Assignment 4 - Sort Efficiency

Start Assignment

* **Due** Monday by 11:59pm

* **Points** 100

* **Submitting** a file upload

* **File Types** zip

* **Available** until Oct 3 at 11:59pm

For this assignment, we’ll look at the efficiency for several sorting algorithms.

*Suggestion: there is a fair amount of work involved in this assignment.  I suggest that you do the four parts of this assignment over the course of several days.*

**Part 1: O(n2) sort**

Pick a O(n2) sorting algorithm (bubble, insertion, or selection) and implement it in a C++ program to sort an array of integers.  There is C++ code in the textbook and online for these algorithms, but I encourage you to resist the temptation to simply copy it.  Add a counter to the sort function to track the number of times that the function accesses elements from the array you are sorting.  For example, if your function has the line **array[i] = max** that counts as one array access.  If your function has the line**if (array[i] < array[i+1])** that counts as two array accesses.

**Part 2: O(n log n) sort**

Do the same thing as part 1, but with a O(n log n) algorithm (merge sort or quick sort).

**Part 3: Counting sort**

Do the same thing as part 1, but with the **counting sort**, as described here.

The counting sort can sort an array of integers that are known to be in a given range of integers by using a second array count to count the number of occurrences of each integer in the array.

As an example, let's say we have an array called scores containing the following ten integers in the range between 0 and 4, inclusive:

scores[] : 1 4 0 4 0 0 3 4 0 3

In this case, the count array would have five elements, one for each possible value 0 through 4.  We initialize count to all zeros.  We then make one pass through scores -- each time we see a given value x in scores, we increment the value in count[x].   When we are done, count[x] will tell us how many instances of the value x we have in scores.  In this example, count would be:

count[] : 4 1 0 2 3

because there are 4 zeros, 1 one, 0 twos, 2 threes, 3 fours in scores.

Once you have this array count, it is straightforward to determine the sorted scores array:

scores[] : 0 0 0 0 1 3 3 4 4 4

and store it back into the original array.

Write a C++ function that implements the counting sort, assuming an array of integers that are in the range 0 to 100, inclusive.

In big O notation, what is the expected efficiency?  Why is counting sort impractical as a general sorting algorithm?

**Part 4: Gather efficiency data**

Now, gather some data on how many array accesses these sorts require.  Write a C++ program which declares int arrays of various lengths given in the table below, and then fills the arrays with random ints between 0 and 100.  Count the number of array accesses your sort algorithms from part 1, 2, and 3 above require to sort each array.  In order to make sure you didn't hit a best case or worst case value, do at least 3 trials.  In other words, randomize the array at least 3 times for each array length and run your sort and record the number of array accesses made.  Compute the average number of array accesses for sorting each array length.  For part 3, counting sort, you do not need to count accesses of the count array, just the array that you are sorting.

When you're finished, you should have a table like the one below, where you’ve filled in the number of array accesses for each pass and the average number of array accesses for your chosen sort algorithms.  Observe the differences in growth between the algorithms as your n gets larger.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sort Algorithm | Array Length | Array Accesses  Pass 1 | Array Accesses  Pass 2 | Array Accesses  Pass 3 | Average Array Accesses |
| Bubble | 8 |  |  |  |  |
| Bubble | 32 |  |  |  |  |
| Bubble | 128 |  |  |  |  |
| Merge | 8 |  |  |  |  |
| Merge | 32 |  |  |  |  |
| Merge | 128 |  |  |  |  |
| Counting | 8 |  |  |  |  |
| Counting | 32 |  |  |  |  |
| Counting | 128 |  |  |  |  |

**Submission**

Submit your implementation of all three sorting algorithms with sample output demonstrating your sorts for each pass in the table.  This can be all in one .cpp file, or multiple .cpp files.  Additionally, submit a plaintext (.txt) or pdf (.pdf) document that has the table containing your data and also the answers to the questions in part 3, the counting sort.

Make one zip file containing the .cpp file(s) and the .txt (or .pdf) file and submit.

cout << "Bubble array length of 8: " << bubbleSortAccessCount << endl; int mergeSortInput[] = {1, 4, 5, 4, 3, 2, 3};

int mergeSortAccessCount = mergeSort(mergeSortInput, 7);

cout << "access count for merge sort was: " << mergeSortAccessCount << endl;

int \*countSortInput = makeRandomArray(8, true);

countingSort(countSortInput, 8);

cout << "Is Sorted: " << isSorted(countSortInput, 8) << endl;

delete[] countSortInput;

int bubbleSortInput[] = {5, 4, 3, 2, 1};

int bubbleSortAccessCount = bubbleSort(bubbleSortInput, 5);

cout << "access count for bubble sort was: " << bubbleSortAccessCount << endl;

int countingSortInput[] = {5, 4, 3, 2, 1};

int countingSortAccessCount = countingSort(countingSortInput, 5);

cout << "access count for counting sort was: " << countingSortAccessCount << endl;

int \*bubbleOneData = makeRandomArray(8, true);

bubbleSortAccessCount = bubbleSort(bubbleOneData, 8);

delete[] bubbleOneData;