Middle East Technical University | Department of Computer Engineering

CENG 315

Algorithms

Fall 2024-2025

Take-Home Exam 2

Due date: 3 November 2024, Sunday, 23.59

1 Problem Definition

You have been playing a word game using the Scrabble letter tiles. At each turn, every player picks a bunch of tiles from the bag, then lays down the tiles on the table to create a word. After everyone has laid down their words, the player having the turn to play tries to sort the words alphabetically as quick as possible.

Since you and your friends have just learned about RadixSort, you decided to add more structure to the game. Using the idea of multi-digit RadixSort algorithm, your task for this homework is to implement a string sorting method, `multi_digit_string_radix_sort`. Your implementation should take `group_size` as parameter, and return the number of `iterations` the algorithm goes through in the loops of the CountingSort algorithm (you need to use CountingSort as a subroutine in the RadixSort) to sort the given array `arr` of strings.

```
long multi_digit_string_radix_sort(std::string *arr,
                                   int size,
                                   bool ascending,
                                   int group_size);
```

The parameter `group_size` is critical. This parameter controls the size of the smallest partition of each string to be considered during the run of the algorithm. For instance, in the case of numbers, multi-digit RadixSort with a `group_size` of 1 is equal to the classical RadixSort. Keep in mind that `group_size` affects the number of buckets that will be created.

You should also consider the `ascending` boolean parameter to decide the order of the sorting. If `ascending = true`, you should sort the strings in dictionary order; if `ascending = false`, it should be the other way around. Additionally, the length of the string array `arr` is given as the `size` parameter.

2 Example Run

Example 1

```
--input--
ascending: 1
group_size: 3
size: 2
arr:
SEN
BEN

--output--
iterations: 19688
size: 2
arr (ordered):
BEN
SEN
```

Example 2

```
--input--
ascending: 0
group_size: 1
size: 12
arr:
GAUSS
EULER
TAO
CANTOR
NEWTON
POINCARE
LAGRANGE
CAUCHY
FOURIER
LAPLACE
LEIBNIZ
FERMAT

--output--
iterations: 496
size: 12
arr (ordered):
TAO
POINCARE
NEWTON
LEIBNIZ
LAPLACE
LAGRANGE
GAUSS
FOURIER
FERMAT
EULER
CAUCHY
CANTOR
```

3 Constraints and Limits

In this exam, the complexity of your implementations will be checked with your reporting of the count of `iterations`. Hence the system limitations are not strict, and are as follows:

- a maximum execution time of 2 minutes
- a 4 MB maximum memory limit
- a stack size of 128 MB for function calls (ie. recursive solutions)

There are some important points to keep in mind:

- Array elements will be strings each of which can contain only the characters as uppercase English letters (i.e. from 'A' to 'Z').
- It will be easier to follow the count of `iterations` if you implement your solution by modifying the pseudocodes given in your book.
- Different from the RadixSort algorithm in your book, it is not guaranteed that the strings in the array will always have the same length. (Hint: You can use an extra bucket during CountingSort routine to handle strings with different lengths.)
- Different than the algorithm for CountingSort in your book, initialize the count array as `int* C = new int[k]` and use the fourth loop for copying the array back. That means, you shouldn't count iterations during initialization, but you should count iterations during copying array back. Otherwise, the return value of the function (`iterations`) will not be evaluated as correct.
- You should count loop iterations in four different loops.
- If `group_size` is not a multiplier of the length of the longest string in `arr`, you should use maximum remaining letter count in CountingSort function. For example, if the `group_size` is 3 and the length of the longest string in the array is 5, in the first pass of the CountingSort, you should use the last 3 letters, in the second pass, you should use the first 2 letters.
- You can make sure that the `size` for the array `arr` and `group_size` will always be given to stay in the limitations of the VPL environment. Since the complexity of your implementation will be checked by your returned number of iterations, we will not test your code with such edge cases.

4 Specifications

- You will implement your solutions in the `the2.cpp` file.
- Do not change the first line of `the2.cpp`, which is `#include "the2.h"`.
- Do not change the arguments and the return value of the given functions in the file `the2.cpp`, but you are free to add other functions to `the2.cpp`.
- Do not include any other library or write include anywhere in your `the2.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.

- You can test your `the2.cpp` on the virtual lab environment. If you click run, your function will be compiled and executed with `test.cpp`. If you click evaluate, you will get feedback for your current work and your work will be temporarily graded with a limited number of inputs.
- The grade you see in VPL is not your final grade, your code will be reevaluated with more inputs after the exam.
- If you want to test your work and see your outputs on your locale you can use the following commands:

```
> g++ test.cpp the2.cpp -Wall -std=c++11 -o test
> ./test
```

5 Regulations

- **Implementation and Submission:** The template files are available in the Virtual Programming Lab (VPL) activity called "THE2" on ODTUCLASS. At this point, you have two options:
  - You can download the template files, complete the implementation, and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
  - You can directly use the editor of the VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submitting a file.

  Please make sure that your code runs on ODTUCLASS. There is no limitation in running your code online. The last save/submission will determine your final grade.
- **Programming Language:** You must code your program in C++. Your submission will be tested on the VPL environment in ODTUCLASS, hence you are expected to make sure your code runs successfully there.
- **Cheating: This assignment is designed to be worked on individually.** Additionally, the use of any LLMs (chatgpt, copilot, the other one that you are thinking about...) and copying code directly from the internet for implementations is strictly forbidden. Your work will be evaluated for cheating, and disciplinary action may be taken if necessary.
- **Evaluation:** Your program will be evaluated automatically using "black-box" testing, so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to worry about invalid cases. **Important Note:** The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official test cases to determine your final grade after the deadline.