Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп'ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №11

з навчальної дисципліни «Алгоритми та структури даних»

Варіант 5

Виконав студент:

Омельніцький Андрій Миколайович
Група: КН-1023б

Перевірив:

Старший викладач

Бульба Сергій Сергійович

Харків - 2024

Зміст

1 Мета роботи

Закріпити теоретичні знання та набути практичного досвіду впорядкування набору статичних та динамічних структур даних.
Теми для попередньої роботи:

- масиви та списки;
- класифікація алгоритмів сортування;
- алгоритми сортування вибором (вибіркою) та включенням.

2 Завдання

Написати програму, що реалізує сортування статичного та/або динамічного набору даних заданим способом згідно з даними табл. 11.1.

Визначити кількість порівнянь та обмінів для наборів даних, що містять різну кількість елементів (20, 1000, 5000, 10000, 50000).

Оцінити час сортування. Дослідити вплив початкової впорядкованості набору даних (відсортований, відсортований у зворотному порядку, випадковий).

3 Хід виконання

Для виконання завдання було обрано мову Rust. Увесь код також додатково був розміщений в GitHub репозитарії: https://github.com/blackgolyb/algos-labs.

3.1 Підготовка до виконання

Для подальшої роботи підготуємо набір функцій, інтерфейсів та макросів.

3.1.1 Logger

Створимо структуру Logger та Metrics. Logger буде відповідати за логування сортування та по закінченню роботи буде видавати Metrics. Metrics в свою чергу буде зберігати в собі результати логування сортування.

```
use std::fmt::Display;

pub struct Metrics {
    pub swaps: i64,
    pub compares: i64,
    pub time: std::time::Duration,
}

impl Display for Metrics {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "Comparisons: {}, Swaps: {}, Execution Time: {:?}",
            self.compares, self.swaps, self.time
        )
    }
}

pub struct Logger {
    swaps: i64,
    compares: i64,
    time: std::time::Instant,
    result_time: std::time::Duration,
}

impl Logger {
    pub fn new() -> Self {
        Self {
            swaps: 0,
            compares: 0,
            time: std::time::Instant::now(),
            result_time: std::time::Duration::new(0, 0),
        }
    }

    pub fn get_metrics(&self) -> Metrics {
        Metrics {
            swaps: self.swaps,
            compares: self.compares,
            time: self.result_time,
        }
    }

    pub fn log_compare(&mut self) {
        self.compares += 1;
    }

    pub fn log_swap(&mut self) {
        self.swaps += 1;
    }

    pub fn start(&mut self) {
        self.time = std::time::Instant::now();
        self.compares = 0;
        self.swaps = 0;
    }

    pub fn end(&mut self) {
        self.result_time = self.time.elapsed();
    }
```

```
61
62      pub fn marge_sub_logger(&mut self, sub_logger: &mut Self) {
63          self.compares += sub_logger.compares;
64          self.swaps += sub_logger.swaps;
65      }
66  }
67
68  pub(super) struct LoggedVec<'a, T> {
69      vec: Vec<T>,
70      logger: &'a mut Logger,
71  }
```

3.1.2 Інтерфейси

Також створимо інтерфейси які повинен буде реалізувати наше сортування: Sort, SortLogging.

```
1   use super::logger::{Logger, Metrics};
2
3   pub trait AsIndex {
4       fn as_index(&self) -> usize;
5   }
6
7   pub trait SortLogging {
8       fn logger(&mut self) -> &mut Logger;
9
10      fn get_metrics(&mut self) -> Metrics {
11          self.logger().get_metrics()
12      }
13
14      fn log_swap(&mut self) {
15          self.logger().log_swap();
16      }
17      fn log_compare(&mut self) {
18          self.logger().log_compare();
19      }
20  }
21
22  pub trait Sort<T>
23  where
24      T: Ord,
25      Self: SortLogging,
26  {
27      fn sort(&mut self, arr: &mut Vec<T>);
28
29      fn swap(&mut self, arr: &mut Vec<T>, a: usize, b: usize) {
30          self.logger().log_swap();
31          arr.swap(a, b)
32      }
33      fn eq(&mut self, arr: &Vec<T>, a: usize, b: usize) -> bool {
34          self.logger().log_compare();
35          arr[a] == arr[b]
36      }
37      fn neq(&mut self, arr: &Vec<T>, a: usize, b: usize) -> bool {
38          self.logger().log_compare();
39          arr[a] != arr[b]
40      }
41      fn gt(&mut self, arr: &Vec<T>, a: usize, b: usize) -> bool {
42          self.logger().log_compare();
43          arr[a] > arr[b]
44      }
45      fn lt(&mut self, arr: &Vec<T>, a: usize, b: usize) -> bool {
46          self.logger().log_compare();
47          arr[a] < arr[b]
48      }
49      fn gte(&mut self, arr: &Vec<T>, a: usize, b: usize) -> bool {
50          self.logger().log_compare();
51          arr[a] >= arr[b]
52      }
53      fn lte(&mut self, arr: &Vec<T>, a: usize, b: usize) -> bool {
54          self.logger().log_compare();
```

```
        arr[a]  <= arr[b]
    }
}
```

3.1.3 Макроси

Для зручного створення сортування напишемо пару макросів які будуть додавати зручний DSL. Так як більшість сортування можна описати функцією яка приймає масив по змінювальному посиланню та сортує всередені цього масива. То ми можемо звести створення сортуваня до створення однієї функції яка буде його реалізовувати, а створення структури та додавання логування покласти на наш DSL.

```
#[macro_export]
macro_rules! init_sort  {
    ($name:ident)  => {
        pub struct  $name {
            logger:  Logger,
        }

        impl $name {
            pub fn new() -> Self  {
                Self  {
                    logger:  Logger::new(),
                }
            }
        }

        type  SortArgs<'a,  T> = (&'a mut Vec<T>, &'a mut $name);

        impl SortLogging for  $name {
            fn  logger(&mut self)  -> &mut Logger {
                &mut self.logger
            }
        }
    };
}

#[macro_export]
macro_rules! sort  {
    (
        $name:ident<$type:ident>  $sort:expr
    ) => {
        init_sort!($name);

        impl Sort<$type> for  $name {
            fn  sort(&mut self,  vec:  &mut Vec<$type>) {
                self.logger.start();
                if  !vec.is_empty()  {
                    $sort((vec,  self));
                }
                self.logger.end();
            }
        }
    };
    (
        $name:ident  $(+ $aditional_trait:ident)*      $sort:expr
    ) => {
        init_sort!($name);

        impl<T> Sort<T> for $name
        where
            T:  Ord $(  + $aditional_trait)*,
        {
            fn  sort(&mut self,  vec:  &mut Vec<T>) {
                self.logger.start();
                if  !vec.is_empty()  {
                    $sort((vec,  self));
```

```
                }
                self.logger.end();
            }
        }
    };
}
```

3.1.4 Тестування

Також додамо декілька функцій для того щоб тестувати наші сортування та зберігати результати їх логування.

```
use csv::Writer;
use rand::Rng;

use super::{Metrics, Sort};

pub fn test_i64<T: Sort<i64>>(sort: &mut T, n: usize) -> Metrics {
    let mut rng = rand::thread_rng();
    let mut arr = Vec::<i64>::new();
    for _ in 0..n {
        let t: usize = rng.gen();
        arr.push((t % (n * 10)) as i64);
    }

    sort.sort(&mut arr);
    assert!(arr.is_sorted());
    sort.get_metrics()
}

pub fn test_perf<T: Sort<i64>>(sort: &mut T, file_name: String, test_cases: Vec<usize>) {
    let mut wtr = Writer::from_path(file_name).unwrap();
    wtr.write_record(&["n", "compares", "swaps", "time"])
        .unwrap();

    for n in test_cases {
        let m = test_i64(sort, n);
        wtr.write_record(&[
            n.to_string(),
            m.compares.to_string(),
            m.swaps.to_string(),
            m.time.as_nanos().to_string(),
        ])
        .unwrap();
    }
}
```

3.2 Сортування

```
use super::super::sort_preamble::*;

sort! {
    BubbleSort | args: SortArgs<T> | {
        let arr = args.0;
        let sort = args.1;

        let n = arr.len();
        for i in 0..n {
            let mut swapped = false;
            for j in 0..(n - i - 1) {
                if sort.gt(arr, j, j + 1) {
                    sort.swap(arr, j, j + 1);
                    swapped = true;
                }
            }
            if !swapped {
                break;
            }
        }
    }
}
```

3.3 Приклад роботи програми

Код програми для перевірки:

```rust
use crate::libs::sort::{testing::test_perf, variants::BubbleSort};

// impl AsIndex for i64 {
//     fn as_index(&self) -> usize {
//         (self / 10) as usize
//     }
// }

pub fn main() {
    let test_cases: Vec<usize> = vec![20, 1000, 5000, 10000, 50000];
    let mut sort = BubbleSort::new();
    let file_name = "bubble.csv".to_string();

    test_perf(&mut sort, file_name, test_cases);
}
```

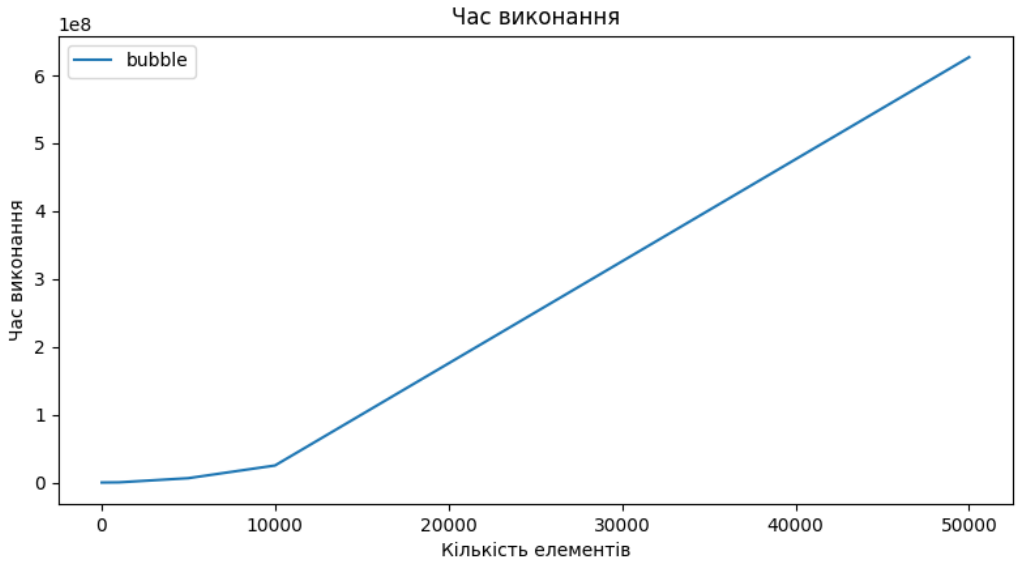

Рис. 1. Залежність часу пошуку від кількості елементів

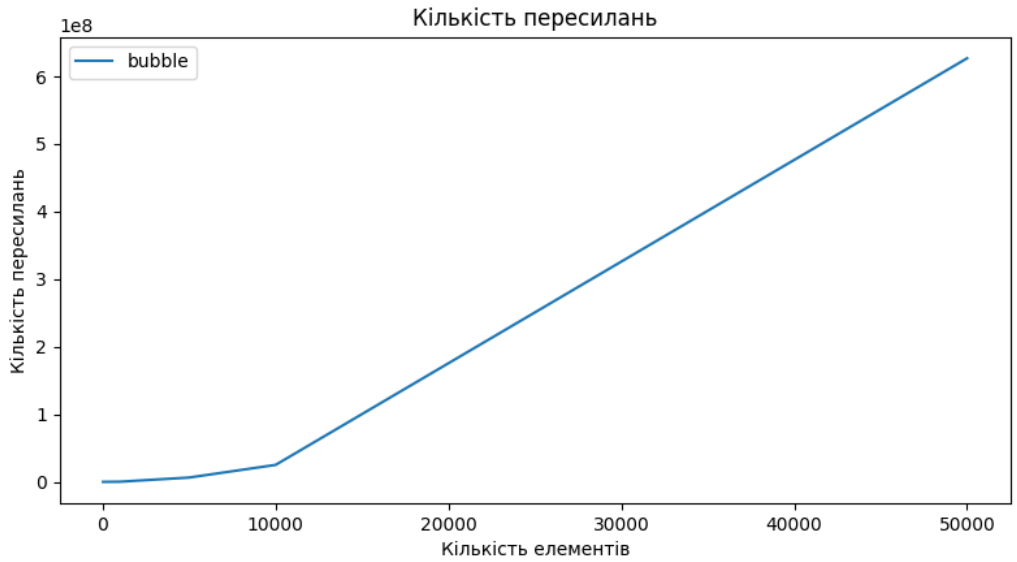

Рис. 2. Залежність кількості пересилань від кількості елементів

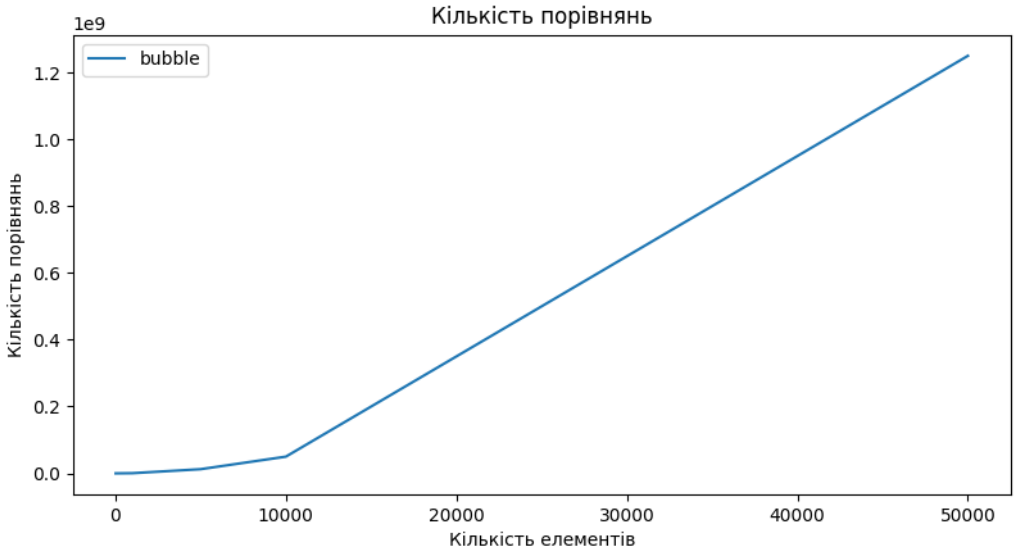

Рис. 3. Залежність кількості порівнянь від кількості елементів

4 Висновки

В ході виконання лабораторної робити було створено сортування bubble sort. Також було протестовано його на різних обємах даних та побудовано графіки для наглядної демонстрації його характеристик.