ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І.І.МЕЧНИКОВА Факультет математики, фізики та інформаційних технологій

**Звіт з лабораторної роботи № 1**

з дисципліни: “Алгоритми та структури даних” на тему:

“Аналіз алгоритмів”

Виконав: студент 2 курсу

денної форми навчання

спеціальності 123 «Комп'ютерна інженерія»

Мовчан Ілля Олександрович

**Лабораторна робота № 1**

**Тема:** Діагностика роботи алгоритмів обчислення різних програм.

**Мета**: зробити часові заміри оптимізованих та не оптимізованих алгоритмів на мові програмування C#, проаналізувати отримані дані

**Програмне забезпечення:** Windows 10, Visual Studio 2022

**Постановка задачі**

1. Реалізувати за допомогою функцій (або статичних методів мови C#) декілька варіантів вирішення завдань.
2. Порівняти їх швидкість та побудувати таблицю залежності часу виконання від n для декількох реалізацій (опціонально побудувати відповідні графіки).
3. Скласти звіт, куди включити ці дані.

**Завдання**

1. [Пошук елемента у масиві](#q5mxbcqd3n9l)
2. [Сортування масиву](#jbvof6kx187)
3. [Пошук простих чисел](#ait16acmexyb)

**Пошук елемента у масиві**

Я реалізував 2 способи пошуку - це лінійний пошук та бінарний пошук. Кожен із них має свої переваги та недоліки, наприклад лінійний пошук буде доволі повільним при пошуку числа у великому масиві, а бінарний пошук буде корректно працювати тільки з відсортованим масивом.

Отже розглянемо два алгоритми пошуку:

1. Лінійний пошук :

public static bool LinearSearch(int[] stack, int needle, int start = 0, int end = -1)

{

if (end == -1) end = stack.Length - 1;

for (int i = start; i <= end; i++)

{

if (stack[i] == needle) return true;

}

return false;

}

1. Бінарний пошук :

public static bool BinarySearch(int[] stack, int needle, int start = 0, int end = -1)

{

if (end == -1) end = stack.Length - 1;

while (start <= end)

{

int mid = (start + end) / 2;

if (stack[mid] == needle) return true;

else if (stack[mid] < needle) start = mid + 1;

else if (stack[mid] > needle) end = mid - 1;

}

return false;

}

Я створив випадковий масив з int.MinValue до int.MaxValue і почав пошук числа за індексом 20 з кінця масиву. Ось які результати я отримав:

* **Не відсортований масив:**

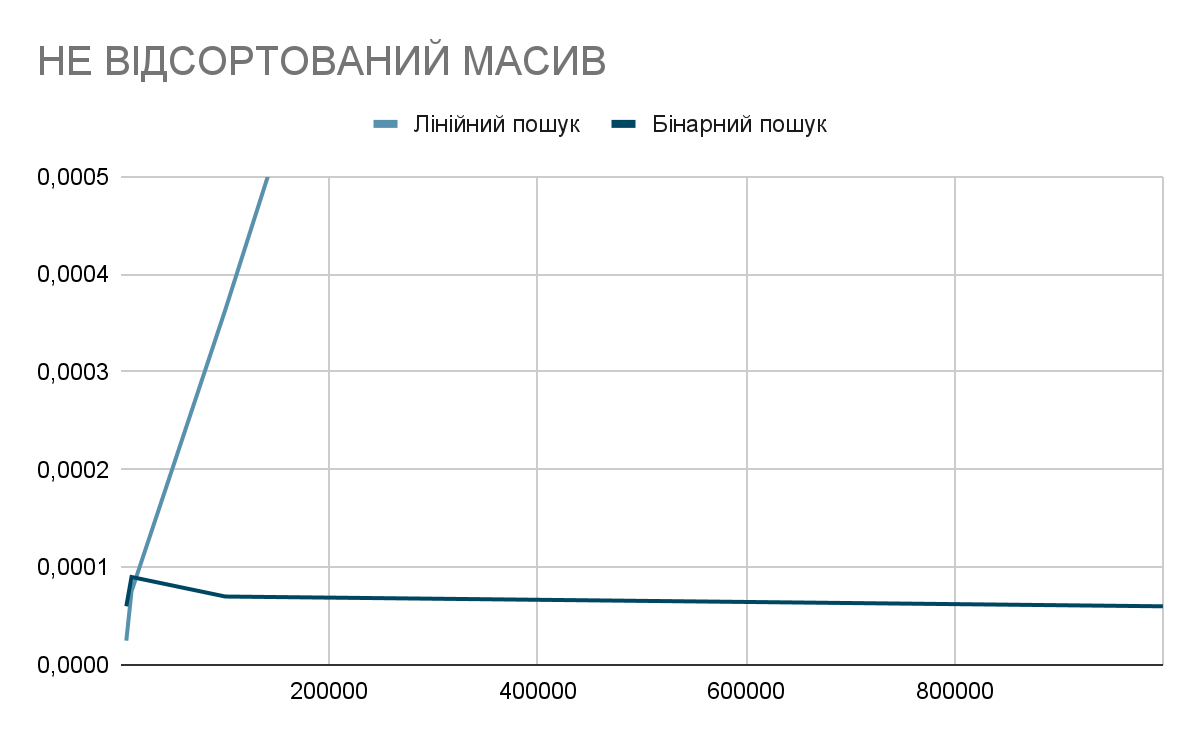
| Розмір масиву | Час | |
| --- | --- | --- |
| Лінійний пошук | Бінарний пошук |
| 5000 | 00:00:00.0000248 | 00:00:00.0000006 |
| 10000 | 00:00:00.0000753 | 00:00:00.0000009 |
| 100000 | 00:00:00.0003635 | 00:00:00.0000007 |
| 1000000 | 00:00:00.0033748 | 00:00:00.0000006 |

(табл. 1)

* **Відсортований масив:**

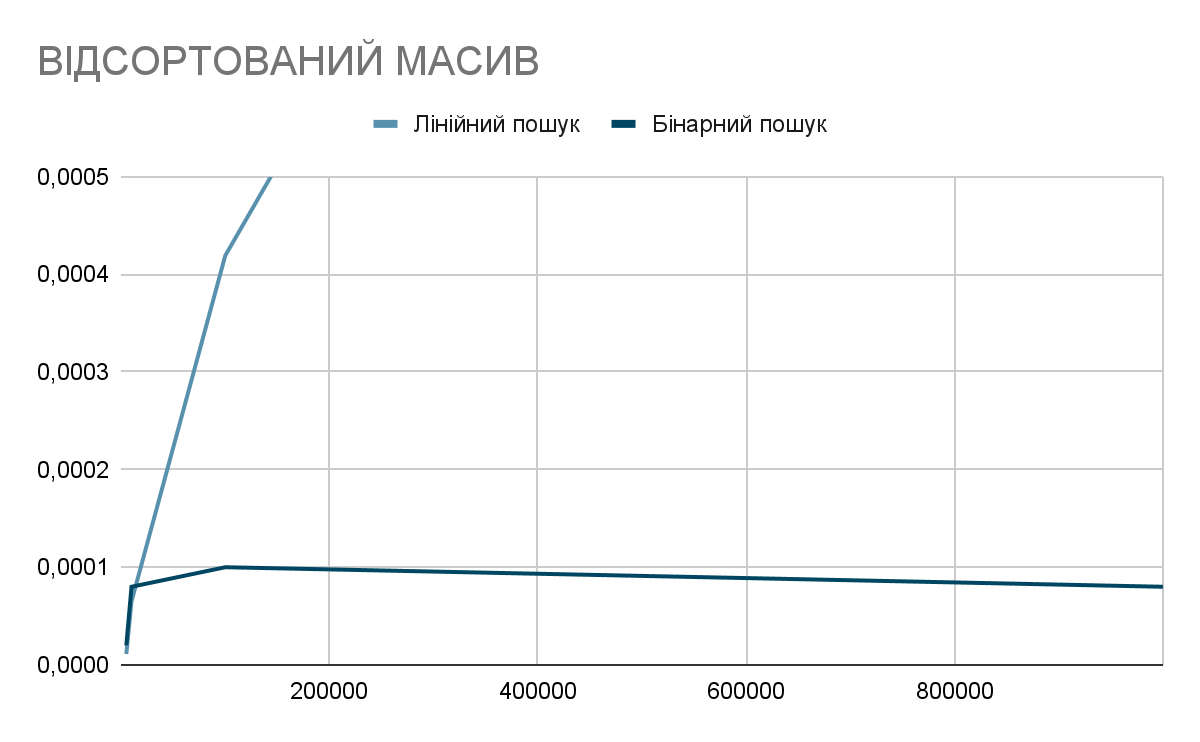
| Розмір масиву | Час | |
| --- | --- | --- |
| Лінійний пошук | Бінарний пошук |
| 5000 | 00:00:00.0000111 | 00:00:00.0000002 |
| 10000 | 00:00:00.0000651 | 00:00:00.0000008 |
| 100000 | 00:00:00.0004191 | 00:00:00.0000013 |
| 1000000 | 00:00:00.0020837 | 00:00:00.0000018 |

(табл. 2)



(рис. 1)

Тепер порівняймо результати у відсортованому масиві:



(рис. 2)

Отже, бінарний пошук у відсортованому масиві буде працювати набагато ефективніше, але буде працювати неправильну у не відсортованому. Натомість лінійний пошук у масиві, буде працювати менш ефективно у великому масиві, але завжди буде видавати вірний результат.

**Сортування масиву**

Сортування масиву - одна із базових операцій взаємодії із масивами. Я зробив 3 методи сортування масиву - **сортування вибором**, **бульбашкове сортування та сортування вставками**

Розглянемо три алгоритми сортування:

1. Сортування вибором:

public static int[] SelectionSort(int[] originalArr)

{

int[] arr = originalArr.ToArray();

for (int j = 0; j < arr.Length; j++)

{

int minIndex = j;

for (int i = j; i < arr.Length; i++)

{

if (arr[minIndex] > arr[i])

{

minIndex = i;

}

}

Swap(ref arr[j], ref arr[minIndex]);

}

return arr;

}

Цей метод використовує алгоритм сортування вибором для впорядкування масиву чисел. Проходячи через масив, він вибирає найменший елемент та обмінює його з першим невідсортованим елементом.

Повторюючи цей процес для кожного елемента масиву, метод забезпечує відсортований результат в порядку зростання.

1. Бульбашкове сортування:

public static int[] BubbleSort(int[] originalArr)

{

int[] arr = originalArr.ToArray();

for (int j = 0; j < arr.Length - 1; j++)

{

bool swapped = false;

for (int i = 0; i < arr.Length - j - 1; i++)

{

if (arr[i] > arr[i + 1])

{

Swap(ref arr[i],ref arr[i+1]);

swapped = true;

}

}

if (!swapped) return arr;

}

return arr;

}

Алгоритм бульбашкового сортування порівнює сусідні елементи і переміщує їх, поки не відсортує весь масив в порядку зростання. Якщо на одному з проходжень внутрішнього циклу не відбудеться жодного обміну, це означає, що масив вже відсортований, і алгоритм завершить свою роботу.

Також було реалізовано метод Swap() для зручності здійснення свопу елементів у двох попередніх алгоритмах:

private static void Swap(ref int a, ref int b)

{

int tmp = a;

a = b;

b = tmp;

}

1. Сортування вставками:

public static int[] InsertionSort(int[] originalArr)

{

int[] arr = originalArr.ToArray();

int index;

int currentNumber;

for (int i = 0; i < arr.Length; i++)

{

index = i;

currentNumber = arr[i];

while(index > 0 && currentNumber < arr[index - 1])

{

arr[index] = arr[index - 1];

index--;

}

arr[index] = currentNumber;

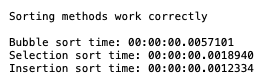
}

return arr;

}

Алгоритм сортування вставкою додає кожен елемент відсортованої частини масиву, вибираючи його з невідсортованої частини і вставляючи на відповідне місце у відсортований частині, поки весь масив не буде відсортований.

Отже після здійснення 3 методів та виводу виміру часу отримуємо:

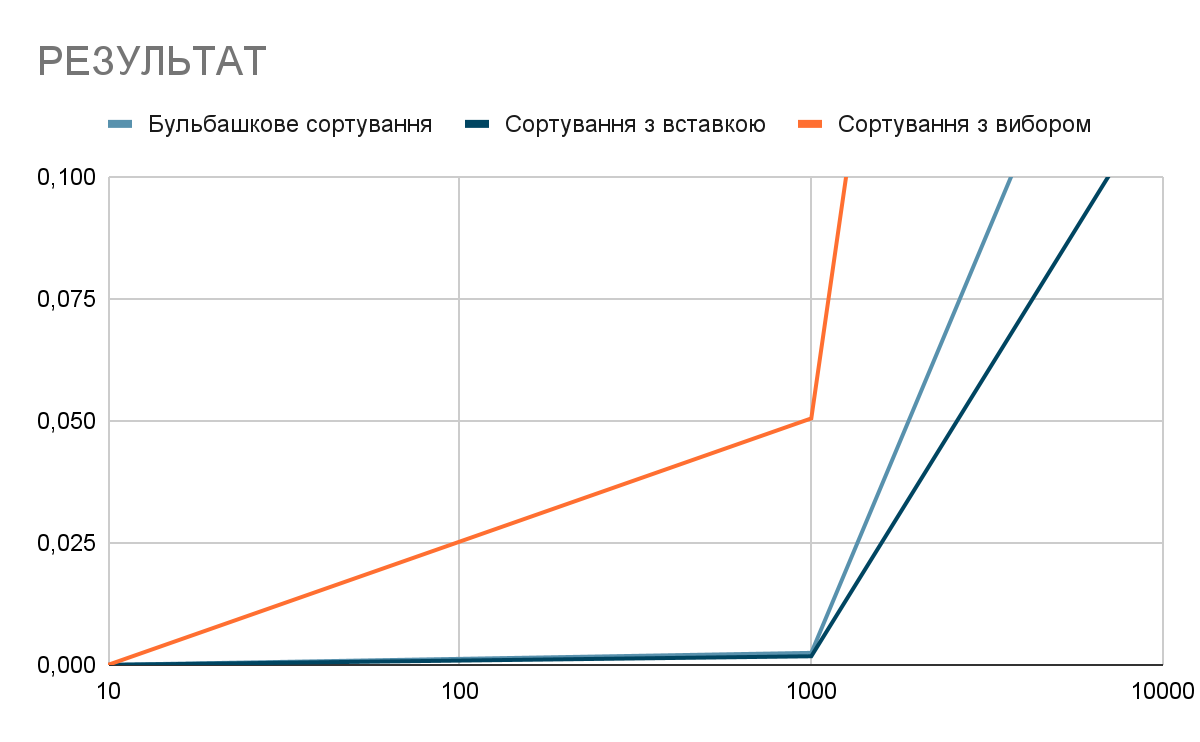


(рис. 4)

Я згенерував масив чисел від int.MinValue до int.MaxValue та простежив за виконанням алгоритмів при різній кількості елементів у масиві:

| Розмір масиву | Час | | |
| --- | --- | --- | --- |
| Сортування з вибором | Бульбашкове сортування | Сортування вставками |
| 10 | 00:00:00.0000093 | 00:00:00.0000040 | 00:00:00.0000010 |
| 1000 | 00:00:00.0050477 | 00:00:00.0024645 | 00:00:00.0017671 |
| 10000 | 00:00:00.5505997 | 00:00:00.1740517 | 00:00:00.1180402 |
| 100000 | 00:00:54.1500755 | 00:00:18.152357 | 00:00:11.6369838 |

(табл. 3)

(рис. 3)

Отже за таблицею та діаграмою ми можемо побачити, що найменш ефективним алгоритмом є сортування з вибором, а найбільш оптимальним - сортування вставками, хоча бульбашкове сортування теж є доволі ефективним та не сильно відстає за швидкістю від сортування з вибором.

**Пошук простих чисел**

Пошук простих чисел - це метод, який перевіряє послідовніть цілих чисел від start до end та записує у тип даних List усі прості числа даного діапазону.

Допоміжним методом став метод IsPrime, який перевіряє число на простоту та виводить true або false:

private static bool IsPrime(int number)

{

if (number <= 1) return false;

for (int i = 2; i <= Math.Sqrt(number); i++)

{

if (number % i == 0) return false;

}

return true;

}

Розглянемо два алгоритми пошуку простих чисел :

1. Неоптимізований метод:

public static List<int> FindPrimesUnoptimized(int start, int end)

{

List<int> primes = new List<int>();

for (int i = start; i <= end; i++)

{

if (IsPrime(i))

{

primes.Add(i);

}

}

return primes;

}

1. Оптимізований метод:

public static List<int> FindPrimesOptimized(int start, int end)

{

bool[] isPrime = new bool[end + 1];

List<int> primes = new List<int>();

for (int i = 2; i <= end; i++)

{

isPrime[i] = true;

}

for (int p = 2; p \* p <= end; p++)

{

if (isPrime[p])

{

for (int i = p \* p; i <= end; i += p)

{

isPrime[i] = false;

}

}

}

for (int i = Math.Max(2, start); i <= end; i++)

{

if (isPrime[i])

{

primes.Add(i);

}

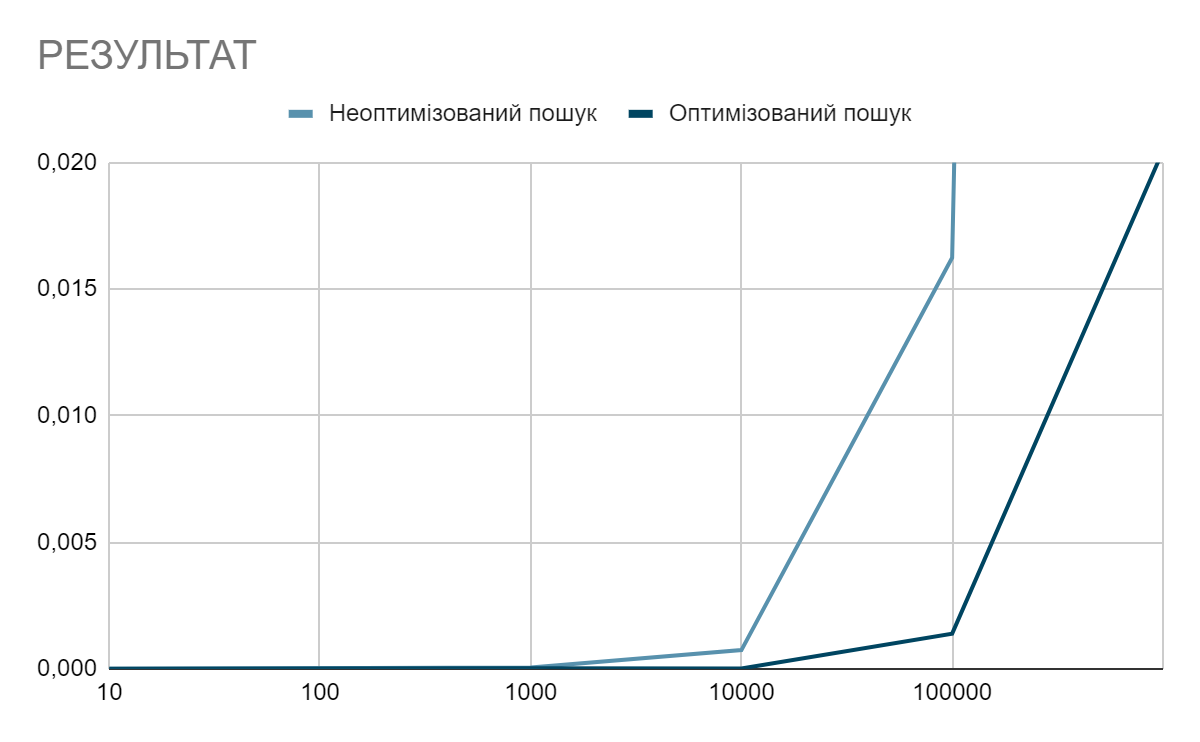
}

return primes;

}

| Послідовність | Час | |
| --- | --- | --- |
| Неоптимізваний пошук | Оптимізваний пошук |
| 100 | 00:00:00.0000067 | 00:00:00.0000035 |
| 1000 | 00:00:00.0000540 | 00:00:00.0000209 |
| 10000 | 00:00:00.0007420 | 00:00:00.0001490 |
| 100000 | 00:00:00.0162381 | 00:00:00.0013943 |
| 1000000 | 00:00:00.3994659 | 00:00:00.0204630 |

(табл. 4)



(рис. 4)

Неоптимізований метод перевіряє, чи число n є простим, ділячи його на будь-яке число від 2 до кореня з n. Якщо число ділиться без залишку на яке-небудь з цих чисел, то воно не є простим. Цей метод перевіряє всі числа від 2 до кореня з n - 1, щоб переконатися, що число n не ділиться націло на них.

Оптимізований метод використовує той факт, що якщо число n не є простим і ділиться на яке-небудь число **a**, то воно також ділиться на **b=n/a**. Тому при перевірці до кореня з n, досить перевірити числа до sqrt(n)

Оптимізований метод починає перевірку з ділення на 2, а потім перевіряє лише непарні числа від 3 до sqrt(n). Це зменшує кількість перевірок вдвічі. Таким чином робить його більш ефективним для великих чисел.

**Висновок**

* цій лабораторній роботі я реалізував по два алгоритми для наступних завдань: пошук елементу в масиві, сортування масиву та пошук простих чисел. Було виміряно час виконання цих алгоритмів, проаналізовано їх складність, а також складено порівняльні таблиці і графіки, з яких видно, який з алгоритмів є найшвидшим. Дякую за увагу!

**Посилання на код**

<https://github.com/Ilmovchan/algorithm/tree/9dbd0a49094b16438478d49d4cda6845c653b5a8/lab%20%231/AlgorithmTest>

