Chair for Clinical Bioinformatics
Pascal Hirsch
Annika Engel
Saurabh Pandey
Ilia Olkhovskii

Programming Course 3rd Assignment

Hand in until 20.12.2023 at 11:59 pm

General remarks

Please make sure that you follow all rules stated in the general remarks PDF in the moodle. The number of barbells ⤡ for each task describes the expected difficulty. The maximum of number of barbells per task will be 4.
Note that tutors and automatic tests will assess error-handling and bounds checking mechanisms of your implementations using different kinds of invalid input.

Task 1 (8 P.): ⤡

1. What is the difference between the keywords `typedef` and `using`? Also, state one usage example for both. (0.5 P.)
2. What are iterators? Name at least 3 different types of iterators and explain their functionalities. (1 P.)
3. What is a crucial difference between an input and a forward iterator? (0.5 P.)
4. Why are iterators to a `std::vector` invalided by `insert`, `push_back` or `erase`? (0.5 P.)
5. Which container would you choose and why if
   a) you need to do sorted traversal and no duplicates are allowed (0.5 P.)
   b) the order is not important and no duplicates are allowed (0.5 P.)
   c) you only need to add elements at the end and want fast traversal (0.5 P.)
   d) you need to provide a dictionary that can associate multiple values with the same key (0.5 P.)
6. How does an STL algorithm usually indicate "not found" or "failure"? (0.5 P.)
7. What does `std::sort(b,e)` use as its sorting criterion? (0.5 P.)
8. Why is the STL container `set` missing a `[]` (subscript) operator? (0.5 P.)

9. Why does `std::map` require its `key_type` to be copy-constructible and its `mapped_type` to be default constructible? (1 P.)

10. Why is `std::forward_list` missing a `size` member function and why is the `empty` member-function still implemented if it is typically equal to `container.size() == 0`? (1 P.)

Task 2 (8 P.): ⤡⤡

Use for this task as many STL functions as possible. Write a program that takes five arguments n, m, s (unsigned int), p (double) and k (unsigned int). Your program should first generate a vector V_1 of n elements (as `double`) that are randomly drawn from a negative binomial distribution (initialized with the probability of a trial generating true p and a number of trial successes k and result type `int`). The random numbers should be generated with the Mersenne-Twister algorithm by Matsumoto and Nishimura (2000) using 64-bit integers, with default parameters and a fixed seed s. Afterwards report on the standard output the mean and sample standard deviation of the generated elements. The output precision of the stream should be set to 3. Subsequently, scale each element x by computing $log_2(|x| + 1)$. Then output up to the 10 largest elements, comma separated on one line. Next, generate a vector V_2 of m elements (as `double`) and modify them in the same way as before, but with a different seed $s_2 = s \cdot 2$. When repeating the procedure make sure to use a new distribution object. Output the same statistics as for V_1. Then, given these two vectors V_1 and V_2, perform a two-sample t-test on the scaled and non-scaled values and output the t-statistic according to the following https://en.wikipedia.org/wiki/Student%27s_t-test#Equal_or_unequal_sample_sizes,_similar_variances_(1/2_%3C_sX1/sX2_%3C_2). Output the total number of degrees of freedom as well. Next, generate a range of sequentially increasing integers from 1 to n. Shuffle them using random numbers generated by the same Mersenne-Twister algorithm as previously, initialized with a seed $s_3 = s \cdot 7$. Multiply the obtained elements with the **unsorted** $\log_2$-scaled values V_1. Finally, sort the resulting vector such that the order is as follows: first numbers for which the nearest integer value is even, sorted in descending order, followed by numbers with nearest integer being odd in ascending order. Print this vector comma separated on the standard output. Invalid parameters given to your program should result in an error message and lead to the termination of the program with exit code 1.

Sample command:

```
./task2 5 8 42 0.5 10
```

Sample output:

```
V1 Mean: 11
V1 Sample standard deviation: 3.16
```

```
V1 Top 5 elements: 3.91, 3.81, 3.7, 3.46, 2.81
V2 Mean: 12.9
V2 Sample standard deviation: 6.79
V2 Top 8 elements: 4.64, 4.39, 3.81, 3.81, 3.81, 3.58, 2.81, 2

Comparing V1 and V2...
Unscaled t-statistic: -0.573
Comparing log2 scaled V1 and V2...
Scaled t-statistic: -0.169
Total degrees of freedom: 11

Sorted vector: 19.5, 10.4, 3.81, 7.4, 11.2
```

Task 3 (14 P.): ↖↘↖↘

Implement a lightweight version of grep. In its essence, grep allows to find arbitrary user-defined text-patterns in a given set of files with several options to specify the strictness of matching and the output formatting. In the following, we describe the use-cases and arguments your implementation should cover to configure the search and report behaviour. To simplify argument parsing, we include the header-only library cxxopts in the respective code-template. As soon as your program finds a match of a pattern in a target file, the entire line where the match occurred should be reported on the standard output.

a) Basic usage:

   `./task3 pattern list of filenames...`

   where `pattern` is a valid regular expression. Each matching, case-sensitive occurence of `pattern` should be reported in each of the referenced filenames in order as given on the command line. Filenames are referenced as white-space separated list and providing only a single filename is considered valid.

b) `./task3 -i pattern list of filenames...`

   Same as in a) but the `pattern` is applied without case-sensitivity.

c) `./task3 -F pattern list of filenames...`

   Here, `pattern` should be treated as a fixed-string instead of a regular expression.

d) `./task3 -f FILENAME list of filenames...`

   Pattern(s) should be read from `FILENAME`. Each line in the referenced file defines one pattern that can be applied. Also, patterns have higher priority than target files, i.e. first all matches for each pattern from `FILENAME` should be reported for a file. In case several patterns match to the same line, only one corresponding

output line should be emitted, i.e. no duplicates. Thereafter, the next file should be checked against the patterns from `FILENAME` and so on.

e) `./task3 -v pattern list of filenames...`

Instead of reporting matches to `pattern`, output all non-matching lines.

f) `./task3 -n pattern list of filenames...`

In addition to the matches, print each line number from the target files where the match occurred, at the front and separated by a colon.

g) `./task3 -R pattern list of filenames...`

If a filename is a directory and `-R` is not provided, you should continue computation and print

`grep: <filename>: Is a directory`

However, if the `-R` flag is enabled you should instead recursively descend into the directory and all subdirectories. Process the files you find in these directories in reverse lexicographic order. This lexicographic ordering does not only focus on the filename but also considers the entire path prefix. Also note that symbolic links need to be resolved.

A user may arbitrarily combine any or all of these options above. It is up to you to make sure that only semantically valid combinations are considered by your implementation, which is expected to return a non-zero error-code otherwise. If multiple filenames are provided prefix each line reporting a match to the corresponding filepath as specified on the command line, separated by a colon. This has precedence over the flag that prints the line number, i.e. the correct order is `<filename>:<line_number>:<matched_line>` .

Example:

Content of input.txt:

```
Here goes the example text.
A year has 365 days.
```

Program call:

```
./task3 -i "^a" input.txt
```

Output:

```
A year has 365 days.
```

Note that you can always compare your implementation against the behavior of the official grep in case you require additional specifications.

Task 4 (10 P.): ↖↘↖

For this task we provide a `graph.hpp` file with a defined interface for an unweighted, directed `Graph` class storing edges between nodes in an adjacency matrix. Edges connecting vertices to themselves are not allowed but a graph may contain cycles. Your task is to implement the functions of the interface and to write tests for each public function using the Catch test framework (https://github.com/catchorg/Catch2). Perform your tests not only on trivial but on sufficiently complex and large graphs with at least 6 vertices and 9 edges. We expect you to test each interface function with both valid and invalid input. To this end, we provide pre-defined signatures for all the different test-cases. Try to prevent writing trivial tests, since in turn, we will assess the robustness of your tests both with a reference implementation and an erroneous implementation of the graph class.

Task 5 (20 (14 +6) P.): ↖↘↖↘ + ↖↘↖↘↖↘

Implement for the `Graph` class of Task 4 a directed depth-first search (dfs) iterator and an undirected dfs (dfsu) iterator starting at `vertex_idx`.

- `dfs_iterator dfs_begin(std::size_t vertex_idx)`
- `dfs_iterator dfs_end(std::size_t vertex_idx)`
- `dfsu_iterator dfsu_begin(std::size_t vertex_idx)`
- `dfsu_iterator dfsu_end(std::size_t vertex_idx)`

Your iterator should support the following basic operations:

- *it should return the underlying object
- ++it should let the iterator point to the next element
- comparison via the == operator and the != operator

Furthermore, overload your iterator functions so that they support constant objects of class Graph as well. Implement the following member functions:

- `const_dfs_iterator dfs_begin(std::size_t vertex_idx) const`
- `const_dfs_iterator dfs_end(std::size_t vertex_idx) const`
- `const_dfsu_iterator dfsu_begin(std::size_t vertex_idx) const`
- `const_dfsu_iterator dfsu_end(std::size_t vertex_idx) const`

To obtain all points for this task try to avoid code duplication across the four iterator classes described above by extracting the inner logic into re-usable functions.