Отчёт

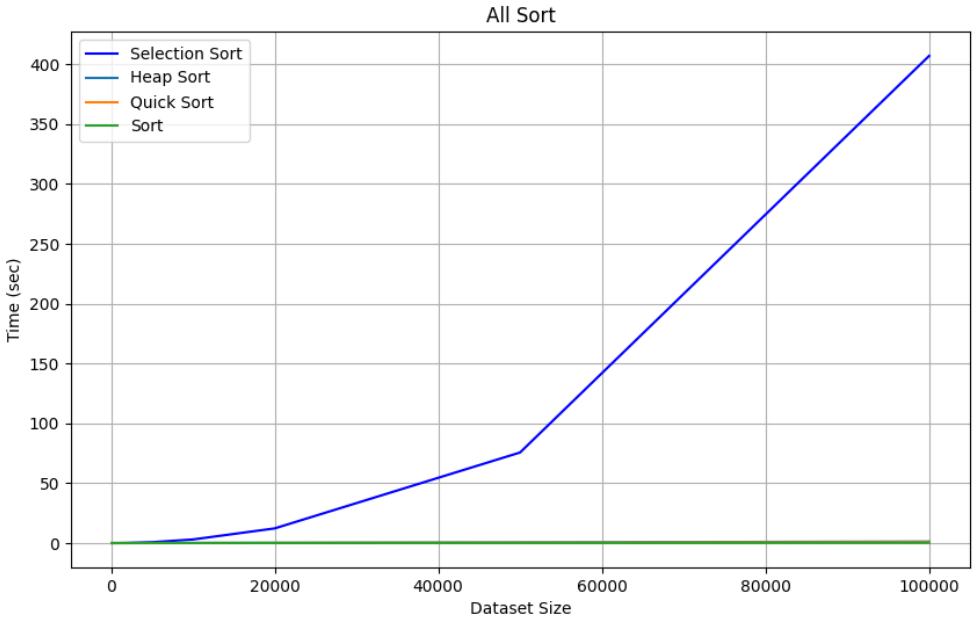
Лабораторная работа №1 по дисциплине
«Методы программирования»

ФИО: Гриднева Екатерина Владимировна
Группа: СКБ222
Вариант: 4

Ссылка на исходный код программы: https://github.com/emokater/sort-benchmark.git

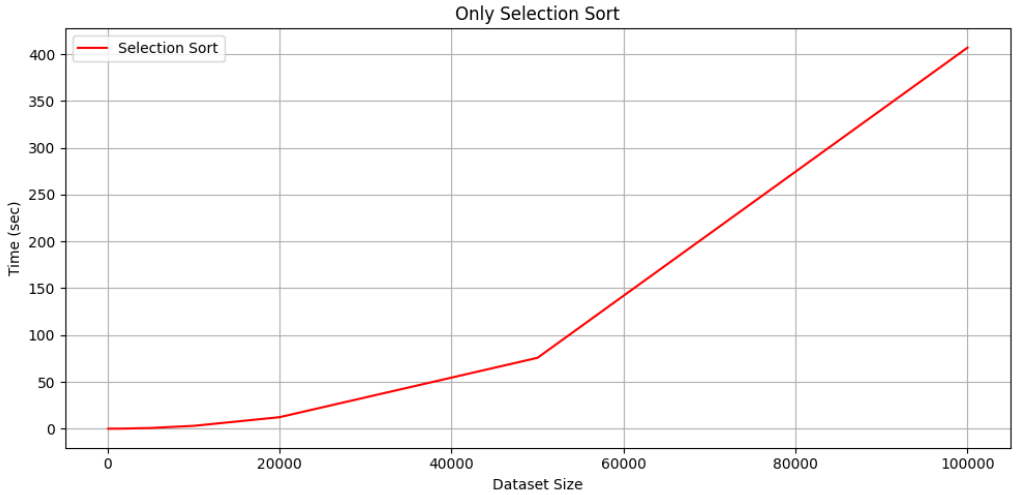
Анализ графиков.

1. **График «All Sort»**. На нём изображено сравнение времени выполнения всех четырёх алгоритмов сортировки в зависимости от размера входного массива.



- ***Сортировка выбором (Selection Sort).***

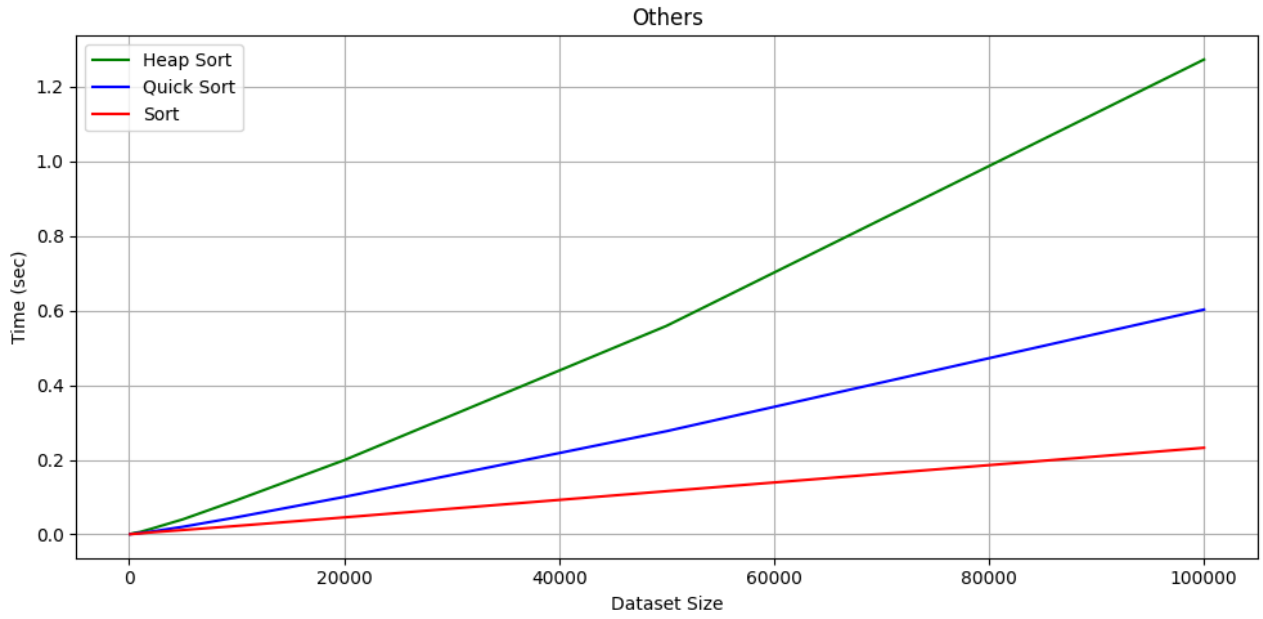
Очевидно из графика, что сильно отстаёт от других алгоритмов даже на сравнительно малых размерах массивов. Ожидаемо, ведь его сложность $O(n^2)$ (по сравнению со сложностью других алгоритмов ($O(n \log n)$)).



- ***Пирамидальная сортировка (Heap Sort), быстрая сортировка (Quick Sort) и встроенная функция сортировки (std::sort).***

Остальные алгоритмы показывают намного лучшую работу, но на этом графике их невозможно адекватно сравнить из-за разницы в масштабах. Поэтому рассмотрим отдельный график без сортировки выбором.

2. График «Others». Этот график показывает время выполнения уже трёх алгоритмов сортировки.



- Красная линия всегда ниже двух других => **std::sort** — самый быстрый. Это логично, так как std::sort — гибрид Quick Sort + Heap Sort + Insertion Sort.
- Синяя линия всегда ниже зелёной => **Quick Sort** работает быстрее **Heap Sort.**
- **Heap Sort** — самый медленный из трёх, не смотря на то, что сложность у всех трёх алгоритмов одинаковая (O(n log n)). Наверно потому, что требует больше операций по поддержанию кучи.

Вывод:

Проведённый эксперимент подтверждает теоретические оценки сложности алгоритмов сортировки.

Selection Sort показал крайне низкую производительность уже при небольших объёмах данных, что делает его непригодным для практического использования при размерах массива более нескольких тысяч элементов.

Остальные алгоритмы показали близкую к линейно-логарифмической производительность, однако *std::sort* оказался самым быстрым за счёт своей гибридной природы.

Quick Sort продемонстрировал лучшую производительность среди «ручных» реализаций, в то время как *Heap Sort* оказался самым медленным из тройки «эффективных».

Таким образом, для практического применения рекомендуется использовать *std::sort*, либо *Quick Sort* при необходимости собственной реализации.

Laboratory Work №1

Generated by Doxygen 1.13.2

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 Flower Class Reference

The Flower class contains information about a flower (name, color, scent intensity, and habitat regions).

`#include <flower.h>`

Public Member Functions

- bool EqFlowers (const Flower &other) const

  Compares the current object with another based on key fields.

Constructors

Default constructor.

- **Flower** (string name, string color, string smell, vector< string > regions)

  Primary constructor of the Flower class.
- **Flower** (const Flower &other)=default

  Copy constructor.
- **Flower** (Flower &&other)=default

  Move constructor.
- ~**Flower** ()=default

  Destructor.

Getters

- string **GetName** () const
- string **GetColor** () const
- string **GetSmell** () const
- vector< string > **GetRegions** () const

Setters

- void **SetName** (string name)
- void **SetColor** (string color)
- void **SetSmell** (string smell)
- void **SetRegions** (vector< string > regions)

Operator Overloading

Comparison based on key fields (name, color, smell).

- bool **operator>** (const Flower &other) const
  "Greater than" operator based on key fields.
- bool **operator<** (const Flower &other) const
  "Less than" operator based on key fields.
- bool **operator>=** (const Flower &other) const
  "Greater than or equal to" operator based on key fields.
- bool **operator<=** (const Flower &other) const
  "Less than or equal to" operator based on key fields.
- Flower & **operator=** (const Flower &other)
  Copy assignment operator.

3.1.1 Member Function Documentation

3.1.1.1 EqFlowers()

```
bool Flower::EqFlowers (
            const Flower & other) const
```

Parameters

other	Reference to another Flower object.

Returns

true if name, color, and smell match; otherwise false.

Two objects are considered "equal" if their name, color, and smell fields are the same. The regions field is ignored in this comparison.

The documentation for this class was generated from the following files:

- flower.h
- flower.cpp

Chapter 4

File Documentation

4.1 flower.h File Reference

Declaration of the Flower class.

```
#include <string>
#include <vector>
```

Classes

- class Flower

  The Flower class contains information about a flower (name, color, scent intensity, and habitat regions).

4.2 flower.h

Go to the documentation of this file.

```
00001
00003
00004 #ifndef FLOWER_H
00005 #define FLOWER_H
00006
00007 #include <string>
00008 #include <vector>
00009
00010 using namespace std;
00011
00013 class Flower {
00014 public:
00018     Flower() = default;
00019     Flower(string name, string color, string smell, vector<string> regions);
00021     Flower(const Flower& other) = default;
00023     Flower(Flower&& other) = default;
00025     ~Flower() = default;
00027
00030     string GetName() const { return name_; }
00031     string GetColor() const { return color_; }
00032     string GetSmell() const { return smell_; }
00033     vector<string> GetRegions() const { return regions_; }
00035
00038     void SetName(string name) { name_ = name; }
00039     void SetColor(string color) { color_ = color; }
00040     void SetSmell(string smell) { smell_ = smell; }
00041     void SetRegions(vector<string> regions) { regions_ = regions; }
00043
00047     bool operator>(const Flower& other) const;
```

```
00048     bool operator<(const Flower& other) const;
00049     bool operator>=(const Flower& other) const;
00050     bool operator<=(const Flower& other) const;
00051     Flower& operator=(const Flower& other);
00053
00060     bool EqFlowers(const Flower& other) const;
00061
00062 private:
00063     string name_;
00064     string color_;
00065     string smell_;
00066     vector<string> regions_;
00067 };
00068
00069 #endif
```

4.3 io.h File Reference

Interface for input/output functions: parsing CSV files and saving sorting results.

```
#include "flower.h"
#include <string>
#include <vector>
```

Functions

- vector< Flower > parserCSV (string filename)

  Reads a CSV file and returns a vector of Flower objects.
- void saveRes (const std::vector< Flower > &source, long size)

  Saves a sorted array of flowers to a CSV file and measures the time.

4.3.1 Function Documentation

4.3.1.1 parserCSV()

```
vector< Flower > parserCSV (
            string filename)
```

Parameters

filename	Path to the input CSV file.

Exceptions

runtime_error	if the file cannot be opened.

Returns

Vector of Flower objects loaded from the file.

The function opens the given CSV file and discards the first line (assumed to be a header).
Each subsequent line must contain exactly four comma-separated fields:

1. name
2. color
3. smell
4. regions — a list of one or more region names enclosed in square brackets, e.g. `[Region1,Region2,...]`

Internally, the parser locates the first three commas to extract the name, color and smell fields.
It then strips the surrounding brackets from the remaining substring and splits it on commas to obtain each region.
A `Flower` is constructed with these values and appended to the result vector.
If the file contains no data lines (only a header or is empty), an empty vector is returned.

4.3.1.2 saveRes()

```
void saveRes (
            const std::vector< Flower > & source,
            long size)
```

Parameters

source	Constant reference to the vector of Flower objects.
size	Size of the array.

Exceptions

runtime_error	In case of a file write error.

The function performs four sorts on copies of the input data:

- Selection sort
- Heap sort
- Quick sort
- std::sort (introsort)

It first creates four separate vectors (tmp1–tmp4) from `source` so that each algorithm works on the same initial dataset. It then builds a base filepath using the provided `size`, and appends algorithm-specific suffixes:

```
sorted_data/<size>_selectSort.txt
sorted_data/<size>_heapSort.txt
sorted_data/<size>_quickSort.txt
sorted_data/<size>_sort.txt
```

A master timing log ("info_time.txt") is opened in append mode. For each algorithm:

1. The function records the start time using std::chrono::high_resolution_clock.
2. It invokes the sort (selectSort, heapSort, quickSort, or std::sort).
3. It records the end time and computes the elapsed duration in seconds.
4. It writes a labeled line with the algorithm number and elapsed time to the timing log.
5. It opens the corresponding output file and writes each Flower in sorted order, one per line, with fields separated by spaces:
   ```
   Name Color Smell Region1 Region2 ...
   ```
6. Closes the file before proceeding to the next algorithm.

If opening or writing to any of the files fails, the function throws a std::runtime_error indicating which path could not be accessed.

4.4 io.h

Go to the documentation of this file.

```
00001
00006
00007 #ifndef IO_H
00008 #define IO_H
00009
00010 #include "flower.h"
00011 #include <string>
00012 #include <vector>
00013
00031 vector<Flower> parserCSV(string filename);
00032
00073 void saveRes(const std::vector<Flower>& source, long size);
00074
00075 #endif
```

4.5 sorts.h File Reference

Implementation of various sorting algorithms (selection sort, heap sort, and quick sort), as well as the auxiliary function mySwap.

Functions

- template<typename T>
  void mySwap (T &x, T &y)

  Auxiliary swap function.
- template<class T>
  void selectSort (T data[ ], long size)

  Selection Sort algorithm.
- template<class T>
  void downHeap (T data[ ], long k, long n)

  "Heapify" operation for heap.
- template<class T>
  void heapSort (T data[ ], long size)

  Heap Sort algorithm.
- template<class T>
  void quickSort (T *data, long n)

  Quick Sort algorithm.

4.5.1 Function Documentation

4.5.1.1 downHeap()

```
template<class T>
void downHeap (
            T data[],
            long k,
            long n)
```

Template Parameters

T	Any type supporting > operator.

Parameters

data	Heap array with n elements.
k	Index of the root of the subheap.
n	Size of the array.

4.5.1.2 heapSort()

```
template<class T>
void heapSort (
            T data[],
            long size)
```

Template Parameters

T	Any type supporting > operator.

Parameters

data	Array of size elements.
size	Number of elements.

4.5.1.3 mySwap()

```
template<typename T>
void mySwap (
            T & x,
            T & y)
```

Parameters

x	First array element.
y	Second array element.

4.5.1.4 quickSort()

```
template<class T>
void quickSort (
             T * data,
             long n)
```

Template Parameters

T	Any type supporting <, > and assignment.

Parameters

data	Pointer to the first element of the array.
n	Number of elements.

4.5.1.5 selectSort()

```
template<class T>
void selectSort (
             T data[],
             long size)
```

Template Parameters

T	Any type supporting < and = operators.

Parameters

data	Array of size elements.
size	Number of elements.

4.6 sorts.h

Go to the documentation of this file.

```
00001
00003
00004 #ifndef SORTS_H
00005 #define SORTS_H
00006
00010 template <typename T> void mySwap(T& x, T& y) {
00011     T temp = x;
00012     x = y;
00013     y = temp;
00014 }
00015
00020 template<class T> void selectSort(T data[], long size) {
00021     T x; // min el
00022     long k; // his index
00023     for (long i = 0; i < size - 1; ++i) {
00024         x = data[i];
00025         k = i;
00026         for (long j = i + 1; j < size; ++j) {
00027             if (data[j] < x) {
00028                 k = j;
```

```
00029                 x = data[j];
00030             }
00031         }
00032
00033         data[k] = data[i];
00034         data[i] = x;
00035     }
00036 }
00037
00043 template<class T> void downHeap(T data[], long k, long n) {
00044     while (true) {
00045         long left = 2 * k + 1;
00046         long right = 2 * k + 2;
00047         long largest = k;
00048
00049         if (left < n && data[left] > data[largest]) {
00050             largest = left;
00051         }
00052
00053         if (right < n && data[right] > data[largest]) {
00054             largest = right;
00055         }
00056
00057         if (largest == k) { break; }
00058
00059         mySwap(data[k], data[largest]);
00060         k = largest;
00061     }
00062 }
00063
00068 template<class T> void heapSort(T data[], long size) {
00069     long i;
00070
00071     for(i = size/2 - 1; i >= 0; --i) {
00072         downHeap(data, i, size - 1);
00073     }
00074
00075     for(i = size - 1; i > 0; --i) {
00076         mySwap(data[i], data[0]);
00077         downHeap(data, 0, i - 1);
00078     }
00079 }
00080
00085 template<class T> void quickSort(T* data, long n) {
00086     long i = 0, j = n - 1;
00087     T p = data[n » 1];
00088
00089     while (i <= j) {
00090         while (data[i] < p) { i++; }
00091         while (data[j] > p) { j--; }
00092
00093         if (i <= j) {
00094             mySwap(data[i], data[j]);
00095             i++;
00096             j--;
00097         }
00098     }
00099
00100     if (i < n) {
00101         quickSort(data + i, n - i);
00102     }
00103
00104     if (j > 0) {
00105         quickSort(data, j + 1);
00106     }
00107 }
00108
00109 #endif
```

4.7 flower.cpp File Reference

Implementation of the Flower class methods and operator overloading.

```
#include "../headers/flower.h"
```

4.8 io.cpp File Reference

Implementation of functions for parsing and exporting the sorted array to a file.

```
#include "../headers/io.h"
#include "../headers/sorts.h"
#include <fstream>
#include <chrono>
#include <algorithm>
#include <stdexcept>
```

Functions

- vector< Flower > parserCSV (string filename)

  Reads a CSV file and returns a vector of Flower objects.
- void saveRes (const std::vector< Flower > &source, long size)

  Saves a sorted array of flowers to a CSV file and measures the time.

4.8.1 Function Documentation

4.8.1.1 parserCSV()

```
vector< Flower > parserCSV (
            string filename)
```

Parameters

filename	Path to the input CSV file.

Exceptions

runtime_error	if the file cannot be opened.

Returns

Vector of Flower objects loaded from the file.

The function opens the given CSV file and discards the first line (assumed to be a header).
Each subsequent line must contain exactly four comma-separated fields:

1. name
2. color
3. smell
4. regions — a list of one or more region names enclosed in square brackets, e.g. `[Region1,Region2,...]`

Internally, the parser locates the first three commas to extract the name, color and smell fields.
It then strips the surrounding brackets from the remaining substring and splits it on commas to obtain each region.
A `Flower` is constructed with these values and appended to the result vector.
If the file contains no data lines (only a header or is empty), an empty vector is returned.

4.8.1.2 saveRes()

```
void saveRes (
            const std::vector< Flower > & source,
            long size)
```

Parameters

source	Constant reference to the vector of Flower objects.
size	Size of the array.

Exceptions

runtime_error	In case of a file write error.

The function performs four sorts on copies of the input data:

- Selection sort
- Heap sort
- Quick sort
- std::sort (introsort)

It first creates four separate vectors (tmp1–tmp4) from `source` so that each algorithm works on the same initial dataset. It then builds a base filepath using the provided `size`, and appends algorithm-specific suffixes:

```
sorted_data/<size>_selectSort.txt
sorted_data/<size>_heapSort.txt
sorted_data/<size>_quickSort.txt
sorted_data/<size>_sort.txt
```

A master timing log ("info_time.txt") is opened in append mode. For each algorithm:

1. The function records the start time using std::chrono::high_resolution_clock.
2. It invokes the sort (selectSort, heapSort, quickSort, or std::sort).
3. It records the end time and computes the elapsed duration in seconds.
4. It writes a labeled line with the algorithm number and elapsed time to the timing log.
5. It opens the corresponding output file and writes each Flower in sorted order, one per line, with fields separated by spaces:
   ```
   Name Color Smell Region1 Region2 ...
   ```
6. Closes the file before proceeding to the next algorithm.

If opening or writing to any of the files fails, the function throws a std::runtime_error indicating which path could not be accessed.

4.9 main.cpp File Reference

Entry point: reading multiple datasets, sorting them, and saving the results.

```
#include "../headers/io.h"
#include <string>
#include <vector>
```

Functions

- int **main** ()

4.9.1 Detailed Description

Iterates through CSV files with different array sizes specified in the `sizes` array. For each file:

1. Calls parserCSV to load data into a vector of Flower objects.
2. Calls saveRes to sort the data, save the result, and measure execution time.

4.10 genGraphs.py File Reference

Parses timing data and plots performance of sorting algorithms.

Functions

- genGraphs.parse_timing_file (filepath)
  Read the timing log and fill the data dict.
- genGraphs.plotAll (df)
  Plot sorting algorithms on one graph.
- **genGraphs.plotSelectionSort** (data)
- **genGraphs.plotOthers** (data)

Variables

- dict genGraphs.data
  Container for parsed timing data.
- **genGraphs.df** = pd.DataFrame(data)

4.10.1 Function Documentation

4.10.1.1 parse_timing_file()

```
genGraphs.parse_timing_file (
              filepath)
```

```
\details
Opens the file at \p filepath, reads each line and:
    - If it starts with 'Datasets', extracts dataset size.
    - If it starts with '1', '2', '3' or '4', extracts the corresponding time.
Populates and returns a dict with keys "Size", "Selection Sort", "Heap Sort", "Quick Sort", "Sort".
```

4.10.1.2 plotAll()

```
genGraphs.plotAll (
              df)
```

Parameters

df	pandas DataFrame with timing data.

4.10.2 Variable Documentation

4.10.2.1 data

`genGraphs.data`

Initial value:

```
00001 =  {
00002     "Size": [],
00003     "Selection Sort": [],
00004     "Heap Sort": [],
00005     "Quick Sort": [],
00006     "Sort": []
00007 }
```

Index