МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

Інститут **КНІТ**
Кафедра **ПЗ**

ЗВІТ

До лабораторної роботи № 1
На тему: "*Метод сортування бульбашкою*"
З дисципліни: "Алгоритми та структури даних"

Лектор:
доцент кафедри ПЗ
Коротєєва Т.О.

Виконав:
студент групи ПЗ-22
Коваленко Д.М.

Прийняв:
асистент кафедри ПЗ
Франко А.В.

«_____» _____ 2022 р.
$\sum$ = _____.....

Львів — 2022

Тема. Метод сортування бульбашкою.

Мета. Вивчити алгоритм сортування бульбашкою. Здійснити програмну реалізацію алгоритму сортування бульбашкою. Дослідити швидкодію алгоритму сортування бульбашкою.

Лабораторне завдання

Створити віконний проект та написати програму, яка реалізує алгоритм сортування бульбашкою.

2. Задано перелік товарів у супермаркеті та їх ціни. Упорядкувати за алфавітом лише ті товари, вартість яких не перевищує середню.

Теоретичні відомості

Загальна задача сортування полягає в наступному: нехай дано множину елементів, яка є індексованою, тобто довільно пронумерованою від 1 до n. Необхідно індексувати цю множину елементів так, щоб з умови $i < j$ витікало $a_i <_j$ - для всіх $i, j = 1..n$. Отже, процес сортування полягає у послідовних перестановках елементів доти, доки їх індексація не узгодиться з їх впорядкованістю.

Алгоритм сортування бульбашкою (англійською «Bubble Sort») відноситься до класу алгоритмів сортування вибіркою. Алгоритм працює наступним чином — у заданому наборі даних (списку чи масиві) порівнюються два сусідні елементи. Якщо один з елементів не відповідає критерію сортування (є більшим, або ж, навпаки, меншим за свого сусіда), то ці два елементи обмінюються місцями. Прохід по масиву продовжується до тих пір, доки дані не будуть відсортованими. Алгоритм отримав свою назву від того, що процес сортування згідно нього нагадує поведінку бульбашок повітря у резервуарі з водою: найлегша бульбашка піднімається до гори першою. Оскільки для роботи з елементами масиву алгоритм використовує лише порівняння, це сортування на основі порівнянь.

Цей метод неефективний для масивів великого розміру. Існує багато модифікацій даного алгоритму.

Покроковий опис роботи алгоритму сортування бульбашкою

Алгоритм В Задано масив елементів R_1 ,R_2 ,...,R_n. F – прапорець перестановок.
Даний алгоритм реорганізує масив у висхідному порядку, тобто для його елементів буде мати місце співвідношення $R_i < R_j$ - для всіх $i, j = 1..n$.
В1. Цикл за індексом проходження. Повторювати кроки В2 і В3 при $i = 1..n - 1$.
В2. Ініціалізація прапорця перестановки: встановити $F = 0$.
В3. Виконання проходження. Повторювати при $j = 1, 2, \ldots, n - i$: якщо $R_{j+1} < R_j$, то встановити F=1 та переставити місцями елементи $R_j <-> R_j + 1$, ; якщо $F = 0$, то завершити виконання алгоритму.
В4. Кінець. Вихід.

Хід роботи

Файл sort.rs

```
use crate::data::Data;

pub struct Sorted;

impl Sorted {
    pub fn sort(v: &mut Vec<Data>) -> Vec<(usize, usize)> {
        remove_above_avg_price(v);
        let mut sorted = false;
        let mut res = Vec::new();
        for i in 0..v.len() {
            for j in 0..(v.len() - 1 - i) {
                sorted = true;
                if v[j] > v[j + 1] {
                    v.swap(j, j + 1);
                    res.push((j, j + 1));
                    sorted = false;
                }
```

```
            }
            if sorted { break }
        }
        res
    }
}

fn remove_above_avg_price(v: &mut Vec<Data>) {
    let avg_price = v.iter().map(|d| d.price).sum::<f32>() / v.len() as f32;
    v.retain(|d| d.price < avg_price);
}
```

Файл data.rs

```
use fake::{faker::lorem::en::*, Dummy, Fake};

#[derive(Debug, Clone, PartialEq, Dummy)]
pub struct Data {
    #[dummy(faker = "Word()")]
    pub name: String,
    pub price: f32,
}

impl PartialOrd for Data {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        self.name.partial_cmp(&other.name)
    }
}
```

Файл main.rs

```
mod data;
mod sort;

use eframe::egui;
use egui_extras::{Size, TableBuilder};

use crate::{data::Data, sort::Sorted};

fn main() {
    let options = eframe::NativeOptions::default();
    eframe::run_native(
    "Sorting",
    options,
    Box::new(|_cc| Box::new(SortingApp::new())),
    );
}

struct SortingApp {
    data: Option<Vec<Data>>,
    prev_data: Option<Vec<Data>>,
    data_len: usize,
    selected: Option<Vec<(usize, usize)>>,
    elapsed: Option<std::time::Duration>,
}

impl SortingApp {
    fn new() -> Self {
        Self {
```

```
            data: Some(fake::vec![Data; 1]),
            prev_data: None,
            data_len: 1,
            selected: None,
            elapsed: None,
        }
    }
}

impl eframe::App for SortingApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        egui::CentralPanel::default().show(ctx, |ui| {
            if ui
            .add(egui::DragValue::new(&mut self.data_len).speed(0.1))
            .changed
            {
                self.data = Some(fake::vec![Data; self.data_len])
            }
            ui.end_row();
            if ui.button("Sort").clicked() {
                self.prev_data = self.data.to_owned();
                match &mut self.data {
                    Some(d) => {
                        use std::time::Instant;
                        let now = Instant::now();
                        self.selected = Some(Sorted::sort(d));
                        self.data_len = d.len();
                        self.elapsed = Some(now.elapsed());
                    }
                    None => {}
                };
            }
            if let Some(elapsed) = self.elapsed {
                ui.label(format!("{:.2?}", elapsed).to_string());
            }
            ui.end_row();
            egui::Grid::new("unique_id")
            .min_col_width(290.)
            .show(ui, |ui| {
                ui.label("after");
                ui.label("before");
                ui.label("Indexes");
                ui.end_row();
            });
            egui::Grid::new("another_unique_id")
            .min_row_height(500.)
            .show(ui, |ui| {
                ui.push_id(1, |ui| {
                    TableBuilder::new(ui)
                    .striped(true)
                    .cell_layout(egui::Layout::left_to_right(egui::Align::Center))
                    .column(Size::initial(20.0).at_least(40.0))
                    .column(Size::initial(150.0).at_least(40.0))
                    .column(Size::initial(100.0).at_least(40.0))
                    .body(|mut body| {
                        if let Some(data) = self.data.as_ref() {
                            for i in 0..data.len() {
                                body.row(30., |mut row| {
                                    row.col(|ui| {
                                        ui.label(i.to_string());
                                    });
```

```
                        row.col(|ui| {
                            if let Some(v) = data.get(i) {
                                ui.label(v.name.clone());
                            }
                        });
                        row.col(|ui| {
                            if let Some(v) = data.get(i) {
                                ui.label(v.price.to_string());
                            }
                        });
                    });
                }
            }
        });
    });

    ui.push_id(2, |ui| {
        TableBuilder::new(ui)
        .striped(true)
        .cell_layout(egui::Layout::left_to_right(egui::Align::Center))
        .column(Size::initial(20.0).at_least(40.0))
        .column(Size::initial(150.0).at_least(40.0))
        .column(Size::initial(100.0).at_least(40.0))
        .body(|mut body| {
            if let Some(data) = self.prev_data.as_ref() {
                for i in 0..data.len() {
                    body.row(30., |mut row| {
                        row.col(|ui| {
                            ui.label(i.to_string());
                        });
                        row.col(|ui| {
                            if let Some(v) = data.get(i) {
                                ui.label(v.name.clone());
                            }
                        });
                        row.col(|ui| {
                            if let Some(v) = data.get(i) {
                                ui.label(v.price.to_string());
                            }
                        });
                    });
                }
            }
        });
    });

    ui.push_id(3, |ui| {
        TableBuilder::new(ui)
        .striped(true)
        .cell_layout(egui::Layout::left_to_right(egui::Align::Center))
        .column(Size::initial(40.0).at_least(40.0))
        .column(Size::initial(40.0).at_least(40.0))
        .body(|mut body| {
            if let Some(selected) = self.selected.as_ref() {
                for i in 0..selected.len() {
                    body.row(30., |mut row| {
                        row.col(|ui| {
                            if let Some(v) = selected.get(i) {
                                ui.label(v.0.to_string());
                            }
                        });
```

```
                                    row.col(|ui| {
                                        if let Some(v) = selected.get(i) {
                                            ui.label(v.1.to_string());
                                        }
                                    });
                                });
                            }
                        }
                    });
                });
            });
        }
    }
}
```

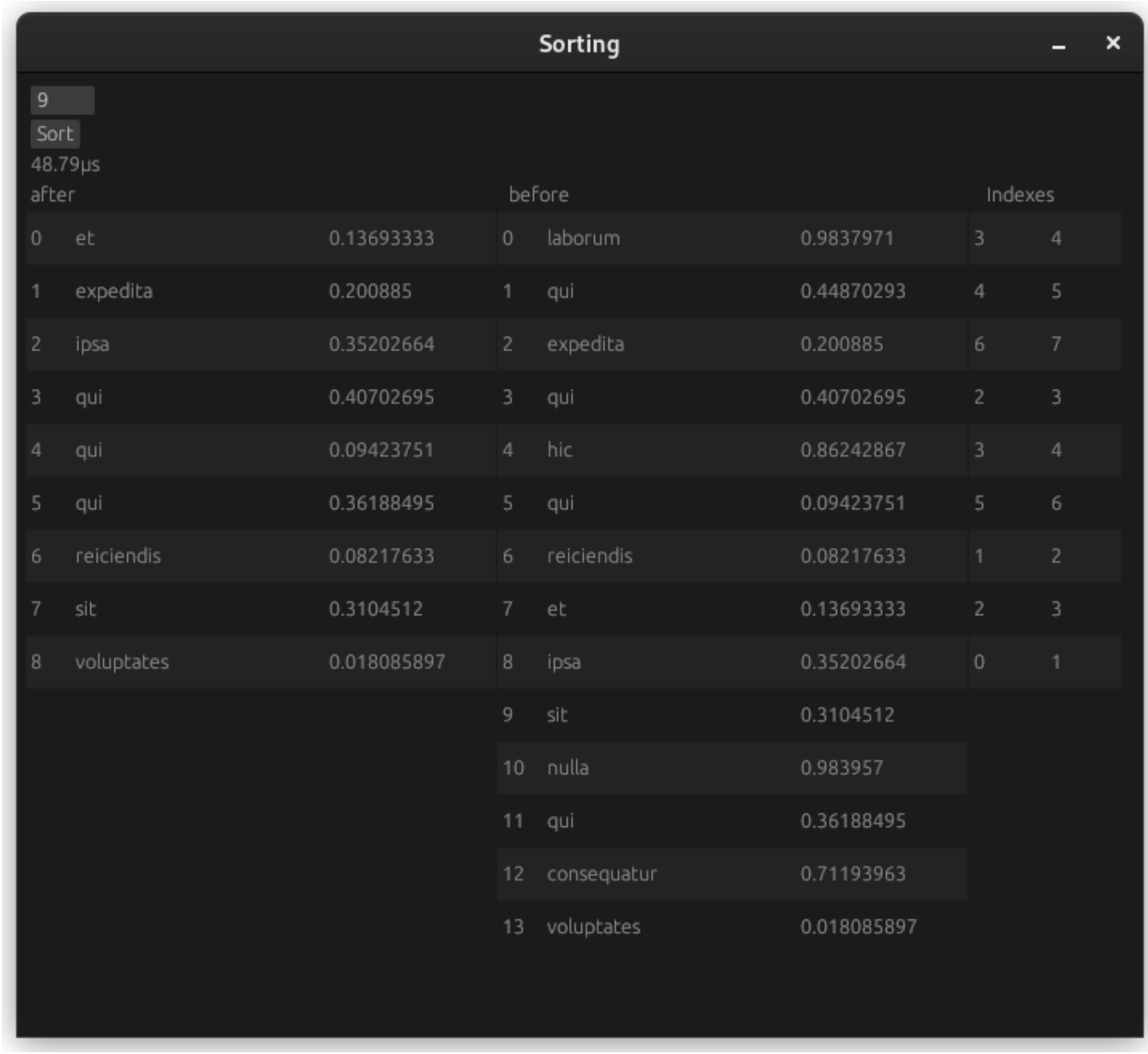

Рис. 1: Виконання програми

Висновок

Під час виконання лабораторної роботи я навчився використовувати алгоритм сортування бульбашкою. Програмно реалізував алгоритм та дослідив його швидкодію.