}

for (int i = n - 1; i >= 0; i--) {

if (A\_len[i] <= d) {

B[C[0] - 1] = A[i];

B\_len[C[0] - 1] = A\_len[i];

C[0]--;

} else {

B[C[A[i][d]] - 1] = A[i];

B\_len[C[A[i][d]] - 1] = A\_len[i];

C[A[i][d]]--;

}

}

for (int i = 0; i < n; i++) {

A[i] = B[i];

A\_len[i] = B\_len[i];

}

}

void radix\_sort\_cs(char\*\* A, int\* A\_len, int n, int m)

{

char\*\* temp = new char\*[n];

int\* temp\_len = new int[n];

for (int i = 0; i < n; i++) {

temp[i] = new char[m + 1];

temp\_len[i] = A\_len[i];

}

for (int d = m - 1; d >= 0; d--) {

counting\_sort\_digit(A, A\_len, temp, temp\_len, n, d);

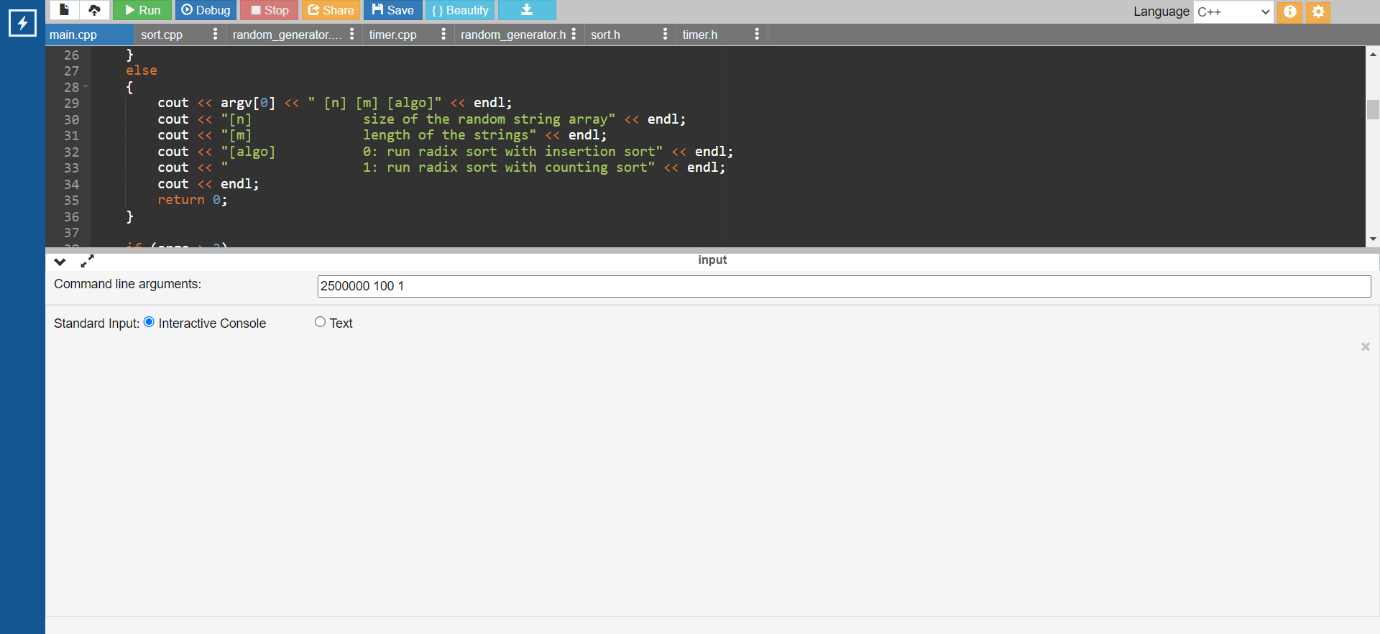
std::swap(A, temp);

std::swap(A\_len, temp\_len);

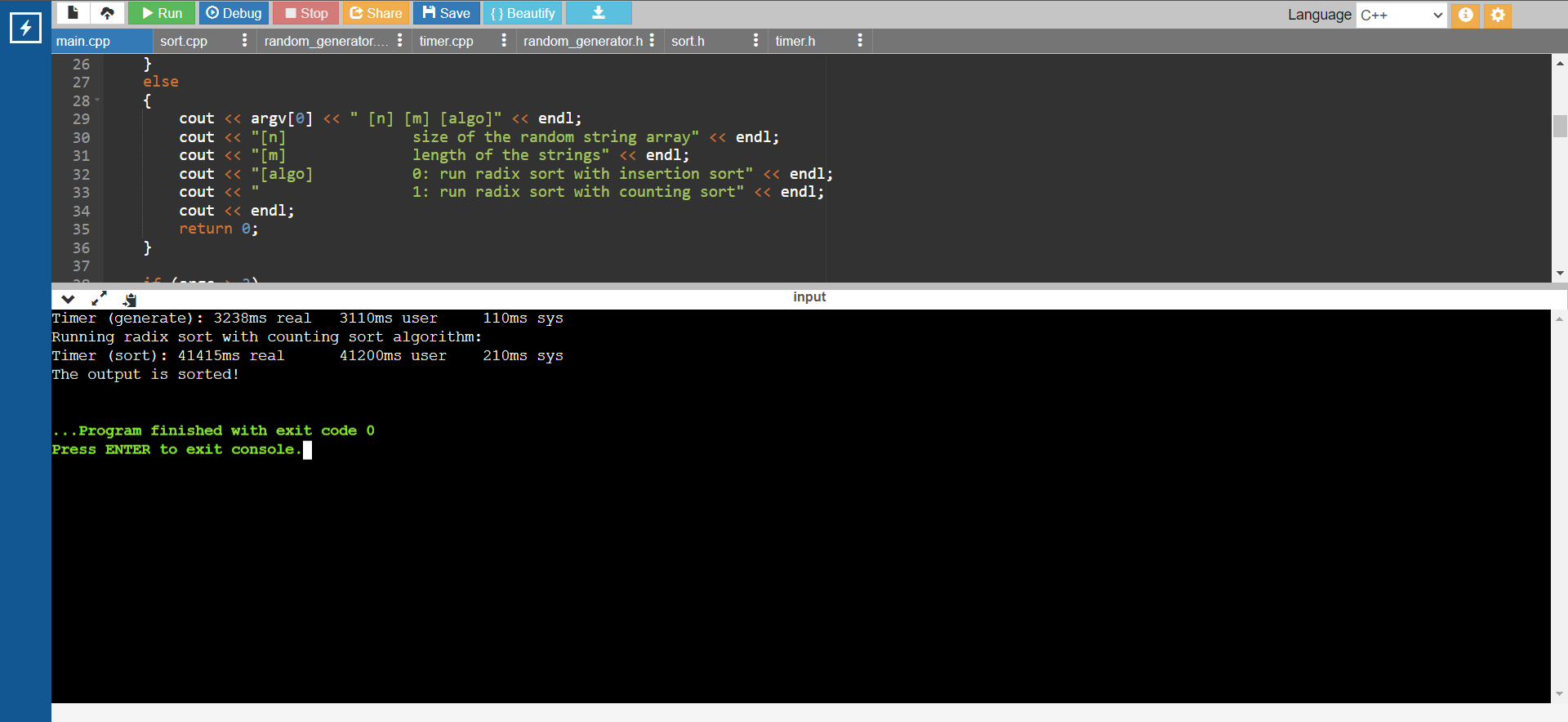
}

}

Now, the task given to me was to measure the runtime performance of this Radix sort for array size *n* = 100000, 250000, 500000,1000000, 2500000, 5000000 and the string length *m*= 50, 70, 90, 100. Then, repeating each test a number of times (usually at least 10 times) and compute the average running time for each combination of *m* and *n*.



**(c)**



**(d)**

The above figures (a) and (b) show calculating the running time of the radix sort with Counting sort algorithm with *n*=2500000 and *m*=100 on an online C++ compiler https://www.onlinegdb.com/. Similarly, we calculate for every parameter value and below is the table of the average running time of the algorithm for each *m* and *n*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **m=50** | **m=70** | **m=90** | **m=100** |
| **n** | **RUNNING TIME T(N)** | | | |
| **100000** | 128.8 | 207 | 211.3 | 217 |
| **250000** | 380.2 | 942 | 1201.5 | 1508 |
| **500000** | 1349.5 | 3485.6 | 4171.16 | 4127.67 |
| **1000000** | 1893.8 | 7741 | 13637.2 | 11580.67 |
| **2500000** | 5165.7 | 18981.8 | 26052.6 | 42907 |
| **5000000** | 10419.6 | - | - | - |

In the above table, all the running time is calculated in milli-seconds (ms). So, using this and creating a scatter plot for each case of *m* and *n*, I got 4 scatter plots. In the graphs, the Y axis is the Running time T(n) in milli-seconds (ms) and the X axis is the *n* values representing size of the random string array.

* CASE 1:

|  |  |
| --- | --- |
| **n** | **m=50** |
| **100000** | 128.8 |
| **250000** | 380.2 |
| **500000** | 1349.5 |
| **1000000** | 1893.8 |
| **2500000** | 5165.7 |
| **5000000** | 10419.6 |

* CASE 2:

|  |  |
| --- | --- |
| **n** | **m=70** |
| **100000** | 207 |
| **250000** | 942 |
| **500000** | 3485.6 |
| **1000000** | 7741 |
| **2500000** | 18981.8 |
| **5000000** | - |

* CASE 3:

|  |  |
| --- | --- |
| **n** | **m=90** |
| **100000** | 211.3 |
| **250000** | 1201.5 |
| **500000** | 4171.16 |
| **1000000** | 13637.2 |
| **2500000** | 26052.6 |
| **5000000** | - |

* CASE 4:

|  |  |
| --- | --- |
| **n** | **m=100** |
| **100000** | 217 |
| **250000** | 1508 |
| **500000** | 4127.67 |
| **1000000** | 11580.67 |
| **2500000** | 42907 |
| **5000000** | - |  |

By comparing the runtime tables and the graph of both versions of radix sort, we can easily conclude that counting sort performs far better than insertion sort when used in Radix sort to sort an array of strings. When we consider the trendline of the graph of the later version of radix sort, we can see the Runtime of the algorithm plummets for the value *n*=500000 for any given value of *m*. But this version can sort large array size of strings with big lengths. Hence, the efficiency of counting sort is more than insertion sort by multiple folds.