Implement a program to do the following:
Input: An unsorted array of size n (user input) and fill it with random numbers.
Output: Sorted array
Procedure:
Write three functions separately that the main function calls one after another, where each function sorts the same array using a different algorithm, namely:
1. Radix sort
2. Bubble sort
3. Merge sort
Experiment: Compute the average runtime for each of the three techniques separately, for n = 10^3, 10^6 and 10^7.
Upload the code and results in a table.

Algo	**1000**	**1000000**	**10000000**
Bubble Sort	0.003799	-	-
Merge Sort	0.000155	0.269600	
Radix Sort	0.000207	-	-

Code

```
/*
Rohan Verma
1510110508
*/


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>


#define ll unsigned long long


void swap(ll * a, ll * b){
```

```
	ll temp = *a;
	*a = *b;
	*b = temp;
}

void print_array(ll* arr, ll n){
	for(ll i = 0; i < n; i++){
		printf("%ld ", arr[i]);
	}
	printf("\n");
}

void bubble_sort(ll arr[], ll size) {
	bool not_done = 1;
	while(not_done) {
		not_done = 0;
		for (ll i = 0; i < size-1; ++i)
		{
			if(arr[i] > arr[i+1]) {
				swap(&arr[i], &arr[i+1]);
				not_done = 1;
			}
		}
	}
}

void merge(ll arr[], ll start, ll mid, ll end) {
	ll size1 = mid-start+1, size2 = end-mid;
```

```
	ll arr1[size1], arr2[size2];

	for (ll i = start, j = 0; i <= mid; ++i, ++j)
		arr1[j] = arr[i];
	for (ll i = mid+1, j = 0; i <= end; ++i, ++j)
		arr2[j] = arr[i];

	ll i, j, k;
	i = j = 0;
	k = start;
	while(i < size1 && j < size2) {
		if(arr1[i] < arr2[j]) {
			arr[k] = arr1[i];
			i++; k++;
		}
		else {
			arr[k] = arr2[j];
			j++; k++;
		}
	}

	for (; i < size1; ++i, ++k)
		arr[k] = arr1[i];
	for (; j < size2; ++j, ++k)
		arr[k] = arr2[j];
}

void mergesort(ll arr[], ll l, ll r) {
```

```
      if(l < r) {
           ll mid = (l+r)/2;
           mergesort(arr, l, mid);
           mergesort(arr, mid+1, r);
           merge(arr, l, mid, r);
      }
}

void merge_sort(ll arr[], ll size) {
     mergesort(arr, 0, size);
}

int max_array(ll * array, ll size){

 ll i;
 ll largestNum = -1;

 for(i = 0; i < size; i++){
  if(array[i] > largestNum)
   largestNum = array[i];
 }

 return largestNum;
}

void radix_sort(ll array[], ll size) {

 // Base 10 is used
```

```
 int i;
 int semiSorted[size];
 int significantDigit = 1;
 int largestNum = max_array(array, size);

 while (largestNum / significantDigit > 0){

  int bucket[10] = { 0 };

  for (i = 0; i < size; i++)
   bucket[(array[i] / significantDigit) % 10]++;

  for (i = 1; i < 10; i++)
       bucket[i] += bucket[i - 1];

  for (i = size - 1; i >= 0; i--)
   semiSorted[--bucket[(array[i] / significantDigit) % 10]] =
array[i];


  for (i = 0; i < size; i++)
       array[i] = semiSorted[i];

  significantDigit *= 10;

 }
}
```

```
ll* generate_random_array(ll n){

	ll * array = (ll*) malloc(sizeof(ll)*n);

	for(ll i = 0; i < n; i++)
		array[i] = rand() % n;

	return array;

}

void sort_array(void (sorting_algo(ll* , ll)), ll* arr, ll n){
	sorting_algo(arr, n);
}

void print_usage(char* argv[]){
	printf("Usage: %s <n> <algorithm>\n", argv[0]);
	printf("<algorithm> :\n 0 = bubble_sort\n 1 = merge_sort\n 2 = radix_sort\n");
}

int main(int argc, char *argv[]) {

	if(argc > 2){

		srand(time(NULL));
```

```
        ll size = atol(argv[1]);
        ll algo = atol(argv[2]);
        if (algo > 2){
            print_usage(argv);
            return 0;
        }
        ll *array = generate_random_array(size);

        void (*sorting_algos[])(ll *, ll) = {bubble_sort, merge_sort,
radix_sort};

        clock_t t = clock();
        sort_array(sorting_algos[algo], array, size);
        t = clock() - t;
        // uncomment to view sorted array
        print_array(array, size);
        double time_elapsed = ((double)t)/CLOCKS_PER_SEC;
        printf("Time Taken: %f\n", time_elapsed);
    }
    else{
        print_usage(argv);
    }
    return 0;

}
```