NAME: EZE, ALLOYSIUS IZUCHUKWU
MATRIC NUMBER: 24PGCD00498
ADVANCED ALGORITHM (CSC 811)
COMPUTER SCIENCE DEPARTMENT, LANDMARK UNIVERSITY
MOBILE:0810 7233 021

SORT ALGORITHM

This assignment describes two sort algorithm which includes Merge Algorithm and Radix Algorithm. These two will be discoursed in details with practical examples below:

1). MERGE SORT

Paradigm: Divide and Conquer

Description: Merge Sort divides the array into halves, recursively sorts each half, and merges the sorted halves to produce the final sorted array.

APPROACH:

Split the array into two halves until each subarray contains only one element.
Merge the subarrays by comparing their elements and arranging them in sorted order.Repeat until all subarrays are merged into one sorted array.

Sample Code:

```
#include <iostream>
// Function to merge two sorted subarrays
void merge(int arr[], int left, int mid, int right) {
// Calculate the sizes of the two subarrays
int n1 = mid - left + 1;
int n2 = right - mid;
// Create temporary arrays to store the subarrays
int leftArr[n1], rightArr[n2];
// Copy the elements from the original array to the temporary arrays
for (int i = 0; i < n1; i++) {
leftArr[i] = arr[left + i];
}
for (int j = 0; j < n2; j++) {
rightArr[j] = arr[mid + 1 + j];
}
// Merge the temporary arrays back into the original array
int i = 0, j = 0, k = left;
while (i < n1 && j < n2) {
if (leftArr[i] <= rightArr[j]) {
arr[k] = leftArr[i];
i++;
} else {
arr[k] = rightArr[j];
j++;
}
k++;
}

// Copy any remaining elements from the temporary arrays
while (i < n1) {
arr[k] = leftArr[i];
i++;
k++;
}
while (j < n2) {
arr[k] = rightArr[j];
j++;
k++;
}}
// Function to implement the Merge Sort algorithm
void mergeSort(int arr[], int left, int right) {
if (left < right) {
// Calculate the middle index
int mid = left + (right - left) / 2;
// Recursively sort the left and right subarrays
mergeSort(arr, left, mid);
mergeSort(arr, mid + 1, right);
// Merge the sorted subarrays
merge(arr, left, mid, right);
}}
// Function to print the array
void printArray(int arr[], int size) {
for (int i = 0; i < size; i++) {
std::cout << arr[i] << " ";
}
std::cout << std::endl;
}
// Driver code
int main() {
int arr[] = {64, 34, 25, 12, 22, 11, 90};
int n = sizeof(arr) / sizeof(arr[0]);
std::cout << "Original array: ";
printArray(arr, n);
mergeSort(arr, 0, n - 1);
std::cout << "Sorted array: ";
printArray(arr, n);
return 0;
}
```

Output:

Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90

The merge function merges two sorted subarrays into a single sorted subarray. The merge Sort function recursively divides the array into smaller subarrays, sorts them, and merges them back together using the merge function. The print Array function prints the elements of the array. In the main function, we define an example array, print the original array, sort it using merge Sort, and print the sorted array.This implementation demonstrates the Merge Sort algorithm, which is a divide-and-conquer algorithm that splits the input array into two halves, recursively sorts them, and merges them back together

Further Demonstration: Sort the array [38, 27, 43, 3, 9, 82, 10]:

1. Split into [38, 27, 43] and [3, 9, 82, 10].
2. Split [38, 27, 43] into [38] and [27, 43], then [27, 43] into [27] and [43].
3. Merge [27] and [43] into [27, 43], then merge [38] and [27, 43] into [27, 38, 43].
4. Split [3, 9, 82, 10] into [3, 9] and [82, 10], then [3, 9] into [3] and [9].
5. Merge [3] and [9] into [3, 9], then [82, 10] into [10, 82], then merge [3, 9] and [10, 82] into [3, 9, 10, 82].
6. Finally, merge [27, 38, 43] and [3, 9, 10, 82] into [3, 9, 10, 27, 38, 43, 82].

Merge Sort is efficient for large arrays with a time complexity of O(nlog⁡n)O(n \log n)O(nlogn).

2). RADIX SORT

Paradigm: The Paradigm used here is Divide and Conquer

Description: Radix Sort sorts numbers by processing their digits individually, starting from the least significant digit (LSD) to the most significant digit (MSD) or vice versa.

Approach:

Identify the maximum number in the array to determine the number of digits to process.Sort the array based on the least significant digit (units place).Proceed to the next significant digit (tens place, hundreds place, etc.). Repeat until all digits have been processed, and the array is sorted.

Advantages:

Extremely efficient for sorting integers or fixed-width strings. Linear time complexity for small ranges of digits.

Disadvantages:

Not suitable for sorting very large integers with many digits. Requires additional memory for auxiliary arrays.

Characteristics of Radix Sort

Stable: Radix sort is a stable sorting algorithm, meaning that the order of equal elements is preserved.

Non-comparative: Radix sort does not compare elements directly, instead, it relies on the digits of the elements to determine the order.

Efficient: Radix sort has a time complexity of O(nk), where n is the number of elements and k is the number of digits in the radix sort.

Time Complexity:

Best, Average, Worst: O(d×(n+b))O(d \times (n + b))O(d×(n+b)), where:

d: Number of digits in the largest number.

n: Number of elements in the array.

b: Base of the number system (e.g., 10 for decimal).

- **Space Complexity**: O(n+b)O(n + b)O(n+b) for temporary storage.

Sample Code

```
#include <iostream>
#include <vector>
#include <algorithm>
// Function to find the maximum element in the array
int findMax(std::vector<int>& arr) {
return *std::max_element(arr.begin(), arr.end());
}
// Function to perform counting sort based on significant places
void countingSort(std::vector<int>& arr, int place) {
const int max = 10;
std::vector<int> output(arr.size());
std::vector<int> count(max);
// Initialize count array with all zeros
for (int i = 0; i < max; ++i) {
count[i] = 0;
}
// Store the count of occurrences in count[]
for (int i = 0; i < arr.size(); ++i) {
count[(arr[i] / place) % 10]++;
}
// Change count[i] so that count[i] now contains actual position of this digit in output array
for (int i = 1; i < max; ++i) {
count[i] += count[i - 1];
}
// Build the output array
for (int i = arr.size() - 1; i >= 0; --i) {
output[count[(arr[i] / place) % 10] - 1] = arr[i];
count[(arr[i] / place) % 10]--;
}
// Copy the output array to arr[], so that arr[] now contains sorted numbers according to current
digit
for (int i = 0; i < arr.size(); ++i) {
arr[i] = output[i];
}}
// Function to implement radix sort
void radixSort(std::vector<int>& arr) {
int max = findMax(arr);
// Perform counting sort for every digit. Note that instead of passing digit number, exp is passed.
exp is 10^i where i is the current digit number
for (int place = 1; max / place > 0; place *= 10) {
countingSort(arr, place);
}}
// Function to print the array
void printArray(const std::vector<int>& arr) {
for (int num : arr) {
std::cout << num << " ";
} std::cout << std::endl;
}
// Driver code
int main() {
std::vector<int> arr = {170, 45, 75, 90, 802, 24, 2, 66};
std::cout << "Original array: ";
printArray(arr);
radixSort(arr);
std::cout << "Sorted array: ";
printArray(arr);

return 0;
}
```

The find Max function finds the maximum element in the array to determine the number of digits. The counting Sort function performs counting sort based on significant places. The radix Sort function implements radix sort by performing counting sort for every digit. The print Array function prints the array.

Output:

```
Original array: 170 45 75 90 802 24 2 66
Sorted array: 2 24 45 66 75 90 170 802
```

Further Demonstration **:**

Sort the array [170, 45, 75, 90, 802, 24, 2, 66].

Step 1: Sort by the least significant digit (units place):

- Extract digits: [0, 5, 5, 0, 2, 4, 2, 6]
- Sort: [170, 90, 802, 2, 24, 45, 75, 66]

Step 2: Sort by the next significant digit (tens place):

- Extract digits: [7, 9, 0, 0, 2, 4, 7, 6]
- Sort: [802, 2, 24, 45, 66, 170, 75, 90]

Step 3: Sort by the most significant digit (hundreds place):

- Extract digits: [8, 0, 0, 0, 0, 1, 0, 0]
- Sort: [2, 24, 45, 66, 75, 90, 170, 802]

Final sorted array: [2, 24, 45, 66, 75, 90, 170, 802]