590 Assignment-3

Code for radix sort using counting sort:

```
void counting_sort_digit(char** A, int* A_len, char** B, int* B_len, int n, int d) {
    const int C = 256;
    int c[C] = {0};

    for (int i = 0; i < n; ++i) {
        if (d < A_len[i]) {
            c[(unsigned char)A[i][d]]++;
        } else {
            c[0]++;
        }
    }
    for (int i = 1; i < C; ++i) {
        c[i] += c[i - 1];
    }
    for (int i = n - 1; i >= 0; --i) {
        int p = (d < A_len[i]) ? (unsigned char)A[i][d] : 0;
        B[c[p] - 1] = A[i];
        B_len[c[p] - 1] = A_len[i];
        c[p]--;
    }
}
void radix_sort_cs(char** A, int* A_len, int n, int m) {
    char** B = new char*[n];
    int* B_len = new int[n];
    int max_len = 0;
    for (int i = 0; i < n; ++i) {
        if (A_len[i] > max_len) {
            max_len = A_len[i];
        }
    }
    for (int d = max_len - 1; d >= 0; --d) {
        counting_sort_digit(A, A_len, B, B_len, n, d);
        for (int i = 0; i < n; ++i) {
            A[i] = B[i];
            A_len[i] = B_len[i];
        }
    }
    delete[] B;
    delete[] B_len;
}
```

Code for Radix sort using insertion sort:

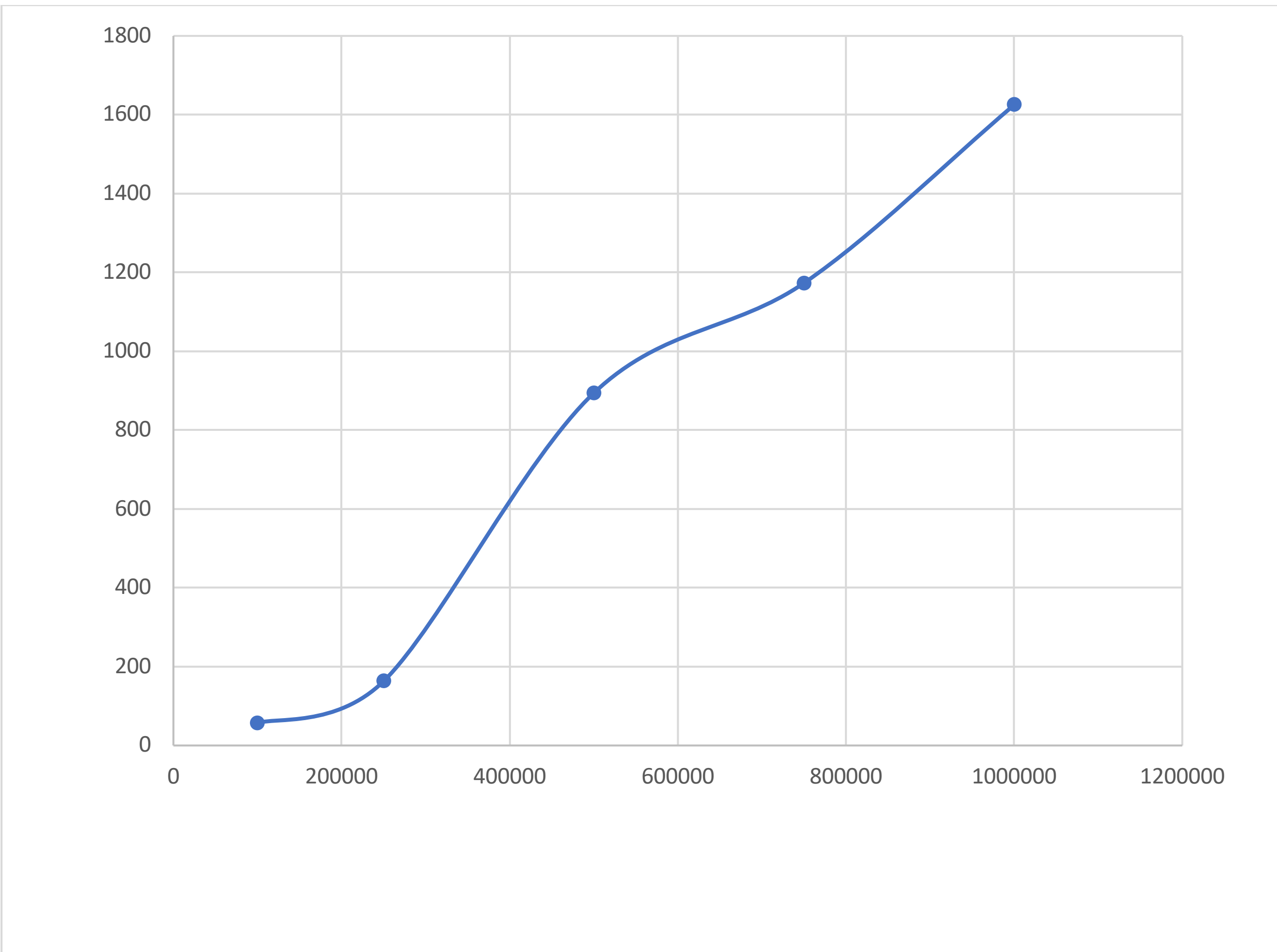
```
void insertion_sort_digit(char** A, int* A_len, int l, int r, int d)
{
    for (int j = l + 1; j <= r; j++) {
        char* key = A[j];
        int key_len = A_len[j];
        int i = j - 1;
        while (i >= l && (d < A_len[i] && A[i][d] > key[d])) {
            A[i + 1] = A[i];
            A_len[i + 1] = A_len[i];
            i = i - 1;
        }

        A[i + 1] = key;
        A_len[i + 1] = key_len;
    }
void radix_sort_is(char** A, int* A_len, int n, int m)
{

    int max_len = 0;
    for (int i = 0; i < n; ++i) {
        if (A_len[i] > max_len) {
            max_len = A_len[i];
        }
    }
    for (int d = max_len - 1; d >= 0; --d) {
        insertion_sort_digit(A, A_len, 0, n - 1, d);
    }
}
```

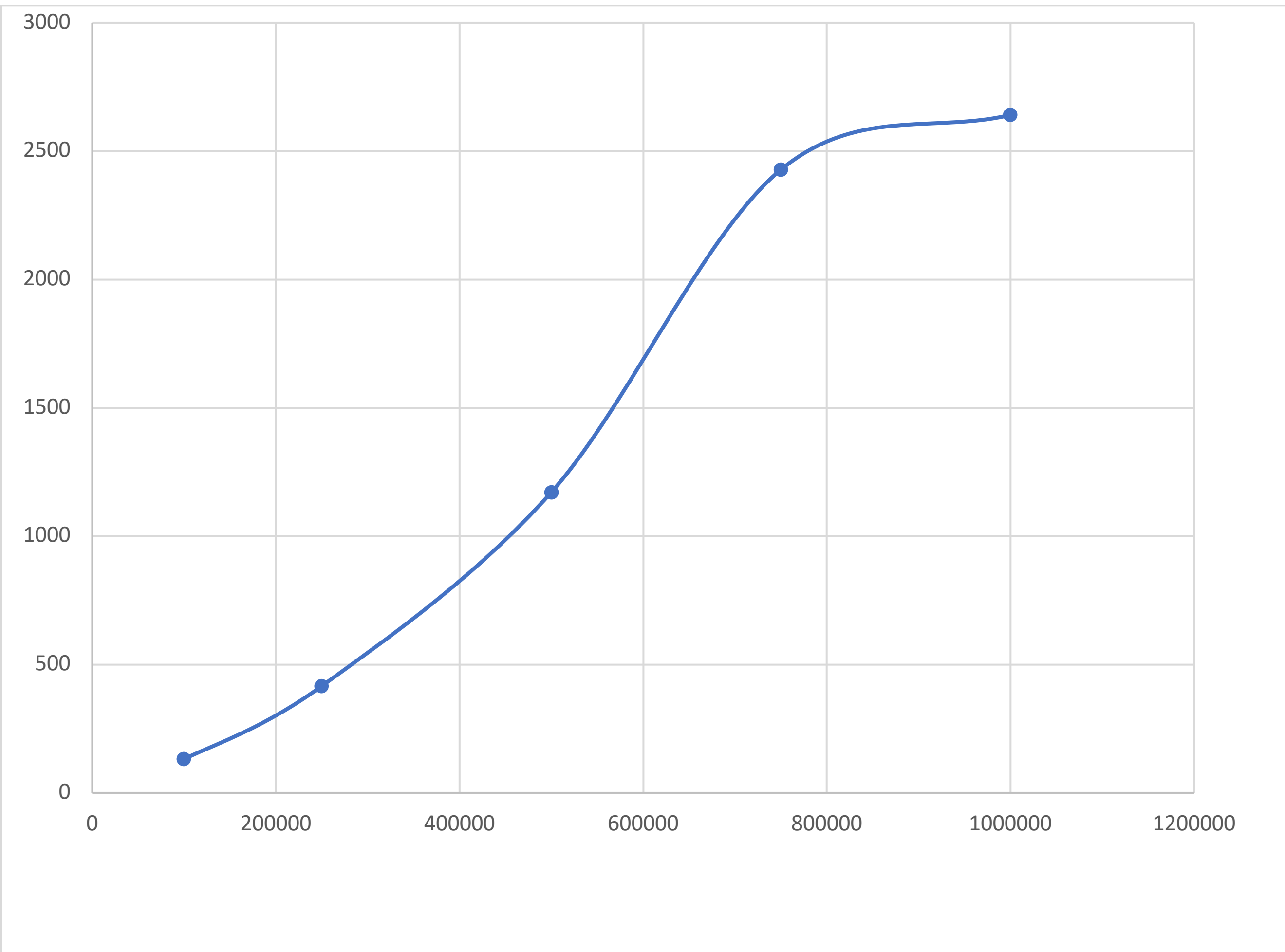
Radix sort using Counting sort for m = 25

n	m = 25
100000	57
250000	155
500000	894
750000	1173
1000000	1625



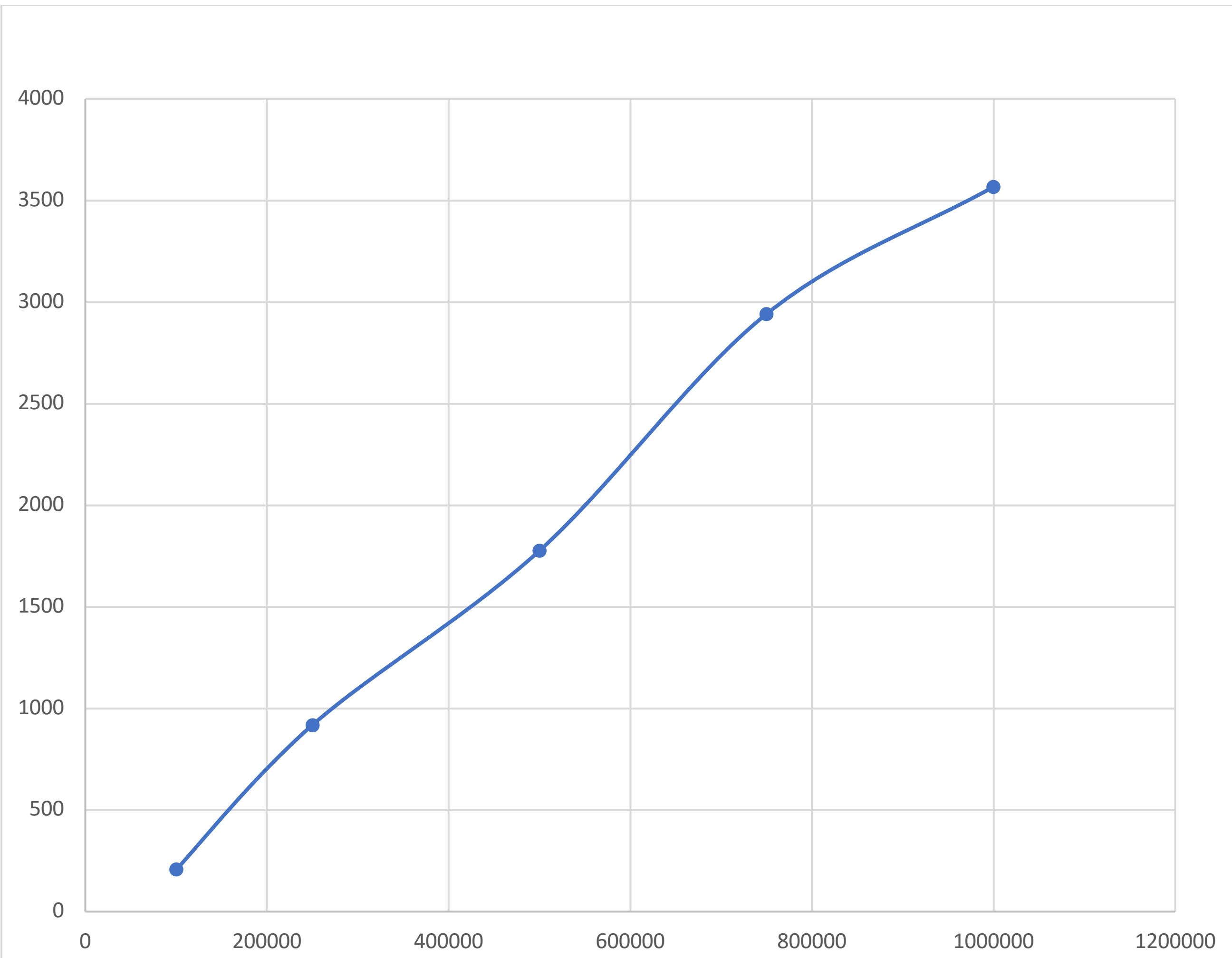
Radix sort using Counting sort for m = 50

n	m = 50
100000	131
250000	416
500000	1171
750000	2428
1000000	2641



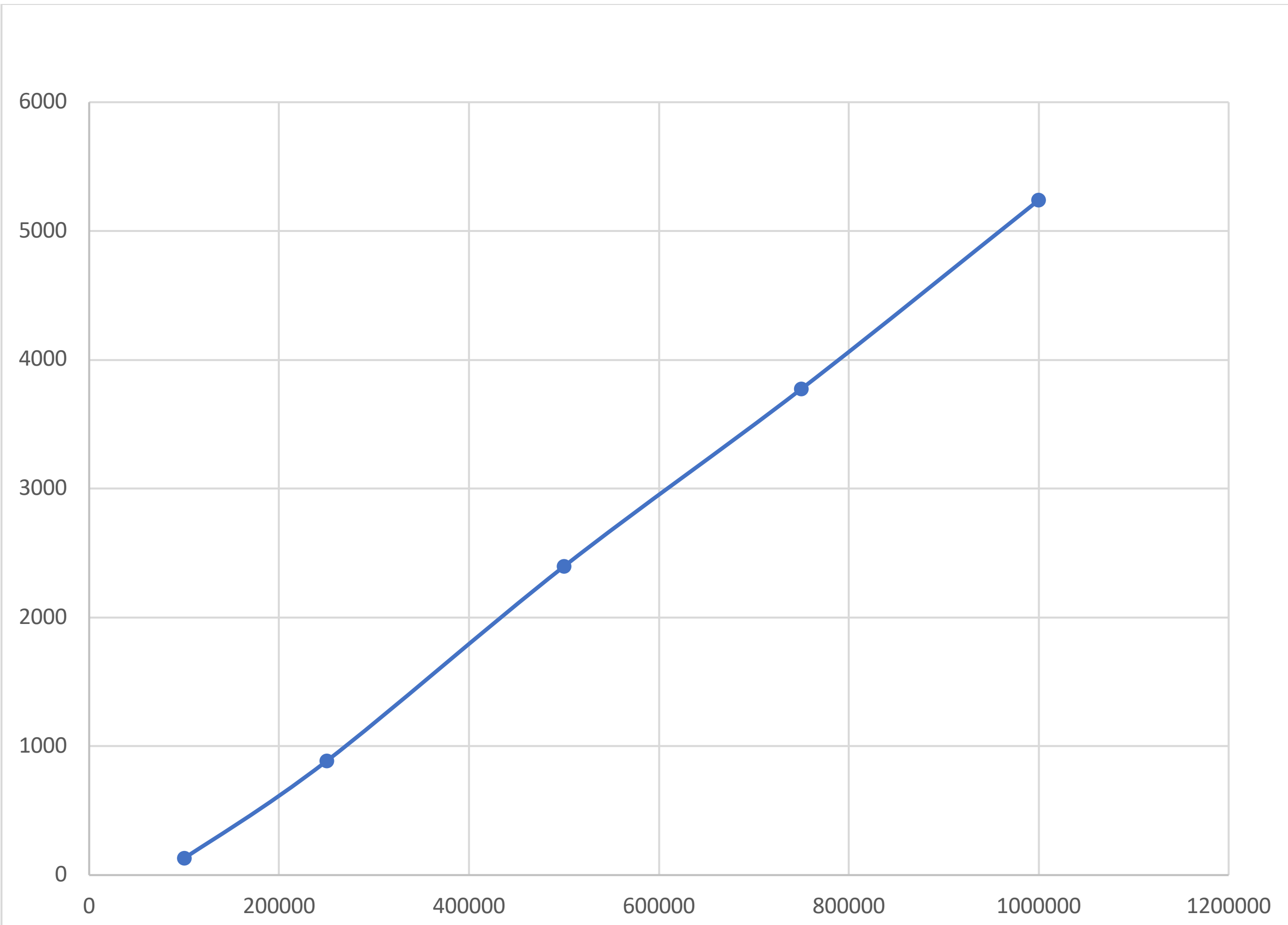
Radix sort using Counting sort for m = 75

n	m = 75
100000	206
250000	918
500000	1776
750000	2941
1000000	3567



Radix sort using Counting sort for m = 100

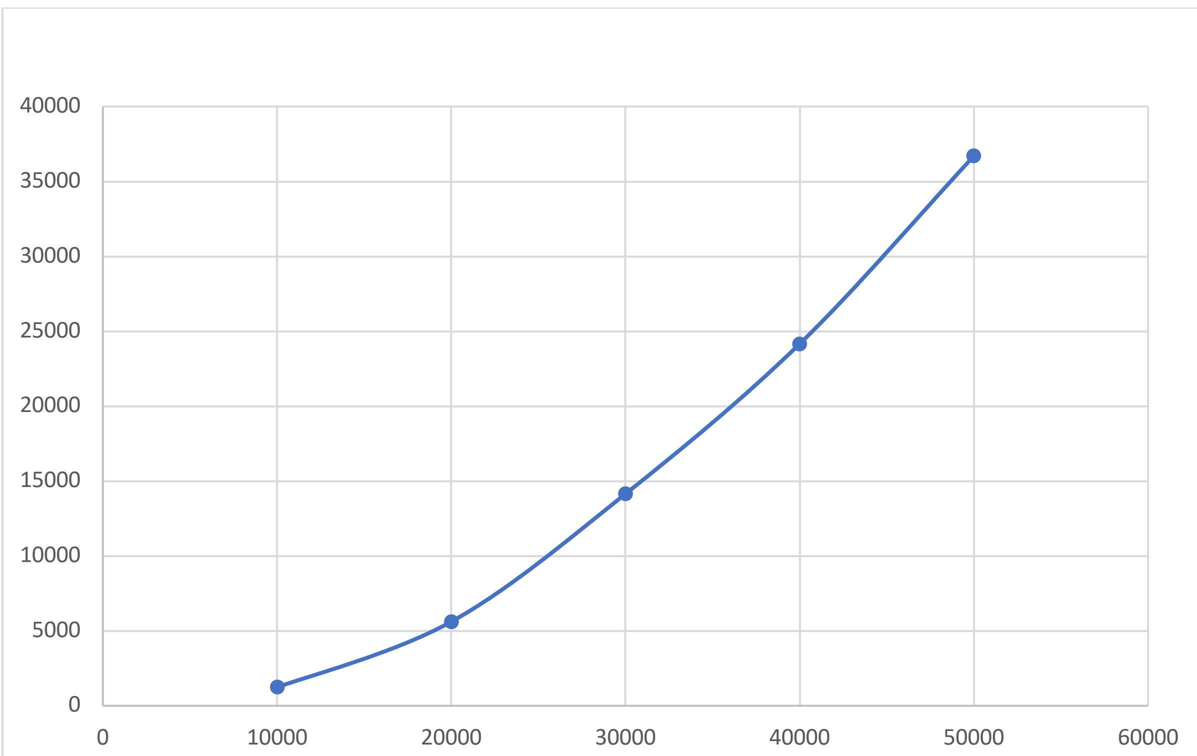
n	m = 100
100000	130
250000	885
500000	2396
750000	3774
1000000	5240



Radix sort operates by sorting elements based on digits or characters, from least significant to most significant. Each character or digit are sorted using counting sort which is stable algorithm.
The time complexity of radix sort using counting sort is O(d(n+k)). d is number of digits, n is number of elements, k is range of input. The time complexity graph shows a linear trend with respect to value of input size n.

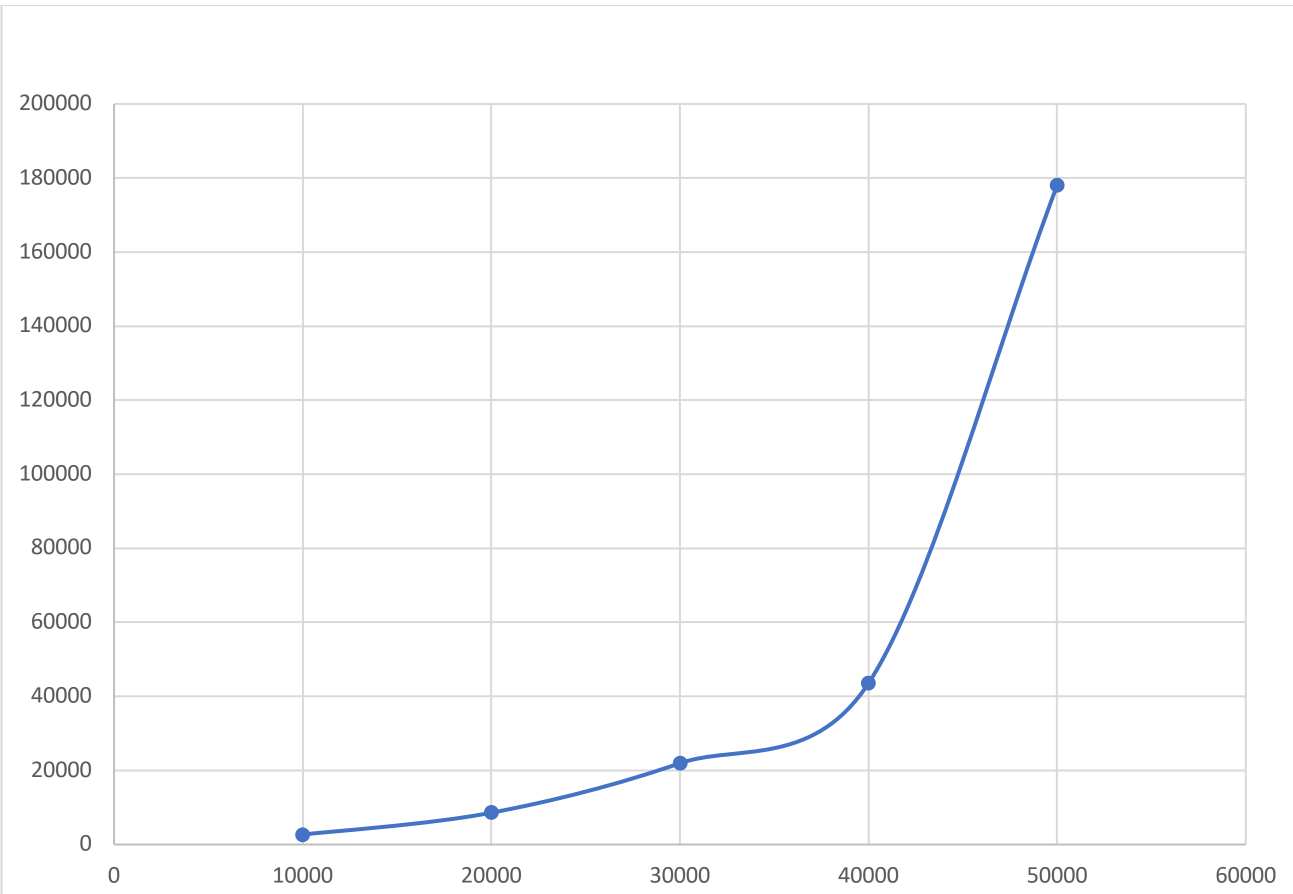
Radix sort using Insertion sort for m = 25

n	m = 25
10000	1232
20000	5613
30000	14145
40000	24160
50000	36700



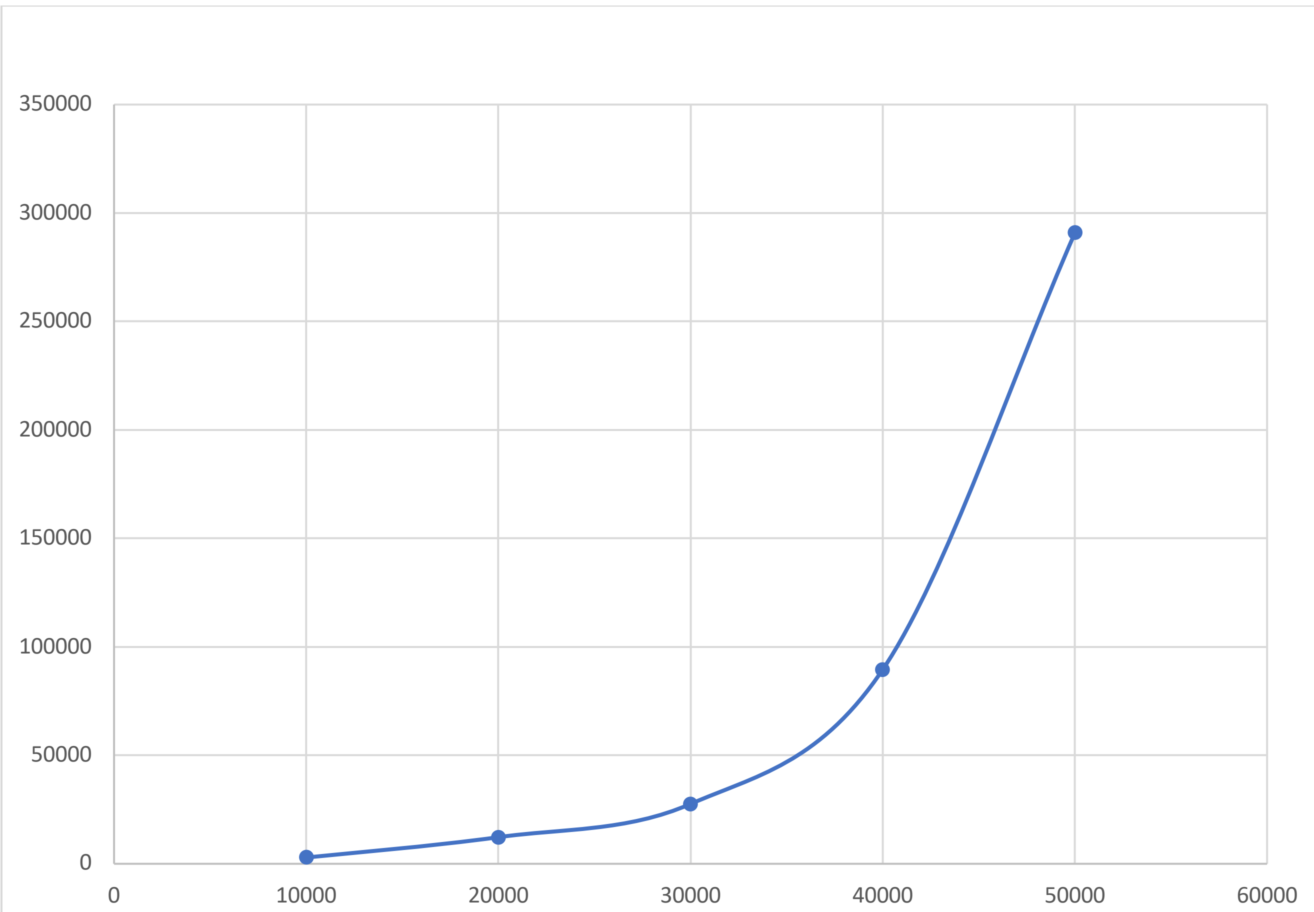
Radix sort using Insertion sort for m = 50

n	m = 50
10000	2693
20000	8584
30000	21948
40000	43564
50000	178036



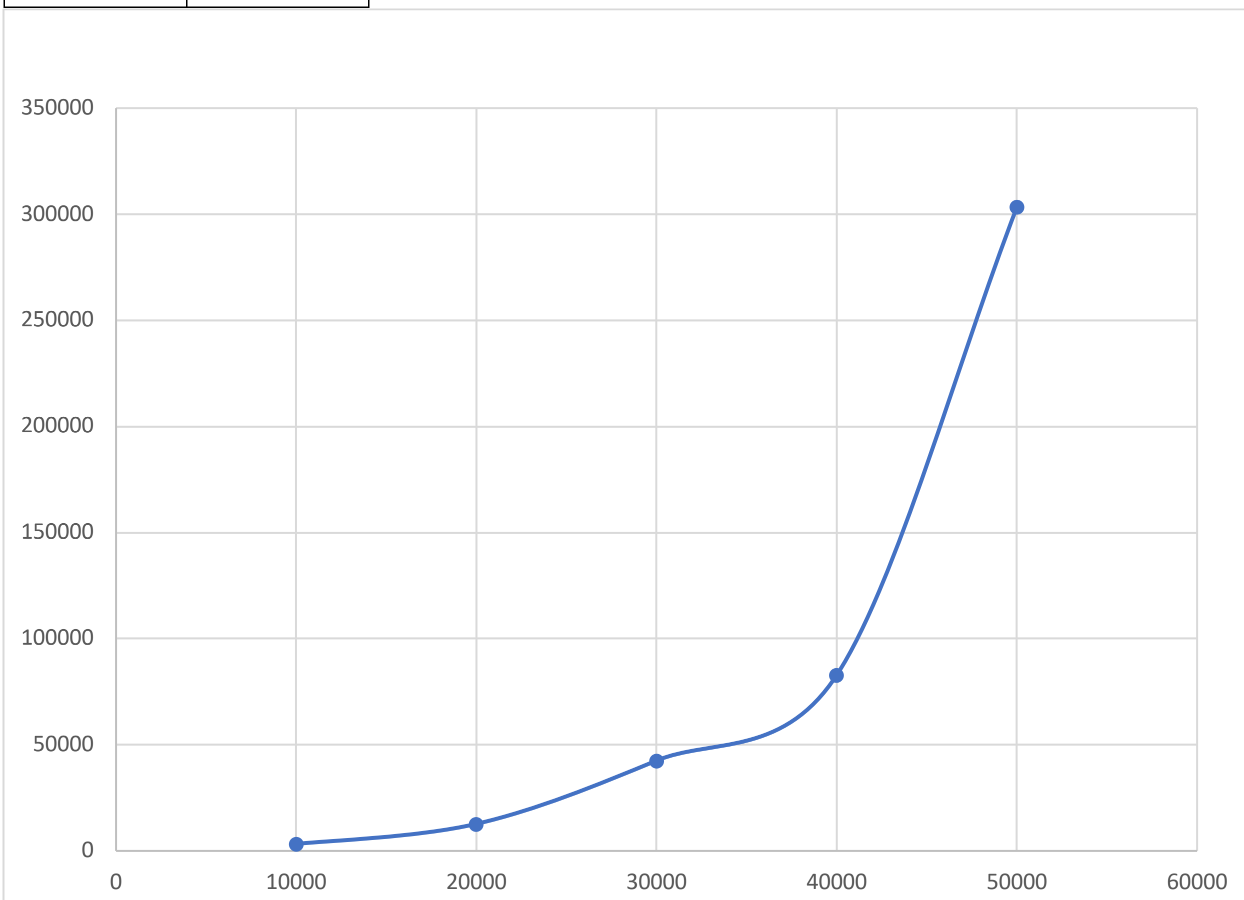
Radix sort using Insertion sort for m = 75

n	m = 75
10000	2888
20000	12268
30000	27592
40000	89538
50000	290844



Radix sort using Insertion sort for m = 100

n	m = 100
10000	3292
20000	12742
30000	42571
40000	82880
50000	303368



Radix sort using insertion sort which uses other stable algorithm that is insertion sort. Insertion sort is efficient for smaller arrays or sorted arrays. The time complexity of radix sort using insertion sort is O(d(n^2)) , which is less efficient than Radix sort using counting sort. The time complexity graph of radix sort using insertion sort varies quadratically with respect to input size n. As the input size increases, the time taken to sort the elements increases quadratically, as insertion sort as quadratic time complexity.

In conclusion Radix sort using counting sort is more efficient as counting sort time complexity varies linearly with input size n. Radix sort using insertion sort is inefficient for larger input sizes due to its quadratic time complexity of insertion sort.