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

Programming Course
4th Assignment

Hand in until 17.01.24 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 (10 P.): 🏋

1. Why is it dangerous to return a reference when you must return an object? (1 P.)
2. What is an implicit interface in the context of compile-time polymorphism? (1 P.)
3. Why should you prefer range member functions to their single-element counterparts? (1 P.)
4. Why should you avoid in-place modification of keys in a set? (1 P.)
5. Why should predicate functions be pure? (1 P.)
6. Why should you avoid the default capture modes of lamdbas? (1 P.)
7. What do `std::move` and `std::forward` do? (2 P.)
8. How are the strings of a `std::set<std::string*>` sorted? How you would make them sorted lexicographically? Provide the necessary code. (2 P.)

Task 2 (12 P.): ⤡⤡

Given are n cities. Each city c_i is connected to its neighboring city c_{i+1} and the distance between c_i and c_j is $|i - j|$.
To enable faster delivery of Christmas presents, beaming stations have been built in some of the cities. However, due to the large energy consumption they have not been turned on yet. Once enabled, each beaming station allows to teleport anything up to a distance $< k$.
However, to perform efficient low temperature delivery of coronavirus vaccines, the exothermic beaming procedure is unsuitable. Instead, expensive catapults have to be built in selected cities, allowing to shoot anything up to a distance $< l$.
Your first task is to find the minimum number of beaming stations that must be enabled to ensure that all cities are within the radius of at least one beaming station, so that their inhabitants receive all gifts in time. Additionally, compute the minimum number of catapults that have to be constructed in order to guarantee that vaccines can be delivered into every city. For each of the two problems, print the minimal number found on the standard output. If there is no solution to one or either of the problems print `-1`.
The input will be provided on the standard input stream. In the first line we will provide the number n of cities and the ranges k and l of the beaming stations and catapults, respectively. The second line is to define the landscape of cities and will consist of n space-separated values $c \in \{0, 1, 2\}^n$, the mth digit denoting whether the mth city (c_m) is able to build a catapult (2), owns a beaming station (1) or neither of these (0).

Sample input:

```
6 2 1
0 1 0 2 1 0
```

Sample output:

```
beamer:2
cata:-1
```

Task 3 (13 P.): ⤡⤡⤡

In this task you should implement the Munkres algorithm that has a runtime complexity of $O(n^3)$ and solves the assignment problem. Appropriate pseudo-code to guide your implementation can be found in **Algorithm 2** in the scientific publication by Riesen & Bunke. The article is also available from our moodle. The corresponding code templates for this assignment sheet contain the files `munkres_algorithm.cpp/.hpp` and `matrix.hpp`. Please provide an implementation for the algorithm having the signature

```
  Matrix<int> run_munkres_algorithm(Matrix<int> c)
```

in `munkres_algorithm.cpp`. The source file `task3.cpp` already contains a simple test-case, which is described below. You may use this file to further test your implementation with different and potentially more complex instances of the assignment problem.

Input matrix:

```
250 400 350
400 600 350
200 400 250
```

Output matrix:

```
0 1 0
0 0 1
1 0 0
```

Task 4 (13 P.): ⤡⤡⤡

In this task you will implement a Bloom filter. It is a space-efficient probabilistic data structure that is used to test whether an element belongs to a set. False positive matches can occur but not false negatives. Due to its properties elements can only be added but not removed. The more elements the Bloom filter contains, the higher gets the false positive rate. Create a templated class
`template<typename Key, unsigned int m, typename Hash> BloomFilter`
that accepts as template parameter the class of the element it will store, the number of bits it uses m and the hash function to use for your `Key` class. Use the MurmurHash function provided in the `murmurhash.hpp`. Your hash function object should provide the following operator:

```
  std::size_t operator()(Key key, unsigned int seed) const
```

The class should be stored in `bloom_filter.hpp`. As data structure for the bit array use `std::bitset`. Your class should provide the following functions:

1. `BloomFilter(unsigned int k)`, k is the number of hash functions
2. `BloomFilter(std::initializer_list<Key>, unsigned int k)`
3. `template<typename It> BloomFilter(It first, It last, unsigned int k)` that inserts all elements defined by the iterator range.
4. `bool insert(const Key&)` that inserts the key and returns if the insertion was successful. If the element was already in the filter then it returns false.
5. `bool contains(const Key&) const` that returns if the element is present.

6. `template<typename It> double false_positive_rate(`
   `It positive_begin, It positive_end,`
   `It negative_begin, It negative_end) const`

   see false positive rate

7. `double space_ratio(uint64_t num_elements) const`, ratio of the space consumed by your filter in comparison to the minimum size required for `num_elements` of the `Key` class

8. `uint64_t approx_size() const` that approximates the number of elements in the Bloom filter via the formula (k is the number of hash functions and X is the number of bits set to one):

$$n^* = -\frac{m}{k} \cdot ln(1 - \frac{X}{m})$$

To determine the number of bytes used to represent e.g. one `int` use the `sizeof` operator.

Task 5 (12 P.):

Write a program that takes an adjacency matrix on the standard input and outputs all maximal cliques on the standard output. The adjacency matrix can be read in with the same matrix class provided for task 3. Implement the Bron-Kerbosch algorithm to detect the maximal cliques.

Sample input:

```
0 1 0 0 1 0
1 0 1 0 1 0
0 1 0 1 0 0
0 0 1 0 1 1
1 1 0 1 0 0
0 0 0 1 0 0
```

Sample output:

```
{0, 1, 4}
{1, 2}
{2, 3}
{3, 4}
{3, 5}
```

Sample input2:

```
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 1 1 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 1 1 1 0 0 0
0 0 0 1 0 0 1 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 1 1 0 0 0
0 0 0 0 0 0 1 1 0 1 0 1 0 0 0
0 0 0 0 0 0 1 1 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
```

Sample output2:

```
{0, 1, 2}
{1, 3}
{3, 4, 5}
{3, 6, 7}
{6, 7, 9, 10, 11}
{8, 9}
{12, 13, 14}
```