МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и струтуры данных»

Тема: Алгоритмы сортировки

Студент гр. 1301 _____ Устинов Г.А.

Преподаватель _____ Родионова Е.А.

Санкт-Петербург

2022

Цель работы

Изучить и реализовать некоторые алгоритмы сортировки.

Задание

Реализовать следующие алгоритмы сортировки:

1. Сортировка вставками (Insertion sort)
2. Сортировка выбором (Selection sort)
3. Пузырьковая сортировка (Bubble sort)
4. Сортировка слиянием (Merge sort)
5. Сортировка Шелла (Shell sort)
6. Быстрая сортировка (Quick sort)

Для каждого алгоритма сортировки указать временную асимптотическую сложность для лучшего, худшего случая и среднего случая, а также пространственную сложность, подкрепив это логическими построениями (в меру своих сил). Свести получившиеся результаты в таблицу. Построить график зависимости времени выполнения от размера входных данных и определить временную асимптотическую сложность для лучшего, худшего, среднего случая практически. Определить наиболее быстрый алгоритм сортировки. Сравнить скорость его выполнения с одним из алгоритмов сортировки из базовых библиотек языка.

Выполнение работы

Для каждого алгоритма была реализована отдельная функция, которая должна выполнять сортировку некоторого промежутка, заданного итераторами, соответсвующими указанному интерфейсу (например, случайного доступа).

Разработанный программный код см. в приложении А.

Сортировка вставками

Принцип работы алгоритма заключается в следующем: делим массив на две части, считая, что одна часть отсортирована, а другая ещё нет. Изначально граница между двумя частями пролегает так, что отсортированная часть массива пуста, а другая содержит все элементы массива. Далее, по очереди просматриваем каждый элемент, находим в отсортированной части позицию, куда его вставить, ставим его на эту позицию и сдвигаем границу отсортированного участка на еденицу. После рассмотрения каждого элемента, все элементы расставлены в отсортированном порядке, и массив отсортирован.

В выполненной реализации поиск места осуществляется не с начала массива, а с текущуего элемента и происходит одновременно с перемещением элементов. Поскольку работа ведётся с массивом или вектором, а не списком, элементы приходится «сдвигать», на то теряется значительная часть производительности.

Лучшим является случай, когда массив отсортирован. Тогда для каждого элемента поиск находит нужное место на первой же итерации, и никаких перестановок не требуется. Поскольку будет отсмотрен каждый элемент, сложность в этом случае является линейной: $\Theta(N)=N$.

Худший случай — массив отсортирован в обратном порядке. Тогда не только поиск займёт наибольшее возможное время — но и будут «сдвинуты» все элементы, предшевствующие текущему. Для каждого n-ного из N элементов будет выполнено n перестановок:

$$\Theta(N)=\sum_{n=1}^{N} n=\frac{(1+N)*N}{2}=\frac{N^2}{2}+\frac{N}{2}.$$

С оценкой сверху: $O(N)=N^2$.

В среднем, можно сказать, что для каждого n-ного элемента выпонится n/2 перестановок. Тогда расчёт сложности для этого случая будет выглядеть так:

$$\Theta(N)=\sum_{n=1}^{N}(\frac{n}{2})=\frac{(1+\frac{N}{2})*N}{2}=\frac{N^2}{4}+\frac{N}{2}.$$

Оценка сверху, опять же, будет равна $O(N)=N^2$.

В написанной реализации, накладные расходы по памяти константны, потому что максимум один элемент во время копирования для сдвига может быть вынесен вне пределов массива, а расходы на счётчик и копии итераторы всегда одни и те же, вне зависимости от размера входных данных.

Эта сортировка, как будет показано ниже, не является самой быстрой из подобных, потому что работа ведётся с тем элементом, который является текущим, это влечёт за собой сдвиг элементов массива, что является самой дорогостоящей операцией.

Сортировка выбором

Ключевое отличие сортировки выбором от сортировки вставками в попытке оптимизации значения, которое втавляется в отсортированную часть. Находится наименьший элемент на неотсортированном участке, который затем встаёт на последнее место в отсортированном массиве, что не требует сдвига элементов массива.

В выполненной реализации после нахождения лучшего элемента, производится обмен значений элементов. Поскольку в любом случае будут просмотрены все неотсортированные элементы для нахождения наилучшего, не имеет значения порядок, в каком они будут просмотрены, а значит перестановка не оказывает дополнительного влияния на производительность, но при этом избавляет от необходимости сдвига массива.

Поскольку на каждой итерации отсматривается весь оставшийся массив для поиска наилучшего элемента для перестановки, лучший, худший и средний случай имеют однаковую оценку, которая будет равна:

$$\Theta(N)=\sum_{n=1}^{N}(N-n)=\frac{N*N}{2}=\frac{N^2}{2}.$$

Накладные расходы по памяти на итераторы копируемый элемент константны.

Среди всех подобных сортировок, именно сортировка вставками показывает самые лучшие результаты на практике благодаря наименьшим постоянным затратам времени на каждой итерации. Не смотря на формальную оценку квадратичной функцией, наиболее дорогостоящая операция обмена двух элементов происходит максимум N раз, а всё остальное — сравнения, которые в общем случае происходят быстро.

Пузырьковая сортирока

Идея пузырьковой сортировки заключается в том, чтобы на каждой итерации менять местами элементы, которые не отсортированы локально. По итогу после некоторого количества проходов по массиву от начала до конца «большие» элементы выдавливаются в конец массива, а «маленькие» - наоборот в начало, где каждый попарно сортируется.

Лучшим случаем является отсортированный массив — в таком случае алгоритм один раз проходит массив от начала до конца, убеждаясь в том, что он отсортирован, и завершается. Оценка для лучшего случая - $\Theta(N)=N$.

В худшем и среднем случае на каждой итерации самый «большой» элемент будет выдавливаться в конец, проходя от 0 до N обменов с более маленькими элементами:

$$\Theta(N)=\sum_{n=1}^{N}(N-n)=\frac{N*N}{2}=\frac{N^2}{2}.$$

Накладные расходы по памяти как и в предыдущих рассмотренных алгоритмах постоянны.

Не смотря на не самую плохую асимптотику среди подобных алгоритмов, пузырьковая сортировка работает значительно хуже, чем сортировки вставками и поиском, потому что на каждой итерации внутреннего цикла она меняет местами до N — 1 элементов.

Сортировка слиянием

Алгоритм работает по принципу «Разделяй и властвуй» - массив разбивается пополам до тех пор, пока решение не приобретёт тривиальный вид — например, два элемента или меньше, — после чего объединяет решения в одно: сливая две половины в один массив. Всего таких разбиений будет $N\log_2(N)$. Тривиальных случаев будет столько, сколько разбиений на 1 или 2 элемента (зависит от округления конкретных значений) - $\frac{N}{2}$. Также, в тривиальных случаях не проводится слияние массивов.

Алгоритм всегда производит разбиения вне зависимости от того, отсортирован массив или нет, поэтому лучший, худший и средний случай имеют одну и ту же оценку, которая исходит из сложности слияния, количества слияний и количества тривиальных случаев-перестановок:

$$\Theta(N)=N(\log_2(N)-1)+\frac{N}{2}=N\log_2(N)-\frac{N}{2}.$$

Оценка сверху равна $O(N)=N\log_2(N)$.

В реализованной версии единовременные расходы по памяти не превышают N, потому что разбиение и тривиальные случаи рассматриваются на изначальном массиве, а дополнительная память выделяется только для объединения решений в одно, что позволяет избежать нарастания затрат больших, чем константные затраты для рекурсивного алгоритма.

Сортировка слиянием работает значительно быстрее, чем рассмотренные ранее сортировки, благодаря более пологой асимптотике решения, поэтому для сравнений на основе эмпирических измерений она помещена в группу «быстрых» сортировок.

Впрочем, среди «быстрых» сортировок, этот алгоритм работает заметно медленнее других, это происходит из-за динамического выделения памяти практически на половине рекурсивных вызовов, что является самом затратной операцией по времени.

Сортировка Шелла

Аналогично пузырьковой сортировке алгоритм последовательно просматривает все пары элементов, находящиеся на заданном расстоянии, попарно сортируя их, затем уменьшает это расстояние и повторяет действия заново. Так происходит до тех пор, пока массив не будет пройдет с расстоянием 1. Тогда массив считается отсортированным.

Быстродействие данного алгоритма зависит от метода выбора расстояния между элементами, которое будет рассматриваться. Для выполнения рыботы был выбран наиболее простой — каждый раз расстояние будет уменьшаться вдвое. Наибольшая же эффективность доказана при уменьшении этого расстояния примерно в 2.25 раз, в таком случае оценка времени достигает показателя $N\log^2(N)$.

При выбранной же последовательности в худшем случае время алгоритма оценивается как $O(N)=N^2$. Для лучшего же случая оценка составляет $O(N)=N\log(N)$.

В реализованной версии накладные расходы по памяти постоянны, и не зависят от размера входных данных.

Среди алгоритмов с квадратичной асимптотикой, сортировка Шелла показывает наихудшие результаты. Предполагаю, что это из-за выбора

последовательности для выбора промежутка сравнения элементов. «Скачки» на построенных графиках также могу объяснить только выбором последовательности промежутков и на погрешность, связанная с её вычислением, которая может увеличить количество итераций при некотором граничном переходе количества элементов.

Быстрая сортировка

Алгоритм быстрой сортировки работает по принципу «Разделяй и властвуй». При каждом вызове функции выбирается некоторое опорное значение, относительно которого сортируются элементы: те что меньше помещаются в левую часть массива, те что больше — в правую. Затем массив разбивается по границе левой и правой частей, и алгоритм повторяетс для каждой из подчастей.

Быстродействие алгоритма зависит от способа выбора опорного элемента. Для реализации было выбрано разбиение Хоара — за опорное значение берётся значение элемента из середины массива. При любом тривиальном выборе может быть выбрано такое значение, что разбиение каждый раз будет делить массив на части длинами N - 1 и 1, что увеличит количество разбиений до N, лишая преимущества разбиения на подмассивы. Выбор среднего элемента может частично решать эту проблему, однако конкретное время выполнения будет зависеть от входных данных.

В худшем случае, когда на каждом разбиении будет выбираться опорным либо наименьший, либо наибольший элемент, и на каждом разбиении нужно будет опоменять все элементы местами, сложность может быть оценена как:

$$\Theta(N) = N(\frac{N}{2}) = \frac{N^2}{2}.$$

В лучшем случае, когда массив отсортирован, сложность алгоритма выглядит так: $\Theta(N)=N\log_2(N)$ — разбиения всегда по середине, перестановки элементов не происходит.

В среднем, разбиения происходят достаточно близко к середине, а количество перестановок элементов на каждом подмассиве не больше N / 2. В таком случае, можно оценить средний случай сверху так:

$$O(N)=\log_2(N)\frac{N}{2}=N\log(N).$$

В написанной реализации накладные расходы по памяти не превышают N - накладные расходы рекурсивного алгоритма в худшем случае разбиений.

Не смотря на квадратичную асимптотику худшего случая, быстрая сортировка оказывается на практике самой быстрой сортировкой, из написанных. Не имея дополнительных потерь времени на динамическое выделение памяти, и имея хорошую асимптотику среднего решения, быстрая сортировка уступает только сортировке стандартной библиотеки.

Об используемой стандартной сортировке

Стандарт языка C++ не указывает, какой алгоритм должен использоваться, только гарантирет выполнение за время $N\log(N)$.

Таблица 1 — Временная и пространственная сложность алгоритмов

Сортировка	Best-case	Avg.-case	Worst-case	Память
Вставками	N	N^2	N^2	1
Выбором	N^2	N^2	N^2	1
Пузырьковая	N	N^2	N^2	1
Слиянием	$N\ logN$	$N\ logN$	$N\ logN$	N
Шелла	$N\ logN$	N^2	N^2	1
Быстрая	$N\ logN$	$N\ logN$	N^2	N

Проведение замеров и построение графиков

Для проведения практических измерений были сгенерированы случайные наборы целых чисел размером в 5000000 элементов. Для группы «медленных» алгоритмов замеры проводились на первых 10000 элементах с шагом в 100 элементов, для «быстрых» - на всех элементах с шагом в 10000 элементов.

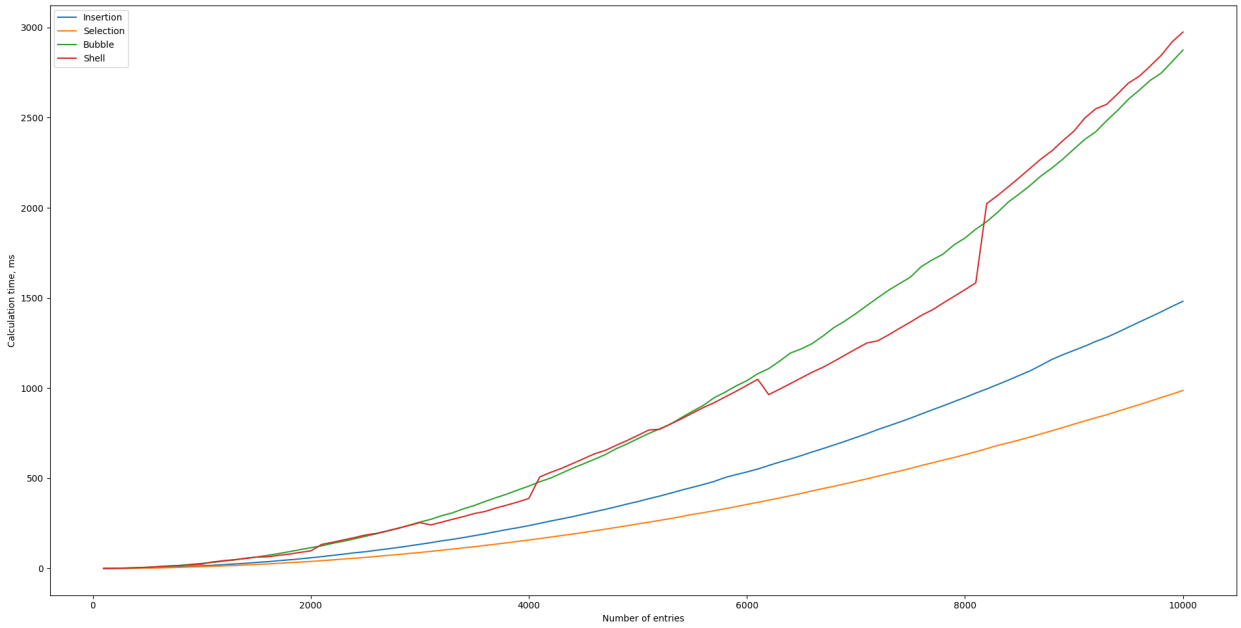

Рисунок 1 — Графики полученных значений времени выполнения для сортировок вставками, выбором, пузырьковой и сортировки Шелла

На Рисунке 1 можно заметить, что графики времени для сортировок вставками и выбором довольно пологие. Благодаря малому количеству действий, сортировка выбором показывает наилучшие результаты, не смотря на худшие показатели асимптоки в лучшем случае, и такие же в худшем и среднем случаях.

Также можно отметить, что начиная с некоторого предела при увеличении количества элементов в двое, время выполнения сортировки Шелла и пузырьковой сортировки вырастает в четыре раза.

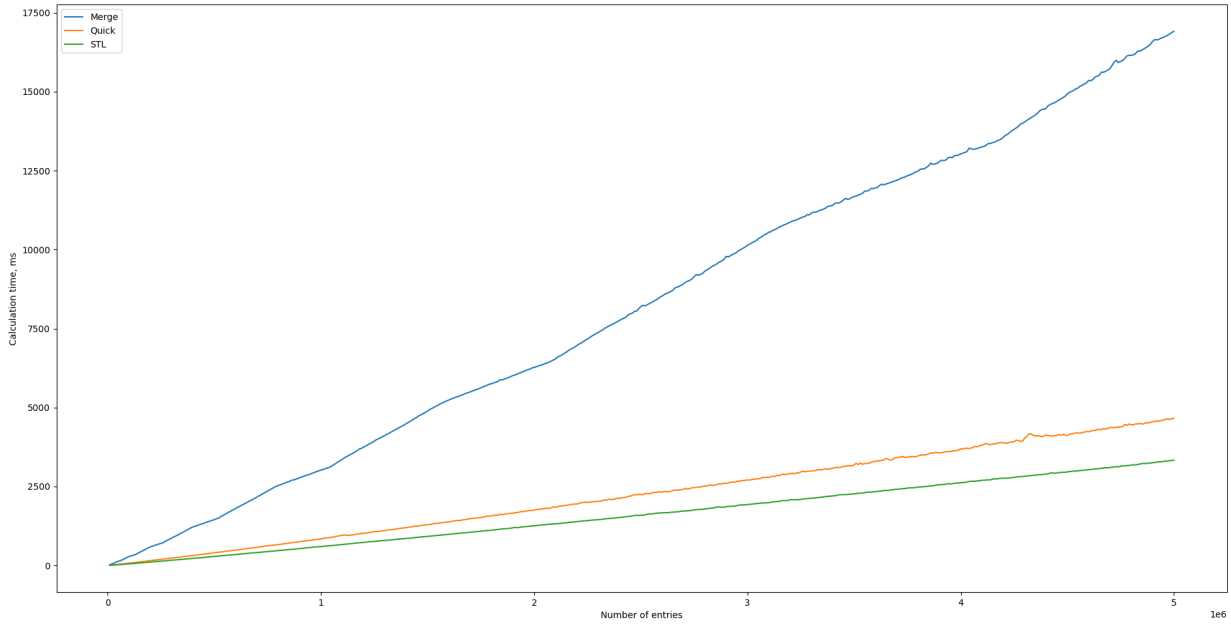

Рисунок 2 — Графики полученных значений времени выполнения для сортировок слиянием, быстрой сортировки и средств стандартной библиотеки

На Рисунке 2 невооружёному глазу кажется, что графики линейны, и отличаются только коэффициентом, что конечно же, не так. При более детальном рассмотрении можно заметить склонность построенных графиков к нелинейному росту, однако замер на большем количестве элементов занимал бы слишком много времени, и потому был признан нецелесообразным.

Можно отметить, насколько выделение дополнительной памяти в сортировке слиянием влияет на быстродействие, по сравнению с быстрой сортировкой.

По численным значениям можно сказать, что реализованный алгоритм быстрой сортировки медленее средств стандартной библиотеки примерно в 1.5 раза при всевозможных длинах входных данных. Скорее всего, «под капотом» стандартной сортировки находится алгоритм сходной асимптотики, но реализация которого проводит все операции быстрее и без потерь.

Выводы

Были изучены и реализованы некоторые распространённые и учебные алгоритмы сортировки, их сложность и особенности. Также была проведена наглядная демонстрация быстродействия алгоритмов на случайных входных данных.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл **src/main.cpp**

```
#include <algorithm>
#include <chrono>
#include <fstream>
#include <iostream>
#include <sstream>
#include <vector>

#include "sorts.hh"

using namespace std;

vector<int> readData()
{
   ifstream in("data.txt");
   size_t n;
   in >> n;
   vector<int> data(n);
   for (size_t i = 0; i < n; ++i)
       in >> data[i];
   return data;
}

void measureInsertionSort(vector<int> &data)
{
   size_t step = 100;
   vector<int> slice;
   ofstream out("insertion.txt");
   for (size_t size = step; size <= 10000; size += step) {
       copy(
           data.begin(),
           data.begin() + size,
           back_inserter(slice)
       );
       chrono::steady_clock timer;
       chrono::time_point<chrono::steady_clock> start = timer.now();
       insertionSort(slice.begin(), slice.end());
       chrono::time_point<chrono::steady_clock> finish = timer.now();
       out << size << " entries sorted in "
           << (finish - start) / chrono::milliseconds(1) << "ms" <<
```

```
endl;
        slice.clear();
    }
}

void measureSelectionSort(vector<int> &data)
{
    size_t step = 100;
    vector<int> slice;
    ofstream out("selection.txt");
    for (size_t size = step; size <= 10000; size += step) {
        copy(
            data.begin(),
            data.begin() + size,
            back_inserter(slice)
        );
        chrono::steady_clock timer;
        chrono::time_point<chrono::steady_clock> start = timer.now();
        selectionSort(slice.begin(), slice.end());
        chrono::time_point<chrono::steady_clock> finish = timer.now();
        out << size << " entries sorted in "
            << (finish - start) / chrono::milliseconds(1) << "ms" <<
endl;
        slice.clear();
    }
}

void measureBubbleSort(vector<int> &data)
{
    size_t step = 100;
    vector<int> slice;
    ofstream out("bubble.txt");
    for (size_t size = step; size <= 10000; size += step) {
        copy(
            data.begin(),
            data.begin() + size,
            back_inserter(slice)
        );
        chrono::steady_clock timer;
        chrono::time_point<chrono::steady_clock> start = timer.now();
        bubbleSort(slice.begin(), slice.end());
        chrono::time_point<chrono::steady_clock> finish = timer.now();
        out << size << " entries sorted in "
            << (finish - start) / chrono::milliseconds(1) << "ms" <<
```

```
endl;
        slice.clear();
    }
}

void measureMergeSort(vector<int> &data)
{
    size_t step = 10000;
    vector<int> slice;
    ofstream out("merge.txt");
    for (size_t size = step; size <= data.size(); size += step) {
        copy(
            data.begin(),
            data.begin() + size,
            back_inserter(slice)
        );
        chrono::steady_clock timer;
        chrono::time_point<chrono::steady_clock> start = timer.now();
        mergeSort(slice.begin(), slice.end());
        chrono::time_point<chrono::steady_clock> finish = timer.now();
        out << size << " entries sorted in "
            << (finish - start) / chrono::milliseconds(1) << "ms" <<
endl;
        slice.clear();
    }
}

void measureShellSort(vector<int> &data)
{
    size_t step = 100;
    vector<int> slice;
    ofstream out("shell.txt");
    for (size_t size = step; size <= 10000; size += step) {
        copy(
            data.begin(),
            data.begin() + size,
            back_inserter(slice)
        );
        chrono::steady_clock timer;
        chrono::time_point<chrono::steady_clock> start = timer.now();
        shellSort(slice.begin(), slice.end());
        chrono::time_point<chrono::steady_clock> finish = timer.now();
        out << size << " entries sorted in "
            << (finish - start) / chrono::milliseconds(1) << "ms" <<
```

```
endl;
        slice.clear();
    }
}

void measureQuickSort(vector<int> &data)
{
    size_t step = 10000;
    vector<int> slice;
    ofstream out("quick.txt");
    for (size_t size = step; size <= data.size(); size += step) {
        copy(
            data.begin(),
            data.begin() + size,
            back_inserter(slice)
        );
        chrono::steady_clock timer;
        chrono::time_point<chrono::steady_clock> start = timer.now();
        quickSort(slice.begin(), slice.end());
        chrono::time_point<chrono::steady_clock> finish = timer.now();
        out << size << " entries sorted in "
            << (finish - start) / chrono::milliseconds(1) << "ms" <<
endl;
        slice.clear();
    }
}

void measureSTLSort(vector<int> &data)
{
    size_t step = 10000;
    vector<int> slice;
    ofstream out("stl.txt");
    for (size_t size = step; size <= data.size(); size += step) {
        copy(
            data.begin(),
            data.begin() + size,
            back_inserter(slice)
        );
        chrono::steady_clock timer;
        chrono::time_point<chrono::steady_clock> start = timer.now();
        sort(slice.begin(), slice.end());
        chrono::time_point<chrono::steady_clock> finish = timer.now();
        out << size << " entries sorted in "
            << (finish - start) / chrono::milliseconds(1) << "ms" <<
```

```
endl;
        slice.clear();
    }
}

int main()
{
    vector<int> data = readData();
    measureInsertionSort(data);
    measureSelectionSort(data);
    measureBubbleSort(data);
    measureMergeSort(data);
    measureShellSort(data);
    measureSTLSort(data);
    measureQuickSort(data);
    return 0;
}
```

Файл **src/sorts.hh**

```
#pragma once
#ifndef SORTS_HH
#define SORTS_HH

#include <functional>
#include <stdexcept>
#include <vector>

template<class RandIt, class Compare>
void insertionSort(RandIt first, RandIt last, Compare cmp)
{
    if (first >= last)
        throw std::range_error("First iterator equals or is behind the
last");
    RandIt shuttle, prev;
    for (RandIt cur = first + 1; cur < last; ++cur) {
        for (shuttle = cur, prev = cur - 1;
             shuttle != first && cmp(*shuttle, *(shuttle - 1));
             --shuttle, --prev)

            std::swap(*shuttle, *(shuttle - 1));
    }
```

```
}

template<class RandIt>
void insertionSort(RandIt first, RandIt last)
{
   insertionSort(first, last, std::less<typename
RandIt::value_type>());
}

template<class BiDirIt, class Compare>
void selectionSort(BiDirIt first, BiDirIt last, Compare cmp)
{
   if (first == last)
       throw std::range_error("First iterator equals the last");
   BiDirIt shuttle;
   BiDirIt most_extreme;
   for (BiDirIt unsorted = first; unsorted != last; ++unsorted) {
       most_extreme = unsorted;
       for (shuttle = unsorted; shuttle != last; ++shuttle)
           if (cmp(*shuttle, *most_extreme)) most_extreme = shuttle;

       if (most_extreme != unsorted) std::swap(*most_extreme,
*unsorted);
   }
}

template<class BiDirIt>
void selectionSort(BiDirIt first, BiDirIt last)
{
   selectionSort(first, last, std::less<typename
BiDirIt::value_type>());
}

template<class BidirIt, class Compare>
void bubbleSort(BidirIt first, BidirIt last, Compare cmp)
{
   if (first == last)
       throw std::range_error("First iterator equals the last");
   BidirIt cur, prev;
   bool sorted;
   do {
       sorted = true;
       cur = prev = first;
       for (++cur; cur != last; ++cur, ++prev) {
```

```
            if (cmp(*cur, *prev)) {
                std::swap(*cur, *prev);
                sorted = false;
            }
        }
    } while (!sorted);
}

template<class BidirIt>
void bubbleSort(BidirIt first, BidirIt last)
{
    bubbleSort(first, last, std::less<typename BidirIt::value_type>());

}

template<class RandIt, class Compare>
void mergeSort(RandIt first, RandIt last, Compare cmp)
{
    if (first >= last)
        throw std::range_error("First iterator equals or is behind the
last");
    const typename RandIt::difference_type len = last - first;
    if (len == 1) return;
    if (len == 2) {
        if (cmp(*(first + 1), *first)) std::swap(*(first + 1), *first);

        return;
    }
    RandIt mid = first + (len / 2 + len % 2);
    mergeSort(first, mid, cmp);
    mergeSort(mid, last, cmp);

    typename std::vector<typename RandIt::value_type> buffer;
    RandIt lcandidate = first, rcandidate = mid;
    while (lcandidate != mid && rcandidate != last) {
        buffer.push_back(
            cmp(*lcandidate, *rcandidate)
                ? *lcandidate++
                : *rcandidate++
        );
    }
    if (lcandidate == mid) lcandidate = rcandidate;
    while (lcandidate != last) buffer.push_back(*lcandidate++);
    lcandidate = first;
```

```
   typename std::vector<typename RandIt::value_type>::iterator it =
buffer.begin();
   while (lcandidate != last) *lcandidate++ = *it++;
}

template<class RandIt>
void mergeSort(RandIt first, RandIt last)
{
   mergeSort(first, last, std::less<typename RandIt::value_type>());
}

template<class RandIt, class Compare>
void shellSort(RandIt first, RandIt last, Compare cmp)
{
   if (first >= last)
       throw std::range_error("First iterator equals or is behind the
last");
   const typename RandIt::difference_type len = last - first;
   typename RandIt::difference_type gap = len;
   RandIt cur, pair;
   while (gap != 1) {
       gap /= 2;
       if (gap < 1) gap = 1;
       for (cur = first + gap; cur != last; ++cur) {
           for (pair = cur - gap; pair >= first; pair -= gap)
               if (cmp(*(pair + gap), *pair))
                   std::swap(*(pair + gap), *pair);
       }
   }
}

template<class RandIt>
void shellSort(RandIt first, RandIt last)
{
   shellSort(first, last, std::less<typename RandIt::value_type>());
}

template<class RandIt, class Compare>
void quickSort(RandIt first, RandIt last, Compare cmp)
{
   if (first >= last)
       throw std::range_error("First iterator equals or is behind the
last");
   const typename RandIt::difference_type len = last - first;
```

```
    if (len == 1) return;
    if (len == 2) {
        if (cmp(*(first + 1), *first)) std::swap(*(first + 1), *first);

        return;
    }
    RandIt pivot = first + len / 2;
    typename RandIt::value_type pivot_val = *pivot;
    RandIt left = first, right = last - 1;
    while (true) {
        while (left < last && cmp(*left, pivot_val)) ++left;
        while (right > first && cmp(pivot_val, *right)) --right;
        if (left >= right) {
            pivot = first == right ? right + 1 : right;
            break;
        }
        std::swap(*left, *right);
        ++left;
        --right;
    }
    quickSort(first, pivot, cmp);
    quickSort(pivot, last, cmp);
}

template<class RandIt>
void quickSort(RandIt first, RandIt last)
{
    quickSort(first, last, std::less<typename RandIt::value_type>());
}

#endif  // SORTS_HH
```