**Question 1**

**Given the following two functions find1() and find2() which are both passed an integer array, arrA[] , and its associated size, size, and return an integer. In addition, the function find1() is passed the value size - 1 for curr initially. (Line numbers are included):**

1. // curr should be size - 1 for the first call

2. int find1(int arrA[], int size, int curr)

3. {

4. if (size == 1)

5. {

6. return (curr);

7. }

8. else if (arrA[curr] < arrA[size - 2])

9. {

10. return (find1(arrA, size - 1, size - 2));

11. }

12. else

13. {

14. return (find1(arrA, size - 1, curr));

15. }

16. }

17. int find2(int arrA[], int size)

18. {

19. int curr = 0;

20. for (int i = 1; i < size; i++)

21. {

22. if (arrA[i] > arrA[curr])

23. {

24. curr = i;

25. }

26. }

27. return (curr);

28. }

**With respect to time complexity analysis, and assuming a worst-case scenario, calculate the number of timesteps of each function as a function of the array size N. Explain your approach and any assumptions, clearly showing how the timesteps are calculated.**

**For find1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line | Cost | numTimes | cost\*numTimes | Total |
| 4 | 1 | N | N |  |
| 6 | 1 | 1 | 1 |  |
| 8 | 1 | N-1 | N-1 |  |
| 10 | 1 | 0 | 0 |  |
| 14 | 1 | N-1 | N-1 |  |
|  |  |  |  | **f(N) = 3N-1** |

L4 will be executed N times, because in worst case scenario is when size will be equal to 0

L6 will be executed once when the condition will be true

L8 will execute N-1,because first if statement will not come true N-1 times

L10 will not be executed, because in worst case scenario size will be equal 1 and the condition in else if wont be true

L14 else statement will be executed N-1 times

In the worst case scenario the maximum element will be at index curr(N-1).

**For find2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Line | Cost | numTimes | cost\*numTimes | Total |
| 19 | 1 | 1 | 1 |  |
| 20 | 1 | N-1 | N-1 |  |
| 22 | 1 | N-1 | N-1 |  |
| 24 | 1 | 0 | 0 |  |
| 27 | 1 | 1 | 1 |  |
|  |  |  |  | **f(N) = 2N** |

I the worst case scenario L24 will not be executed, because maximum element will be at index curr = 0;

Question 2

The following function merge(), merges two sorted portions (from lb to mid and from mid+1 to ub) of an array arrA[]. (Line numbers are included).

L1. void merge(int arrA[], int lb, int mid, int ub)

L2. {

L3.     int i, j, k;

L4.     int size = ub - lb + 1;

L5.     int \*arrC = (int \*)calloc(size, sizeof(int));

L6.     for (i = lb, j = mid + 1, k = 0; i <= mid && j <= ub; k++)

L7.     {

L8.         if (arrA[i] <= arrA[j])

L9.         {

L10.             arrC[k] = arrA[i++];

L11.         }

L12.         else

L13.         {

L14.             arrC[k] = arrA[j++];

L15.         }

L16.     } // end for loop

L17.     while (i <= mid)

L18.     {

L19.         arrC[k++] = arrA[i++];

L20.     }

L21.     while (j <= ub)

L22.     {

L23.         arrC[k++] = arrA[j++];

L24.     }

L25.     for (i = lb, k = 0; i <= ub; i++, k++)

L26.     {

L27.         arrA[i] = arrC[k];

L28.     }

L29. }

Using some sample data, and with reference to the code line numbers, explain, in your own words, how the function merge() works.

In L3 variables **i**, **j** and **k** are being declared.

In **L4** size of the array is being calculated by subtracting lower band from upper band and adding one . It will be later used in **L4** to dynamically allocate memory to **arrC** of size that is calculated in L3.

In **L6** a loop is initialized starting from **i** set to **lb**, **j** set to **mid + 1**, and **k** set to **0**. This loop runs until either **i** reaches **mid** or **j** reaches **ub**, whichever comes first.

Inside the loop**(L8-L15)**, elements are being compared at indices **i** and **j** of **arrA[].** If the element at **arrA[i]** is less than or equal to the element at **arrA[j],** **arrA[i]** is being assignedto **arrC[k]** and **i** is incremented. Otherwise, **arrA[j]** is assignedto **arrC[k]** and increment **j**.

**(L17-L24)** After exiting the loop there is possibility that there are some remaining elements in either of the two portions. Two separate while loops are used to copy any remaining elements from either portion into arrC[].

**(L25-L28)** Then elements from **arrC** are being merged back into the original array staring from index **lb** up to **ub**

**Initialization:**

lb = 0

mid = 3

ub = 7

**Declaration and Memory Allocation:**

size = 7 - 0 + 1 = 8

Allocate memory for arrC[] of size 8.

**Merging Process:**

Initialize i, j, and k:

i = 0, j = mid + 1 = 4, k = 0

Iteration 1:

Compare arrA[0] (2) and arrA[4] (1) - false.

So, arrC[0] = arrA[4] = 1 and increment j.

Result: arrC[] = {1, \_, \_, \_, \_, \_, \_, \_}

Iteration 2:

Compare arrA[0] (2) and arrA[5] (3) - true.

So, arrC[1] = arrA[0] = 2 and increment i.

Result: arrC[] = {1, 2, \_, \_, \_, \_, \_, \_}

Iteration 3:

Compare arrA[1] (4) and arrA[5] (3) - false.

So, arrC[2] = arrA[5] = 3 and increment j.

Result: arrC[] = {1, 2, 3, \_, \_, \_, \_, \_}

Iteration 4:

Compare arrA[1] (4) and arrA[6] (5) - true.

So, arrC[3] = arrA[1] = 4 and increment i.

Result: arrC[] = {1, 2, 3, 4, \_, \_, \_, \_}

Iteration 5:

Compare arrA[2] (6) and arrA[6] (5) - false.

So, arrC[4] = arrA[6] = 5 and increment j.

Result: arrC[] = {1, 2, 3, 4, 5, \_, \_, \_}

Iteration 6:

Compare arrA[2] (6) and arrA[7] (7) - true.

So, arrC[5] = arrA[2] = 6 and increment i.

Result: arrC[] = {1, 2, 3, 4, 5, 6, \_, \_}

Iteration 7:

Compare arrA[3] (8) and arrA[7] (7) - false.

So, arrC[6] = arrA[7] = 7 and increment j.

Result: arrC[] = {1, 2, 3, 4, 5, 6, 7, \_}

**Copying Remaining Elements:**

No remaining elements in the first portion (i > mid), but there are elements remaining in the second portion.

Copy these remaining elements into arrC[].

**Copying Back to Original Array:**

Copy elements from arrC[] back to arrA[] starting from index lb to ub.

**After completing the merge() function, arrA[] will be {1, 2, 3, 4, 5, 6, 7, 8}.**

Question 3

Given the test file, file1.txt (with 10,000 integers), perform a comparison of the two sorting techniques Merge Sort and Quick Sort considering different values of N (array size) in terms of:

1. time taken
2. number of swaps/data moves
3. number of comparisons
4. number of function calls (recursive if using, and non-recursive)

Present your results in a meaningful and clear way so that it is easy to see differences between the algorithms for the same value of N. You may re-use your code from assignment 1.

In my testing approach I was evaluating 4 sorting algorithms. Testing is performed on array of different size ranging from 1000 to 10000 in increments of 1000, for each size. Numbers are read from given file.

Counting swaps, comparisons and function calls is executed in each soring algorithm. The time taken to sort the array is measured by recording the time before and after the sorting algorithm is executed, and then calculating the difference.

After sorting an array and gathering all data, the sorted array is checked to ensure that it is sorted correctly.

Results are being saved to .csv file.

Below I’ve included analysis of gathered data and parts of code. Full code and csv files are available in the GitHub repository on branch **Assignment-2-Submission**

<https://github.com/Olszewski-Jakub/Sorting-Algorithms-Performance-Analysis/tree/Assignment-2-Submission>

[Csv file with gathered data](https://github.com/Olszewski-Jakub/Sorting-Algorithms-Performance-Analysis/blob/Assignment-2-Submission/files/result/result_file1.csv)

[Sorting algorithms](https://github.com/Olszewski-Jakub/Sorting-Algorithms-Performance-Analysis/blob/Assignment-2-Submission/src/SortingAlgorithms.c)

[Algorithm execution and data gathering](https://github.com/Olszewski-Jakub/Sorting-Algorithms-Performance-Analysis/blob/Assignment-2-Submission/src/SortingOperations.c)

|  |  |  |
| --- | --- | --- |
| Row Labels | Merge Sort | Quick Sort |
| 1000 | 0 | 0 |
| 2000 | 0,001 | 0 |
| 3000 | 0,001 | 0 |
| 4000 | 0,003 | 0,001 |
| 5000 | 0,003 | 0,001 |
| 6000 | 0,002 | 0,002 |
| 7000 | 0,005 | 0,002 |
| 8000 | 0,005 | 0,002 |
| 9000 | 0,004 | 0,003 |
| 10000 | 0,006 | 0,003 |

Data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sorting Algorithm | Size | Swaps | Comparisons | Function Calls | Time Taken |
| Merge Sort | 1000 | 9976 | 8727 | 2998 | 0 |
| Merge Sort | 2000 | 21952 | 19445 | 5998 | 0,001 |
| Merge Sort | 3000 | 34904 | 30972 | 8998 | 0,001 |
| Merge Sort | 4000 | 47904 | 42886 | 11998 | 0,003 |
| Merge Sort | 5000 | 61808 | 55269 | 14998 | 0,003 |
| Merge Sort | 6000 | 75808 | 67903 | 17998 | 0,002 |
| Merge Sort | 7000 | 89808 | 80710 | 20998 | 0,005 |
| Merge Sort | 8000 | 103808 | 93732 | 23998 | 0,005 |
| Merge Sort | 9000 | 118616 | 107032 | 26998 | 0,004 |
| Merge Sort | 10000 | 133616 | 120530 | 29998 | 0,006 |
| Quick Sort | 1000 | 5060 | 11226 | 1990 | 0 |
| Quick Sort | 2000 | 12170 | 26550 | 4471 | 0 |
| Quick Sort | 3000 | 17744 | 43682 | 7258 | 0 |
| Quick Sort | 4000 | 26741 | 64342 | 10117 | 0,001 |
| Quick Sort | 5000 | 31774 | 88044 | 13054 | 0,001 |
| Quick Sort | 6000 | 40238 | 111284 | 16009 | 0,002 |
| Quick Sort | 7000 | 45251 | 137563 | 18937 | 0,002 |
| Quick Sort | 8000 | 49257 | 162389 | 21880 | 0,002 |
| Quick Sort | 9000 | 52462 | 185817 | 24775 | 0,003 |
| Quick Sort | 10000 | 69361 | 222955 | 27742 | 0,003 |

**Sorting Algorithms**

**A screenshot of a computer

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

I am aware of what plagiarism is and include this here to confirm that this work is my own

