Chair for Clinical Bioinformatics
M. Sc. Georges Schmartz
M. Sc. Pascal Hirsch
B. Sc. Matthias Flotho
B. Sc. Saurabh Pandey

Programming Course
3rd Assignment

Hand in until 07.12.2021 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 (12 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 formula. 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 (25 (15 + 10) P.): (🏋🏋+🏋🏋🏋🏋)

Write a program that takes as first argument a FASTA file, as second argument the k-mer size and as third argument the skip size n and outputs the number of n-skip-k-mers found in the sequences on the standard output. A n-skip-k-mer is defined as a sequence of length k that includes one out of every n+1 bases. The output should contain the header of the sequence and then the uppercase n-skip-k-mer and its count (space separated), sorted lexicographically by the n-skip-k-mers. The FASTA file should only contain DNA sequences. If invalid sequences are found print their header on the standard error stream. Upper or lowercase DNA count to the same k-mer. To obtain all points your program should run in less than 2 seconds for k=10, n=1 and in less than 1 minute for k=33, n=1 on the CIP pool computers with the provided FASTA. You might want to use different containers for different sizes of k. For measuring the runtime of your program, don't forget to redirect the output to a file (or best /dev/null). Printing the output on the terminal takes some additional time that we don't want to measure. You can measure the runtime under Linux via the following command:

```
time ./task3 in.fa 10 1 > /dev/null
```

Hint: depending on the size of k it might be a good idea to hash the n-skip-k-mer s via the following function:

$$H(s,k) = \sum_{i=0}^{k-1} h(s_i) * 4^{k-i-1}$$

with

$$h(b) = \begin{cases} 0 & \text{if b=A} \\ 1 & \text{if b=C} \\ 2 & \text{if b=G} \\ 3 & \text{if b=T} \end{cases}$$

Sample input:

```
>seq1
```

```
ACGGAGCA
>seq2
ACGAG
>seq3
GGGGGG
```

Sample output for k = 3 and n = 1:

```
seq1
AGA 1
CGG 1
GAC 1
GGA 1

seq2
AGG 1

seq3
GGG 2
```

Task 4 (25 (25 + 10) P.): ⤡⤡⤡ + ⤡⤡⤡⤡

For this task you should extend the provided `Matrix` class by a reverse column iterator and diagonal iterator (only valid on square matrices). A basic iterator for lists was shown in lecture 6 slide 36. Provide the following member functions:

1. `col_reverse_iterator col_rbegin(std::size_t col_num)`
2. `col_reverse_iterator col_rend(std::size_t col_num)`
3. `diag_iterator diag_begin()`
4. `diag_iterator diag_end()`

Your iterator should support the following basic operations:

1. `*it should return the underlying object`
2. `++it should let the iterator point to the next element`
3. `comparison via the == operator and the != operator`

To obtain all points overload your iterator functions so that they work with constant `Matrix` objects as well (via `const_col_reverse_iterator` and `const_diag_iterator`). Therefore provide the following member functions:

1. `const_col_reverse_iterator col_rbegin(std::size_t col_num) const`
2. `const_col_reverse_iterator col_rend(std::size_t col_num) const`

3. `const_col_reverse_iterator col_rend(std::size_t col_num) const`

4. `const_diag_iterator diag_end() const`