**Department of Computing**

**School of Electrical Engineering and Computer Science**

**CS250 – Data Structures and Algorithms**



**Lab 12: Merge & Quick Sort**

**Submission Details**

|  |  |
| --- | --- |
| Name | CMS ID |
| Muhammad Umer | 345834 |
| Group | GP – 1 |
| Lab Engineer | Anum Asif |
| Faculty Member | Bostan Khan |
| Class | BEE12 |
| Date | 29/04/2024 |
| Time | 10:00 am – 12:50 pm |

**Table of Contents**

[2 Merge & Quick Sort 3](#_Toc165279027)

[2.1 Introduction 3](#_Toc165279028)

[2.2 Objectives 3](#_Toc165279029)

[2.3 Tools/Software Requirement 3](#_Toc165279030)

[2.4 Deliverables 3](#_Toc165279031)

[2.5 Description 3](#_Toc165279032)

[3 Lab Tasks 5](#_Toc165279033)

[3.1 Task 1 5](#_Toc165279034)

[3.2 Task 2 (average case complexity) 6](#_Toc165279035)

[3.3 Task 3 (best- and worst-case complexity) 9](#_Toc165279036)

# Merge & Quick Sort

## Introduction

In this lab, you will implement merge and quick sort.

## Objectives

Objective of this lab is to implement quick sort then compare the running times for sorting.

## Tools/Software Requirement

* Visual Studio C++

## Deliverables

Compile a single word document by filling in the solution parts and submit this file on LMS. The name of word document should follow this format. i.e., YourFullName(reg)\_Lab#. You must show the implementation of the tasks in a complete manner to get your work graded.

## Description

**Merge Sort**

Merge sort is another important sorting algorithm that we have seen. Unlike insertion sort, it is not an in-place sorting algorithm. The pseudo code for merge sort is shown below:



Merge (Arr, n1, mid, n2)

a=n1, b=mid, c=n1 ,B;

while a <= mid and b<=n2

if Arr[a]<Arr[b]

B[c++]=Arr[a++];

else

B[c++]=Arr[b++];

while a<mid

B[c++]=Arr[a++];

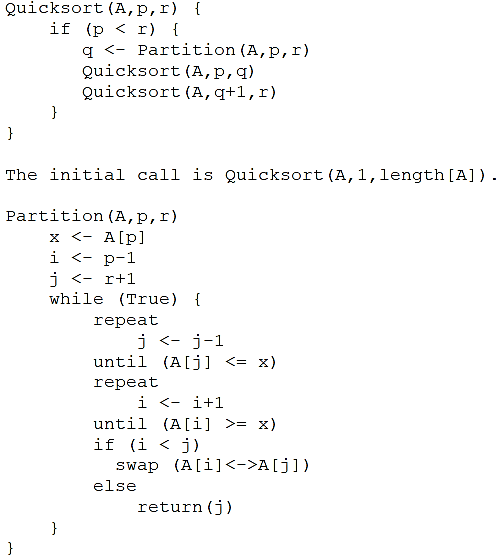
while b<n2

B[c++]=Arr[b++];

for a=n1; a<n2; a++

Arr[a]=B[a];

**Quick Sort**

****

# Lab Tasks

## Task 1

Implement **M**erge sort and **Q**uick sort algorithms in C++.

Code: Merge Sort

*void* merge(*int* *arr*[], *int* *n1*, *int* *mid*, *int* *n2*) {

*int* a = *n1*, b = *mid*, c = *n1*, B[*n2*];

    while (a < *mid* && b < *n2*) {

        if (*arr*[a] < *arr*[b]) {

            B[c++] = *arr*[a++];

        } else {

            B[c++] = *arr*[b++];

        }

    }

    while (a < *mid*) {

        B[c++] = *arr*[a++];

    }

    while (b < *n2*) {

        B[c++] = *arr*[b++];

    }

    for (a = *n1*; a < *n2*; a++) {

*arr*[a] = B[a];

    }

}

*void* mergeSort(*int* *arr*[], *int* *first*, *int* *last*) {

    if (*first* < *last* - 1) {

*int* mid = (*first* + *last*) / 2;

        mergeSort(*arr*, *first*, mid);

        mergeSort(*arr*, mid, *last*);

        merge(*arr*, *first*, mid, *last*);

    }

}

Code: Quick Sort

*int* partition(*int* *arr*[], *int* *p*, *int* *r*) {

*int* x = *arr*[*r*];

*int* i = *p* - 1;

    for (*int* j = *p*; j <= *r* - 1; j++) {

        if (*arr*[j] <= x) {

            i++;

            swap(*arr*[i], *arr*[j]);

        }

    }

    swap(*arr*[i + 1], *arr*[*r*]);

    return (i + 1);

}

*void* quickSort(*int* *arr*[], *int* *p*, *int* *r*) {

    if (*p* < *r*) {

*int* q = partition(*arr*, *p*, *r*);

        quickSort(*arr*, *p*, q - 1);

        quickSort(*arr*, q + 1, *r*);

    }

}

Main Function

*int* main() {

*int* arr\_quick[] = {5, 3, 8, 4, 2, 7, 1, 10, 6, 9};

*int* arr\_merge[] = {5, 3, 8, 4, 2, 7, 1, 10, 6, 9};

    cout << "Quick Sort:" << endl;

    quickSort(arr\_quick, 0, 9);

    for (*int* i = 0; i < 10; i++) {

        cout << arr\_quick[i] << " ";

    }

    cout << endl;

    cout << "Merge Sort:" << endl;

    mergeSort(arr\_merge, 0, 10);

    for (*int* i = 0; i < 10; i++) {

        cout << arr\_merge[i] << " ";

    }

    cout << endl;

    return 0;

}

Output

root@Zonularity:/home/zonularity/dsa/lab\_12# ./task\_1

Quick Sort:

1 2 3 4 5 6 7 8 9 10

Merge Sort:

1 2 3 4 5 6 7 8 9 10

## Task 2 (average case complexity)

The next step is to compare the two algorithms. Generate arrays of random numbers in the range 1 to 100 with sizes 100, 1000, 10000, and 100000. Compare the running times of the two algorithms on each array. How do they compare? Are the results what you expected, and why? Answer the questions in the solution section.

Code

#include <stdlib.h>

#include <time.h>

#include <iostream>

using *namespace* std;

*void* merge(*int* *arr*[], *int* *n1*, *int* *mid*, *int* *n2*) {

*int* a = *n1*, b = *mid*, c = *n1*, B[*n2*];

    while (a < *mid* && b < *n2*) {

        if (*arr*[a] < *arr*[b]) {

            B[c++] = *arr*[a++];

        } else {

            B[c++] = *arr*[b++];

        }

    }

    while (a < *mid*) {

        B[c++] = *arr*[a++];

    }

    while (b < *n2*) {

        B[c++] = *arr*[b++];

    }

    for (a = *n1*; a < *n2*; a++) {

*arr*[a] = B[a];

    }

}

*void* mergeSort(*int* *arr*[], *int* *first*, *int* *last*) {

    if (*first* < *last* - 1) {

*int* mid = (*first* + *last*) / 2;

        mergeSort(*arr*, *first*, mid);

        mergeSort(*arr*, mid, *last*);

        merge(*arr*, *first*, mid, *last*);

    }

}

*int* partition(*int* *arr*[], *int* *p*, *int* *r*) {

*int* x = *arr*[*r*];

*int* i = *p* - 1;

    for (*int* j = *p*; j <= *r* - 1; j++) {

        if (*arr*[j] <= x) {

            i++;

            swap(*arr*[i], *arr*[j]);

        }

    }

    swap(*arr*[i + 1], *arr*[*r*]);

    return (i + 1);

}

*void* quickSort(*int* *arr*[], *int* *p*, *int* *r*) {

    if (*p* < *r*) {

*int* q = partition(*arr*, *p*, *r*);

        quickSort(*arr*, *p*, q - 1);

        quickSort(*arr*, q + 1, *r*);

    }

}

*int* main() {

    srand(time(NULL));

*int* sizes[] = {100, 1000, 10000, 100000};

    for (*int* size : sizes) {

*int* arr[size];

        for (*int* i = 0; i < size; i++) {

            arr[i] = rand() % 100 + 1;

        }

*clock\_t* start = clock();

        quickSort(arr, 0, size);

*clock\_t* end = clock();

        cout << "Quick sort on size " << size << " took "

             << (*double*)(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl;

        start = clock();

        mergeSort(arr, 0, size);

        end = clock();

        cout << "Merge sort on size " << size << " took "

             << (*double*)(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl

             << endl;

    }

    return 0;

}

Output

root@Zonularity:/home/zonularity/dsa/lab\_12# ./task\_2

Quick sort on size 100 took 1.7e-05 seconds

Merge sort on size 100 took 1.1e-05 seconds

Quick sort on size 1000 took 0.000203 seconds

Merge sort on size 1000 took 0.000108 seconds

Quick sort on size 10000 took 0.009193 seconds

Merge sort on size 10000 took 0.001208 seconds

Quick sort on size 100000 took 0.665041 seconds

Merge sort on size 100000 took 0.091762 seconds

Insights

Despite the average case time complexity of both quick and merge sort being , in our case, merge sort is faster than quick sort. This is because quick sort has a worst-case time complexity of ; we are generating arrays of random numbers, so the worst-case time complexity is more likely to occur.

## Task 3 (best- and worst-case complexity)

Now sort the arrays using stl::sort, once in ascending order and then in descending order. Given both sorted arrays as inputs to all algorithms and compute their running time. The running time of which algorithm shows most variations based on the structure of the input and why? Answer the questions in the solution section.

Code

#include <time.h>

#include <algorithm>

#include <iostream>

using *namespace* std;

*void* merge(*int* *arr*[], *int* *n1*, *int* *mid*, *int* *n2*) {

*int* a = *n1*, b = *mid*, c = *n1*, B[*n2*];

    while (a < *mid* && b < *n2*) {

        if (*arr*[a] < *arr*[b]) {

            B[c++] = *arr*[a++];

        } else {

            B[c++] = *arr*[b++];

        }

    }

    while (a < *mid*) {

        B[c++] = *arr*[a++];

    }

    while (b < *n2*) {

        B[c++] = *arr*[b++];

    }

    for (a = *n1*; a < *n2*; a++) {

*arr*[a] = B[a];

    }

}

*void* mergeSort(*int* *arr*[], *int* *first*, *int* *last*) {

    if (*first* < *last* - 1) {

*int* mid = (*first* + *last*) / 2;

        mergeSort(*arr*, *first*, mid);

        mergeSort(*arr*, mid, *last*);

        merge(*arr*, *first*, mid, *last*);

    }

}

*int* partition(*int* *arr*[], *int* *p*, *int* *r*) {

*int* x = *arr*[*r*];

*int* i = *p* - 1;

    for (*int* j = *p*; j <= *r* - 1; j++) {

        if (*arr*[j] <= x) {

            i++;

            swap(*arr*[i], *arr*[j]);

        }

    }

    swap(*arr*[i + 1], *arr*[*r*]);

    return (i + 1);

}

*void* quickSort(*int* *arr*[], *int* *p*, *int* *r*) {

    if (*p* < *r*) {

*int* q = partition(*arr*, *p*, *r*);

        quickSort(*arr*, *p*, q - 1);

        quickSort(*arr*, q + 1, *r*);

    }

}

*int* main() {

*int* n = 10000;

*int* arr1[n], arr2[n];

    for (*int* i = 0; i < n; i++) {

        arr1[i] = i;

        arr2[i] = n - i;

    }

*clock\_t* start, end;

*double* timeTaken;

    start = clock();

    sort(arr1, arr1 + n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "STL sort in ascending order: " << timeTaken << endl;

    start = clock();

    sort(arr2, arr2 + n, greater<*int*>());

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "STL sort in descending order: " << timeTaken << endl

         << endl;

    for (*int* i = 0; i < n; i++) {

        arr1[i] = i;

        arr2[i] = n - i;

    }

    start = clock();

    quickSort(arr1, 0, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by quick sort (ascending): " << timeTaken << endl;

    start = clock();

    quickSort(arr2, 0, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by quick sort (descending): " << timeTaken << endl

         << endl;

    for (*int* i = 0; i < n; i++) {

        arr1[i] = i;

        arr2[i] = n - i;

    }

    start = clock();

    mergeSort(arr1, 0, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by merge sort (ascending): " << timeTaken << endl;

    start = clock();

    mergeSort(arr2, 0, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by merge sort (descending): " << timeTaken << endl

         << endl;

    return 0;

}

Output

root@Zonularity:/home/zonularity/dsa/lab\_12# ./task\_3

STL sort in ascending order: 0.000978

STL sort in descending order: 0.001333

Time taken by quick sort (ascending): 0.6456

Time taken by quick sort (descending): 0.454865

Time taken by merge sort (ascending): 0.001168

Time taken by merge sort (descending): 0.001143

Reason

The running time of **merge** sort remains the same for both ascending and descending order. This is because merge sort divides the array into two halves and then merges them back together. The time taken to merge the two halves is the same for both ascending and descending order.

**Quick** sort, on the other hand, shows a significant difference in running time for ascending and descending order. This is because quick sort picks the last element as the pivot and then partitions the array around the pivot. In the best case, the pivot is the median element, and the array is divided into two halves of equal size. In the worst case, the pivot is the smallest or largest element, and the array is divided into two halves of unequal size. This difference in partitioning causes the running time of quick sort to vary based on the structure of the input.