

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Name | Email | Group |
| 133722 | Salma Mohamed Yassin | [Salma133722@bue.edu.eg](mailto:Salma133722@bue.edu.eg) | A6 |
| 129128 | Nouran Moataz Mosallam | [Nouran129128@bue.edu.eg](mailto:Nouran129128@bue.edu.eg) | A5 |

Table of Contents:

1. Implementation of the merge sort and the quick sort algorithms 2

2. Complexity Explaining for each algorithm 7

3. Comparison between the two algorithms8

1. Actual number steps 8
2. Which algorithm generates the solution first8
3. Explanation of the complexity of each algorithm8

4. Why one algorithm is the better the other9

1. Implementation:
2. // Salma Mohamed Yassin – 133722
3. // Nouran Moataz Mosallam – 129128
4. #include<iostream>
5. #include <ctime>
6. #include <thread>
7. #include <iomanip>
9. using namespace std;
11. int \*Arr;                       // Initializing the dynamic array for the Merge Sort Function
12. int \*Arr2;                      // Initializing a copy of the dynamic array for the Quick Sort Function
13. int size;                       // An integer holding the dynamic arrays' size
15. int counterMerge = 0;           // Counts the Merge Sort steps
16. int counterQuick = 0;           // Counts the Quick Sort steps
18. clock\_t mergeStart;             // To holds the starting time of the merge sort function
19. clock\_t quickStart;             // To holds the starting time of the quick sort function
21. long double mergeTotal;             // to Calculates the duration of the mergeSort function
22. long double quickTotal;             // to Calculates the duration of the quickSort function
24. void display(int \*arr, int size, int sort){     */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
25. if (sort == 1)                              */\*    A function to  \*/*
26. cout << "**\n**Merge";                      */\* display the array \*/*
27. if (sort == 2)                              */\*      elements     \*/*
28. cout << "**\n**Quick";                      */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
30. cout << "Sorted: ";
31. for (int i = 0; i < size; i++){
32. cout << arr[i] << " ";
33. }
35. cout << endl;
37. if (sort == 1){
38. cout << "Merge Sort Counter: " << counterMerge;     // Display the number of steps in mergeSort
39. }
40. else if (sort == 2){
41. cout << "Quick Sort Counter: " << counterQuick;     // Display the number of steps in quickSort
42. }
43. cout << endl;
44. cout << endl;
46. }
48. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
49. */\* Merge Sort \*/*
50. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
52. void merge(int \*arr, int \*arrLeft, int \*arrRight, int sizeLeft, int sizeRight, int sizeArray){
54. int i = 0;                                  // i to manage the index of the temporary left array
55. int j = 0;                                  // j to manage the index of the temporary right array
56. int k = 0;                                  // k to manage the index of the real array
58. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
59. */\* Sorting and Merging \*/*
60. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
62. while (i < sizeLeft && j < sizeRight) {     // Loop untill the end of the Right array or the Left array is reached
63. counterMerge++;
65. if (arrLeft[i] < arrRight[j])           // If the number in the left array is less then the number in the right
66. arr[k++] = arrLeft[i++];            // put the number which was in the left array in the real array and increment both the real and the left arrays' indexes
68. else                                    // If the number in the right array is less then the number in the left
69. arr[k++] = arrRight[j++];           // put the number which was in the right array in the real array and increment both the real and the right arrays' indexes
70. }
72. while (i < sizeLeft) {                      // If the Left array still contains elements
73. arr[k++] = arrLeft[i++];                // put them in the real array
74. }
76. while (j < sizeRight) {                     // If the right array still contains elements
77. arr[k++] = arrRight[j++];               // put them in the real array
78. }
79. }
81. void mergeSort(int \*arr, int size){
82. int \*arrLeft;                               // Initializing two temporary dynamic
83. int \*arrRight;                              // arrays to hold the two arrays to be merged
85. int sizeLeft;                               // The two arrays' sizes
86. int sizeRight;
88. if (size > 1) {                             // checking if the size of the arrays divide bigger than 1
90. if (size % 2 == 0){                     // if the size is even divide both arrays into 2 equal parts
91. sizeLeft = size / 2;
92. sizeRight = size / 2;
93. }
94. else{                                   // else if the size is odd divide them so that the left array would have 1 more number than the right one
95. sizeLeft = (size / 2) + 1;
96. sizeRight = size / 2;
98. }
100. arrLeft = new int[sizeLeft];            // Give the Left array its size
101. arrRight = new int[sizeRight];          // Give the Right array its size
103. */\*\*\*\*\*\*\*\*\*\*\*\*/*
104. */\* Dividing \*/*
105. */\*\*\*\*\*\*\*\*\*\*\*\*/*
107. for (int i = 0; i < sizeLeft; i++){                        // Putting the first half of the numbers of the real array in the temporary left array
108. arrLeft[i] = arr[i];
109. }
111. int j = 0;                                                 // j works as an index number for the temporary Right array
113. for (int i = sizeLeft; i < size; i++){                     // Putting the second half of the numbers of the real array in the temporary right array
114. arrRight[j++] = arr[i];
115. }
117. */\*\*\*\*\*\*\*\*\*\*\*\*\*/*
118. */\* Recursion \*/*
119. */\*\*\*\*\*\*\*\*\*\*\*\*\*/*
121. mergeSort(arrLeft, sizeLeft);                              // Recursively call the Merge Sort function for the Left array untill the size is less than or equals 1
122. mergeSort(arrRight, sizeRight);                            // Recursively call the Merge Sort function for the Right array untill the size is less than or equals 1
123. merge(arr, arrLeft, arrRight, sizeLeft, sizeRight, size);  // Call the Merge function to sort and merge both halfs of the array
125. }
126. }
128. int MergeSort(){
130. mergeSort(Arr, size);                                         // Calling the mergeSort Function
132. mergeTotal = (clock() - mergeStart) / (double)CLOCKS\_PER\_SEC; // Calculating the mergeSort Run-time in seconds
134. cout << endl;
135. cout << "Merge Sort Completed!" << endl;
136. return 0;
137. }
139. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
140. */\* Quick Sort \*/*
141. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
143. void Swap(int &arr1, int &arr2){
145. int temp;                       */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
146. temp = arr1;                    */\*A function to swap \*/*
147. arr1 = arr2;                    */\*  any two integers \*/*
148. arr2 = temp;                    */\*   by refrence     \*/*
149. }                                   */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

152. int partition(int \*arr, int firstIndex, int lastIndex){     // The partition function takes the array that should be sorted, the first and last indexes of the elements of the array
154. int i = firstIndex - 1;                                 // An index that starts before the first element of the part that needs to be sorted
156. int pivot = arr[lastIndex];                             // The last element in the partition in which we compare the other element with
158. for (int j = firstIndex; j < lastIndex; j++){           // A loop that goes through all the elements in the partition
159. if (arr[j] <= pivot){                               // If the element of the array in index j is less than or equal the specified pivot
160. i++;                                            // i gets incremented
161. Swap(arr[i], arr[j]);                           // then the elemets of the array in index i and j gets swapped
162. counterQuick++;
163. }
164. }
166. Swap(arr[i + 1], arr[lastIndex]);                       // Swaps the pivot with the element at i+1 to put the pivot at its rightful place
167. return i + 1;                                           // Returns the new pivot's position to go through the following partitions
168. }
170. void quickSort(int \*arr, int firstIndex, int lastIndex){    // The quick sort function takes the array that should be sorted, the first and last indexes of the elemnts that needs to be sorted
172. if (firstIndex < lastIndex){                            // Checks if in the two sent indexes, the first is less than the last
174. int p = partition(arr, firstIndex, lastIndex);      // Calls partition function to sort this partition
176. quickSort(arr, firstIndex, p - 1);                  // Recursively calls itself in order to sort the first part of the array
177. quickSort(arr, p + 1, lastIndex);                   // Recursively calls itself in order to sort the second part of the array
178. }
180. }
182. int QuickSort(){
184. quickSort(Arr2, 0, size - 1);                                   // Calling the quickSort Function
186. quickTotal = (clock() - quickStart) / (double)CLOCKS\_PER\_SEC;   // Calculating the quickSort Run-time
188. cout << endl;
189. cout << "Quick Sort Completed!" << endl;
190. return 0;
192. }
194. int main(){
196. //ios\_base::sync\_with\_stdio(false); // to have the cout and cin run faster
198. cout << "Enter size of Array: ";
199. cin >> size;                        // Aquiring the size of the array from the user
201. cout << endl;
203. //srand(time(NULL)); //makes sure that numbers are not the same in each run
205. Arr = new int[size];                // Giving the dynamic array its size
206. Arr2 = new int[size];
208. for (int i = 0; i < size; i++){     // A loop to let the user enter Elements to be sorted
210. cout << "Enter number: ";
211. cin >> Arr[i];
212. //Arr[i] = rand() % 100; // generates random numbers;
213. }
215. cout << endl;
217. cout << "unsorted array :";
218. for (int i = 0; i < size; i++){
220. Arr2[i] = Arr[i];
221. cout << Arr2[i]<<" ";
222. }


226. mergeStart = clock();
227. thread Merge(MergeSort);            // A function for displaying the sorted numbers in the array by the mergeSort algorithm
228. quickStart = clock();
229. thread Quick(QuickSort);            // A function for displaying the sorted numbers in the array by the quickSort algrithm



234. Merge.join();
235. Quick.join();


239. display(Arr, size, 1);
240. display(Arr2, size, 2);
241. cout << "Time Elapsed for Merge: " << fixed << setprecision(8) << mergeTotal << "s" << endl;
242. cout << "Time Elapsed for Quick: " << fixed << setprecision(8) << quickTotal << "s" << endl;
244. //printf("Time Elapsed for Merge: %.22lf ms \n", mergeTotal\*1000); // Printing out the run-time of the program
245. //printf("Time Elapsed for Quick: %.22lf ms \n", quickTotal\*1000); // Printing out the run-time of the program
247. return 0;
248. }

2. Complexity:

Merge Sort:

Partition step is of number of steps: c

Recursive calls is of number of steps: 2\*F(n/2)

Merging is of number of steps: n

So the recurrence formula will be as follows:

F(n) = 2\*F(n/2) + cn

According to the master theorem

F(n) = aF(n/b) + nd then

Then a = 2, b =2, and d=1

Then (a = 2) = (bd=21=2) which is the second case : F( n ) = O(nd log2 n)

The F(n) = O("nd log2 n") = O("n1 log2 n") = O("n∗log2 n")

Worst- case performance: O(*n* log *n*)

Best-case performance: O(*n* log *n*) typical, O(*n*) natural variant

Average performance: O(*n* log *n*)

Quick Sort:

Best case has partitions are as evenly balanced as possible: their sizes either are equal of size ((n-1)/2) if the number of elements is odd or are within 1 of each other if the number of elements is even. So the recursive call on both partitions is of complexity 2\*c\*(n/2)=cn, then 22\*c(n/22)=cn, then 23\*c(n/23)=cn, and so on.

The algorithm formula is f(n)=2\*f(n/2)+cn

Worst case has the most unbalanced partitions possible. The sequence of n elements S(n) in this case is partitioned to {pivot, S(n-1)}. So the recursive call on S(n) is of complexity cn, and the recursive call on S(n-1) is of complexity c(n-1), and the recursive call on S(n-2) is of complexity c(n-2), and so on.

The algorithm formula is f(n)=f(n-1) + cn

Worst- case performance: O(*n*2)

Best-case performance: O(*n* log *n*)

Average performance: O(*n* log *n*)

3. Comparison between both algorithms:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithms | Number of Steps | Which generates first | Test Case |
| Merge Sort | According to our test case #1 the number of steps of the Merge Sort: 538 steps  The Merge sort is slower than the Quick sort in smaller test-cases  Worst-case: O(n\*log(n)) | According to our test case the duration of the run time of the merge sort function is 0.003s but the duration of the quick sort function takes less than 1/100000000s that the time is showing as zero seconds.  In agreement with tried out test cases; the Merge sort works best with very large test cases than the quick sort does | Test Case #1:  Enter size of Array: 100  unsorted array :88 74 27 97 98 57 81 88 38 4 79 83 72 33 77 59 66 6 96 37 95 27 36 42 75 32 58 99 28 10 81 96 24 65 87 83 21 4 77 71 74 48 67 25 16 18 64 1 79 64 6 30 87 66 36 8 65 93 57 6 30 15 65 31 77 0 57 59 27 49 99 6 12 35 74 69 65 30 91 98 70 39 49 29 49 19 43 2 62 32 91 49 55 33 33 0 31 21 12 32  Quick Sort Completed!  Merge Sort Completed!  MergeSorted: 0 0 1 2 4 4 6 6 6 6 8 10 12 12 15 16 18 19 21 21 24 25 27 27 27 28 29 30 30 30 31 31 32 32 32 33 33 33 35 36 36 37 38 39 42 43 48 49 49 49 49 55 57 57 57 58 59 59 62 64 64 65 65 65 65 66 66 67 69 70 71 72 74 74 74 75 77 77 77 79 79 81 81 83 83 87 87 88 88 91 91 93 95 96 96 97 98 98 99 99  Merge Sort Counter: 538  QuickSorted: 0 0 1 2 4 4 6 6 6 6 8 10 12 12 15 16 18 19 21 21 24 25 27 27 27 28 29 30 30 30 31 31 32 32 32 33 33 33 35 36 36 37 38 39 42 43 48 49 49 49 49 55 57 57 57 58 59 59 62 64 64 65 65 65 65 66 66 67 69 70 71 72 74 74 74 75 77 77 77 79 79 81 81 83 83 87 87 88 88 91 91 93 95 96 96 97 98 98 99 99  Quick Sort Counter: 361  Time Elapsed for Merge: 0.00200000s  Time Elapsed for Quick: 0.00100000s |
| Quick Sort | Consequently the number of steps of the quick sort function: 361 steps  The quicksort is very fast with smaller cases  Worst case: O(n2)  Which is so much worse than the merge sort’s worst case | According to our test cases the quick sort  Quick sort is typically faster than merge sort. However, when the data set is huge or if the array is already sorted, merge sort is the clear winner in terms of speed. | Test Case #2:  Enter size of Array: 50  unsorted array: 3 14 6 42 94 23 60 58 83 80 64 85 56 47 42 80 71 17 47 6 53 84 59 19 60 16 86 48 6 36 17 28 62 67 90 23 19 22 41 57 31 39 37 85 6 18 26 26 2 75  Quick Sort Completed!  Merge Sort Completed!  MergeSorted: 2 3 6 6 6 6 14 16 17 17 18 19 19 22 23 23 26 26 28 31 36 37 39 41 42 42 47 47 48 53 56 57 58 59 60 60 62 64 67 71 75 80 80 83 84 85 85 86 90 94  Merge Sort Counter: 229  QuickSorted: 2 3 6 6 6 6 14 16 17 17 18 19 19 22 23 23 26 26 28 31 36 37 39 41 42 42 47 47 48 53 56 57 58 59 60 60 62 64 67 71 75 80 80 83 84 85 85 86 90 94  Quick Sort Counter: 167  Time Elapsed for Merge: 0.00200000s  Time Elapsed for Quick: 0.00100000s |

4. Why is one algorithm is better than the other?

The quick sort works best when cases are small or needs lots of sorting, as it takes much less time and steps than the merge sort. However, the merge sort works best with very large cases as it will take less time and steps than the quick sort.