RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.

During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the n^{th} pass, where n is the length of the name with maximum number of letters.

After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digts.

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS =
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:           SET I =   and INITIALIZE buckets
Step 6:           Repeat Steps 7 to 9 while I<N-1
Step 7:                 SET DIGIT = digit at PASSth place in A[I]
Step 8:                 Add A[I] to the bucket numbered DIGIT
Step 9:                 INCEREMENT bucket count for bucket numbered DIGIT
                  [END OF LOOP]
Step 1 :          Collect the numbers in the bucket
      o [END OF LOOP]
Step 11: END
```

Algorithm for radix sort

Example 14.7 Sort the numbers given below using radix sort. 345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

n the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result.

After the third pass, the list can be given as 123, 345, 472, 555, 567, 654, 808, 911, 924.

Complexity of Radix Sort

To calculate the complexity of radix sort algorithm, assume that there are nnumbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes O(kn) time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

Pros and Cons of Radix Sort

Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.

But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters.

Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task.

Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Program to implement Radix Sort

```
#include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
int main()
{
int arr[size], i, n;
printf("\n Enter the number of elements in the array: ");
scanf("%d", &n);
printf("\n Enter the elements of the array: ");
for(i=0;i<n;i++)
{
scanf("%d", &arr[i]);
}
radix_sort(arr, n);
printf("\n The sorted array is: \n");
for(i=0;i<n;i++)
printf(" %d\t", arr[i]);
return(0);
getch();
}
int largest(int arr[], int n)
{
int large=arr[0], i;
for(i=1;i<n;i++)
{
if(arr[i]>large)
large = arr[i];
}
return large;
}
void radix_sort(int arr[], int n)
{
int bucket[size][size], bucket_count[size];
int i, j, k, remainder, NOP=0, divisor=1, large, pass;
large = largest(arr, n);
while(large>0)
{
NOP++;
large/=size;
}
for(pass=0;pass<NOP;pass++) // Initialize the buckets
```

```
{
for(i=0;i<size;i++)
bucket_count[i]=0;
for(i=0;i<n;i++)
{
// sort the numbers according to the digit at passth place
remainder = (arr[i]/divisor)%size;
bucket[remainder][bucket_count[remainder]] = arr[i];
bucket_count[remainder] += 1;
}
// collect the numbers after PASS pass
i=0;
for(k=0;k<size;k++)
{
for(j=0;j<bucket_count[k];j++)
{
arr[i] = bucket[k][j];
i++;
}
}
printf("\n array after %d pass", pass);
for(int l=0;l<n;l++)
        printf(" %d ",arr[l]);
divisor *= size;
}
}
```