**Оцінка результатів роботи програми**

Програма написана на мові програмування C#. Тестування програми було виконано на процесорі Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz.

*Завдання:* Розробити програму, яка визначає час сортування масиву. Виконати порівняння різних методів сортування (мінімум 6) при різних умовах, а сам:

1. Порівняти час сортування для масивів з кількістю елементів 10, 1000, 10 000, 1 000 000, ~~100 000 000~~ (випадкові з великого інтервалу)
2. Порівняти час сортування для масивів з кількістю елементів 10, 1000, 10 000, 1 000 000, ~~100 000 000~~ (випадкові з малого інтервалу)
3. Порівняти час сортування за зростанням для масивів з кількістю елементів 10, 1000, 10 000, 1 000 000, ~~100 000 000~~ (випадкові з великого інтервалу, які попередньо впорядковані за спаданням)
4. Порівняти час сортування за зростанням для масивів з кількістю елементів 10, 1000, 10 000, 1 000 000, ~~100 000 000~~ (випадкові з малого інтервалу, які попередньо впорядковані за спаданням)

Написаний алгоритм дуже простий – відповідно легко доповнюється і модифікується, наприклад для збільшення кількості методів сортування або кількості елементів в масиві.

В основу алгоритмі покладена така структура:

*Лістинг:*

internal class Program

{

static Random random = new Random();

static void Main(string[] args)

{

int[] sizes = { 10, 1000, 10000, 1000000 }; // Можна додати більше елементів

var sortingAlgorithms = new Dictionary<string, Action<int[]>>

{

//Колекція усіз методів сортування, можна збільшити

{ "Insertion Sort", InsertionSort },

{ "Bubble Sort", BubbleSort },

{ "Quick Sort", QuickSort },

{ "Merge Sort", MergeSort },

{ "Timsort", Timsort },

{ "Binary Tree Sort", BinaryTreeSort }

};

//Виведення даних в табличному вигляді

Console.WriteLine("Таблиця часу сортування (мс):\n");

Console.WriteLine("Метод сортування\t Рандом. елем. (великий iнтервал)\t\t" +

"Рандом. елем. (малий iнтервал)\t\t" +

"Впоряд. за спадом (великий iнтервал)\t\t" +

"Впоряд. за спадом (малий iнтервал)");

foreach (var size in sizes)

{

Console.WriteLine("---------------------------------------------------------------------------------------------------" +

"---------------------------------------------------------------------------------------------------");

Console.WriteLine($"Количество елементов: {size}" +

$"\t\tmm:ss:ms" +

$" \t\tmm:ss:ms" +

$" \t\t\tmm:ss:ms" +

$" \t\t\tmm:ss:ms");

foreach (var algo in sortingAlgorithms)

{

// Рандомний масив з великого інтервалу

int[] largeRangeArray = GenerateRandomArray(size, int.MinValue, int.MaxValue);

TimeSpan largeRangeTime = MeasureSortTime(algo.Value, largeRangeArray);

// Рандомний масив з малого інтервалу

int[] smallRangeArray = GenerateRandomArray(size, -100, 100);

TimeSpan smallRangeTime = MeasureSortTime(algo.Value, smallRangeArray);

// Впорядкований за спаданням масив з великого інтервалу

int[] largeRangeDescendingArray = GenerateRandomArray(size, int.MinValue, int.MaxValue);

Array.Sort(largeRangeDescendingArray);

Array.Reverse(largeRangeDescendingArray);

TimeSpan largeRangeDescendingTime = MeasureSortTime(algo.Value, largeRangeDescendingArray);

// Впорядкований за спаданням масив з малого інтервалу

int[] smallRangeDescendingArray = GenerateRandomArray(size, -100, 100);

Array.Sort(smallRangeDescendingArray);

Array.Reverse(smallRangeDescendingArray);

TimeSpan smallRangeDescendingTime = MeasureSortTime(algo.Value, smallRangeDescendingArray);

Console.WriteLine($"{algo.Key.PadRight(20)}\t\t\t" +

$"{largeRangeTime.ToString(@"mm\:ss\:fff")}" +

$" \t\t" +

$"{smallRangeTime.ToString(@"mm\:ss\:fff")}" +

$" \t\t\t" +

$"{largeRangeDescendingTime.ToString(@"mm\:ss\:fff")}" +

$" \t\t\t" +

$"{smallRangeDescendingTime.ToString(@"mm\:ss\:fff")} ");

}

Console.WriteLine();

}

}

static int[] GenerateRandomArray(int size, int minValue, int maxValue)

{

int[] array = new int[size];

for (int i = 0; i < size; i++)

{

array[i] = random.Next(minValue, maxValue);

}

return array;

}

static TimeSpan MeasureSortTime(Action<int[]> sortAlgorithm, int[] array)

{

int[] arrayCopy = (int[])array.Clone();

Stopwatch stopwatch = Stopwatch.StartNew();

sortAlgorithm(arrayCopy);

stopwatch.Stop();

return stopwatch.Elapsed;

}

//Далі алгоритми сортування…

}

Структура коду дуже така:

1. Спочатку створюємо масив в якому буде знаходитися кількість елементів масиву для кожного сортування. Також створимо колекцію (словник) з алгоритмами сортування.
2. Надалі запускаємо два вкладених друг в друга цикли які відповідно проходяться по масиву з кількістю елементів і списку алгоритмів.
3. В тілі вкладеного списку запускаємо 2 методи. Перший створює масив заповнений випадковими числами у вказаному діапазоні, а другий, запускає метод з відповідним алгоритмом сортування, в який вкладається раніше сформований масив випадкових чисел і метод з алгоритмом сортування з колекції алгоритмів. Як результат виконання другий метод повертає час сортування масиву.
4. Щоб реалізувати умову сортування масиву який вже впорядкований за спаданням, було використано вбудований метод сортування з класу ***Array***. Слід зазначити що час затрачений на таке сортування ні в якому разі не впливає на результати, бо виконується перед початком запуску таймера.
5. Для реалізації підрахунку часу сортування використано клас ***TimeSpan.***
6. Також слід зазначити що така процедура повторюється 4 рази Для кожного пункту завдання, а це значить що масив для одного алгоритму сортування на кожному з цих 4 етапів однаковий, тому точність сортування максимальна.

Для виконання цього завдання були обрані 6 алгоритмів:

* + - *InsertionSort*
    - *BubbleSort*
    - *QuickSort*
    - *MergeSort*
    - *Timsort*
    - *BinaryTreeSort (реалізований за допомогою SortedSet)*

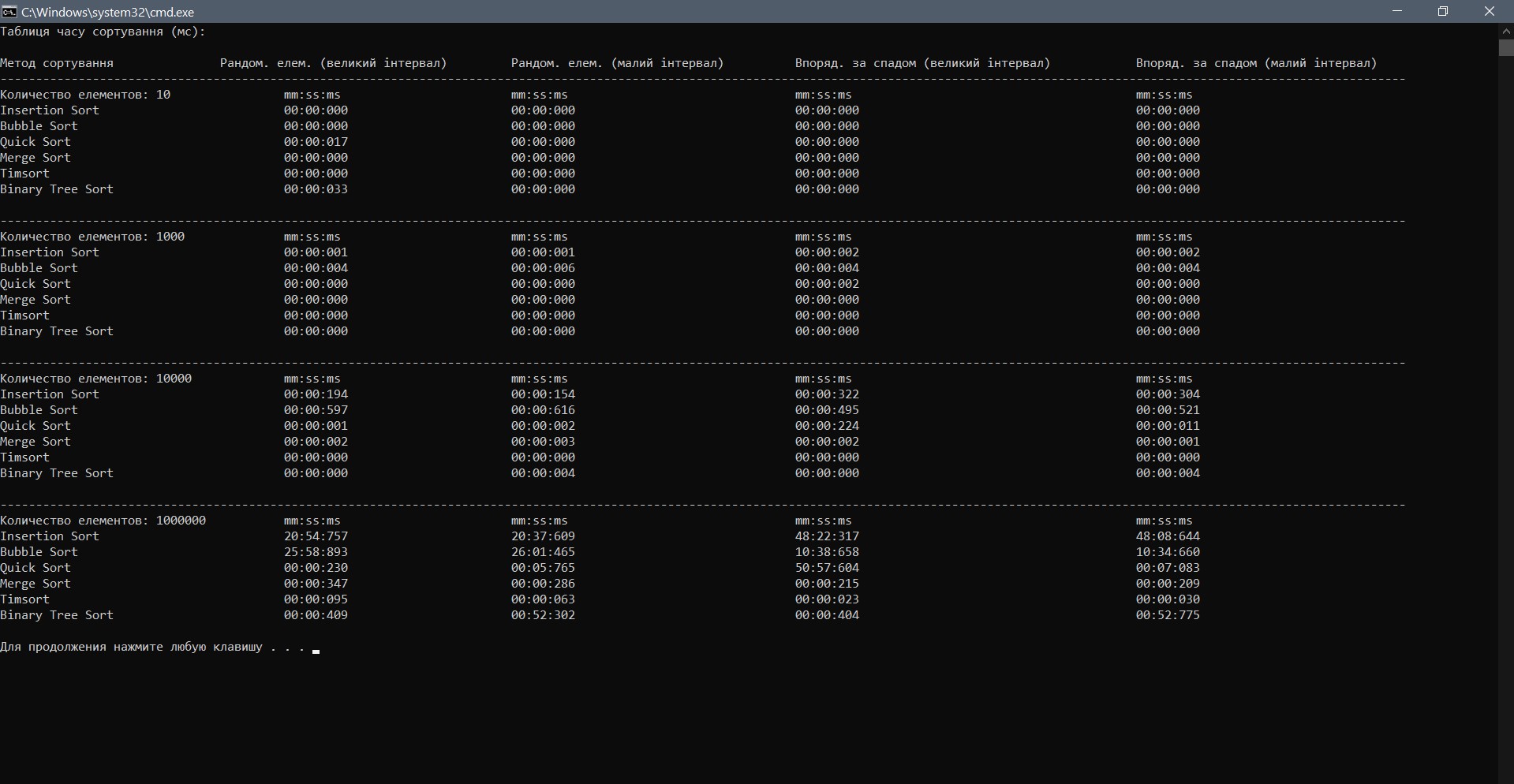


Рис. 1 – Результати виконання усіх алгоритмів сортування за всіма встановленими критеріями завдання

Виходячи з результатів роботи алгоритмів можна зробити висновок, що масиви к кількістю елементів менше ніж 10 000 упорядковуються швидше ніж 0.5 с, тому особливу увагу слід приділити останнім рядкам де використовується 1 000 000 елементів.

Ось основні спостереження виходячи з результатів:

1. **Insertion Sort і Bubble Sort**:
   * Ці алгоритми мають квадратичну складність , тому вони погано справляються з великими масивами.
     + *Insertion Sort* зайняв **20 хвилин 54 секунди** для випадкових елементів (великий інтервал).
     + *Bubble Sort* зайняв навіть більше – **26 хвилин 1 секунда**.
   * Також видно, що ці алгоритми займають ще більше часу для масивів, відсортованих за спаданням. Наприклад, для 1 000 000 елементів з великого інтервалу часу, упорядкованих за спаданням:
     + *Insertion Sort* зайняв **48 хвилин 8 секунд**.
     + *Bubble Sort* зайняв **більше 1 години 26 хвилин**.
   * Це ще раз підкреслює, що ці алгоритми не підходять для великих масивів або масивів із поганою початковою впорядкованістю.
2. **Quick Sort**:
   * *Quick Sort* показує значно кращі результати порівняно з *Insertion Sort* і *Bubble Sort*, оскільки його середня часова складність .
   * Для масиву з 1 000 000 випадкових елементів (великий інтервал) він працює за **5.765 секунд**, що набагато швидше, ніж *Insertion Sort* і *Bubble Sort*.
   * Проте для масиву, відсортованого за спаданням, час роботи зростає до **10 хвилин 57 секунд**. Це пов'язано з тим, що Quick Sort чутливий до вибору опорного елемента. У таких випадках він може деградувати до , якщо опорний елемент вибирається не оптимально.
3. **Merge Sort**:
   * *Merge Sort* показує стабільно хорошу продуктивність і добре справляється з масивами всіх типів. Це пояснюється його часовою складністю , яка не залежить від початкового впорядкування даних.
   * Наприклад, для 1 000 000 випадкових елементів (великий інтервал) він завершує роботу за **286 мс**, а для масиву, відсортованого за спаданням, – **215 мс**.
   * *Merge Sort* показує себе як ефективний алгоритм для великих масивів, оскільки він не чутливий до початкового стану масиву.
4. **Timsort**:
   * *Timsort* також має часову складність і є адаптивним алгоритмом. Він добре підходить для майже відсортованих даних або для масивів, де є певна часткова впорядкованість.
   * Час роботи для 1 000 000 випадкових елементів (великий інтервал) склав лише **63 мс**, а для масиву, відсортованого за спаданням, – **23 мс**.
   * *Timsort* є одним з найшвидших алгоритмів серед протестованих, особливо на великих наборах даних.
5. **Binary Tree Sort**:
   * *Binary Tree Sort*, який реалізований за допомогою *SortedSet*, має часову складність в середньому.
   * Його результати також дуже хороші. Наприклад, для масиву з 1 000 000 випадкових елементів (великий інтервал) він завершує сортування за **409 мс**, а для масиву, відсортованого за спаданням – **404 мс**.
   * Це також ефективний варіант для великих масивів, хоча він трохи поступається *Timsort* за швидкістю.

**Висновки**

Для малих масивів (10, 1000 елементів) різниця в часі між алгоритмами не настільки помітна, і більшість алгоритмів працюють дуже швидко.

Для середніх і великих масивів (10 000 та 1 000 000 елементів) чітко видно переваги алгоритмів із часовою складністю над алгоритмами .

*Timsort* і *Merge Sort* показали найкращу продуктивність для великих масивів, оскільки вони стабільні за часом виконання.

*Quick Sort* може працювати дуже швидко на випадкових даних, але може втрачати продуктивність на відсортованих масивах або масивах, де вибір опорного елемента є неефективним.

*Binary Tree Sort* на основі *SortedSet* також є хорошим вибором для великих масивів, хоча він трохи повільніший за *Timsort*.

**Рекомендації**

Для великих масивів (понад 10 000 елементів) рекомендується використовувати *Timsort* або *Merge Sort*, які забезпечують стабільно високу продуктивність.

*Quick Sort* може бути ефективним для випадкових масивів, але його слід використовувати обережно для масивів з поганим початковим впорядкуванням.

*Bubble Sort* та *Insertion Sort* не підходять для великих масивів і можуть використовуватися лише для дуже малих масивів, де швидкість не є критичною.