Problem 2. Basic STL Exercises

Please go to Piazza Resources to download the materials for this problem.

This problem consists of several small exercises on STL iterators and algorithms. In all of the exercises:

- You are not allowed to write loops or recursions. You must make full use of the standard library algorithms to achieve the goals.
- You are not allowed to define other classes or functions. Write lambda expressions if necessary.

Subtask 1: `dropDuplicates`

Write a function `dropDuplicates(strings)`, where `strings` is a vector of strings, that eliminates the duplicate elements in `strings`. After the function call, `strings` contains the same set of strings (possibly sorted) as original, but each string appears only once.

For example:

```
std::vector<std::string> words{"the", "quick", "red", "fox", "jumps", "over", "the", "slow",
"red", "turtle"};
dropDuplicates(words);
for (const auto &word : words)
  std::cout << word << ' ';
```

Possible output:

```
fox jumps over quick red slow the turtle
```

You may need the following functions:

- `std::sort`
- `std::unique`
- `std::vector<T>::erase`

Subtask 2: `partitionByLength`

Write a function `partitionByLength(strings, k)`, where `strings` is a vector of strings and `k` is of type `std::size_t`. Partition the vector into two parts (with the elements rearranged) so that the first part contains all the strings with length no greater than `k` and the second part contains the rest of the strings. Return an iterator just past the last element of the first part.

The order of the strings may be changed, which does not matter.

For example,

```
std::vector<std::string> words{"fox", "jumps", "over", "quick", "red", "slow", "the", "turtle"};
auto pos = takeShort(words, 4);
for (auto it = words.begin(); it != pos; ++it)
  std::cout << *it << ' ';
```

Possible output: (The order does not matter.)

```
fox slow over the red
```

The standard library defines an algorithm named `std::partition` that takes a predicate and partitions the container so that values for which the predicate is `true` appear in the first part and those for which the predicate is `false` appear in the second part. The algorithm returns an iterator just past the last element for which the predicate returned `true`.

Write a lambda expression and pass it to `std::partition` to achieve the goal.

Subtask 3: `generatePermutations`

Write a function `generatePermutations(n, os)`, where `n` is a positive integer and `os` is an ostream, that generates all the permutations of $\{1, 2, \cdots. n\}$ in lexicographical order and print them to `os`. For example:

```
generatePermutations(3, std::cout);
```

Possible output:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

You are provided with the following function framework.

```
std::vector<int> numbers(n);

// TODO: Fill `numbers` with {1, 2, ..., n}.

do {
  // TODO: Print the numbers, separated by a space.

  os << '\n';
} while (/* TODO: Generate the next permutation */);
```

Do not modify the framework. All you need to do is to fill in the three blanks marked `TODO`, according to the following rules:

1. For the first blank, use a standard library algorithm defined in the header `<numeric>` to fill `numbers` with `1`, `2`, ..., `n`.
2. For the second blank, write a lambda and pass it to `std::for_each` to print the numbers in `numbers`. It is OK if your output contains a trailing space like `1 2 3`. Note that you should print things to `os`, which is not necessarily `std::cout`.
3. For the third blank, we need to find the *next* permutation of `numbers` in lexicographical order, or terminate the loop if the "next permutation" does not exist. This should be done straightforwardly by a standard library function defined in the header `<algorithm>`.

In other words, what you write in this function should be nothing but three standard library function calls.