Combinatorics Project 2.
Student ID: **2018280070** Name: **Peter Garamvoelgyi**

1. Problem statement

This assignment was about implementing one or more algorithms for generating permutations. In the following sections, I am going to give a brief explanation of three common algorithms (lexicographic ordering, Steinhaus-Johnson-Trotter, and inversions). Then I will proceed to discuss implementation details using C++. Finally, an experimental comparison of the three algorithms is presented.

2. Algorithms

In this section, I am going to give a brief overview of the implemented algorithms. Furthermore, some of the pros and cons of each method are discussed.

Lexicographic ordering

When a set has an ordering defined on it, we can use it to define an ordering on collections of items drawn from the set. Let us regard two *words:*

$$A = [a_1, a_2, \dots, a_n] \quad b = [b_1, b_2, \dots, b_m]$$

To define the order of A and B:

1. First, find the first index i for which $a_i \neq b_i$.
2. If no such i exists, then the shorter of the two words is *smaller*, i.e.

$$\begin{array}{ll} A < B & \text{if } n < m \\ A = B & \text{if } n = m \\ B < A & \text{otherwise} \end{array}$$

3. If there is such an i, then

$$\begin{array}{ll} A < B & \text{if } a_i < b_i \\ B < A & \text{otherwise} \end{array}$$

This algorithm uses a method for enumerating all permutations of a collection of distinct elements following their lexicographic ordering. The algorithm goes as follows.

1. Starting from the *right*, find the first element where the sequence decreases. This element divides the item into a *prefix* and a *suffix*.
2. Swap this element with the smallest item larger than it from the suffix.
3. Shuffle the suffix so that it is as small as possible (i.e. sort it).

Discussion:

- \+ **Intuitive ordering**: The order of the generated permutations corresponds to most developer's expectations.
- \+ **Zero storage overhead**: Given a permutation, we can generate the next one without any additional input.
- – **Items need to be comparable**, i.e. there need to exist a total ordering defined on the type of the items.

Steinhaus-Johnson-Trotter (SJT) algorithm

In the SJT algorithm, we first assign an arrow to each item. Arrows are either LEFT or RIGHT ones, with LEFT being the initial state for each item. Let us define *mobile items* as items whose arrow points to a smaller item.

$$\begin{array}{cccccc} \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow \\ 2 & \mathbf{6} & \mathbf{3} & 1 & \mathbf{5} & 4 \end{array}$$

In this example, the numbers 6, 3, and 5 are mobile. Given this definition, we can describe one step of the algorithm as follows.

1. First, find the largest mobile item m. If there are no more mobile items then terminate.
2. Then, switch m and the adjacent item its arrow points to.
3. Finally, switch the direction of the arrows associated with items larger than m.

Using these steps, we can enumerate all permutations of the given items (given that initially they are sorted in ascending order).

Discussion:

- \+ **Consecutive permutations are neighbors**, i.e. they only differ by two adjacent items being swapped.
- – **Storage overhead**: In addition to the current permutation, we also need to keep track of the *arrows*, one for each item. This doubles the storage requirements.
- – **Items need to be comparable** (see above).

Generating from inversions

The pair (i_k, i_l) on the permutation $i_1 i_2 \dots i_n$ is called an inversion if $k < l$ and $i_k > i_l$, i.e. the given pair of numbers are out of their natural order.

For the items $[1, 2, 3, 4, 5]$, 1 can cause up to 4 inversions, 2 can cause up to 3, etc. From the inversion number, we can algorithmically reconstruct the permutation, i.e. there is a bijection between these two. For example, for the iversion sequence $(1:1)\ (2:2)\ (3:0)\ (4:1)\ (5:0)$, we can reconstruct 31524.

Given these facts, we can use inversions for generating permutations. In each step:

1. First, increment the inversion sequence.
2. Then, generate the corresponding permutation.

A possible ordering of the inversions for the items above would be like this:

$$00000\ 10000\ \ldots\ 40000\ 01000\ \ldots\ 03000\ 13000\ \ldots\ 43000\ \ldots\ 00100\ \ldots$$

... where the i'th digit corresponds to the inversion number of item i.

Discussion:

- \+ **Low overhead**: While we need to store the current inversion, there is no need to store the current permutation, as it can be generated directly from the inversion.
- – **Items need to be comparable** (see above).

3. Implementation strategy

For implementing the algorithms above, I first defined a common interface `PermutationGenerator`:

```
template <typename T>
class PermutationGenerator
{
public:
    virtual bool next() = 0;
    virtual std::vector<T> get() const = 0;
};
```

The `next` method would step to the next permutation, while `get` returns the current one. The subclasses are called `Lexicographic`, `SJT`, and `Inversion`.

Representing these generators using classes is especially useful for SJT and Inversion, where we need to keep track of the associated state (arrows and inversions, respectively). Classes provide encapsulation and data hiding. Moreover, defining a common interface increases reusability and makes testing easier, too.

I used the lightweight Catch C++ testing library for Test Driven Development (TDD). Test cases are defined like this

```
SECTION("[1, 2, 3, 4, 5] next permutation is correct")
{
    Lexicographic<int> g = {1, 2, 3, 4, 5};

    bool success = g.next();
    REQUIRE(success == true);

    auto perm = g.get();
    REQUIRE(perm == std::vector<int>({1, 2, 3, 5, 4}));
}
```

Having defined many tests, including corner cases, helped me quickly detect implementation errors.

4. Performance profiling

I used the built-in `chrono` stdlib library to compare execution times of the three algorithms and the C++ built-in `next_permutation` function (this one is using lexicographic ordering). The idea was to define a sequence, and generate all permutations using all the generators. Then, the execution time is divided by the number of permutations, thus giving us the average time required for generating a single permutation.

An example output for the sequence `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}` is shown below.

```
          -O0 (no opt)                  |          -O3 (optimized)
StdLib: 65.451 nanoseconds              |          StdLib: 3.1994 nanoseconds
Lexico: 69.732 nanoseconds              |          Lexico: 10.186 nanoseconds
SJT:    483.21 nanoseconds              |          SJT:    71.351 nanoseconds
Inv:    38.515 nanoseconds              |          Inv:    1.0765 nanoseconds
```

As we can see, our lexicographic generator performs comparably well with the standard library implementation, although the difference gets bigger with optimization enabled. SJT is significantly slower, while the inversion one is significantly faster. Note, however, that the test only measures time for `next`, not for `get`, so Inversion gets some leverage. Including both `next` and `get` in the enumeration, we get different results:

```
          -O0 (no opt)                  |          -O3 (optimized)
StdLib: 276.75 nanoseconds              |          StdLib: 86.375 nanoseconds
Lexico: 283.69 nanoseconds              |          Lexico: 92.040 nanoseconds
SJT:    682.41 nanoseconds              |          SJT:    144.63 nanoseconds
Inv:    1057.3 nanoseconds              |          Inv:    236.86 nanoseconds
```

This shows that Inversion is good for quickly stepping over permutations but does not perform so well once we actually want to use those permutations.

Admittedly, there is still plenty of room for improvement. The large difference between the two test cases above is mostly due to copying, i.e. it is not inherent to the algorithms. Returning a copy might not even be necessary for many (most) use cases. Furthermore, the algorithms themselves could be optimized and tuned further.

5. Novel points

- TDD-based implementation of permutation algorithms
- Performance comparison of 3 (+1) implementations.
- Detailed discussion.

6. Conclusion

This assignment has been a good opportunity to gain a deeper understanding of common algorithms for generating permutations, while also practicing TDD using C++. The algorithms have comparable performance with different pros and cons for each. Choice should be made according to the specific use case, with these trade-offs kept in mind.