Radix sort algorithm

***Source:* https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms/Sorting_Algorithms**

1. *What is the Radix Sort algorithm?*

Radix sort is an integer sorting algorithm that sorts data by grouping the individual digits of the data according to their radix (base) and then sorting the data according to each digit. This is done by repeatedly sorting the data based on each digit, starting with the least significant digit and moving on to the most significant digit.

In C++, radix sort can be implemented using an array of linked lists to store the data, with one list for each digit. The data is then repeatedly passed through the array of lists, with each pass sorting the data based on the next digit. The end result is a sorted list of data. It is also can be implemented using counting sort.

The idea of Radix Sort is to do digit-by-digit sorting starting from the least significant digit to the most significant digit. Radix sort uses counting sort as a subroutine to sort.

Some of the key points for Radix sort are given below:

- It makes assumptions about the data like the data must be between a range of elements.
- Input array must have the elements with the same radix and width.
- Radix sort works on sorting based on an individual digit or letter position.
- We must start sorting from the rightmost position and use a stable algorithm at each position.
- Radix sort is not an in-place algorithm as it uses a temporary count array.

2. *Code example.*

```
#include <iostream>
#include <cmath>

void radixSort(int arr[], int n) {
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit.
    for (int exp = 1; m/exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}
void countingSort(int arr[], int n, int exp) {
    int output[n];
    int i, count[10] = {0};
    // Store count of occurrences in count[]
    for (i = 0; i < n; i++) {
        count[ (arr[i]/exp)%10 ]++;
    }
    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }
    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}
```

```
int getMax(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > mx) {
            mx = arr[i];
        }
    }
    return mx;
}

int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr)/sizeof(arr[0]);

    radixSort(arr, n);

    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

This program will output the sorted array:

```
2 24 45 66 75 90 170 802
```

3. Code explanation.

The example above shows a C++ implementation of the radix sort algorithm. The radix sort algorithm is used to sort an array of integers by repeatedly sorting the array based on the digits of the integers, starting with the least significant digit and moving on to the most significant digit.

The **`radixSort()`** function takes an array of integers and their size as input and first finds the maximum number in the array using the **`getMax()`** function. The maximum number is used to determine the number of digits in the largest number in the array, which is needed to know how many times to iterate over the array to sort all digits.

Next, the **radixSort()** function repeatedly calls the **countingSort()** function, which sorts the array based on a specific digit. The **countingSort()** function uses the counting sort algorithm to sort the array.

The **countingSort()** function takes an array of integers, its size, and the current digit **(exp)** being considered as input. It creates an output array and a count array to store the count of occurrences of each digit. It then counts the occurrences of each digit in the input array by iterating over the input array and incrementing the count of the corresponding digit in the count array.

Then the **countingSort()** function modifies the count array so that it now contains the actual positions of each digit in the output array. It then builds the output array by iterating over the input array backward and placing each element in its correct position in the output array based on the count array.

Finally, the **countingSort()** function copies the output array to the input array so that the input array now contains the elements sorted according to the current digit.

The outer loop of **radixSort()** function will run log(k) where k is the largest integer, because at every iteration it sorts the array based on the next significant digit, this number of iterations is dependent on the largest number that is k.

The **countingSort()** function will run in **O(n)** because the number of iterations of the loop is directly proportional to the size of the input array.

Overall the Radix Sort will run in **O(nk)** where n is the number of elements and k is the number of digits in the largest number.

It's worth noting that in this example, I've assumed that the input array contains only non-negative integers, as the counting sort assumes positive integers as input.

4. Time complexity.

The time complexity of radix sort is O(n*k), where n is the number of elements to be sorted and k is the number of digits in the largest number.

The outer loop of the **radixSort()** function iterates log(k) times, where k is the largest number in the array. On each iteration, it calls the **countingSort()** function, which takes **O(n)** time to sort the array based on the current digit. Since the outer loop iterates **log(k)** times, the total time complexity of the **radixSort()** function is **O(n*log(k)).**

It's worth noting that in this implementation I've used counting sort as a sub-routine, the time complexity of counting sort is **O(n+k)** where **n** is the number of elements to be sorted and **k** is the range of possible key values.

In summary, the radix sort algorithm has a time complexity of **O(n*k)** which makes it an efficient algorithm when used to sort a large number of integers with the same number of digits.

I wish you happy learning,

Your Rosen Rusev