|  |  |
| --- | --- |
| **Course Code:** CT2352 | **Course Name:** Lab - Design & Analysis of Algorithms |

|  |  |
| --- | --- |
| **Name of the Student:**  S Akshansh | **Sem/Sec:**  6th/A |
| **Branch:**  Computer Technology | **Roll no.:** 72  **Enrolment no.:** 19010927 |

**PRACTICAL NO. 4**

**AIM:** Write a program to sort an integer array using merge sort and quick sort and also give its time complexity.

**THEORY:**

**MERGE SORT**

* Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.
* Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.
* It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

**Applications of Merge Sort**

1. [Merge Sort is useful for sorting linked lists in O(nLogn) time](https://www.geeksforgeeks.org/merge-sort-for-linked-list/)
2. [Inversion Count Problem](https://www.geeksforgeeks.org/counting-inversions/)
3. Used in [External Sorting](http://en.wikipedia.org/wiki/External_sorting)

**Algorithm for Merge Sort:**

MergeSort(arr[], l, r)

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = l+ (r-l)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

|  |  |
| --- | --- |
| **Time Complexity** |  |
| Best | O(n\*log n) |
| Worst | O(n\*log n) |
| Average | O(n\*log n) |

**QUICK SORT**

Quicksort is a sorting algorithm based on the divide and conquer approach where

* An array is divided into subarrays by selecting a pivot element (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

* The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
* At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

**Applications of Quick Sort:**

* Commercial Computing is used in various government and private organizations for the purpose of sorting various data like sorting files by name/date/price, sorting of students by their roll no., sorting of account profile by given id, etc.
* The [sorting](https://www.geeksforgeeks.org/sorting-algorithms/)algorithm is used for information searching and as Quicksort is the fastest algorithm so it is widely used as a better way of searching.
* It is used everywhere where a stable sort is not needed.
* Quicksort is a [cache-friendly](https://www.geeksforgeeks.org/cache-memory-in-computer-organization/) algorithm as it has a good locality of reference when used for arrays.

|  |  |
| --- | --- |
| **Time Complexity** | |
| Best | O(n\*log n) |
| Worst | O(n2) |
| Average | O(n\*log n) |

**Algorithm for Quick Sort:**

quickSort(array, leftmostIndex, rightmostIndex)

if (leftmostIndex < rightmostIndex)

pivotIndex <- partition(array,leftmostIndex, rightmostIndex)

quickSort(array, leftmostIndex, pivotIndex - 1)

quickSort(array, pivotIndex, rightmostIndex)

The partition algorithm rearranges the sub-arrays in a place.

partition(array, leftmostIndex, rightmostIndex)

set rightmostIndex as pivotIndex

storeIndex <- leftmostIndex - 1

for i <- leftmostIndex + 1 to rightmostIndex

if element[i] < pivotElement

swap element[i] and element[storeIndex]

storeIndex++

swap pivotElement and element[storeIndex+1]

return storeIndex + 1

**CODE:**

**Merge Sort:**

#include<iostream>

using namespace std;

void swapping(int &a, int &b) {     //swap the content of a and b

   int temp;

   temp = a;

   a = b;

   b = temp;

}

void display(int \*array, int size) {

   for(int i = 0; i<size; i++)

      cout << array[i] << " ";

   cout << endl;

}

void merge(int \*array, int l, int m, int r) {

   int i, j, k, nl, nr;

   //size of left and right sub-arrays

   nl = m-l+1; nr = r-m;

   int larr[nl], rarr[nr];

   //fill left and right sub-arrays

   for(i = 0; i<nl; i++)

      larr[i] = array[l+i];

   for(j = 0; j<nr; j++)

      rarr[j] = array[m+1+j];

   i = 0; j = 0; k = l;

   //marge temp arrays to real array

   while(i < nl && j<nr) {

      if(larr[i] <= rarr[j]) {

         array[k] = larr[i];

         i++;

      }else{

         array[k] = rarr[j];

         j++;

      }

      k++;

   }

while(i<nl) { //extra element in left array

array[k] = larr[i];

i++; k++;

}

while(j<nr) { //extra element in right array

array[k] = rarr[j];

j++; k++;

}

}

void mergeSort(int \*array, int l, int r) {

int m;

if(l < r) {

int m = l+(r-l)/2;

// Sort first and second arrays

mergeSort(array, l, m);

mergeSort(array, m+1, r);

merge(array, l, m, r);

}

}

int main() {

int n;

cout << "Enter the number of elements: ";

cin >> n;

int arr[n]; //create an array with given number of elements

cout << "Enter elements:" << endl;

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

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

mergeSort(arr, 0, n-1); //(n-1) for last index

cout << "Array after Sorting: ";

display(arr, n);

}

**Output:**

Text

Description automatically generated

**Quick Sort:**

// C++ Implementation of the Quick Sort Algorithm.

#include <iostream>

using namespace std;

int partition(int arr[], int start, int end)

{

int pivot = arr[start];

int count = 0;

for (int i = start + 1; i <= end; i++) {

if (arr[i] <= pivot)

count++;

}

// Giving pivot element its correct position

int pivotIndex = start + count;

swap(arr[pivotIndex], arr[start]);

// Sorting left and right parts of the pivot element

int i = start, j = end;

while (i < pivotIndex && j > pivotIndex)

{

while (arr[i] <= pivot) {

i++;

}

while (arr[j] > pivot) {

j--;

}

if (i < pivotIndex && j > pivotIndex) {

swap(arr[i++], arr[j--]);

}

}

return pivotIndex;

}

void quickSort(int arr[], int start, int end)

{

// base case

if (start >= end)

return;

// partitioning the array

int p = partition(arr, start, end);

// Sorting the left part

quickSort(arr, start, p - 1);

// Sorting the right part

quickSort(arr, p + 1, end);

}

int main()

{

int arr[] = { 9, 3, 4, 2, 1, 8 };

int n = 6;

quickSort(arr, 0, n - 1);

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

cout << arr[i] << " ";

}

return 0;

}

**Output:**

**Graphical user interface, text, application

Description automatically generated**

**CONCLUSION:** In this practical I executed program to sort an integer array using merge sort and quick sort and also provided its time complexity.