[CENG 315 All Sections] Algorithms

Description Submission view

THE2

Available from: Friday, November 4, 2022, 12:00 PM
Due date: Friday, November 4, 2022, 11:59 PM
Requested files: the2.cpp, test.cpp, input01.txt, input02.txt, input03.txt, input04.txt, outsolm01.txt, outsolm02.txt, outsolm03.txt, outsolm04.txt, the2_solution.cpp (Download)
Maximum number of files: 10
Type of work: Individual work

Specifications:

- There is 1 **task** 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 **multiDigitRadixSort()** 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
- 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.

```
 long multiDigitRadixSort(int* arr, bool ascending, int arraySize, int groupSize, int
maxDigitLength);
```

In this exam, you are asked to complete the function definitions to sort the given array **arr** with **descending or ascending** order with respect to given boolean parameter **ascending** by using **multiDigitRadixSort**.

multiDigitRadixSort is a derivative of radix sort that uses multi digit counting sort as a stable sort function. In each counting sort, you should use the number of digits given by **groupSize** parameter.

For example, arr[4] = {4367, 1234, 7890}. If groupSize is 1, then the function is exactly equal to classical radix sort. That means, the first counting sort will use the last digits of the numbers which are 7, 4 and 0. If groupSize is equal to 2, then the first counting sort will use the last two digits of the numbers which are 67, 34 and 90.

If groupSize is not a multiplier of maxDigitLength, you should use maximum remaining digit count in counting sort function. For example, groupSize is 3 and maximum number in the array is 65789 , that means maxDigitLength is 5. In the first counting sort, you should use least significant 3 digits, in the second counting sort you should use most significant 2 digits.

You should return the number of iterations during counting sort.(The counting sort algorithm in our book has 4 different loops. Different than the algorithm for Counting Sort in your book, initialize count array as int C[k] = {0} and use the fourth loop for copying the array back. Otherwise the return value of the function will not be evaluated as correct.)

Do not forget to initialize your count array to zero. Otherwise, you may see some unstable behaviours.

Inputs:

- **bool ascending:** If ascending is true, sort the array with ascending order. Otherwise, sort the array with descending order.
- **int arraySize:** The number of elements in the array
- **int groupSize:** The number of digits that each counting sort uses
- **int maxDigitLength:** Maximum number of digits that any element in the array has

Constraints:

- Maximum array size is 100000.
- Array elements are positive integers.
- The number of digits can be different in each element.
- Maximum groupSize is 10.

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 iterations correctly.

Example IO:

You can find example I/O files under your working directory. The example files is exactly same as the ones used in evaluation function. Input parser function is given to you in test.cpp file.

The first line of the output file is iteration number. The second line is array size and rest is sorted array. You can test your function on your locale by using these input and output files.

Requested files

the2.cpp

```cpp
#include "the2.h"
#include <cmath>
// You may write your own helper functions here

long multiDigitRadixSort(int* arr, bool ascending, int arraySize, int groupSize, int maxDigitLength){
    long numberOfIterations = 0;

    return numberOfIterations;
}
```

test.cpp

```cpp
//This file is entirely for your test purposes.
//This will not be evaluated, you can change it and experiment with it as you want.
#include <iostream>
#include <fstream>
#include <random>
#include <ctime>
#include "the2.h"

// the2.h only contains declaration of the function:
// int multiDigitRadixSort(int* arr, bool ascending, int arraySize, int groupSize, int maxDigitLength)

using namespace std;

void randomFill(int*& arr, int size, int minval, int interval){
    arr = new int [size];
    for (int i = 0; i < size; i++)
    {
        arr[i] = minval + (random() % interval);
    }
}

void print_to_file(int* arr, int size, long numberOfIterations){
    ofstream ofile;
    ofile.open("outsol01.txt");
    ofile << numberOfIterations << endl;
    ofile << size << endl;
    for(int i = 0; i < size; i++)
        ofile << arr[i] << endl;
}

void read_from_file(int*& arr, int& size, bool& ascending, int& groupSize, int& maxDigitLength){

    char addr[]= "input01.txt";
    ifstream infile (addr);
    if (!infile.is_open())
    {
        cout << "File \'"<< addr
            << "\' can not be opened. Make sure that this file exists." <<endl;
        return;
    }
  infile >> ascending;
  infile >> groupSize;
  infile >> maxDigitLength;
  infile >> size;
    arr = new int [size];

    for (int i=0; i<size;i++) {

        infile >> arr[i];
    }
}


void test(int* arr, bool ascending, int groupSize, int maxDigitLength, int array_size){

    clock_t begin, end;
    double duration;
    long iterations=0;
    // Print initial array
    cout << "Array size: " << array_size << endl
         << "Group size: : " << groupSize << endl
         << "Max Digit Length: " << maxDigitLength << endl
         << "Ascending: ";
    if(ascending){
        cout << "True" << endl << endl;
    }else{
        cout << "False" << endl << endl;
    }
    cout << "Initial Array: {";
    for(int i=0; i<array_size; i++){
        cout << arr[i];
        if(i != array_size-1) cout << ", ";
    }
    cout << "}" << endl;

    // Function call and and calculate the duration
    if ((begin = clock() ) ==-1)
        cerr << "clock error" << endl;

    iterations = multiDigitRadixSort(arr, ascending, array_size, groupSize, maxDigitLength);

    if ((end = clock() ) ==-1)
        cerr << "clock error" << endl;


    cout << "Sorted Array: {";
    for(int i=0; i<array_size; i++){
        cout << arr[i];
        if(i != array_size-1) cout << ", ";
    }
```

```
    cout << "}" << endl << endl;

    duration = ((double) end - begin) / CLOCKS_PER_SEC;
    cout << "Duration: " << duration << " seconds." <<endl;
    cout << "Number of iterations: " << iterations << endl;
    print_to_file(arr, array_size, iterations);
    // Calculation and output end
}

int main(){
    int size = 5;
    int groupSize = 1;
    int maxDigitLength = 3;
    int minval = 0;
    int interval = 900;
    int *arr;
    bool ascending = true;
    // Randomly generate initial array:
    srandom(time(0));
    randomFill(arr, size, minval, interval);
    // Read the test inputs. input01.txt through input04.txt exists.
    // read_from_file(arr, size, ascending, groupSize, maxDigitLength);
    test(arr, ascending, groupSize, maxDigitLength, size);
    cout << endl;
    return 0;
}

```

input01.txt

```
1
1
3
5
901
119
459
888
765
```

input02.txt

```
0
2
6
7
345678
784532
121212
347965
999870
431111
900000

```

input03.txt

```
1
3
3
5
123
987
654
98
5

```

input04.txt

```
0
3
5
6
45678
45
987
3423
1
8888

```

outsolm01.txt

```
72
5
119
459
765
888
901

```

outsolm02.txt

```
360
7
999870
900000
784532
431111
347965
345678
121212

```

outsolm03.txt

```
1014
5
5
98
123
654
987

```

outsolm04.txt

```
1134
6
45678
8888
3423
987
45
1

```

the2_solution.cpp

```cpp
#include <cmath>
#include <iostream>
#include "the2.h"


int* getSubInt(int* arr, int arraySize, int startIndex, int subLength){
    int* subInt = new int[arraySize];
    for(int i=0; i < arraySize; i++){
        subInt[i] = (arr[i] % int(pow(10, startIndex))) / pow(10, startIndex-subLength) ;
        // std::cout << "Digit: " << i << ": "<< subInt[i] << std::endl;
    }
    return subInt;
}

int countingSort(int* arr, int arraySize, int startIndex, int subLength){
    int* result = new int[arraySize];
    for(int i = 0; i < arraySize; i++) result[i] = 0;
    int numberOfIterations = 0;

    int* indices = getSubInt(arr, arraySize, startIndex, subLength);

    int k = (int) pow(10, subLength);

    int* counts = new int[k];
    for(int i = 0; i < k; i++) counts[i] = 0;


    // filling the counts
    for(int i = 0; i < arraySize; i++){
        counts[indices[i]] += 1;
        numberOfIterations++;
    }

    // processing the counts
    for(int i = 1; i < k; i++){
        counts[i] = counts[i] + counts[i-1];
        numberOfIterations++;
    }

    // placing the result
    int j = arraySize;
    while (j > 0){
        result[counts[indices[j-1]]] = arr[j-1];
        counts[indices[j-1]]--;
        numberOfIterations++;
        j--;
    }

    // copying back to original array
    for(int i = 0; i < arraySize; i++){
        arr[i] = result[i+1];
        numberOfIterations++;
    }

    delete []result;
    delete []counts;

    return numberOfIterations;
}


long multiDigitRadixSort(int* arr, bool ascending, int arraySize, int groupSize, int maxDigitLength){
    int* temp = new int[arraySize];

    for(int i = 0; i < arraySize; i++){
        temp[i] = arr[i];
    }

    int startIndex = groupSize;
    int remainder = maxDigitLength % groupSize;
    int numberOfIterations = 0;
    int pass = 1;
    while (startIndex <= maxDigitLength){
        numberOfIterations += countingSort(temp, arraySize, startIndex, groupSize);
        startIndex += groupSize;
        pass++;
    }

    if (remainder){
        numberOfIterations += countingSort(temp, arraySize, maxDigitLength, remainder);
    }

    for(int i = 0; i < arraySize; i++){
        if (ascending){
            arr[i] = temp[i];
        }else{
            arr[arraySize-(i+1)] = temp[i];
        }
    }

```

```
91        return numberOfIterations;
92    }
93
```

VPL