**Assignment Number**

**Problem Statement**

Program in C to sort a bunch of numbers in ascending order by using radix sort algorithm.

**Theory**

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. A positional notation is required, but because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers. Radix sort is classified as a “sorting by distribution” algorithm, because the sorting operation is carried out by means of distribution based on consituent components in the elements.

**Algorithm**

**Input :** An array A[1, 2, …, n] where n is the number of elements.

**Output :** Elements of the array A sorted in ascending order.

**Data Structure Used :**

1. An auxiliary two dimensional array, buckets, with dimension 20 x n
2. An auxiliary array working as a pointer to each bucket subarray, bucketPointer, with dimension 20

**Steps :**

1. Set max = AbsMax(A) // AbsMax is a function which returns the

// maximum between the absolute values of the elements present in A

1. Set passes = NumDigits(max) // NumDigits is a function which // returns the number of digits in an integer value
2. While(passes > 0)
   1. For j = 1 to 20 do
      1. Set bucketPointer[j] = 0

EndFor

* 1. For j = 1 to count do
     1. Set element = A[j]
     2. Set digit = Extract(element, passes) // Extract is a function which

// extracts a digit from an integer value at a particular position

* + 1. If element >= 0
       1. Set digit = digit + 10
    2. Else
       1. Set digit = 9 – digit
    3. Set buckets[digit][bucketPointer[digit]] = element
    4. Set bucketPointer[digit] = bucketPointer[digit] + 1

EndFor

* 1. Set digit = 0, arrPointer = 0, tempPointer = 0
  2. While digit < 20
     1. if(tempPointer = bucketPointer[digit])
        1. Set tempPointer = 0
        2. Set digit = digit + 1
        3. Continue
     2. Set arr[arrPointer] = buckets[digit][tempPointer]
     3. Set arrPointer = arrPointer + 1
     4. Set tempPointer = tempPointer + 1

EndWhile

* 1. Set passes = passes – 1

EndWhile

**Source Code**

#include <stdlib.h>

#include <stdio.h>

// Radix sort (decimal)

// ===================

#define el\_abs(x) (x < 0 ? -x : x)

void sort\_radix(int \*arr, int count){

// Find the maximum number

int64\_t max = el\_abs(arr[0]);

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

if(max < el\_abs(arr[i]))

max = el\_abs(arr[i]);

}

// Count the number of digits in it, and hence number of passes

int passes = 0;

while(max > 0){

max /= 10; passes++;

}

// Create 20 buckets, for digits -ve 0-9 and +ve 0-9 respectively

int buckets[20][count];

int partExtractor = 10, digitExtractor = 1;

// Start pass

while(passes > 0){

// Pointer to buckets

int bucketPointer[20] = {0};

// Extraction loop

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

int64\_t element = arr[i]; // Get the element at ith position

int64\_t digit = ((el\_abs(element))% partExtractor) / digitExtractor;

// Extract the digit

if(element >= 0)

digit += 10;

else

digit = 9 - digit;

buckets[digit][bucketPointer[digit]] = element; // Put the element into

// required bucket

bucketPointer[digit]++; // Increase the bucketPointer for that digit

}

// Put them again in the original array

int digit = 0, arrPointer = 0, tempPointer = 0;

while(digit < 20){

if(tempPointer == bucketPointer[digit]){ // tempPointer should always

// be less than

tempPointer = 0; // bucketPointer if the bucket at digit

digit++; // has atleast one element

continue;

}

// Put the element from bucket to the original array

arr[arrPointer] = buckets[digit][tempPointer];

arrPointer++;

tempPointer++;

}

// Increment

partExtractor \*= 10;

digitExtractor \*= 10;

passes--;

}

}

static void print\_list(int \*arr, int count){

printf("{ %d", arr[0]);

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

printf(", %d", arr[i]);

}

printf(" }");

}

int main(){

int num;

printf("\nEnter the number of elements : ");

scanf("%d", &num);

if(num < 1){

printf("\nThere should be atleast 1 element!\n");

return 1;

}

int \*arr = (int \*)malloc(sizeof(int) \* num);

for(int temp = 0;temp < num;temp++){

printf("\nEnter element %d : ", (temp + 1));

scanf("%d", &arr[temp]);

}

printf("\nBefore sorting : ");

print\_list(arr, num);

sort\_radix(arr, num);

printf("\nAfter sorting : ");

print\_list(arr, num);

printf("\n");

free(arr);

return 0;

}

**Input and Output**

**Set 1:**

Enter the number of elements : 5

Enter element 1 : 3881

Enter element 2 : 3391

Enter element 3 : 48201

Enter element 4 : 228

Enter element 5 : 48

Before sorting : { 3881, 3391, 48201, 228, 48 }

After sorting : { 48, 228, 3391, 3881, 48201 }

**Set 2:**

Enter the number of elements : 10

Enter element 1 : -482

Enter element 2 : 821

Enter element 3 : 3891

Enter element 4 : -3874

Enter element 5 : 3381

Enter element 6 : 3911

Enter element 7 : -5891

Enter element 8 : 3891

Enter element 9 : 3912

Enter element 10 : -84712

Before sorting : { -482, 821, 3891, -3874, 3381, 3911, -5891, 3891, 3912, -84712 }

After sorting : { -84712, -5891, -3874, -482, 821, 3381, 3891, 3891, 3911, 3912 }

**Discussion**

1. Radix sort is largely dependent on maximum number of digits in all the elements in the input array. If there is a few items will large number of digits, the algorithm will be slower altogther.
2. Radix sort complexity is *O*(*wn*) for *n* keys which are integers of word size *w*. Sometimes *w* is presented as a constant, which would make radix sort better (for sufficiently large *n*) than the best comparison-based sorting algorithms, which all perform *O*(*n* log *n*) comparisons to sort *n* keys. However, in general *w* cannot be considered a constant: if all *n* keys are distinct, then *w* has to be at least log *n* for a random-access machine to be able to store them in memory, which gives at best a time complexity *O*(*n* log *n*). That would seem to make radix sort at most equally efficient as the best comparison-based sorts (and worse if keys are much longer than log *n*).