# МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Національний університет “Львівська політехніка”**



**Інститут післядипломної освіти**

**ЗВІТ**

**Про виконання лабораторної роботи №1**

**«Метод швидкого сортування»**

**з дисципліни «Алгоритми і структури даних»**

Виконав:

слухач групи ПЗС-11

Гринчук Тарас

Прийняла:

доц. Кортєєва Т.О.

« »\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2014 р.

∑ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

ЛЬВІВ – 2014

**Тема роботи**: Метод швидкого сортування.

**Мета роботи**: Вивчити [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary) швидкого сортування. Здійснити програмну реалізацію [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)у швидкого сортування. Дослідити швидкодію [алгоритм](http://vns.lp.edu.ua/mod/glossary/showentry.php?courseid=472&eid=19102&displayformat=dictionary)у швидкого сортування.

## 1. Теоретичні відомості

Швидке сортування (англійською «Quick Sort») — алгоритм сортування, добре відомий, як алгоритм розроблений Чарльзом Хоаром, який не потребує додаткової пам’яті і виконує у середньому O(n∙log(n)) операцій, однак, у найгіршому випадку робитьO(n2) порівнянь. Оскільки алгоритм використовує дуже прості цикли і операції, він працює швидше інших алгоритмів, що мають таку ж асимптотичну оцінку складності.

В основі алгоритму лежить принцип «розділяй та володарюй» (англійською «Divide and Conquer»). Ідея алгоритму полягає в переставлянні елементів масиву таким чином, щоб його можна було розділити на дві частини і кожний елемент з першої частини був не більший за будь-який елемент з другої. Впорядкування кожної з частин вудбувається рекурсивно. Алгоритм швидкого сортування може бути реалізований як на масиві, так і на двобічнму списку.

Швидке сортування є алгоритмом на основі порівнянь, і не є стабільним.

Алгоритм швидкого сортування було розроблено Чарльзом Хоаром у 1962 році під час роботи у маленькій британській компанії Elliott Brothers.

В класичному варіанті, запропонованому Хоаром, з масиву обирався один елемент, і весь масив розбивався на дві частини по принципу: в першій частині — ті що не більші даного елементу, в другій частині — ті що не менші даного елемента.

Час роботи алгоритму сортування залежить від збалансованості, що характеризує розбиття. Збалансованість, у свою чергу залежить від того, який елемент обрано як опорний (відносно якого елемента виконується розбиття). Якщо розбиття збалансоване, то асимптотично алгоритм працює так само швидко як і алгоритм сортування злиттям. У найгіршому випадку, асимптотична поведінкаалгоритму настільки ж погана, як і в алгоритму сортування включенням.

• Найгірше розбиття. Найгірша поведінка має місце у тому випадку, коли процедура, що виконує розбиття, породжує одну підзадачу з (n – 1) елементом, а другу — з 0 елементами. Нехай таке незбалансоване розбиття виникає при кожному рекурсивному виклику. Для самого розбиття потрібен час Θ(n). Тоді рекурентне співвідношення для часу роботи, можна записати наступним чином: T(n) = T(n – 1) + T(0) + Θ(n) = T(n – 1) + Θ(n). Розв’язком такого співвідношення є: T(n) =Θ(n2).

• Найкраще розбиття. В найкращому випадку процедура поділу ділить задачу на дві підзадачі, розмір кожної з яких не перевищує (n / 2). Час роботи, описується нерівністю: T(n) ≤ 2∙T(n / 2) + Θ(n). Тоді: T(n) = O(n∙log(n)) — асимптотично найкращий час.

• Середній випадок. Математичне очікування часу роботи алгоритму на всіх можливих вхідних масивах є O(n∙log(n)), тобто середній випадок ближчий до найкращого.

В середньому алгоритм працює дуже швидко, але на практиці, не всі можливі вхідні масиви мають однакову імовірність. Тоді, шляхом додання рандомізації вдається отримати середній час роботи в будь-якому випадку. В рандомізованному алгоритмі, при кожному розбитті в якості опорного обирається випадковий елемент.

Реалізація алгоритму:

Є два вказівники і і j , причому на початку i = 1, j = n. Довільним чином вибирається за базовий будь-який запис з таблиці (перший, середній або останній). Нехай, наприклад, це буде перший запис X. Встановлюємо і=1 (від першого) , j =n (від останнього), якщо знайдено Кi >=X і Кj < X, то потрібний обмін Кi <-> Кj при умові що i<j . Після першого обміну збільшуємо і на одиницю та шукаємо Кi >=X. Якщо такий елемент знайдено то j зменшуємо на одиницю і шукаємо Кj < X . Проводимо наступний обмін. Якщо Кi >=X не знайдено, а i>=j , то перша ітерація закінчена. Отже, алгоритм працює аналогіч­но "спалюванню свічки з обох кінців" . Таблиця буде розділена наступним чином: R1 ,R2 ,…,Ri-1, Ri ,Ri+1 ,…,Rn причому Kj < X , l=1,…,i-1; Km <= X, m=i+1,…,n. В лівій частині таблиці будуть стояти всі записи, що є меншими від базового, а в правій – що є більшими та сам базовий запис. Потім до кожної з цих підтаблиць застосовується даний метод рекурсивно. Рекурсія закінчується, коли всі під таблиці будуть складатися з одного запису, або вся таблиця буде впорядкована.

**2. Текст програми на мові програмування JAVA**

package \_asd\_lab1;

import java.util.\*;

/\*\*

\* Л/р № 1

\* Метод швидкого сортування

\* @author Taras

\*/

public class \_ASD\_Lab1 {

//чи показувати проміжні результати

static final boolean showIntermediateResults = true;

//поділ масиву

static int partition(int arr[], int left, int right)

{

int i = left, j = right;

int tmp;

//беремо значення елемента в середині відрізка

int pivot = arr[(left + right) / 2];

while (i <= j) {

//в лівій половині повинні залишитись менші за pivot

while (arr[i] < pivot)

i++;

//в правій половині повинні залишитись більші за pivot

while (arr[j] > pivot)

j--;

//якщо це не так - проводимо заміну

if (i <= j) {

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i++;

j--;

//виведемо проміжний результат, якщо це потрібно

if(showIntermediateResults) show(arr);

}

}

return i;

}

//функція сортування (рекурсивна)

static void quickSort(int arr[], int left, int right) {

//отримаємо індекс для поділу масиву на дві частини

int index = partition(arr, left, right);

if (left < index - 1)

//рекурсивно сортуємо ліву половину масиву

quickSort(arr, left, index - 1);

if (index < right)

//... або праву.

quickSort(arr, index, right);

}

//вивід одновимірного масиву на екран

static void show(int[] arr) {

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

System.out.print("" + arr[i] + " ");

System.out.println();

}

//перевірка чи масив відсортований у порядку зростяння значень елементів

static boolean checkResults(int[] arr) {

for(int i = 1; i < arr.length; i++)

if(arr[i] < arr[i-1]) return false;

return true;

}

//основна функція програми

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Array size: ");

if(!sc.hasNextInt()) {

// Якщо користувач ввів не ціле число виводимо відповідне повідомлення

System.out.println("It's not integer value! Program terminate...");

return;

}

// зчитуємо ціле число і створюємо масив

int[] a = new int[sc.nextInt()];

Calendar c = Calendar.getInstance();

Random rand = new Random(c.getTimeInMillis());

//ініціалізація едлементів масиву випадковими числами від 0 до 99

for(int i = 0; i < a.length; i++)

a[i] = rand.nextInt(100);

System.out.println("\nBefore sorting:");

show(a);

//засічемо час початку сортування

c = Calendar.getInstance();

long timeStart = c.getTimeInMillis();

//функція сортування (рекурсивна)

quickSort(a, 0, a.length-1);

//засічемо час завершення сортування

c = Calendar.getInstance();

long timeStop = c.getTimeInMillis();

System.out.println("\nAfter sorting:");

show(a);

System.out.println("\nThe result is " + checkResults(a) +

". Time elapsed: " + (timeStop - timeStart) + " msec.");

}

}

**3. Результат виконання програми**

Запустимо програму на виконання без виведення проміжних результатів на екран: для масиву розміром 20 елементів (рис. 3.1) та 1000 елементів (рис. 3.2).

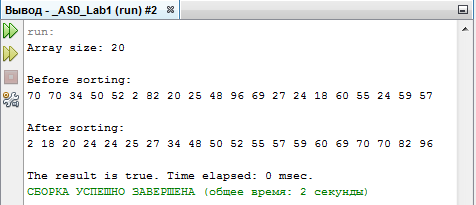


Рис. 3.1. Результат виконання програми

для масиву розміром 20 елементів

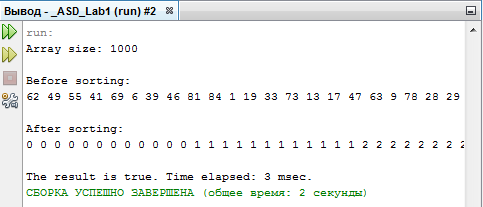


Рис. 3.2. Результат виконання програми

для масиву розміром 1000 елементів

Як бачимо на рисунках, програма виконує сортування в обох випадках правильно, про що свідчить повідомлення про перевірку результату. В першому випадку виконання алгоритму сортування займає час менше одної мілісекунд, а у другому – 3 мілісекунди.

Запустимо програму на виконання в режимі виведення проміжних результатів на екран: для масиву розміром 20 елементів (рис. 3.3) та 1000 елементів (рис. 3.4).

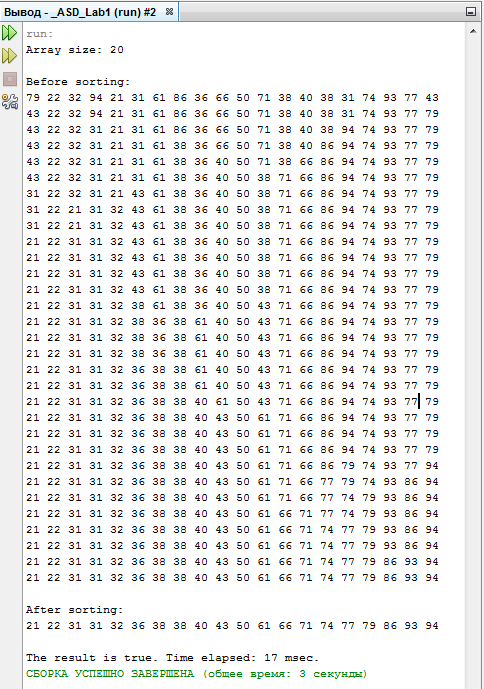


Рис. 3.3. Результат виконання програми

для масиву розміром 20 елементів

(в режимі виводу проміжних даних)

Як бачимо, час виконання програми для сортування 20-ти чисел, суттєво збільшився і становить 17 мілісекунд. Таку різницю можна пояснити значними затратами виводу більшого обсягу інформації на дисплей. Спробуємо вивести всі проміжні дані для масиву, розміром 1000 елементів (рис. 3.4):

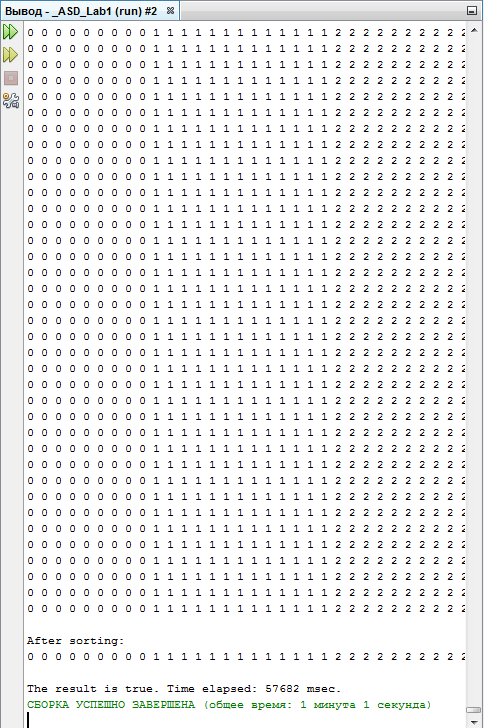


Рис. 3.4. Результат виконання програми

для масиву розміром 1000 елементів

(в режимі виводу проміжних даних)

В даному прикладі різниця в часі виконання – ще більша і становить 57682 мілісекунд.

## ВИСНОВКИ

На даній лабораторній роботі я навчився програмно реалізовувати алгоритм швидкого сортування, удосконалив свої навички у написанні підпрограм для роботи з масивами, а також оволодів основними функціями роботи з часом та генератором випадкових чисел у JAVA.