THE2

Available from: Friday, November 12, 2021, 11:59 AM
Due date: Friday, November 12, 2021, 11:59 PM
Requested files: the2.cpp, test.cpp (Download)
Type of work: Individual work

Specifications:

- There are 3 **tasks** to be solved in **12 hours** in this take home exam.
- You will implement your solutions in **the2.cpp** file.
- You are free to add other functions to **the2.cpp**
- Do **not** change the first line of **the2.cpp**, which is **#include "the2.h"**
- Do **not** change the arguments and return value of the functions **quickSort()** and **quickSort3()** in the file **the2.cpp**
- Do **not** include any other library or write include anywhere in your **the2.cpp** file (not even in comments).

- You are given **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 the2.cpp -Wall -std=c++11 -o test

> ./test
```

- You can test your **the2.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 **completely different** inputs after the exam.

The system has the following limits:

- a maximum execution time of 32 seconds (your functions should return in less than 1 seconds for the largest inputs)
- a 192 MB maximum memory limit
- an execution file size of 1M.

- 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.

```
void quickSort(unsigned short* arr, long &swap, double & avg_dist, double & max_dist, bool hoare, int size);
void quickSort3(unsigned short *arr, long &swap, long &comparison, int size);
```

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

- *Quicksort with Classical Partitioning*[30pts] is called using the function **quickSort()** with $**hoare**$=**false**. It should sort the array in **descending** order, count the number of $**swap**$ executed during sorting process, calculate the average distance between swap positions $**avg_dist**$, find the max distance between swap positions $**max_dist**$(which are all 0 if there are no swaps).
- *Quicksort with Hoare Partitioning*[30pts] is called using the function **quickSort()** with $**hoare**$=**true**. It should sort the array in **descending** order, count the number of $**swap**$ executed during sorting process, calculate the average distance between swap positions $**avg_dist**$, find the max distance between swap positions $**max_dist**$.
- *3-Way Quicksort*[40pts] is called using the function **quickSort3()**. It should sort the array in **descending** order, count the number of $**swap**$ executed during sorting process and count the number of $**comparison**$ executed during sorting proccess (Comparisons are only between the values to be sorted only, not your auxiliary comparisons) (which are all 0 if there are no swaps and comparisons).

For all 3 tasks follow these pseudocodes exactly:

```
# PSEUDOCODE FOR QUICKSORT WITH CLASSICAL PARTITIONING
PARTITION(arr[0:size-1])

    X←arr[size-1]
    i←-1
    for j←0 to size-2                              // The last element excluded
        do if arr[j]≥x
            then i←i+1
                 swap arr[i]↔arr[j]
    swap arr[i+1]↔arr[size-1]
    return i+1

QUICKSORT-CLASSICAL(arr[0:size-1])

    if size>1
        then P←PARTITION(arr[0:size-1])
             QUICKSORT-CLASSICAL(arr[0:P-1])       //P is excluded on recursive calls
             QUICKSORT-CLASSICAL(arr[P+1:size-1])
```

```
# PSEUDOCODE FOR QUICKSORT WITH HOARE PARTITIONING
HOARE(arr[0:size-1])

    X←arr[floor((size-1)/2)]       // i.e. 1 when size=3,4 ---- 2 when size=5,6
    i←-1
    j←size
    while True
        do  repeat j←j-1
                until arr[j]≥x
            repeat i←i+1
                until arr[i]≤x
            if i<j
                then swap arr[i]↔arr[j]
                else return j

QUICKSORT-HOARE(arr[0:size-1])

    if size>1
        then P←HOARE(arr[0:size-1])
             QUICKSORT-HOARE(arr[0:P])           //P is now included
             QUICKSORT-HOARE(arr[P+1:size-1])
```

```
# PSEUDOCODE FOR 3WAY QUICKSORT
PARTITION-3WAY(arr[0:size-1])
    i←0
    j←0
    p←size-1
    while i<p
        do if arr[i]>arr[size-1]
            then swap arr[i]↔arr[j]
                 i←i+1
                 j←j+1
            else if arr[i]=arr[size-1]
                    then p←p-1
                         swap arr[i]↔arr[p]
                    else i←i+1
    m=min(p-j,size-p)
    swap arr[j:j+m-1]↔arr[size-m:size-1]         //swap m elements, increment swap count by m
    L←j
    R←p-j

QUICKSORT-3WAY(arr[0:size-1])

    if size>1
        then (L,R)←PARTITION-3WAY(arr[0:size-1])
             QUICKSORT-3WAY(arr[0:L-1])          //We now exclude equal pivots in the middle
             QUICKSORT-3WAY(arr[size-R:size-1])
```

- Note that the algorithms are all in **descending** order.
- We expect quicksort with classical partitioning to be negatively affected if there are many equal elements, and 3-way quicksort to affected positively for the same condition.
- You may notice that there will be swaps which both sides are pointed by the **same** indexes. You do not need to handle anything. Just like other swaps, apply the swap, increment your swap variable and update your average distance.

Constraints:

- Maximum array size differs according to the function to be used and the interval. See **test.cpp** for more details.

Evaluation:

- After your exam, black box evaluation will be carried out. You will get full points if you set all the variables as stated.

Example IO:

1)

initial array = {0, 3} size=2

sorted array = {3, 0}

for quicksort with classical partitioning; swap=1, avg_dist=1, max_dist=1
for quicksort with hoare partitioning; swap=1, avg_dist=1, max_dist=1

for 3way quicksort; swap=1, comparison=2

2)

initial array = {4, 3, 2, 1} size=4

sorted array = {4, 3, 2, 1}

for quicksort with classical partitioning; swap=9, avg_dist=0, max_dist=0
for quicksort with hoare partitioning; swap=0, avg_dist=0, max_dist=0

for 3way quicksort; swap=6, comparison=6

3)

initial array = {18, 18, 18, 18} size=4

sorted array = {18, 18, 18, 18}

for quicksort with classical partitioning; swap=9, avg_dist=0, max_dist=0
for quicksort with hoare partitioning; swap=4, avg_dist=1.5, max_dist=3

for 3way quicksort; swap=3, comparison=6

4)

initial array = {2, 1, 14, 6, 3, 0, 99, 3} size=8

sorted array = {99, 14, 6, 3, 3, 2, 1, 0}

for quicksort with classical partitioning; swap=11, avg_dist=1.81818, max_dist=3
for quicksort with hoare partitioning; swap=7, avg_dist=2.14286, max_dist=6

for 3way quicksort; swap=10, comparison=22

TEST EVALUATION:

Due to the limitation of our programming environment, larger inputs can not be stored. Therefore, we create them when needed. The test evaluation has 2 phases. The first phase has the same inputs given here to check if your codes work fully correct on small inputs. If your code works perfectly on at least one of three tasks, it will also be tested on the second phase for the task(s) that works correct. The second phase on the other hand, creates and sorts larger arrays that are on boundaries. (Note that the tests give 50 pts for each phase. However, the real inputs will be like the ones on the second phase which means if your code works only on phase 1, it is possible for your real grade to be 0 afterwards).