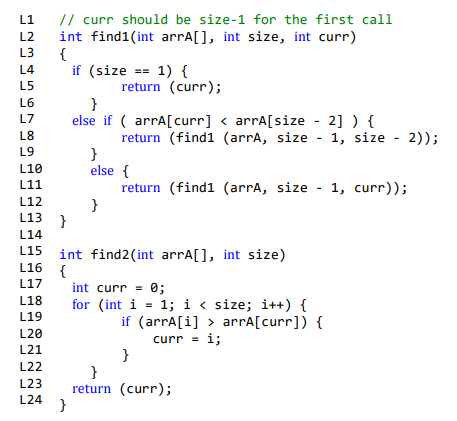
Question 1

i).

A recursive function is one that references themselves. A function that may call itself during its execution. This can lead to an infinite loop unless the recursive function is well defined. A well defined recursive function has a “base case,” and must “move towards” the base case (reduce). Basically the recursive function cannot be infinite.



FOR find1()

Base case: At the base case the function will not call itself again.

The base case check is at line 4. The base case return is on line 5. When size is 1 the function will return a value and no more recursive calls are made.

Reduction: Every call of the function will reduce it towards the base case.

This happens at Line 8 and line 11 because size is reduced by 1 when the recursive function is called again. This means eventually size will equal 1, making line 4 true, ending the recursive function.

ii).

find1()

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Cost | numTimes | Cost\*numTimes |
| L4 | 1 | n | n |
| L5 | 1 | 1 | 1 |
| L7 | 1 | n - 1 | n - 1 |
| L8 or L11 | 1 | n - 1 | n - 1 |
|  |  |  |  |
|  |  | Total: | f(n) = 3n - 1 |

L4: Always checked. True n - 1 times and false 1 time.

L5: Only happens once.

L7: Always checked, besides the last function call when L4 is true.

L8 and L11: One of them will run because of the “else”.

find2()

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Cost | numTimes | Cost\*numTimes |
| L17 | 1 | 1 | 1 |
| L18 | 1 | n | n |
| L19 | 1 | n - 1 | n - 1 |
| L20 | 1 | n - 1 | n - 1 |
| L23 | 1 | 1 | 1 |
|  |  |  |  |
|  |  | Total: | f(n) = 3n |

size = n.

L17: Happens once.

L18: True n - 1 times. Then false 1 time. So it runs n times.

L19: Checked n - 1 times.

L20: Worst case scenario, arrA[i] is always bigger than arrA[curr]. (sorted in ascending order). Therefore, L19 is always true, therefore L20 happens the same number of times as L19. So L20 runs n - 1 times.

L23: Happens once.

Question 2

Imagine we split the range of elements given in arrA into two sub arrays (lower bound -> mid AND mid+1 -> upper bound). (This is done in the declarations made in the for loop in L20). We are going to populate a new array, arrC, one element of arrC at a time using the 2 sub-arrays of arrA. A counter declared as “k” in the code given will keep track of the population of arrC. The left-most unused (not already placed into arrC) element for the left sub-array of arrA is compared to the left-most unused element for the right sub-array of arrA - the element with the lower value is placed into arrC. A counter is incremented to keep track of the left-most unused element for each of the sub-arrays - “i++” in L22 and “j++” in L24.

First loop…

arrA

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 6 | 3 | 5 | 7 | 8 |

^ last of left sub-array

Compare the red values.

arrC

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |  |

The lower value goes into arrC. And the counter for its sub-array increments. (i++).

… after last loop ( i <= mid is no longer true so the loop is exited).

arrA

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 6 | 3 | 5 | 7 | 8 |

^ last of left sub-array

arrC

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |  |  |

When one of the sub-arrays reaches its end, L20 will return false. This sub-array has been fully used, so the rest of the other sub-array must be placed into arrC (this is because the rest of the values in the other sub-array will all be bigger than the values of the sub-array that is at its end). Either L27 will run or L29 will run.

The rest of the sub-array is placed into arrC. In this example L28 is true 2 times. and L29 happens twice.

arrA

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 6 | 3 | 5 | 7 | 8 |

^ last of left sub-array

arrC

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Finally, place the elements of arrC into arrA, overwriting the unsorted section of arrA with the sorted array that has been calculated and stored in arrC. (L30 and L31).

arrA

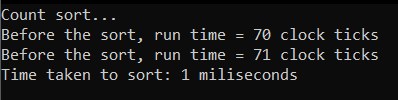
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 6 | 3 | 5 | 7 | 8 |

**document continues on next page…**

Question 3

a)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Count sort** | **Merge sort** | **Quick sort** |
| Recursive calls |  | 9 998 | 9 002 |
| Swaps |  | 61 808 | 34 636 |
| Comparisons |  | 55 296 | 84 507 |
| Time taken (ms) | 1 | 5 | 4 |



b)

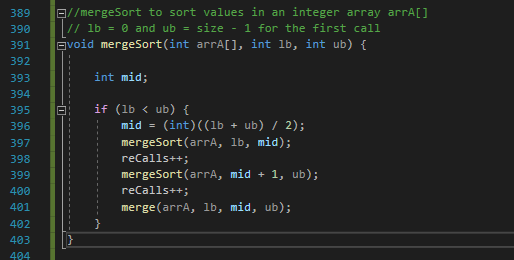
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Count sort** | **Merge sort** | **Quicksort** |
| **Advantages** | Fastest. | Works for general data. | Works for general data. |
| **Disadvantages** | Only works for integer data. | REQUIRES SPACE for the temporary array (arrC).  Slowest. | A bad pivot choice makes quicksort perform worse. |
| **Time taken (ms)** | **1** | **5** | **4** |

**Merge sort**

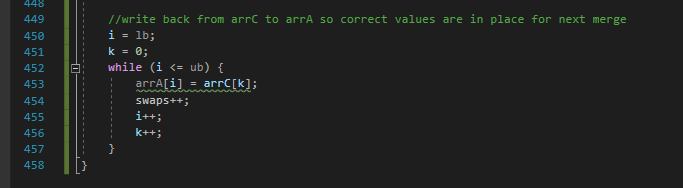
Text

Description automatically generated

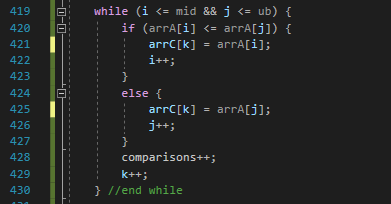
Recursive call counter increments in mergeSort() function. Line 398 and line 400.



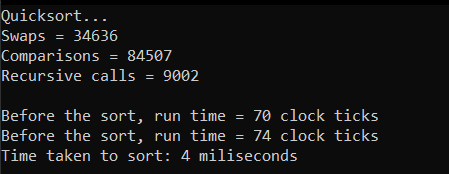
Swap counter increments in write-back section of code in merge() function. Line 454.



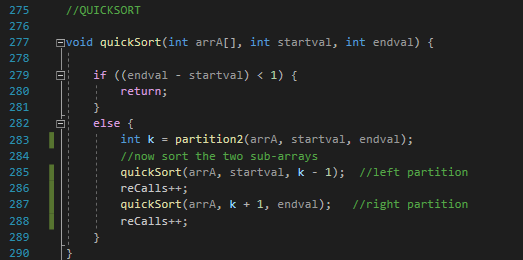
Comparisons counter increments in merge() function. Line 428.



**Quicksort**

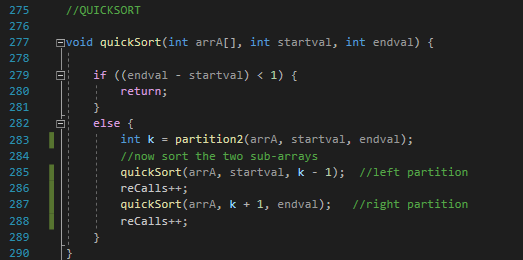


Recursive calls counter increments in quicksort() function. Line 286 and line 288.



Swap counter increments in partition2() function. Line 367 and line 373.

Comparisons counter increments in partition2() function. Line 370.



Question 4

Plagiarism Declaration: “I am aware of what plagiarism is and include this here to confirm that this work is my own.”