Лабораторная работа № 6. Внутренние сортировки

for, while, do-while, range based for loop, chrono

Указания:

1. Выполните задания, при этом алгоритмы сортировки, операцию обмена элементов, а также ввод и вывод массива оформите в виде отдельных функций. Разместите эти функции в отдельных файлах, например, sorting_algorithms.cpp, fiodata.cpp. Подготовьте соответствующий заголовочные файлы.
2. Для хранения данных рекомендуется использовать std::vector.
3. При вычислении времени работы алгоритмов убедитесь, что вы используете режим конфигурации Release, а не Debug. Во время режима Debug оптимизация обычно отключена, а она может оказывать значительное влияние на результаты.
4. Изучите код примеров: https://github.com/avelana/cpp-examples/tree/master/04-functions/DemoChrono , https://github.com/avelana/cpp-examples/tree/master/04-functions/HomeWork/BubbleSortWithTemplate
5. **Разработать Unit-тесты для тестирования работы алгоритмов сортировки.**

Задания:

Задание 1. Разработка алгоритмов внутренней сортировки (сортировки массивов)

Дан массив целых чисел $a[n]$, $n > 0$. Разработать приложение, которое выполняет сортировку массива **по возрастанию** с помощью различных алгоритмов сортировки.

1. Выполнить сортировку элементов массива с помощью «**сортировки вставками**» (см. Лекции).
2. Выполнить сортировку элементов массива с помощью «**сортировки Шелла**». Сортировка Шелла является *усовершенствованным вариантом сортировки вставками.* Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга (d). Иными словами – это сортировка вставками с предварительными «грубыми» проходами.
3. Выполнить сортировку элементов массива с помощью «**сортировки пузырьком**» (см. Лекции).
4. Выполнить сортировку элементов массива с помощью «**чётно-нечётной сортировки**». *На первом проходе элементы с нечётным ключом сравниваем с соседями, стоящими на чётных местах (1-й сравниваем со 2-м, затем 3-й с 4-м, 5-й с 6-м и так далее). Затем наоборот – элементы с четными индексами сравниваем/меняем с элементами, имеющими нечётные индексы. Затем снова «нечёт-чёт», потом опять «чёт-нечет». Процесс останавливается тогда, когда после подряд двух проходов по массиву («нечётно-чётному» и «чётно-нечётному») не произошло ни одного обмена.*
5. Выполнить сортировку элементов массива с помощью «**шейкерной сортировки**» (см. Лекции).

6. Выполнить сортировку элементов массива с помощью «**сортировки расчёской**». Сортировка расчёской является *усовершенствованным вариантом сортировки пузырьком.* Основная идея «расчёски» в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального.
7. Выполнить сортировку элементов массива с помощью «**сортировки выбором**» (см. Лекции).
8. Выполнить сортировку элементов массива с помощью «**двухсторонней сортировки выбором**». *Проходя по неотсортированной части массива, мы кроме максимума также попутно находим и минимум. Минимум ставим на первое место, максимум на последнее. Таким образом, неотсортированная часть при каждой итерации уменьшается сразу на два элемента. На первый взгляд кажется, что это ускоряет алгоритм в 2 раза – после каждого прохода неотсортированный подмассив уменьшается не с одной, а сразу с двух сторон. Но при этом в 2 раза увеличилось количество сравнений, а число обменов осталось неизменным.*
9. Выполнить сортировку элементов массива с помощью «**быстрой сортировки**» (см. Лекции).
10. Выполнить сортировку элементов массива с помощью «**сортировки слиянием**» (см. Лекции).

1	**вставкой**	void insertionSort (std::vector<T>& a)
2	**Шелла**	void shellSort (std::vector<T>& a)
3	**пузырьковая**	void bubbleSort (std::vector<T>& a)
4	**чет-нечет**	void oddEvenSort (std::vector<T>& a)
5	**шейкерная**	void cocktailSort (std::vector<T>& a)
6	**расческой**	void combSort (std::vector<T>& a)
7	**выбором**	void selectionSort (std::vector<T>& a)
8	**двусторонним выбором**	void doubleSelectionSort (std::vector<T>& a)
9	**быстрая сортировка**	void quickSort (std::vector<T>& a)
10	**сортировка слиянием**	void mergeSort (std::vector<T>& a)

Литература:

1. Кнут Д. Э. Искусство программирования. Том 3. Сортировка и поиск. – 2-е изд. – Москва: Вильямс, 2007. – Т. 3. – 832 с.
2. Седжвик, Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик. - К.: Издательство «ДиаСофт», 2001.- 688 с. 3. Идеи вышеперечисленных алгоритмов сортировки массивов описаны в статьях:
    a. https://habr.com/post/204600/
    b. https://habr.com/ru/post/422085/

c. https://habr.com/ru/post/415935/
d. https://habr.com/ru/post/414653/

Задание 2. Разработка функционала для сравнительного анализ алгоритмов сортировки

Дополнить приложение, разработанное ранее функционалом для сравнения производительности работы алгоритмов сортировки:

- оценить число операция сравнения и число операций обмена (перемещений) элементов (например, используйте глобальные переменные – счетчики, значение которых устанавливается в соответствующих функциях).
- время работы алгоритмов сортировки. Для оценки времени работы алгоритма используйте, например, библиотеку chrono (см. Страуструп, стр. 1102).

Зам. С помощью chrono нельзя вывести время и дату в читаемом виде. Можно, например, использовать time_t или использовать другую библиотеку, например date.h от разработчика chrono.

Литература:

1. Страуструп, Бьярне. Программирование: принципы и практика с использованием C++, 2-е изд. : Пер. с англ. - М.: ООО "И.Д.Вильямс", 2016. - 1328 с.
2. Функции для работы с датой и временем https://ru.cppreference.com/w/cpp/chrono
3. Основные концепции библиотеки chrono (C++) https://habr.com/ru/post/324984/
4. Библиотека date.h https://howardhinnant.github.io/date/date.html

Пример оценки времени выполнения алгоритма вычисления n-го числа Фибоначчи:

```
#include <iostream>
#include <chrono>

using namespace std::chrono;
 int fibonacci(int n) {   if (n < 3) return 1;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
  // ЗАСЕКАЕМ ВРЕМЯ НАЧАЛА РАБОТЫ АЛГОРИТМА

  // system_clock – представляет время системы.
  // system_clock::now() - возвращает момент времени (time_point), в который этот метод вызван.
  // std::chrono::time_point - это момент времени, который представляет продолжительность времени,
```

```
  // которое прошло с начала эпохи конкретных часов.


//std::chrono::time_point<std::chrono::system_clock>start=std::chrono::system_clock::now();
  // ЭТО ЖЕ, но короче:
  //auto start = std::chrono::system_clock::now();
  // ЕЩЕ короче, за счет использования namespace:   auto start = system_clock::now();

  // ВЫПОЛНЕНИЕ АЛГОРИТМА   int result = fibonacci(42);

  // ЗАСЕКАЕМ ВРЕМЯ ОКОНЧАНИЯ РАБОТЫ АЛГОРИТМА
  //std::chrono::time_point<std::chrono::system_clock> end=std::chrono::system_clock::now();    auto end = system_clock::now();

  // вычисляем продолжительность работы в секундах или милисекундах   auto elapsed_seconds =
duration_cast<seconds>(end - start).count();
  auto elapsed_milliseconds = duration_cast<milliseconds>(end - start).count();
  // Преобразование time_point в число, например для вывода на экран,
  // можно осуществить через C-тип time_t:   std::time_t end_time =
system_clock::to_time_t(end);
   std::cout << "Calculations are finished on " << std::ctime(&end_time)            << "Algorithm execution
time: " << elapsed_seconds << "s\n"
           << "Algorithm execution time: " << elapsed_milliseconds << "ms\n"; }
```

Результат:

```
Calculations are finished on Sun Sep 22 16:32:34 2019
Algorithm execution time: 1s
Algorithm execution time: 1545ms
```

Задание 3. Проведение эксперимента

1. Разработайте функции (и разместите их в отдельно файле data_generation.cpp, подготовьте заголовочный файл) для заполнения массива тестовыми данными. Сгенерированные данные необходимо записывать в файл, имя которого определяется на основе количества данных для проведения эксперимента: например: input_10_int_random_1.txt, input_1000_int_ascending.txt (или input_10_double_random_1.txt, input_1000_double_ascending.txt). Всего 20 файлов для разных случаев (автоматизируйте этот процесс).

2. Проведите экспериментальное сравнение производительности работы алгоритмов сортировки (Зам. *сравнение алгоритмов должно проводится* ***на одном и том же входном наборе*** *(элементы входного массива должны быть одинаковыми)*) для n = 10, 100, 1 000, 10 000 и следующем порядке входных элементов:

− элементы упорядочены по возрастанию.

− элементы упорядочены по убыванию.

− случайный набор элементов (3 разных случайных набора).

3. Результат каждой сортировки записать в выходной файл(ы).

4. Функции сортировки оформить с помощью шаблонов.

5. Результаты экспериментов оформить на основе нескольких запусков программы в виде сводных таблиц по образцу (**см. Lab06-Результаты эксперимента.xlsx**).