МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФРАСТРУКТУРИ ТА ТЕХНОЛОГІЙ
ІНСТИТУТ УПРАВЛІННЯ, ТЕХНОЛОГІЙ ТА ПРАВА
ФАКУЛЬТЕТ УПРАВЛІННЯ І ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

Лабораторна робота №3

з дисципліни «Фундаментальні комп’ютерні алгоритми»

з теми: «ДОСЛІДЖЕННЯ МЕТОДІВ АНАЛІЗУ АЛГОРИТМІВ»

Варіант 13

Виконав:
Студент групи
ІПЗд-23121 маг.
Петренко Д.М.
Перевірив:
Доцент кафедри ІТ
Ткаченко О.А.

Київ – 2024

Лабораторна робота №3

Мета: дослідити методи аналізу ефективності алгоритмів та набути практичних навичок з емпіричного дослідження швидкодії алгоритмів залежно від обсягу та структурованості вхідних даних.

ЗАВДАННЯ:

Завдання першого рівня

Виконати такі дії:

- реалізувати заданий алгоритм (Сортування - низхідне злиття) для одновимірного масиву чисел;
- отримати час виконання алгоритму для наборів даних розміром N, N^2 , N^3, де N= 100;
- побудувати графік залежностей часу виконання алгоритму від кількості елементів набору даних.

Завдання другого рівня

Виконати такі дії:

- реалізувати два алгоритми для одного набору (Сортування – порозрядне, сортування - висхідне злиття)
- отримати час виконання алгоритмів для наборів даних розміром A, A^2, A^3, де A =100;
- побудувати графіки залежностей часу виконання від кількості елементів набору даних для двох реалізацій алгоритму.

Завдання третього рівня

Виконати такі дії:

- реалізувати заданий алгоритм (Висхідного злиття) для набору даних з 10000 елементів;
- визначити, який порядок розташування елементів у наборах даних дає найменший та найбільший час виконання реалізованих алгоритмів;

- сформувати набори даних розміром 10000 елементів, які розташовані в порядку, що відповідає найменшому, найбільшому та середньому часу виконання;
- отримати час виконання алгоритмів для сформованих наборів даних;
- побудувати графік залежностей часу виконання від порядку розташування елементів набору даних.

ХІД РОБОТИ:

Завдання 1-го рівня:

Створюємо програму на мові програмування Java, що виконує низхідне злиття та виводить значення в наносекундах відносно кількості елементів набору даних.

```
public class MergeSortPerformance {
    Run | Debug
    public static void main(String[] args) {
        int[] sizes = {100, 10000, 1000000}; // Розміри масивів N, N^2, N^3
        int numberOfArrays = 10; // Кількість масивів для кожного розміру

        // Масив для зберігання часу виконання для кожного розміру
        long[] executionTimes = new long[sizes.length];

        // Вимірюємо час виконання для кожного розміру масиву
        for (int i = 0; i < sizes.length; i++) {
            long totalTime = 0;
            for (int j = 0; j < numberOfArrays; j++) {
                int[] data = generateRandomArray(sizes[i]);
                long startTime = System.nanoTime();
                mergeSort(data);
                long endTime = System.nanoTime();
                totalTime += endTime - startTime;
            }
            // Зберігаємо середній час виконання для даного розміру масиву
            executionTimes[i] = totalTime / numberOfArrays;
        }

        // Виводимо результат
        for (int i = 0; i < sizes.length; i++) {
            System.out.println("Size: " + sizes[i] + ", Time: " + executionTimes[i] + " nanose
        }
    }
```

У функції main реалізовано іціалізацію масив **sizes**, що містить розміри масивів, для яких буде вимірюватись час виконання алгоритму сортування. Встановлюється змінну **numberOfArrays**, яка вказує на кількість масивів, які будуть сгенеровані та відсортовані для кожного розміру.

Створюється масив **executionTimes** для зберігання середнього часу виконання для кожного розміру масиву. Запускається цикл, що проходить по

кожному розміру масиву в масиві **sizes**. Усередині цього циклу є внутрішній цикл, який генерує та сортує **numberOfArrays** масивів розміру, вказаного в поточному кроці, використовуючи функцію **generateRandomArray** для генерації масиву та вимірює час виконання алгоритму сортування. Після завершення внутрішнього циклу середній час виконання обчислюється шляхом ділення суми часу виконання на кількість масивів.

Час виконання для кожного розміру масиву зберігається в масиві **executionTimes**. На останньому кроці виводиться результат: для кожного розміру масиву виводиться його розмір та середній час виконання алгоритму сортування в наносекундах.

Наступним кроком є реалізація алгоритму сортування низхідним злиттям:

```
// Сортування злиттям (bottom-down)
public static void mergeSort(int[] arr) {
    int[] aux = new int[arr.length];
    mergeSort(arr, aux, low:0, arr.length - 1);
}

// Bottom-down mergesort для підмасиву arr[low..high]
private static void mergeSort(int[] arr, int[] aux, int low, int high) {
    if (low >= high) return;
    int mid = low + (high - low) / 2;
    mergeSort(arr, aux, low, mid);
    mergeSort(arr, aux, mid + 1, high);
    merge(arr, aux, low, mid, high);
}

// Злиття двох підмасивів arr[low..mid] та arr[mid+1..high]
private static void merge(int[] arr, int[] aux, int low, int mid, int high) {
    // Копіюємо елементи в допоміжний масив
    for (int k = low; k <= high; k++) {
        aux[k] = arr[k];
    }
    // Злиття допоміжних масивів назад у arr
    int i = low, j = mid + 1;
    for (int k = low; k <= high; k++) {
        if      (i > mid)              arr[k] = aux[j++]; // Перший підмасив закінчився
        else if (j > high)             arr[k] = aux[i++]; // Другий підмасив закінчився
        else if (aux[j] < aux[i])     arr[k] = aux[j++]; // Елемент з другого підмасиву ме
        else                           arr[k] = aux[i++]; // Елемент з першого підмасиву м
    }
}
```

Метод **mergeSort(int[] arr)** є вхідною точкою для сортування масиву **arr**. Він ініціалізує допоміжний масив **aux** такого ж розміру, що і вихідний масив **arr**, і викликає внутрішній метод **mergeSort**, щоб розділити та відсортувати масив.

Приватний метод **mergeSort(int[] arr, int[] aux, int low, int high)** виконує сортування bottom-up для підмасиву **arr[low..high]**. Він рекурсивно ділить масив навпіл і викликає себе для обох підмасивів, а потім зливає їх.

Приватний метод **merge(int[] arr, int[] aux, int low, int mid, int high)** виконує злиття двох підмасивів **arr[low..mid]** та **arr[mid+1..high]**. Він копіює елементи з вихідного масиву **arr** у допоміжний масив **aux**, а потім зливає вміст допоміжного масиву назад у вихідний масив **arr**, зберігаючи порядок сортування.

Останнім є реалізація генерації випадкових чисел:

```java
// Генерація випадкового масиву заданого розміру
public static int[] generateRandomArray(int size) {
    int[] array = new int[size];
    Random random = new Random();
    for (int i = 0; i < size; i++) {
        array[i] = random.nextInt(bound:1000); // Генеруємо випадкові числа від 0 до 999
    }
    return array;
}
```

Дивимось результат виконання коду:

```
Size: 100, Time: 27070 nanoseconds
Size: 10000, Time: 1011630 nanoseconds
Size: 1000000, Time: 76250590 nanoseconds
```

Вносимо ці дані в Ексель та будуємо графік:

Завдання 2-го рівня:

Створюємо програму на мові програмування Java, що виконує алгоритм низхідного сортування злиттям, алгоритм порозрядного сортування (по молодшій цифрі (LSD)) та виводить значення в наносекундах відносно кількості елементів набору даних.

```
public static void main(String[] args) {
    int[] sizes = {100, 10000, 1000000}; // Розміри масивів N, N^2, N^3
    int numberOfArrays = 10; // Кількість масивів для кожного розміру

    // Масив для зберігання часу виконання для кожного розміру
    long[][] executionTimes = new long[sizes.length][2];

    // Вимірюємо час виконання для кожного розміру масиву
    for (int i = 0; i < sizes.length; i++) {
        for (int j = 0; j < numberOfArrays; j++) {
            int[] data = generateRandomArray(sizes[i]);

            // Реалізація порозрядного сортування
            int[] dataRadix = Arrays.copyOf(data, data.length);
            long startTimeRadix = System.nanoTime();
            radixLSDSort(dataRadix);
            long endTimeRadix = System.nanoTime();
            executionTimes[i][0] += endTimeRadix - startTimeRadix;

            // Реалізація сортування висхідним злиттям
            int[] dataMerge = Arrays.copyOf(data, data.length);
            long startTimeMerge = System.nanoTime();
            mergeSort(dataMerge);
            long endTimeMerge = System.nanoTime();
            executionTimes[i][1] += endTimeMerge - startTimeMerge;
        }
        // Зберігаємо середній час виконання для даного розміру масиву
        executionTimes[i][0] /= numberOfArrays;
        executionTimes[i][1] /= numberOfArrays;
    }

    // Виводимо результати
    System.out.println(x:"Результати:");
    for (int i = 0; i < sizes.length; i++) {
        System.out.println("Size: " + sizes[i] + "; Time for Radix LSD Sort: " + execution
    }
}
```

Ця частина коду вимірює час виконання двох різних алгоритмів сортування (порозрядного сортування і сортування висхідним злиттям) для кожного розміру масиву даних, які задані у масиві **sizes**.

Зберігається час виконання для кожного розміру масиву в масиві **executionTimes**. Виконується внутрішня петля для кожного розміру масиву, де вимірює час виконання алгоритмів сортування для 10 різних масивів,

згенерованих випадковим чином. Реалізуються алгоритми сортування для кожного з цих масивів і вимірюється час їх виконання.

Обчислюється середній час виконання кожного алгоритму для кожного розміру масиву. Виводиться результати в консоль, показуючи час виконання для кожного розміру масиву і кожного алгоритму сортування.

Реалізація порозрядного сортування (Radix LSD):

```java
public static void radixLSDSort(int[] array) {
    if (array.length == 0) {
        return;
    }

    // Визначаємо мінімальне і максимальне значення
    int minValue = array[0];
    int maxValue = array[0];
    for (int i = 1; i < array.length; i++) {
        if (array[i] < minValue) {
            minValue = array[i];
        } else if (array[i] > maxValue) {
            maxValue = array[i];
        }
    }

    // Виконуємо сортування по кожному розряду, починаючи з менш значущого розряду
    int exponent = 1;
    while ((maxValue - minValue) / exponent >= 1) {
        countingSortByDigit(array, radix:10, exponent, minValue);
        exponent *= 10;
    }
}
```

```java
private static void countingSortByDigit(int[] array, int radix, int exponent, int minValue
    int bucketIndex;
    int[] buckets = new int[radix];
    int[] output = new int[array.length];

    // Ініціалізуємо корзини
    Arrays.fill(buckets, val:0);

    // Рахуємо кількість елементів у кожній корзині
    for (int i = 0; i < array.length; i++) {
        bucketIndex = (array[i] / exponent) % radix;
        buckets[bucketIndex]++;
    }

    // Обчислюємо кумулятиви
    for (int i = 1; i < radix; i++) {
        buckets[i] += buckets[i - 1];
    }

    // Переміщуємо записи
    for (int i = array.length - 1; i >= 0; i--) {
        bucketIndex = (array[i] / exponent) % radix;
        output[--buckets[bucketIndex]] = array[i];
    }

    // Копіюємо назад
    System.arraycopy(output, srcPos:0, array, destPos:0, array.length);
}
```

Ця частина коду реалізує порозрядне сортування (Radix LSD) для масиву цілих чисел. Основні кроки алгоритму:

1. Визначається мінімальне та максимальне значення у масиві для визначення кількості розрядів чисел.

2. Виконується сортування по кожному розряду чисел, починаючи з менш значущого розряду (найменшого розряду).

3. Для сортування по кожному розряду використовується алгоритм лічильного сортування (counting sort), який рахує кількість елементів у кожній "корзині" (bucket), а потім переміщує елементи у відсортований масив за допомогою цих кількостей.

4. Кожний раз після сортування по одному розряду збільшується значення розряду (exponent), аж поки не буде відсортований весь масив за всіма розрядами.

Реалізація сортування висхідним злиттям (Bottom-up mergesort):

```java
// Реалізація сортування висхідним злиттям (Merge sort)
public static void mergeSort(int[] arr) {
    if (arr == null || arr.length <= 1) {
        return;
    }
    int[] aux = new int[arr.length];
    mergeSort(arr, aux, low:0, arr.length - 1);
}

private static void mergeSort(int[] arr, int[] aux, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        mergeSort(arr, aux, low, mid);
        mergeSort(arr, aux, mid + 1, high);
        merge(arr, aux, low, mid, high);
    }
}

private static void merge(int[] arr, int[] aux, int low, int mid, int high) {
    // Копіюємо елементи в допоміжний масив
    for (int k = low; k <= high; k++) {
        aux[k] = arr[k];
    }
    // Злиття допоміжних масивів назад у arr
    int i = low, j = mid + 1;
    for (int k = low; k <= high; k++) {
        if (i > mid) {
            arr[k] = aux[j++];
        } else if (j > high) {
            arr[k] = aux[i++];
        } else if (aux[j] < aux[i]) {
            arr[k] = aux[j++];
        } else {
            arr[k] = aux[i++];
        }
    }
}
```

Ця частина коду реалізує сортування висхідним злиттям (Merge sort) для масиву цілих чисел. Основні кроки алгоритму такі:

1. Перевіряється базовий випадок, коли масив порожній або містить лише один елемент. Якщо так, сортування не потрібне і функція завершується.

2. Створюється допоміжний масив aux того ж розміру, що і вихідний масив arr.

3. Рекурсивно викликається функція mergeSort для сортування лівої та правої половин масиву arr.

4. Після рекурсивних викликів викликається функція merge для злиття відсортованих підмасивів лівої та правої частин масиву arr.

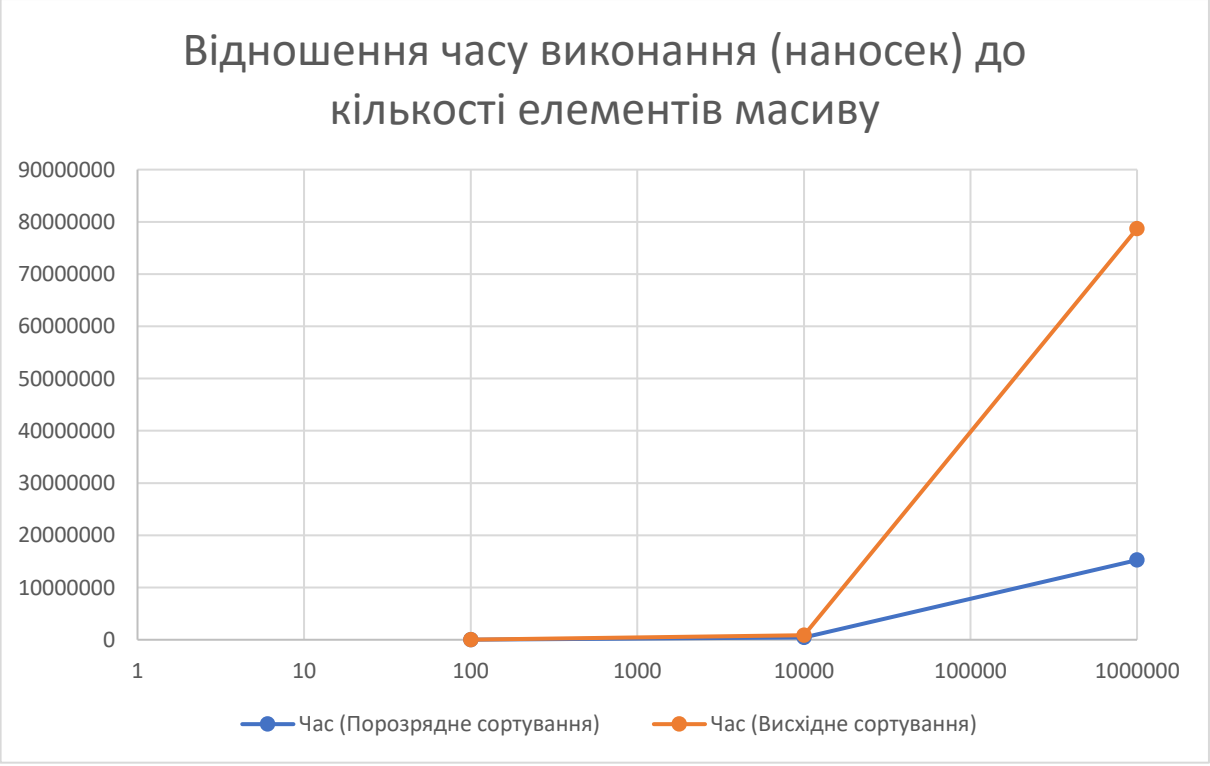
Функція merge виконує наступні дії:

- Копіює елементи з масиву arr в допоміжний масив aux.
- Злиття відсортованих підмасивів, розташованих у допоміжному масиві, назад у масив arr.

Дивимось результат виконання коду:

```
Результати:
Size: 100; Time for Radix LSD Sort: 13720 nanoseconds;     Time for Merge Sort: 22230 nanoseconds
Size: 10000; Time for Radix LSD Sort: 425010 nanoseconds;     Time for Merge Sort: 887300 nanoseconds
Size: 1000000; Time for Radix LSD Sort: 15252160 nanoseconds;     Time for Merge Sort: 78698750 nanoseconds
```

Вносимо ці дані в Ексель та будуємо графік:



Завдання 3-го рівня:

Створено програму на мові програмування Java, що виконує алгоритм висхідного сортування злиттям, генерує три різні набори даних розміром 10000 елементів: один у відсортованому порядку (за зростанням), один у зворотньо відсортованому порядку (за спаданням), і один випадковий набір даних. Для кожного набору даних вимірюється час виконання алгоритму сортування.

```
public static void main(String[] args) {
    int[] sizes = {10000}; // Розміри масивів
    int numberOfArrays = 10; // Кількість масивів для кожного розміру

    // Масив для зберігання часу виконання для кожного розміру та типу набору даних
    long[][][] executionTimes = new long[sizes.length][3][2]; // 3 типи наборів даних: відсортований, випадковий, зворотній порядок

    // Генератор випадкових чисел
    Random random = new Random();

    // Вимірюємо час виконання для кожного розміру масиву та кожного типу набору даних
    for (int i = 0; i < sizes.length; i++) {
        for (int j = 0; j < numberOfArrays; j++) {
            // Генеруємо дані для кожного типу набору даних
            int[] dataSorted = generateSortedArray(sizes[i]); Random random - Main.main(String[])
            int[] dataRandom = generateRandomArray(sizes[i], random);
            int[] dataReverse = generateReverseSortedArray(sizes[i]);

            // Вимірюємо час виконання для кожного типу набору даних
            measureExecutionTime(dataSorted, executionTimes[i][0]);
            measureExecutionTime(dataRandom, executionTimes[i][1]);
            measureExecutionTime(dataReverse, executionTimes[i][2]);
        }
        // Зберігаємо середній час виконання для даного розміру масиву та кожного типу набору даних
        for (int k = 0; k < 3; k++) {
            executionTimes[i][k][0] /= numberOfArrays;
            executionTimes[i][k][1] /= numberOfArrays;
        }
    }

    // Виводимо результати
    System.out.println(x:"Результати:");
    for (int i = 0; i < sizes.length; i++) {
        System.out.println("Size: " + sizes[i]);
        System.out.println("Sorted Order: Time for Merge Sort: " + executionTimes[i][0][0] + " nanoseconds");
        System.out.println("Random Order: Time for Merge Sort: " + executionTimes[i][1][0] + " nanoseconds");
        System.out.println("Reverse Order: Time for Merge Sort: " + executionTimes[i][2][0] + " nanoseconds");
    }
}
```

В даній частині коду йде визначення розмірів масивів та кількість масивів для кожного розміру. Створюється тривимірний масив **executionTimes** для зберігання часу виконання для кожного розміру масиву та кожного типу набору даних (відсортований, випадковий, зворотній порядок).

Генеруються випадкові дані для кожного типу набору даних (відсортований, випадковий, зворотній порядок).

Вимірюється час виконання алгоритму сортування для кожного типу набору даних і зберігає його у **executionTimes**. Обчислюється середній час виконання для кожного типу набору даних.

Виводяться результати, вказуючи час виконання сортування для кожного типу набору даних (відсортований, випадковий, зворотній порядок) для кожного розміру масиву.

```
// Вимірює час виконання сортування для даного масиву
private static void measureExecutionTime(int[] data, long[] executionTime) {
    long startTime = System.nanoTime();
    mergeSort(data);
    long endTime = System.nanoTime();
    executionTime[0] += endTime - startTime;
}

// Генерує відсортований масив
private static int[] generateSortedArray(int size) {
    int[] array = new int[size];
    for (int i = 0; i < size; i++) {
        array[i] = i;
    }
    return array;
}

// Генерує випадковий масив
private static int[] generateRandomArray(int size, Random random) {
    int[] array = new int[size];
    for (int i = 0; i < size; i++) {
        array[i] = random.nextInt();
    }
    return array;
}

// Генерує масив в зворотньому порядку
private static int[] generateReverseSortedArray(int size) {
    int[] array = new int[size];
    for (int i = 0; i < size; i++) {
        array[i] = size - i;
    }
    return array;
}
```

Ця частина коду містить наступні функції:

1. **measureExecutionTime(int[] data, long[] executionTime)**: Вимірює час виконання сортування для даного масиву. Вона приймає масив **data** для сортування та масив **executionTime**, до якого буде додано час виконання. Ця функція викликає **mergeSort(data)**, вимірює час до початку та після виконання сортування, та додає різницю до **executionTime**.

2. **generateSortedArray(int size)**: Генерує відсортований масив розміром **size**. Заповнює масив послідовними числами від 0 до **size - 1**.

3. **generateRandomArray(int size, Random random)**: Генерує випадковий масив розміром **size** за допомогою генератора випадкових чисел **random.nextInt()**. Кожен елемент масиву є випадковим цілим числом.

4. **generateReverseSortedArray(int size)**: Генерує масив, в якому елементи розміщені в зворотньому порядку, від **size** до 1.

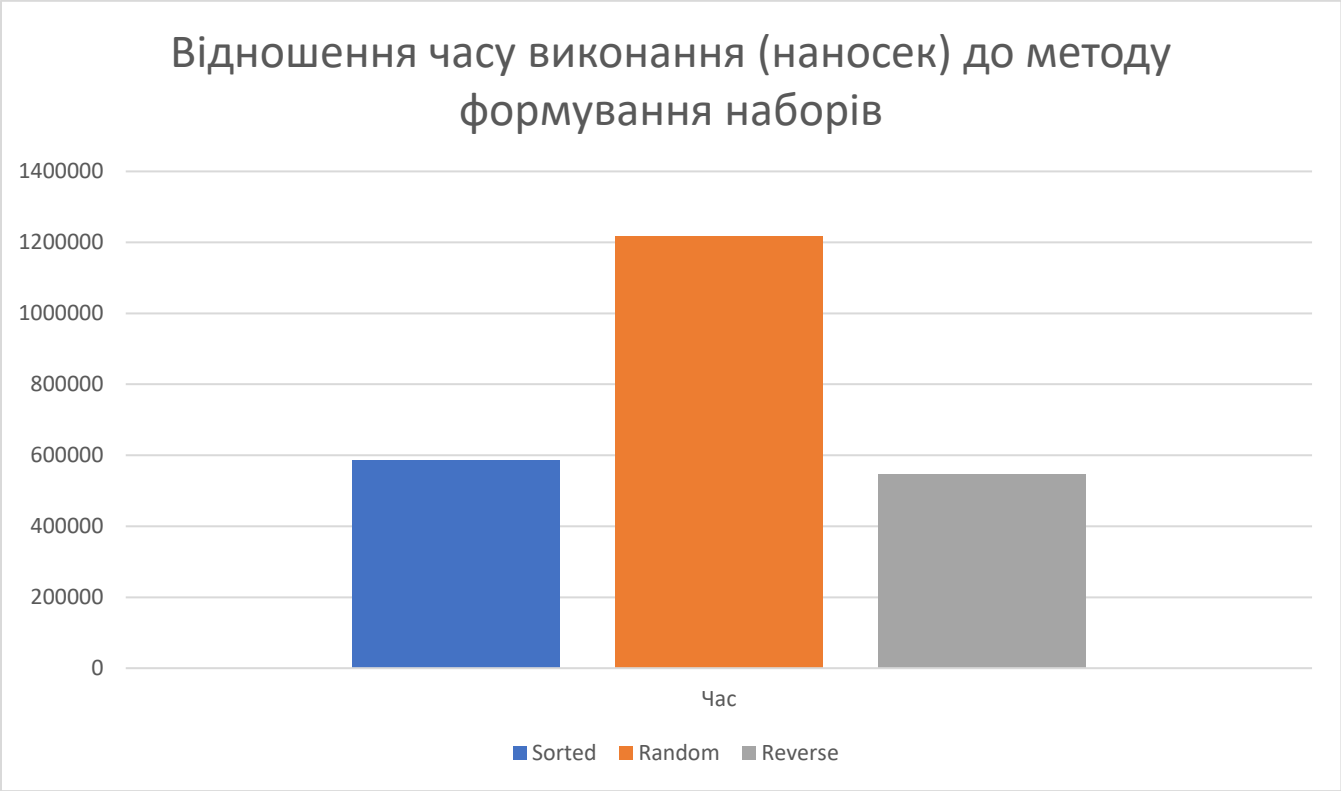
Остання частина коду реалізує алгоритм висхідного злиття:

```
// Реалізація сортування висхідним злиттям (Merge sort)
public static void mergeSort(int[] arr) {
    int[] aux = new int[arr.length];
    mergeSort(arr, aux, low:0, arr.length - 1);
}

// Bottom-down mergesort для підмасиву arr[low..high]
private static void mergeSort(int[] arr, int[] aux, int low, int high) {
    if (low >= high) return;
    int mid = low + (high - low) / 2;
    mergeSort(arr, aux, low, mid);
    mergeSort(arr, aux, mid + 1, high);
    merge(arr, aux, low, mid, high);
}

// Злиття двох підмасивів arr[low..mid] та arr[mid+1..high]
private static void merge(int[] arr, int[] aux, int low, int mid, int high) {
    // Копіюємо елементи в допоміжний масив
    for (int k = low; k <= high; k++) {
        aux[k] = arr[k];
    }
    // Злиття допоміжних масивів назад у arr
    int i = low, j = mid + 1;
    for (int k = low; k <= high; k++) {
        if      (i > mid)              arr[k] = aux[j++]; // Перший підмасив закінчився
        else if (j > high)             arr[k] = aux[i++]; // Другий підмасив закінчився
        else if (aux[j] < aux[i])     arr[k] = aux[j++]; // Елемент з другого підмасиву менший
        else                           arr[k] = aux[i++]; // Елемент з першого підмасиву менший
    }
}
```

Дивимось результат виконання коду:

```
Результати:
Size: 10000
Sorted Order: Time for Merge Sort: 585330 nanoseconds
Random Order: Time for Merge Sort: 1217950 nanoseconds
Reverse Order: Time for Merge Sort: 545690 nanoseconds
```

Вносимо ці дані в Ексель та будуємо графік:



ВИСНОВОК:

В ході лабораторної роботи було проведено дослідження ефективності алгоритмів сортування на прикладі трьох різних завдань.

У першому завданні було реалізовано алгоритм сортування низхідного злиття для одновимірного масиву чисел. Час виконання цього алгоритму було виміряно для наборів даних розміром N, N^2 та N^3, де N = 100. Далі був побудований графік залежностей часу виконання алгоритму від кількості елементів у наборі даних.

У другому завданні були реалізовані два алгоритми сортування: порозрядне сортування та сортування висхідним злиттям. Час виконання цих алгоритмів було виміряно для наборів даних розміром A, A^2, A^3, де A = 100. Потім були побудовані графіки залежностей часу виконання від кількості елементів у наборі даних для обох реалізацій алгоритму.

У третьому завданні було реалізовано алгоритм сортування висхідним злиттям для набору даних розміром 10000 елементів. Було виявлено, що ефективність цього алгоритму сильно залежить від порядку розташування елементів у наборі даних. Було визначено, що найменший час виконання досягається при відсортованому наборі даних, а найбільший - при наборі даних, розташованому в довільному порядку. Сформовано набори даних розміром 10000 елементів, і для кожного набору був виміряний час виконання алгоритму сортування висхідним злиттям. На основі цих вимірювань був побудований графік залежності часу виконання від порядку розташування елементів у наборі даних.