THE1

Available from: Friday, November 5, 2021, 11:59 AM
Due date: Friday, November 5, 2021, 11:59 PM
Requested files: the1.cpp, test.cpp (Download)
Type of work: Individual work

Specifications:

- There are **2 tasks** to be solved in **12 hours** in this take home exam.
- You will implement your solutions in **the1.cpp** file.
- You are free to add other functions to **the1.cpp**
- Do **not** change the first line of **the1.cpp**, which is **#include "the1.h"**
- Do **not** change the arguments and return value of the functions **crossMergeSort()** and **sillySort()** in the file **the1.cpp**
- Do **not** include any other library or write include anywhere in your **the1.cpp** file (not even in comments).

- You are given a test.cpp file to **test** your work on **Odtuclass** or your **locale**. You can and you are encouraged to modify this file to add different test cases.
- If you want to **test** your work and see your outputs you can **compile** your work on your locale as:

```
>g++ test.cpp the1.cpp -Wall -std=c++11 -o test
> ./test
```

- You can test your **the1.cpp** on virtual lab environment. If you click **run**, your function will be compiled and executed with test.cpp. If you click **evaluate**, you will get a feedback for your current work and your work will be **temporarly** graded for **limited** number of inputs.
- The grade you see in lab is **not** your final grade, your code will be reevaluated with **different** inputs after the exam.

The system has the following limits:

- a maximum execution time of 1 minute (your functions should return in less than 1 seconds for the largest inputs)
- a 256 MB maximum memory limit
- a stack size of 64 MB for function calls (ie. recursive solutions)

- Each task has a complexity constraint explained in respective sections.
- Solutions with longer running times will not be graded.

- If you are sure that your solution works in the expected complexity constrains but your evaluation fails due to limits in the lab environment, the constant factors may be the problem.
- If you solution is correct, the time and memory limits may be adjusted to accept your solution after the lab. Please send an email if that is the case for you.

```
int sillySort(int* arr, long &comparison, long & swap, int size);
int crossMergeSort(int *arr, long &comparison, int size);
```

In this exam, you are asked to complete the function definitions to sort the given array $**arr**$ with **ascending** order.

- **sillySort()** should count the number of $**comparison**$ and $**swap**$ executed during sorting proccess (Comparisons are only between the values to be sorted only, not your auxiliary comparisons) and return the number of calls of **sillySort()** (which is 1 in minimum).
- **crossMergeSort()** should count the number of $**comparison**$ executed during sorting proccess (Comparisons are only between the values to be sorted only, not your auxiliary comparisons) and return the number of calls of **crossMergeSort()** (which is 1 in minimum).

Silly sorting algortihm (**sillySort()**) is as follows:

- assume the input array A[1..N] is divided into 4 quarters as q1=A[1..N/4], q2=A[N/4+1..N/2], q3=A[N/2+1..3N/4], q4=A[3N/4+1..N]
- do 6 recursive calls as follows when N>=4 otherwise sort the list with N<4 elements directly.

    1. sillysort: q1 and q2 (sillysort A[1..N/2])
    2. sillysort: q2 and q3 (sillysort A[N/4+1..3N/4])
    3. sillysort: q3 and q4 (sillysort A[N/2+1..N])
    4. sillysort: q1 and q2
    5. sillysort: q2 and q3
    6. sillysort: q1 and q2

- when the input size N<=2 no recursion. (do nothing for N=0 or 1 and just apply swap when N =2)
- to make things simpler we will only use N as a power of 2 on our tests (although not necessary with non-rec termination conditions).
- It is an in-place algorithm, so no merging is needed. Nothing else is needed after the recursive calls.
- count the swap between any 2 elements of the array A, such as swapping A[i] and A[j].
- count the comparison between any 2 elements of the array A, such as A[i]>A[j]
- return the total number of calls to **sillySort**()

Cross merge sort (**crossMergeSort()**) is a variation of k-way merge sort, where k is 4 and the partitions are merged in a different order:

- Assume the input array has N elements which is a power of 2. If the input array A[1..N] has more than or equal to 4 elements it is divied into 4 quarters as q1=A[1..N/4], q2=A[N/4+1..N/2], q3=A[N/2+1..3N/4], q4=A[3N/4+1..N]
- do 4 recursive calls as follows:

    1. cross merge sort q1
    2. cross merge sort q2
    3. cross merge sort q3
    4. cross merge sort q4

Then,

- merge q1 and q3 into h1
- merge q2 and q4 into h2

- merge h1 and h2 into resulting array

- If the input array has exactly 2 elements, just compare these elements and swap if necessary.
- If the input array has exactly 1 element, do nothing.

- You can use the following pseudocode as a base for merge function:

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

- Note that the pseudocode reads sequential data, but in our approach will merge data that are seperated from each other. Thus, the function needs modification.
- In case of equality, pick the element from leftside array(i.e. choose from q1, q2, and h1).

- Hint: when merging 2 arrays with length n, there can be minimum n comparisons and maximum 2n-1 comparisons.
- count the comparison between any 2 elements of the array A, such as A[i]>A[j]
- return the total number of calls to **crossMergeSort()**

Constraints:

- Maximum array size is 2^11 for **sillySort()** and 2^16 for **crossMergeSort()**.

Evaluation:

- After your exam, black box evaluation will be carried out. You will get full points if you fill the $**arr**$ variable as stated and return the number of comparisons, function calls and swaps correctly for the cases that will be tested. **sillySort()** and **crossMergeSort()** are 50 points each.

Example IO:

```
1)
initial array = {-1, -3} size=2
sorted array = {-3, 1}
for crossMergeSort; num_of_calls=1, comparison=1
for sillySort; num_of_calls=1, comparison=1, swap=1
2)
initial array = {1, 2, 3, 4} size=4
sorted array = {1, 2, 3, 4}
for crossMergeSort; num_of_calls=5, comparison=5
for sillySort; num_of_calls=7, comparison=6, swap=0
3)
initial array = {7, 7, 7, 7} size=4
sorted array = {7, 7, 7, 7}
for crossMergeSort; num_of_calls=5, comparison=4
for sillySort; num_of_calls=7, comparison=6, swap=0
4)
initial array = {0, -5, 2, 6, 4, 18, 22, -14} size=8
sorted array = {-14, -5, 0, 2, 4, 6, 18, 22}
for crossMergeSort; num_of_calls=5, comparison=16
for sillySort; num_of_calls=43, comparison=36, swap=9
```