КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

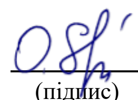
Кваліфікаційна робота

на здобуття освітнього рівня бакалавра
за спеціальністю 121 Інженерія програмного забезпечення
на тему:
ЕКСПЕРИМЕНТАЛЬНИЙ АНАЛІЗ СКЛАДНОСТІ АЛГОРИТМІВ

Виконав студент 4-го курсу
Андрій БЛАГИЙ

(підпис)

Науковий керівник:
доцент, кандидат фізико-математичних наук
Оксана ШКІЛЬНЯК


(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«___»________________202_р.,
протокол №____
Завідувач кафедри
Олександр ПРОВОТАР ______

(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 50 сторінок, 3 таблиці 33 ілюстрацій, 16 джерел посилань, 1 додаток.

АЛГОРИТМ, GARBAGE COLLECTOR, JAVA, JIT-КОМПІЛЯТОР, MICROBENCHMARK, ПОРЯДОК СКЛАДНОСТІ, СЕРЕДНІЙ ВИПАДОК, СКЛАДНІСТЬ, СОРТУВАННЯ, ЧАСОВА СКЛАДНІСТЬ.

Об’єктом роботи є дослідження та аналіз підходів до створення коду, який призначений для виклику різних методів, щоб дізнатись тривалість роботи методу. Предметом роботи є програмний код, який коректно визначає час роботи алгоритмів, та аналіз результатів.

Метою роботи є дослідження методів обчислення часу роботи підпрограми та вимірювання часу роботи алгоритмів сортування з подальшим аналізом.

Методи розроблення: виклик методів, за допомогою спеціалізованих бібліотек, щоб дізнатись тривалість виконання методів у максимально близьких до реальних умов. Інструменти розроблення: комерційне інтегроване середовище розробки Intellij IDEA Ultimate Edition (безкоштовно поширюване для студентів), мова програмування Java, інструмент для вимірювання часу роботи підпрограми Java Microbenchmark Harness.

Результати роботи: проаналізовано доцільність оптимізації алгоритмів, досліджено методи вимірювання часу роботи алгоритмів, розроблений код для вимірювання часу роботи алгоритмів сортування та проаналізовано результат.

Результат роботи може застосовуватись у навчальному процесі в курсі «Алгоритми та складність» а також «Комп’ютерна графіка», як пояснення доцільності вивчати швидші алгоритми, а також в рамках курсу «Об'єктно-орієнтоване програмування», як демонстрація роботи GC та JIT-компілятора.

ЗМІСТ

Скорочення та умовні позначення 4
ВСТУП 5
РОЗДІЛ 1. Теоретичні відомості про алгоритми 8
1.1 Історія розвитку алгоритмів 8
1.2 Часова та просторова складність алгоритмів 10
1.2.1 Оцінка порядку (О велике) 10
1.2.2 Найкращий та найгірший випадки 13
РОЗДІЛ 2. Інструменти та практичні підходи до підрахунку часу виконання підпрограми 15
2.1 Примітивне вимірювання часу виконання 15
2.2 Причини викидів даних 18
2.2.1 Вплив JIT компілятора 18
2.2.2 Вплив garbage collector 20
2.3Дослідження підходів до вимірювання часу виконання підпрограми 24
2.4 Java Microbenchmark Harness 25
РОЗДІЛ 3. Створення проекту для аналізу складності алгоритмів сортування 32
3.1 Створення проекту та його структура 32
3.1.1 Реалізації алгоритмів сортування 32
3.1.2 Реалізація вимірювання часу дії алгоритмів сортування 34
3.2 Отримані результати тестування 38
3.2.1 Неефективні алгоритми сортування 39
3.2.2 Швидкі алгоритми сортування 42
3.2.3 Сортування без порівнянь 43
ВИСНОВКИ 45
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 47
ДОДАТОК А 49

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

GC(Garbage Colector) – збирач сміття

JIT-compiler (Just in time compiller) – компіляція «на льоту»

JRE (Java Runtime Environment) – середовище виконання для Java

JVM (Java Virtual Machine) – віртуальна машина Java

IO (Input/Output) – ввід/вивід інформації

ВСТУП

Оцінка сучасного стану об'єкта розробки. Вимірювання часу роботи підпрограми на мові Java не є таким тривіальним процесом. Адже недостатньо підрахувати час до виклику методу та після, і обчислити різницю, оскільки на цей час впливає дуже багато факторів. Саме тому необхідно правильно вимірювати час роботи підпрограми і для цього у 2013 році інженерами, які розробляли JRE, було написано JMH, який дозволяє у близьких до реальних умов вимірювати час роботи підпрограми.

Актуальність роботи та підстави для її виконання. Активний розвиток комп'ютерної техніки, зокрема зростання потужності процесорів, забезпечував зростання швидкості дії програм. Проте останнім часом це зростання пригальмувало. Через це для зростання швидкості дії програм потрібно оптимізувати їх складові, які тривають найбільшу частину часу. Для цього потрібно вміти визначати правильно час роботи підпрограми, щоб можна було її оптимізувати. Саме тому доцільно проаналізувати що впливає на роботу програми під час власне виконання програми та підходи до вимірювання часу роботи підпрограми, щоб можна було порівнювати в максимально коректний спосіб. Порівнювати будемо на прикладі алгоритмів сортування, оскільки теоретична і практична часова складність може відрізнятись.

Мета й завдання роботи. Метою кваліфікаційної роботи є дослідження методів аналізу часової складності підпрограм і аналіз часової складності алгоритмів сортування з різними параметрами. Для досягнення цієї мети було поставлено такі завдання:

- Дослідити доцільність оптимізації алгоритмів.
- Дослідити які параметри варто враховувати при підрахунку часу роботи реалізації алгоритмів сортування.

- Дослідити недоліки визначення часу роботи підпрограми при примітивному вимірюванні.
- Структурувати вимоги до коректного вимірювання часу роботи підпрограми.
- Дослідити інструменти вимірювання часу роботи підпрограми.
- Розробити програмний код, який коректно визначатиме час дії алгоритму.
- Проаналізувати часову складність реалізацій алгоритмів у залежності від параметрів алгоритмів та від апаратних параметрів комп'ютера.

Об'єкт, методи й засоби розроблення. Об'єктом розроблення є програмний код, який аналізує часову складність реалізацій алгоритмів, у максимально близьких до реальних умов.

Розробці передував аналіз примітивного підходу до підрахунку часу, пошук помилок у такому підході та структуруванні вимог до коректного підрахунку часу. Після цього був здійснений аналіз існуючого інструменту, який при правильному використовуванні, забезпечує коректний підрахунок часу роботи підпрограми та аналіз його функціоналу.

Інструментом розробки було обрано комерційне інтегроване середовище розробки (IDE) Intellij IDEA Ultimate Edition, яке є безкоштовним для студентів, мова програмування – Java, інструмент для визначення тривалості роботи підпрограми – Java Microbenchmark Harness.

Можливі сфери застосування. Аналіз даних алгоритмів може гарно використовуватись як пояснення доцільності вивчати оптимізовані алгоритми під час навчального процесу в рамках курсу «Алгоритми та складність». Також демонстрація роботи GC і JIT-компілятора і їх вплив на швидкодію програми може використовуватись під час курсу «Об'єктно орієнтоване програмування». Вплив кешу процесору та його розміру може використовуватись для більш

наочного розуміння теми управління пам’яттю під час курсу «Системне програмування та операційні системи»

Взаємозв’язок з іншими роботами. За методами розробки та інструментальними засобами робота виконувалась сумісно із роботами з наступних дисциплін: «Алгоритми та складність» та «Об’єктно орієнтоване програмування».

РОЗДІЛ 1. ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО АЛГОРИТМИ

1.1 Історія розвитку алгоритмів

Алгоритм [1] – це послідовність чітко визначених інструкцій, призначених для вирішення деякої задачі. Якщо ж говорити неформально, то алгоритм– це будь-яка коректно визначена обчислювальна процедура, на вхід (input) якої подається деяке значення або набір значень, а результатом виконання якої є вихідне (output) значення або набір значень. Таким чином, алгоритм є послідовністю обчислювальних кроків, що перетворюють вхідні величини у вихідні.

Саме слово «алгоритм» походить від імені перського вченого Аль-Хорезми [2]. Близько 825 року він написав книгу про позиційну десяткову систему числення. У першій половині XII століття ця книга в латинському перекладі потрапила в Європу. Перекладач назвав її «Algoritmi de numero Indorum», тобто «Алгоритми про індійську лічбу». Таким чином латинізоване ім'я вченого потрапило в назву книги. Сьогодні вважається, що слово «алгоритм» потрапило в європейські мови саме завдяки цьому перекладу.

Першим же ж алгоритмом в його інтуїтивному розумінні (тобто кінцева послідовність примітивних дій, яка вирішує поставлене завдання), який дійшов до наших часів вважається запропонований Евклідом в III столітті до нашої ери алгоритм знаходження найбільшого спільного дільника двох чисел [3]. Протягом тривалого часу, приблизно до початку XX століття, слово «алгоритм» вживалось у сталому виразі «алгоритм Евкліда». Для опису покрокового вирішення інших математичних задач використовувалось слово «метод».

У 1843 році був описаний перший алгоритм на аналітичній машині Чарльза Беббіджа [4]. Це був алгоритм обчислення чисел Бернуллі написаний Адою

Лавлейс. Його визнали першою програмою, яка була реалізована для виконання на ЕОМ.

Проте бурхливий розвиток дослідження алгоритмів починається з 1930-х років. Стівен Кліні у 1935 році розробив перше формулювання теорії рекурсії [5]. Алан Тюрінг та Еміль Пост, незалежно один від одного, в 1936 році опублікували роботи, в яких були визначено поняття алгоритму.

Формальні засоби для представлення алгоритмів мали не тільки теоретичне значення. Розробка алгоритмічних мов програмування обчислювальних пристроїв розпочалась у 1951 році із публікації німецького інженера Ганса Рутісхаузера. Після створення перших мов програмування розробка алгоритмів мала уже більш практичне значення, як і питання їх оптимізації.

Проте активний розвиток обчислювальної техніки міг перекрити потребу в швидших алгоритмах. Так в 1965 році Гордон Мур помітив закономірність зростання кількості транзисторів на мікросхемі [6]. Адже кожен рік виходила нова модель мікросхеми, і кількість транзисторів зростала вдвічі. Це спостереження було названо «закон Мура». Проте вже за 10 років періодичність зростання кількості транзисторів була збільшена до двох років.

Це зростання кількості транзисторів було обумовлено не збільшенням кристалу чипу, а зменшенням розміру транзистору. Уже зараз розмір транзистору у сучасних процесорах досягає 10 нм. Анонсовані 7 нм та 5 нм процесори. А 5 нм це шар товщиною в 20 атомів. Очевидно, що в ближчому майбутньому закон Мура перестане працювати. Єдина можливість здійснити новий прорив у кількості транзисторів це створення квантових комп’ютерів. Проте поки до їх створення ще дуже далеко. Тому в найближчому майбутньому технології не зможуть активно розвиватись, і тоді оптимізація алгоритмів буде як ніколи раніше в пріоритеті.

1.2 Часова та просторова складність алгоритмів

Існує кілька способів виміряти складність алгоритму. Програмісти зазвичай зосереджуються на часу виконання, проте доволі важливими також є вимоги до об'єму пам'яті та вільного місця на диску. Використання швидшого алгоритму, проте який потребує більше пам'яті, ніж у комп'ютера не приведе до очікуваних результатів.

Багато алгоритмів пропонують вибір між використанням пам'яті чи швидкістю [7]. Задачу можна вирішити швидко, маючи в своєму розпорядженні велику кількість пам'яті. Типовим прикладом може бути алгоритм пошуку найкоротшого шляху. Представивши карту міста у вигляді графу, можна написати алгоритм для пошуку найкоротшої відстані між двома будь-якими точками в цьому графі. Щоб не підраховувати кожен раз цю відстань, ми можемо підрахувати найкоротші відстані між всіма точками і зберегти ці результати в таблиці. Коли нам треба буде дізнатись найкоротшу відстань між парою точок, ми зможемо просто отримати це значення із таблиці. Результат буде отриманий миттєво, проте ця таблиця буде використовувати велику кількість пам'яті. Оскільки карта великого міста може містити сотні тисяч точок, тому кількість елементів таблиці буде вимірюватись у десятках мільярдів клітинок.

У цій роботі будемо аналізувати часову складність алгоритмів.

1.2.1 Оцінка порядку (O велике)

При порівнянні різних алгоритмів важливо знати, як їх складність залежить від об'єму вхідних даних. Припустимо, що при сортуванні тисячі елементів, один алгоритм працюватиме 1 секунду, а при мільйону – 10 с , а інший – 2 с і 5 с відповідно. У таких умовах не можна однозначно сказати який алгоритм кращий.

Для цього складність алгоритму оцінюють порядком величини. Алгоритм має складність O(f(n)), якщо при збільшенні порядку вхідних даних n складність алгоритму зростає з тією ж швидкістю, що і функція f(n).

Оцінюючи порядок складності алгоритму, потрібно брати лише найбільшу частину, оскільки при безкінечному зростанні розміру вхідних даних вона буде вносити найбільший вклад в зростання часу роботи алгоритму (рисунок 1). Наприклад, нехай кількість операцій буде $N^2 + N$. Тоді при зростанні N, вплив другого доданку буде зменшуватись. Тому складність буде $O(N^2)$, оскільки другим доданком ми нехтуємо. Також, при підрахуванні O можна не враховувати константні множники, тобто алгоритм з кількістю операцій $3N^2$ матиме складність $O(N^2)$.

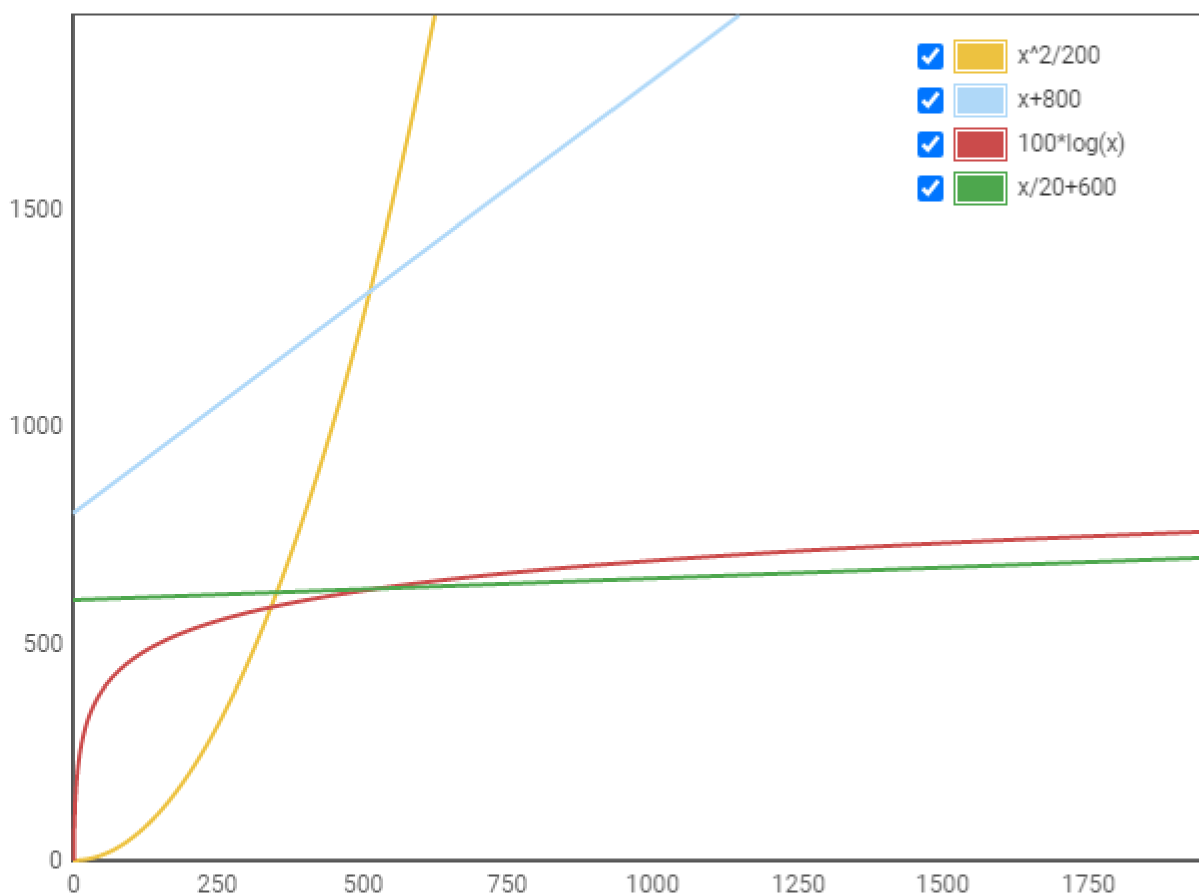


Рисунок 1 – Графіки зростання функцій

Зазвичай, складність алгоритмів буде визначатись однією із наступних функцій (перелічені в порядку зростання складності):

1. C:

   Константна складність зазвичай буде в простих алгоритмах або отриманні даних з певного масиву/матриці. Наприклад, отримання

відстані між двома точками, які ми попередньо підрахували і зберегли в матриці (з п.1.2.1).

2. ln(N):

   Логарифмічна складність з'являється зазвичай у результаті зменшення розміру задачі на певне значення на кожному кроці ітерації алгоритму. В логарифмічному алгоритмі неможлива робота із усіма вхідними даними.

3. N:

   Така складність буде в алгоритмів, які сканують список, який складається із N елементів, наприклад, алгоритм пошуку елементів, які відповідають певній умові, методом прямого перебору.

4. N*ln(N):

   Така складність буде в алгоритмів декомпозиції, тобто в тих, які працюють за принципом «розділяй та володарюй». Зазвичай це алгоритми швидкого сортування.

5. N^2:

   Квадратична складність буде в алгоритмів, які містять два вкладені цикли. Типові приклади це – прості алгоритми сортування та цілий ряд операцій, які виконуються над матрицями розміру N*N.

6. N^3:

   Кубічна складність буде в алгоритмів які містять три вкладені цикли. Зазвичай це складні алгоритми лінійної алгебри, такі як множення матриць.

7. C^N, C>1:

   Експоненціальна складність буде в алгоритмів, які виконують обробку всіх підмножин деякої множини, яка складається з N елементів. Часто під терміном «експоненціальна» мають на увазі значно вищі, ніж експонента, порядки зростання.

8. N!:

Факторіальна складність є типовою для алгоритмів, які виконують обробку всіх можливих перестановок множини, яка складається з N елементів.

1.2.2 Найкращий та найгірший випадки

Час роботи алгоритму також може залежати від вхідних даних. Для кращого розуміння розглянемо алгоритм пошуку елемента в списку. Очевидно, що найкращий випадок буде, коли шуканий елемент є першим у вхідному списку. В гіршому випадку нам потрібно буде перевірити кожен елемент, тобто шуканий елемент є останнім в списку, або відсутній взагалі. Проте в середньому, якщо припустити що шуканий елемент є в списку, і кожен елемент списку однаково ймовірно є шуканим, то пошук відвідає n/2 елементів.

Якщо ж розглянути більш складний алгоритм, наприклад, сортування вставкою, то кращий, середній та гірший випадки вхідних даних також будуть присутні. Наведемо приклад алгоритму сортування вставкою (рисунок 2) написаний мовою Java.

```
void sort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Рисунок 2 – Алгоритм сортування вставкою на мові Java

У залежності від того, наскільки відсортованим прийде вхідний масив, кількість ітерацій буде кардинально відрізнятись. Коли вхідний масив прийде повністю відсортований, то тіло циклу `while` ні разу не виконається (оскільки `arr[i-1]<arr[i]` при всіх значеннях `i`), і складність у цьому випадку буде O(n), тобто лінійна. Проте коли вхідний масив прийде відсортований у зворотному порядку, то цикл `while` виконається `i` разів, для кожного значення `i`, тоді складність алгоритму буде $O(n^2)$, тобто квадратична. У середньому ж, цикл `while` виконається `i/2` разів, оскільки в середньому потрібно буде переглянути половину елементів масиву `arr[1..i]`. Оскільки всього буде таких n ітерацій, то відповідно складність буде також квадратична.

Як уже можна помітити, порядок складності середнього випадку майже нічим не відрізняється від найгіршого випадку. Проте існують алгоритми, в яких в гіршому випадку складність буде зростати дуже сильно, і тоді процес виконання алгоритму може закінчитись непередбачуваними помилками.

РОЗДІЛ 2. Інструменти та практичні підходи до підрахунку часу виконання підпрограми

2.1 Примітивне вимірювання часу виконання

Мовою програмування, на якій ми будемо писати алгоритми та вимірювати їх час виконання, ми вибрали Java. Адже вона зараз активно використовується для серверної частини додатків, де потрібно працювати з великою кількістю даних [8].

Перед тим як почати вимірювати час виконання алгоритмів слід більш детально розібратись, як це слід робити правильно. На перший погляд все просто:

1. визначити час перед виконанням алгоритму;
2. виконати алгоритм;
3. визначити час після виконання алгоритму.

Різниця часу перед виконанням алгоритму та після і буде часом виконання алгоритму. Давайте підрахуємо час виконання алгоритму з п. 1.2.2 для найгіршого випадку (тобто відсортований у зворотному порядку масив) на масиві довжиною 1000 елементів. Час будемо міряти у наносекундах. Результат виконання зобразимо у таблиці 1.

Таблиця 1 – Час виконання алгоритму сортування в залежності від порядкового номеру ітерації сортування

Номер ітерації	Час, нс
1	15,389,700
2	5,651,400
3	4,327,700
4	1123,600
5	863,400
6	732,500
7	675,800
8	2,033,900
9	954,500

При послідовному виконанні цього алгоритму кілька разів ми отримали кардинально суперечливі дані. Адже час виконання алгоритму при кожному наступному виконанні зменшується. А також незрозуміле зростання часу виконання на 8 кроці. Проте всі ці дані насправді цілком логічні. Адже спершу віртуальній машині слід розігрітись [9]. Адже в процесі розігріву запускається JIT компілятор, який оптимізує виконуваний код. Через це рекомендують спершу виконати кілька разів алгоритм, щоб уникнути зростання швидкодії в процесі аналізу. А на 8 кроці, найбільш ймовірно, був запущений збір сміття [10].

Також ми виміряли час роботи алгоритму на масивах розміру 5 та 10 елементів. Результати вимірювання зображені в таблиці 2.

Таблиця 2 – Час сортування масивів довжиною 5 та 10 елементів

Номер запуску	Час виконання для масиву розміру 5, нс	Час виконання для масиву розміру 10, нс
1	60,700	66,500
2	4,700	6,700
3	3,500	5,600
4	6,400	5,300
5	3,700	5,000
6	3,300	4,900
7	2,700	5,600
8	2,900	5,200
9	3,300	5,200

Як ми вже знаємо, перші запуски можна відкинути, оскільки відбувається прогрів. Проте якщо проаналізувати всі наступні, то можна побачити що час сортування вдвічі більшого масиву більший менш ніж вдвічі. Хоча, згідно із пунктом 1.2.2, для даного алгоритму для найгіршого випадку складність має рости квадратично.

Проте в даному підрахунку часу роботи алгоритму була допущена помилка, оскільки метод System.nanoTime() теж триває певний час, який в даному випадку не є значно менший за час роботи алгоритму, тому вносить значний вклад в час, який ми виміряли [11]. Тому слід міряти час лише для великих масивів даних, щоб вклад методу System.nanoTime() був мінімальний.

Також під час виконання алгоритму операційна система могла переключитись на інший потік, або ж була запущена процедура збору сміття (Garbage collector). Тому слід трішки виправити метод підрахунку часу виконання алгоритму. Одним із варіантів корегування буде виконання алгоритму більше ніж один раз для більш точного підрахунку. Розглянемо попередній

алгоритм, проте будемо міряти час виконання не 1, а кілька запусків, для масиву довжиною 1000 елементів. У результуючій таблиці будемо показувати час однієї ітерації алгоритму. Ці дані представимо у таблиці 3.

Таблиця 3 – Час виконання однієї ітерації алгоритму в залежності від кількості запусків

Кількість запусків	Час виконання однієї операції, нс
1	13,919,000
10	1,683,420
100	452,953
1000	703,987
10000	401,367
100000	322,453
1000000	344,967

2.2 Причини викидів даних

2.2.1 Вплив JIT компілятора

JIT компілятор [12] – це компонент середовища виконання Java, який покращує продуктивність програм Java під час роботи. Ніщо в JVM не впливає на продуктивність більше, ніж компілятор, і вибір компілятора є одним із перших рішень, прийнятих під час запуску програми Java - незалежно від того, ви Java-розробник, чи кінцевий користувач.

Ключ до слогану Java "Write once, run everywhere" є байт-код. Шлях перетворення байт-коду у відповідні нативні інструкції для додатку має величезний вплив на швидкість додатку. Адже байт-код може бути інтерпретований, скомпільований у нативний код, або ж виконаний на процесорі, архітектура якого відповідає специфікації байт-коду. Інтерпретація байт-коду є

стандартною реалізацією JVM (віртуальна машина Java), проте вона уповільнює виконання програми. Для підвищення продуктивності, JIT-компілятори взаємодіють з JVM під час виконання програми, та компілюють відповідні послідовності байтів у нативний машинний код. При використанні JIT-компілятора, апаратне забезпечення може виконувати нативний код, замість того, щоб JVM повторно інтерпретував одну і ту ж послідовність байт-коду і включав штраф за тривалий процес інтерпретації. Це призводить до зменшення часу виконання програми, якщо методи виконуються часто. Проте, якщо методи використовуються недостатньо часто, то використання JIT-компілятора може призвести до збільшення часу виконання, оскільки час компіляції байт-коду буде додано до загального часу виконання.

JIT-компілятор виконує певні оптимізації під час компіляції байт-коду. Оскільки він перекладає ряд байт-кодів у нативні інструкції, він може виконати ряд простих оптимізацій. Наприклад, аналіз даних, перекласти операції зі стеку на операції на регістрах, зменшення часу доступу до памяті шляхом розподілу регістрів, усунення мертвого коду, тощо. Чим вище ступінь оптимізації коду, тим більше часу потрібно на цю оптимізацію. Саме тому JIT-компілятор не може собі дозволити виконувати всі оптимізації, які може виконувати статичний компілятор, як через накладні витрати, так і тому що він має лише частинку програми, тобто не має всієї інформації.

JIT-компілятор це компонент JRE (Java Runtime Environment), який покращує продуктивність Java програм під час виконання. Java програми складаються із класів, які містять нейтральний для платформи байт-код, який може інтерпретуватись JVM на багатьох різних комп’ютерних платформах. Під час виконання JVM завантажує файли класів, визначає семантику кожного окремого байт-коду та виконує відповідні обчислення. Додатковий процесор та використання пам'яті під час інтерпретації означає, що Java-програма працює повільніше, ніж нативна програма. Компілятор JIT допомагає покращити

продуктивність програм Java, компілюючи байт-код у нативний машинний код під час виконання (рисунок 3).

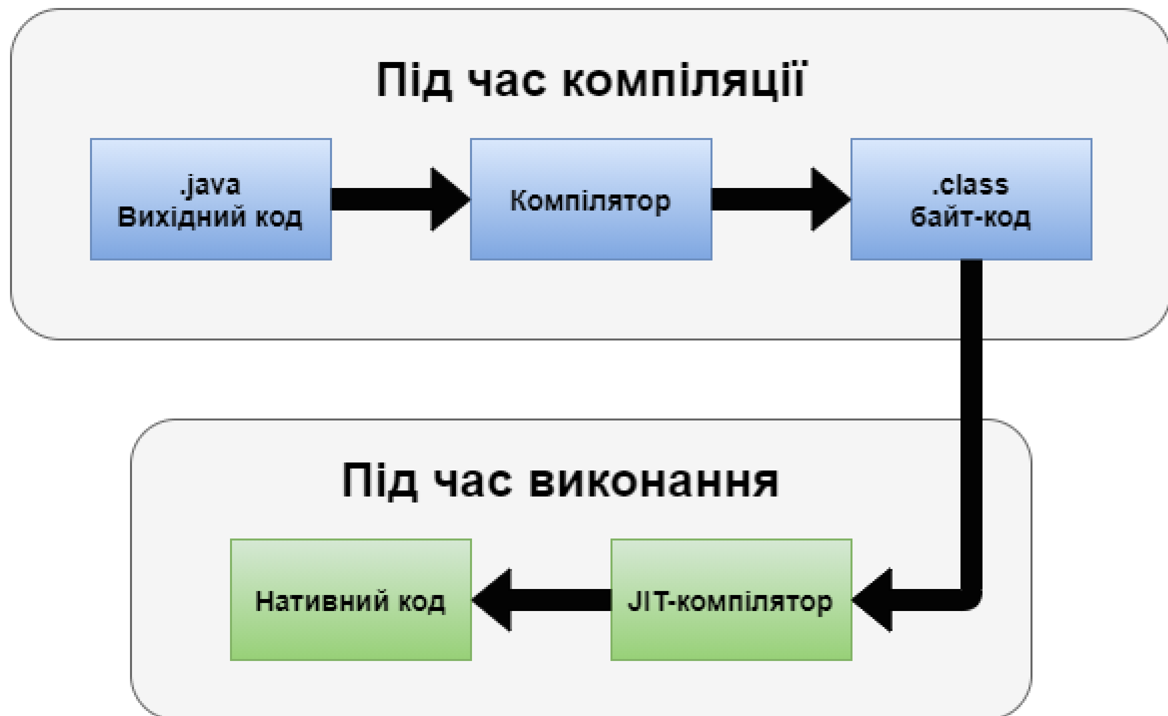


Рисунок 3 – Компіляція вихідного коду у проміжний байткод та у нативний код

JIT-компілятор включено за замовчуванням і він може активуватись при виклику метода в Java. Теоретично, якщо він скомпілює всі методи в програмі, то час її виконання наблизиться до часу виконання нативної програми. Проте ця компіляція займе доволі багато часу і ресурсів.

Коли був вибраний метод для компіляції, JVM передає свій байт-код до JIT-компілятора. Він повинен розуміти семантику і синтаксис байт-коду перш ніж він зможе скомпілювати метод. Для цього викликається метод Trace tree. Після цього компілятор аналізує дерево методу, оптимізує і компілює його. При цьому ці оптимізації можуть залежати від архітектури пристрою.

2.2.2 Вплив garbage collector

Збір сміття [13] – це процес відновлення невикористаної пам’яті під час виконання програми шляхом знищення невикористовуваних об’єктів.

У таких мовах як С, С++ розробник повинен сам відповідати за створення і видалення об’єктів, які є непотрібні. Іноді розробник забуває видалити зайві об’єкти. І тоді виділена під них пам'ять не звільняється, і пам'ять використовувана під програму продовжує зростати, допоки не залишиться пам’яті для виділення. Таке називають витоком пам’яті (memory leak). І тоді програма завершиться незаплановано із OutOfMemoryErrors. Для того, щоб видалити об’єкт у С та С++ використовуються методи free та delete відповідно. У Java ж, як і у більшості високорівневих мов програмування, збір сміття виконується автоматично під час роботи програми. Це позбавляє від необхідності видаляти пам'ять вручну і дозволяє уникнути витоку пам’яті.

Для очищення пам’яті використовується Garbage Collector (збирач сміття). Коли програми Java працюють на JVM, об’єкти створюються в купі, яка є частиною пам’яті, виділеної програмою. Протягом життя програми Java створюються та випускаються нові об'єкти. Зрештою, деякі об’єкти більше не потрібні. Можна сказати, що в будь-який момент часу пам’ять купи складається з двох типів об’єктів:

- Живі – це об’єкти, які ще використовуються, і до них можна звернутись.
- Мертві – це об’єкти, які більше не використовуються, і на них ніхто більше не посилається.

Збирач сміття збирає ці мертві об’єкти та видаляє їх, щоб звільнити пам'ять. Для того, щоб їх віднайти, від проходиться по графу об’єктів у пам’яті, починаючи з коренів, і слідуючи за посиланнями на інші об’єкти. Він позначає кожен об’єкт, який він відвідав як живий (рисунок 4). Об’єкти, які він не зміг відвідати, є сміттям і будуть знищені.

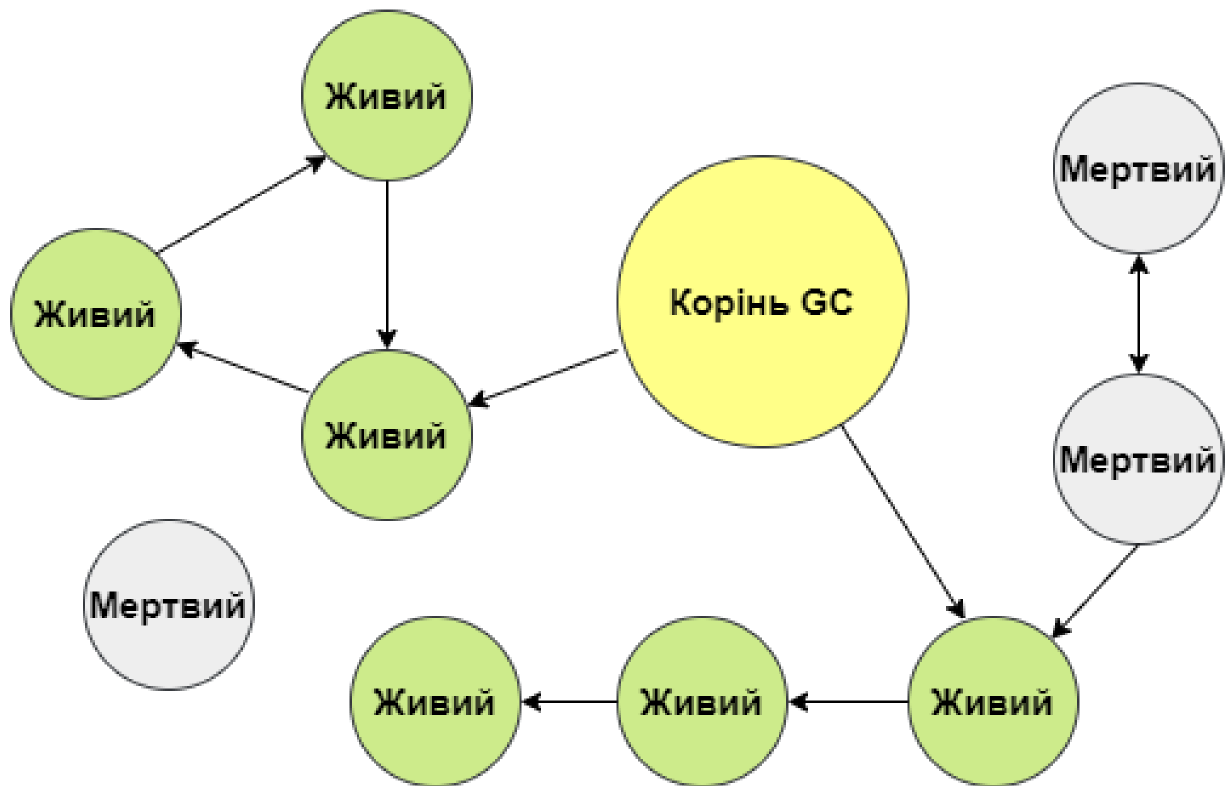


Рисунок 4 – Схематичне зображення графу живих та мертвих об’єктів

Після того, як всі об’єкти були позначені як відвідані чи ні (рисунок 5), потрібно видалити всі невідвідані об’єкти (рисунок 6).

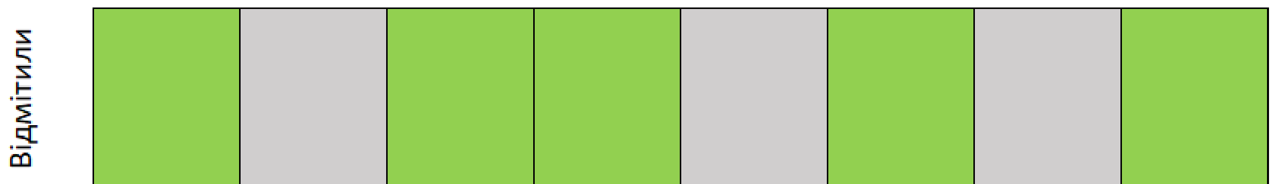


Рисунок 5 – Схематичне зображення живих та мертвих об’єктів у оперативній пам’яті

Рисунок 6 – Схематичне зображення живих об’єктів у оперативній пам’яті одразу після видалення мертвих об’єктів

Мертві об'єкти не обов'язково будуть знаходитись поряд, і після видалення буде отриманий фрагментований простір пам'яті (рисунок 6). Її можна ущільнити, щоб всі об'єкти знаходились на початку купи (рисунок 7). Адже після процесу ущільнення полегшується послідовний розподіл пам'яті для нових об'єктів.

Рисунок 7 – Ущільнення живих об'єктів у оперативній пам'яті

При запуску збірки сміття зазвичай всі процеси зупиняються. Через це користувачам здається, що програма зависає (рисунок 8). Для ручного виклику очищення сміття може використовуватись метод System.gc(), проте в звичайних ситуаціях це робити не рекомендується.

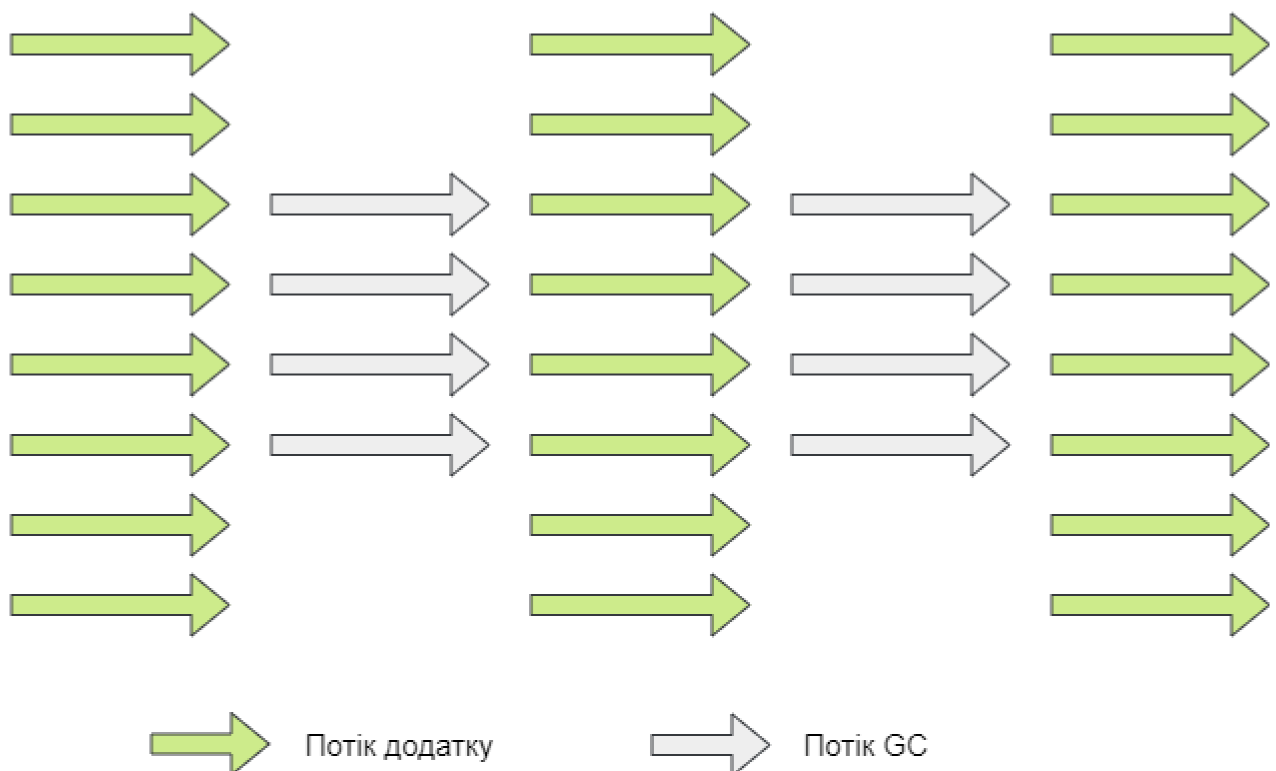


Рисунок 8 – Схематичне зображення потоків роботи програми та збирача сміття

2.3Дослідження підходів до вимірювання часу виконання підпрограми

У попередніх підрозділах було проаналізовано примітивне вимірювання часу роботи підпрограми, які є недоліки, чому вони виникають і як можна їх уникнути. Зараз постараємось це все підсумувати та дізнатись основні підходи до вимірювання часу роботи підпрограми.

1. Включати фазу прогріву. Адже під час цієї фази JIT-компілятор оптимізує виконуваний код, через що він в перші ітерації може працювати повільніше, ніж в наступних.
2. Збільшити розмір пам'яті, яку буде використовувати програма, для того, щоб Garbage Collector рідше запускався під час підрахування часу роботи алгоритму.
3. Між різними тестами запускати System.gc(), щоб різні тести отримували однаково чистий простір пам'яті, яку він буде використовувати, щоб уникнути розбіжності між тестами в залежності від послідовності запуску.
4. Запускати тест достатньо довго, щоб збільшити точність підрахунку часу роботи алгоритму.
5. Використовувати результати роботи підпрограми, щоб JIT-компілятор не видалив виконання всього алгоритму в фазі оптимізації.
6. Не виводити текст під час засікання часу, так як доступ до IO триває не мало часу.
7. Детально аналізувати дані, які ми отримаємо, щоб уникнути раптового зростання на одній із ітерацій. Запускати кілька разів.
8. Запускати тест на машині, яка не завантажена іншою роботою.

Якщо якийсь із цих пунктів порушений, то вимір часу роботи підпрограми буде некоректний, оскільки на час роботи впливатиме не тільки власне код, а і інші показники. Звісно, можна виконати всі ці пункти вручну, написавши багато інфраструктурного коду. Проте можна скористатись готовими бібліотеками. Однією із них є Java Microbenchmark Harness.

2.4 Java Microbenchmark Harness

Java Microbenchmark Harness або JMH [14] – це інструмент для створення мікробенчмарків у Java. Інструмент створений тією ж командою, яка розробила віртуальну машину Java, тому що вони знали, як влаштована «під капотом» JVM і який найкращий спосіб впровадити аналіз часу роботи підпрограми в максимально близьких до реальних умов.

Для того, щоб почати користуватись JMH, його потрібно підключити до проекту, додавши у pom.xml наступні залежності (рисунок 9). Використаємо останню релізну версію 1.29. Знайти найновішу версію можна на https://mvnrepository.com.

```
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.29</version>
</dependency>
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>1.29</version>
    <scope>provided</scope>
</dependency>
```

Рисунок 9 – Залежності які слід додати щоб користуватись JMH

Тепер, для того, щоб проаналізувати час роботи методу, над ним потрібно написати анотацію Benchmark над методами, які хочемо проаналізувати. Приклад використання такої анотації зображено на рисунку 10.

```java
@Benchmark
public void init() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

Рисунок 10 – Приклад використання анотації Benchmark

А також написати main метод, в якому будемо запускати процес порівняльного тестування (рисунок 11).

```java
public static void main(String[] args) throws Exception {
    Options opt = new OptionsBuilder()
            .include(BenchMark.class.getSimpleName())
            .build();
    new Runner(opt).run();
}
```

Рисунок 11 – Приклад main методу

Після запуску головного методу можна побачити наступний результат (рисунок 12) та доволі великі логи (додаток А).

```
Result "benchmark.BenchMark.init":
  0.990 ±(99.9%) 0.001 ops/s [Average]
  (min, avg, max) = (0.988, 0.990, 0.991), stdev = 0.001
  CI (99.9%): [0.989, 0.990] (assumes normal distribution)


# Run complete. Total time: 00:08:30

REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on
why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial
experiments, perform baseline and negative tests that provide experimental control, make sure
the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.

Benchmark       Mode  Cnt  Score   Error  Units
BenchMark.init  thrpt   25  0.990 ± 0.001  ops/s
```

Рисунок 12 – Кінцевий вивід запуску головного методу для тестового бенчмарку

Як можна побачити з рисунку 12, в середньому 0.99 операцій методу init (рисунок 10) встигає виконатись за 1 секунду. І похибка становить приблизно 0.1% або ж одну тисячну операції за секунду.

Проте це лиш маленька частинка того, що може JMH. Якщо більш детально переглянути вивід (додаток А), то можна побачити різні терміни, наприклад, fork, warmup iteration та iteration [15].

Fork – це значення, скільки разів треба повторити випробування. Адже у одному випробуванні може бути викид даних. Це значення можна задати через анотацію Fork над методом або класом і вказати там кількість (рисунок 13). За замовчуванням ця кількість дорівнює 5.

```
@Benchmark
@Fork(3)
public void init() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

Рисунок 13 – Приклад коду, де кількість повторень випробування перевизначили. Тепер буде 3 випробування.

Warmup – це кількість запусків методу в рамках одного випробування, результати якого він відкидає. Це і є фаза розігріву JVM, в рамках якої буде завантаження класів, JIT-компіляція, і тд. Це значення можна задати через анотацію Warmup над методом або класом і вказати там кількість (рисунок 14). За замовчуванням ця кількість дорівнює 5

```
@Benchmark
@Warmup(iterations = 7)
public void init() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

.

Рисунок 14 – Приклад коду, де кількість запусків методу в фазі розігріву перевизначили. Тепер буде 7 таких запусків.

Iteration – це кількість запусків методу в рамках одного випробування, результати якого будуть враховуватись. Цю кількість можна задати через анотацію Measurement над методом або класом, і вказати там кількість (рисунок 15) . За замовчуванням ця кількість дорівнює 5.

```
@Benchmark
@Measurement(iterations = 2)
public void init() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

Рисунок 15 – Приклад коду, де кількість ітерацій перевизначили. Тепер таких запусків буде 2.

Окрім кількостей ітерацій, за допомогою даних анотацій можна задати тривалість ітерації та кількість викликів в рамках однієї ітерації (рисунок 16). За замовчуванням це 10 секунд та необмежена кількість відповідно.

```
@Benchmark
@Warmup(time = 1, timeUnit = TimeUnit.HOURS, batchSize = 100)
@Measurement(time = 10, timeUnit = TimeUnit.MINUTES, batchSize = 15)
public void init() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

Рисунок 16– Приклад коду, де перевизначили тривалість ітерацій та максимальну кількість ітерацій для розігріву та тесту

Якщо метод не закінчить працювати після закінчення часу ітерації, програма зачекає на закінчення методу 10 хвилин. Якщо ж не дочекається, то даний тест завершиться з помилкою. Це значення також можна перевизначити (рисунок 17).

```
@Benchmark
@OutputTimeUnit(TimeUnit.HOURS)
public void init() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

Рисунок 17 - Приклад коду, де час до аварійного завершення тесту перевизначили

Також можна вибрати режим тестування. В JMH всього є 4 режими, а також один спільний:

1. Throughput. Цей режим використовується для вимірювання пропускної здатності коду, тобто кількості випадків виконання методу за певний проміжок часу. Зазвичай цей проміжок часу – це секунда, проте його можна змінити.
2. AverageTime. Цей режим показує середній час, необхідний для запуску тестового методу.
3. SampleTime. Цей режим працює за допомогою методів безперервного виклику, та випадковим чином вибирає час потрібний для виклику. Тобто, він сам регулює частоту дискретизації, але може пропустити деякі паузи. Він також, як і AverageTime, показує час, який потрібний на одну операцію.
4. SingleShotTime. Цей режим одноразово викликає метод без фази розігріву. Використовується, коли хочеться дізнатись час холодного старту.
5. All. Цей режим по черзі викликає всі вище перелічені режими. Використовується для отримання консолідованого результату усіх режимів, а також при тестуванні самого JMH на етапі розробки.

Також можна змінити одиниці виміру часу, за допомогою анотації OutputTimeUnit над класом або методом, вказавши там необхідну одиницю вимірювання (рисунок 18).

```
@Benchmark
@OutputTimeUnit(TimeUnit.MINUTES)
public void init() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

Рисунок 18 – Приклад коду, де одиниці виміру часу перевизначили

Всі ці кастомізації можна зробити не тільки через анотації. Їх також можна зробити через OptionsBuilder, і вони там матимуть більшу силу. Також через OptionsBuilder можна вказати додаткові опції, наприклад, які бенчмарки включити або виключити, чи включати збирач сміття між кожною ітерацією,

кількість потоків, у яких будуть запускатись бенчмарки, змінити опції jvm, а також куди вивести результат, наприклад, в файл певного формату, або консоль, назву цього файлу, назву файлу з більш детальними даними.

Якщо ж буде потрібно ініціалізувати змінні, не в рамках тесту, можна скористатись змінними стану. Це такі змінні, які декларуються у спеціальних класах стану і можуть передаватись в тест в якості параметру. Щоб об'явити клас стану, потрібно над ним додати анотацію State. Також у цього об'єкту повинен бути публічний непараметризований конструктор, і сам клас повинен бути публічним. Цей об'єкт може бути унікальним для кожної ітерації, або ж використовуватись повторно у цьому потоці, групі потоків, або ж використовуватись для всіх тестів.

Також у ньому можна об'явити метод налаштування та скидання. Їх слід помітити анотацією Setup та TearDown відповідно [11]. Ці методи будуть викликані перед та після виклику методів з тестами. І його виклик не буде входити у вимірювання часу виконання алгоритму. При цьому можна задати різний рівень, коли слід викликати метод. Всього є три рівні:

1. Trial. Це рівень за замовчуванням. Метод викликається один раз для повного циклу тесту, тобто для ітерацій розігріву та вимірювання часу.
2. Iteration. Метод викликається один раз для кожної ітерації.
3. Invocation. Метод буде викликатись для кожного виклику тестового методу. Не рекомендується недосвідченим користувачам, оскільки цей метод повинен викликатись між викликами методу, який тестуємо, та його час не повинен впливати. Тому тут буде вимірюватись час кожного окремо взятого тесту, а як відомо, отримання поточного часу теж дороговартісна операція, і тому слід дуже обережно використовувати цей рівень виклику методів.

Також можна параметризувати тестовий метод. Для цього слід використати анотацію Param над полем класу стану, щоб сказати JMH вставити туди значення із параметру анотації (рисунок 19).

```java
@Param({"100", "1000"})
public int time;

@Benchmark
public void init() throws InterruptedException {
    Thread.sleep(time);
}
```

Рисунок 19 – Приклад використання анотації Param

Тоді тестовий метод буде викликаний для кожної комбінації вхідних параметрів (рисунок 20). Також за допомогою цієї анотації уникається оптимізація від JIT-компілятора на підрахунок математичного виразу, оскільки він не знатиме точно, яке значення завжди буде використовуватись у цьому методі [11].

```
Benchmark        (time)   Mode  Cnt  Score   Error  Units
BenchMark.init      100  thrpt    3  9.234 ± 0.510  ops/s
BenchMark.init     1000  thrpt    3  0.993 ± 0.017  ops/s
```

Рисунок 20 – Результат вимірювання параметризованого методу init

РОЗДІЛ 3. СТВОРЕННЯ ПРОЕКТУ ДЛЯ АНАЛІЗУ СКЛАДНОСТІ АЛГОРИТМІВ СОРТУВАННЯ

3.1 Створення проекту та його структура

Для правильного підрахунку часу виконання алгоритмів спершу слід створити проект і його правильно сконфігурувати. В якості інтегрованого середовища розробки було використано Intellij IDEA (Ultimate Edition), а в якості засобу збірки – Maven. В файл pom.xml було додано залежність на JMH. Для самого підрахунку часу виконання було розроблено різні програмні класи.

3.1.1 Реалізації алгоритмів сортування

Першим розробленим класом є SortAlgorithms. У ньому є статичний метод на вхід якому подається вибраний алгоритм сортування та масив, який слід відсортувати. В залежності від вибраного алгоритму викличеться відповідний метод сортування. У ньому реалізовані наступні алгоритми сортування:

- Bubble sort [16] – сортування бульбашкою. Це найпопулярніший простий алгоритм сортування, який багаторазово перебирає список, порівнює сусідні елементи та обмінює їх місцями, якщо вони в неправильному порядку. Проходження по списку повторюється, поки він не буде відсортований.
- Selection sort – сортування вибором. Це доволі простий алгоритм сортування, в якому вхідний список поділений на дві частини: відсортовану та ні. Під час ітерації алгоритм шукає найменший елемент у невідсортованій частині та міняє його з крайнім зліва елементом, і зменшує її довжину на 1. Таким чином довжина відсортованої частини буде зростати.

- Insertion sort – сортування вставкою. В цьому алгоритмі вхідний список також поділений на дві частини, проте на відміну від попереднього, береться перший елемент з невідсортованої частини та вставляється в середину відсортованої.
- Merge sort – сортування злиттям. Це алгоритм оснований на принципі «розділяй та володярюй». Вхідний масив ділять навпіл, сортують дві половинки окремо, і потім їх об'єднують. При цьому ці половинки можуть сортуватись як і сортуванням злиття (тобто рекурсивно викликати себе), так і більш простими алгоритмами. Також це сортування можна розпаралелити, тобто запускати сортування половинок масиву в різних потоках.
- Quick sort – швидке сортування. Цей алгоритм також оснований на принципі «розділяй та володярюй». В цьому алгоритмі сортування масив також розділяється на дві частини відносно певного елементу масиву. І кожен елемент буде або більший або менший від вибраного та піде у правий чи лівий підмасив відповідно. Ці підмасиви сортуються рекурсивно таким сортуванням, або будь-яким іншим. Після сортування цих підмасивів сам масив вже буде відсортований і не потрібно буде виконувати операції об'єднання.
- Heap sort – сортування купою. Цей алгоритм називають вдосконаленим сортуванням вибору. Під час ітерації алгоритм так само ділить вхідний масив на відсортовану та невідсортовану область, проте найбільший елемент у невідсортованій області шукається не лінійним пошуком, а шляхом використання купи.
- Counting sort – сортування підрахунком. В цьому сортуванні не використовується порівняння елементів. Натомість, для числового значення кожного елементу збільшується відповідний індекс у масиві підрахунків. І на основі цього масиву, буде створюватись відсортований масив.

3.1.2 Реалізація вимірювання часу дії алгоритмів сортування

Також було реалізовано клас SortAlgorithmsBenchMark, у якому якраз відбувається підрахунок часу виконання алгоритму. Для цього в ньому є метод main, який запускає цю перевірку. Для його налаштування ми вибрали використання OptionsBuilder, оскільки вся інформація зберігається в одному окремо визначеному місці, і нам не потрібна додаткова гнучкість, тому що сам тест буде лише один. Його ми налаштували наступним чином (рисунок 21).

```
public static void main(String[] args) throws RunnerException {
    Options options = new OptionsBuilder()
            .include(SortAlgorithmsBenchMark.class.getSimpleName())
            .forks(1)
            .shouldDoGC(true)

            .warmupTime(new TimeValue( time: 10, TimeUnit.SECONDS))
            .warmupIterations(5)
            .timeout(TimeValue.hours(1))
            .measurementTime(new TimeValue( time: 10, TimeUnit.SECONDS))
            .measurementIterations(5)

            .mode(Mode.AverageTime)
            .timeUnit(TimeUnit.MILLISECONDS)

            .resultFormat(ResultFormatType.CSV)
            .result("out/sort/t_2.csv")
            .output("out/sort/t_2.txt")
            .build();
    new Runner(options).run();
}
```

Рисунок 21 – Налаштування OptionsBuilder

Сам тест виглядає наступним чином (рисунок 22). Над ним можна помітити лише анотацію Benchmark. Інші анотації не добавляли, оскільки конфігурація вся була виконана в головному методі. В поточному методі викликається статичний метод сортування масиву, туди передається вибраний

алгоритм та сам масив. Результат сортування передається в Blackhole, щоб JIT-компілятор бачив що результат методу використовується, та не видаляв його.

```
@Benchmark
public void sortIntegerArray(Blackhole bh) {
    bh.consume(SortAlgorithms.sort(algorithm, intArray));
}
```

Рисунок 22 – Метод який викликатиме алгоритми сортування

Як було сказано раніше, використовуються різні алгоритми сортування. Ми це зробили за допомогою enum класу, в якому перелічили алгоритми. Для того щоб тест використовував різні алгоритми сортування, ми створили поле даного enum класу, в класі SortAlgorithmsBenchMark, і над ним поставили анотацію Param з назвами цих алгоритмів (рисунок 23).

```
@Param({"BUBBLE_SORT", "SELECTION_SORT", "INSERT_SORT",
        "QUICK_SORT", "MERGE_SORT", "HEAP_SORT",
        "COUNT_SORT",
})
public SortAlgorithmEnum algorithm;
```

Рисунок 23 – Поле, в яке будуть вставлятись назви алгоритмів

У результаті всі перелічені текстові значення будуть перетворені у відповідне значення SortAlgorithmEnum класу та вставлені у дане поле класу. Для того, щоб ця анотація спрацювала, ми над цим класом поставили анотацію State (рисунок 24). Вибрали Scope – Benchmark. Тобто екземпляр даного класу буде створюватись лише один для даного тесту.

```
@State(Scope.Benchmark)
public class SortAlgorithmsBenchMark {
```

Рисунок 24 – Тестовий клас, який помічений анотацією State

Для того, щоб можна було протестувати на масивах різної довжини, також була використана анотація Param, в яку ми передали довжини масивів у тестовому вигляді (рисунок 25). При цьому додати ще одну довжину масиву в тест є доволі простою операцією.

```java
@Param({"100",
        "500",
        "1000",
        "5000",
        "10000",
        "50000",
        "100000",
})
public int arraySize;
```

Рисунок 25 – Поле, в яке будуть вставлятись довжини масивів

Довжину масиву ми вже можемо підставляти, проте потрібно ще створювати масив. Для цього ми використали анотацію Setup (рисунок 26). Рівень вибрали Trial, тобто для кожного нового варіанту параметрів буде викликатись один раз цей метод, який буде створювати масив цифр потрібної для даного випадку довжини. При цьому цей метод не буде викликатись надто часто, щоб не перевантажувати GC присутністю зайвих об’єктів в пам’яті.

```java
@Setup(Level.Trial)
public void initArray() {
    intArray = new Integer[arraySize];
}
```

Рисунок 26 – Метод конфігурації даного класу, який створюватиме масив потрібної довжини

Сам масив ми вже створюємо, проте його слід заповнювати цифрами. При цьому, як ми пам’ятаємо з розділу 1.2.2, існує кращий, гірший та середній випадок. Звісно, для деяких алгоритмів (наприклад Quick sort) найгірший випадок – це не обов’язково відсортований в зворотному порядку масив, проте в більшості алгоритмів сортування, це саме такий випадок. Тому в нас є три варіанти створення масиву: заповнити в порядку зростання від 1 до arraySize, заповнити цими даними в випадковому порядку та заповнити ними в зворотному.

Для того, щоб заповнити, скористаємось також анотацією Setup, проте цей раз рівень виберемо Invocation. Ми вибрали його, тому що нам потрібно

заповнити даними масив перед кожним викликом методу в тому порядку, який нам потрібен. Очевидно, що метод сортування не повинен створювати нові об'єкти, якщо це ним не передбачено, тому що ця операція може випадково викликати Garbage Collector, і тому сортування може тривати довше часу. Саме тому реалізовані алгоритми сортування зазвичай сортують дані прямо у вхідному масиві. Тобто після одноразового виклику методу сортування, масив буде відсортований, і його перед новим тестом слід заново заповнити даними в тому порядку, який нам потрібен, оскільки в інакшому випадку тест був би некоректним. Для вибору способу заповнення також ми створили enum клас. В ньому буде 3 варіанти, а саме: відсортований, рандомний порядок та відсортований в зворотному порядку (рисунок 27).

```
@Param({"SORTED", "RANDOM", "INVERT"})
public InitialSorting initialSorting;
```

Рисунок 27 – Поле, в яке буде вставлятись початковий порядок елементів

І у методі, який буде заповнювати створений масив значеннями, напишемо наступний код (рисунок 28). У ньому за допомогою статичного методу setAll класу Arrays заповнюється масив. На вхід цей метод приймає масив, який слід заповнити, і лямбду, за допомогою якої він заповнить. Ця лямбда приймає на вхід порядковий номер клітинки, яку зараз заповнює, а повертає те значення, яким слід дану клітинку заповнити. Для відсортованого списку це буде той самий номер клітинки. Для відсортованого в зворотному порядку – розмір масиву мінус порядковий номер клітинки. Для рандомно ініціалізованого списку ми вирішили заповнити масив так як відсортований список та потім його перемішати. При цьому всі дані будуть в одному і тому ж діапазоні значень.

```java
@Setup(Level.Invocation)
public void sortArray() {
    switch (initialSorting) {
        case SORTED:
            Arrays.setAll(intArray, i -> i);
            break;
        case RANDOM:
            Arrays.setAll(intArray, i -> i);
            Collections.shuffle(Arrays.asList(intArray));
            break;
        case INVERT:
            Arrays.setAll(intArray, i -> arraySize - i);
            break;
    }
}
```

Рисунок 28 – Заповнення масиву в залежності від необхідної початкової відсортованості

3.2 Отримані результати тестування

Під час запуску головного методу ми отримали доволі велику таблицю. Дані в такому вигляді доволі складно аналізувати, тому ми вирішили зобразити їх на графіку і аналізувати вже його. Тут, та надалі, якщо не сказано інакше, по осі x буде розмір масиву, а по осі y – час виконання алгоритму сортування у мілісекундах, а різними кольорами зображено різні алгоритми сортування, або ж різні параметри вхідних масивів.

Оскільки в переважній більшості ситуацій буде середній випадок, то першим на графіку зобразимо всі алгоритми з середнім випадком, тобто вхідний масив приходить перемішаний (рисунок 29). Для зручності аналізу графік було представлено у логарифмічній шкалі.

Як можна помітити, алгоритми розділились на декілька груп. У першій групі знаходяться неефективні алгоритми сортування, тобто сортування бульбашкою, вибором та вставкою. В другу групу попали ефективні алгоритми сортування, тобто сортування злиттям, швидке сортування та сортування купою.

І до третьої групи належить сортування підрахунком. Проаналізуємо більш уважно кожну групу.

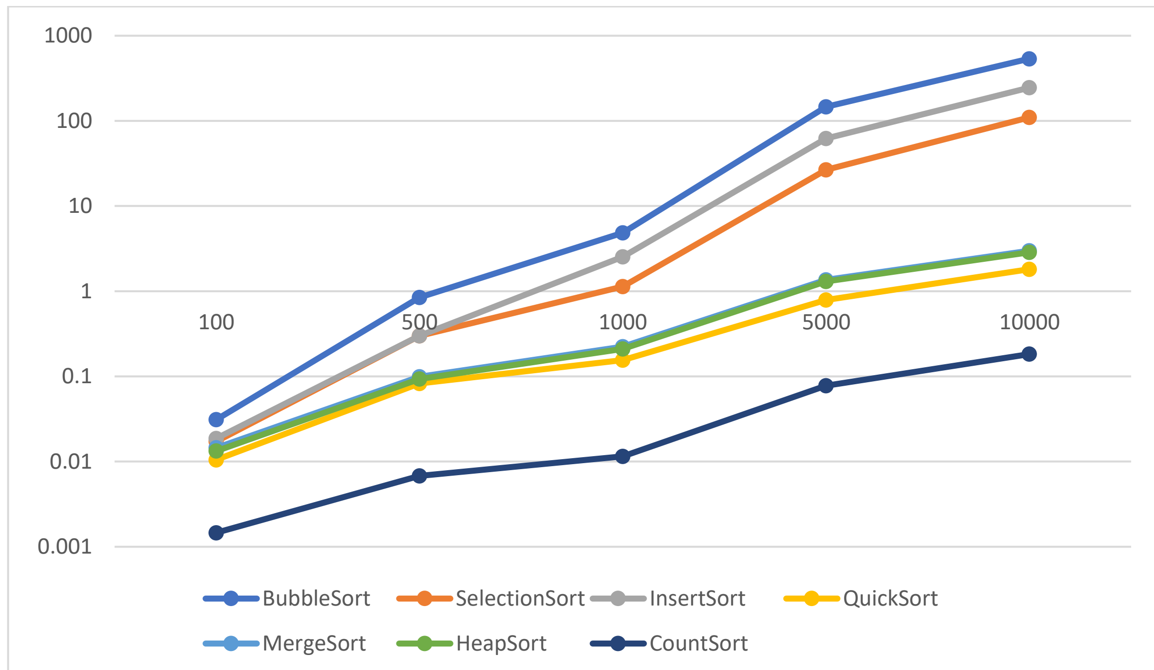


Рисунок 29 – Часова складність у мілісекундах реалізації алгоритмів для рандомізованого масиву

3.2.1 Неефективні алгоритми сортування

Якщо проаналізувати більш детально алгоритми, які попали в цю групу, то можна помітити, що у всіх них є два вкладені цикли. Проте час виконання цих алгоритмів доволі сильно відрізняється між собою. Зобразимо на двох графіках час виконання алгоритмів для кращого випадку та для середнього і гіршого випадків. Було обрано два графіки, щоб було краще аналізувати. При цьому перший графік буде в логарифмічній шкалі.

Як можна помітити, в кращому випадку, тобто коли вхідний масив відсортований, сортування бульбашкою та сортування вставкою займають майже однаково часу та на кілька порядків менше, ніж сортування вибором (рисунок 30). Це в принципі і логічно, оскільки у сортуванні бульбашкою, як і у сортуванні вставкою не відбудеться жодна операція заміни місцями елементів, і n-1 операцій порівняння. Проте у сортуванні вибором на кожній ітерації

відбудеться пошук найменшого елементу, і тому там відбудеться n*(n-1) операцій порівняння, і тому часова складність реалізації алгоритму буде рости. Як можна помітити, на масиві довжиною 100 час роботи алгоритму відрізняється приблизно на півтора порядки, проте на масиві 1000 елементів – уже на 2.5 порядки, а на масиві 1000 – на 3.5 порядків, що підтверджує квадратичну складність цього алгоритму у кращому випадку, в порівнянні з лінійною у інших двох.

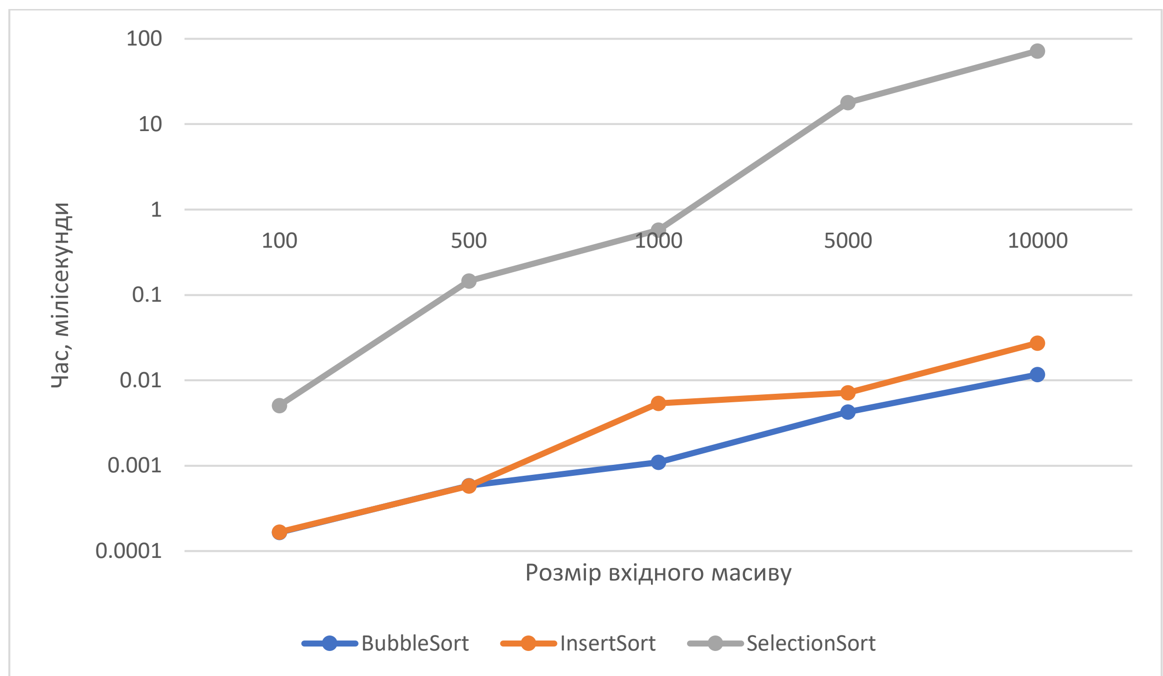


Рисунок 30 – Часова складність неефективних алгоритмів для відсортованого масиву

Графік для рандомізованого та інвертованого випадків зобразимо на рисунку 31. Зразу можна помітити, що часова складність реалізацій алгоритмів не відрізняється на порядки, з чого можна зробити висновок, що у них всіх один порядок складності, і саме тому для аналізу гарно підходить лінійна шкала.

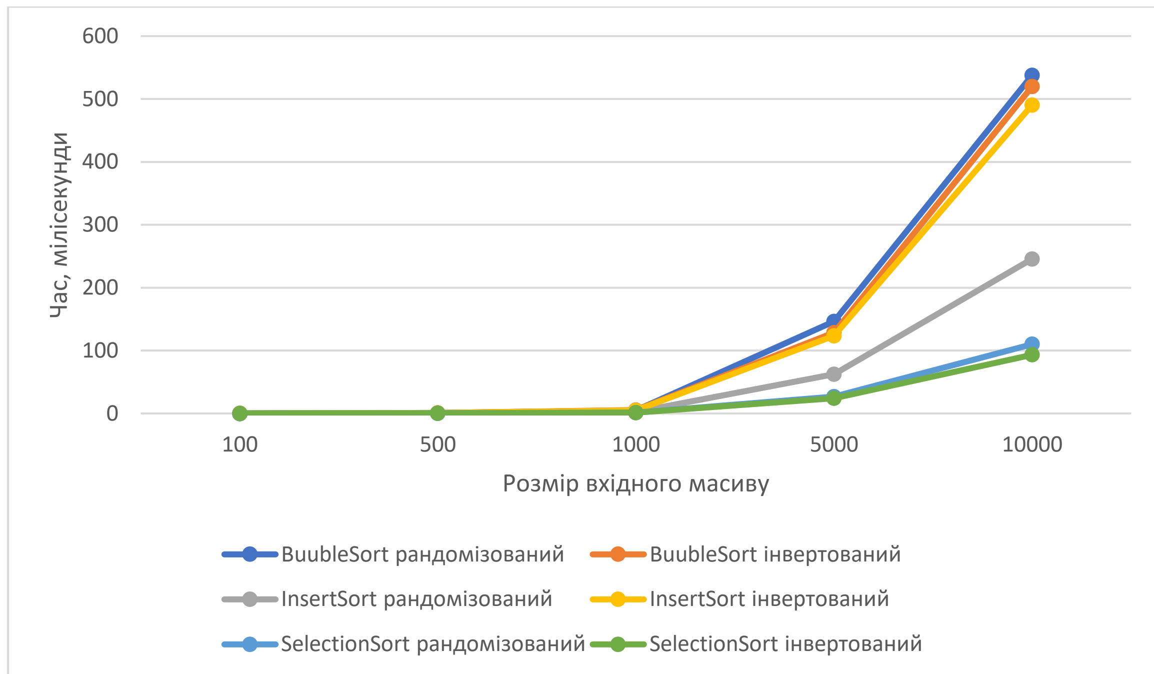


Рисунок 31 – Часова складність реалізацій неефективних алгоритмів для рандомізованого та інвертованого масиву

Як можна помітити, для середнього і найгіршого випадків, часова складність реалізації алгоритму сортування вибором є найменшою з цього переліку алгоритмів. Також вона не сильно відрізняється від найкращого випадку. Відмінність буде лише в тому, що тут виконується операція обміну місцями елементів, а в кращому випадку вона не виконувалась. Також цікавим є те, що в гіршому випадку час роботи алгоритму є меншим, ніж в середньому. Це пов’язано з тим, що обмін елементів, коли вони не є загружені в кеші процесора, є дорожчим, ніж коли вони є загружені. А в інвертованому масиві елемент, який є найменший на ітерації, буде вкінці, або ж ближче до кінця, а не всередині масиву, тому не потрібно вигружати іншу частину масиву в кеш процесору.

Сортування бульбашкою ж в гіршому і середньому випадку майже не відрізняється, оскільки там кількість ітерацій головного циклу буде майже однаковою, а вкладений цикл буде виконуватись таку саму кількість разів, не зважаючи чи мінятиме він місцями елементи.

У сортуванні вставкою у гіршому випадку час виконання вдвічі довше, ніж в середньому. Це пов’язано з тим, що в середньому у внутрішньому вкладеному

циклі буде i/2 операцій обміну елементів, де i – це довжина відсортованої частини масиву.

Отримана оцінка корелює із теоретичними оцінками, з чого можна зробити висновок про коректність даних вимірювань. Тому подальші графіки будуть показувати не час виконання у мілісекундах, а результат ділення отриманого часу виконання на теоретичну оціночну кількість операцій, тобто зміну часової вартості елементарної операції в залежності від довжини масиву. На неефективних масивах цю зміну зображати не будемо, оскільки ними все одно на практиці не користуються.

3.2.2 Швидкі алгоритми сортування

До цієї категорії алгоритмів увійшли ті, у яких порівняння відбувається не всіх елементів з усіма, а кількість елементів на кожній ітерації зменшується приблизно вдвічі. Оскільки теоретична оцінка цих алгоритмів відома, а саме O(N*ln(N)), зобразимо на графіку коефіцієнт C, за формулою $C(N) = \frac{t(N)}{N*\ln(N)}$, тобто час елементарної операції в наносекундах в залежності від розміру масиву.

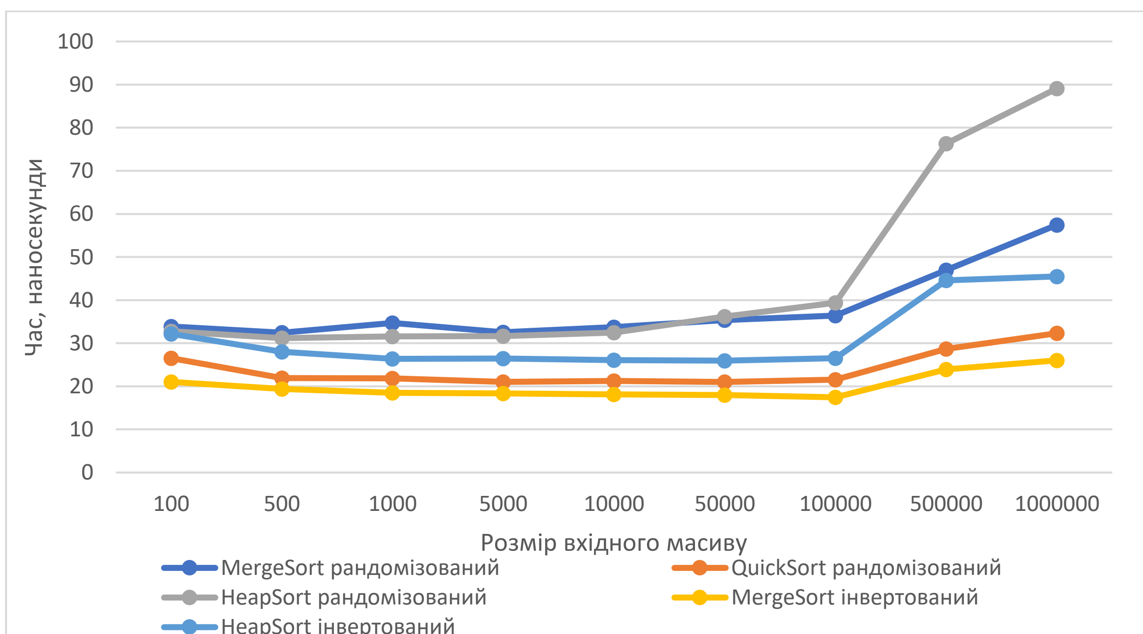


Рисунок 32 – Часова складність елементарної операції в залежності від розміру вхідного масиву

Оскільки швидке сортування при інвертованому масиві зростає значно швидше, і завершується помилкою при великих значеннях, то його зображати на графіку не будемо.

Як можна помітити, при великих вхідних масивах, коефіцієнт C починає зростати аж у кілька разів. Це пов’язано з тим, що чим більших вхідний масив, тим частіше потрібно загружати його нову частину у кеш процесору. Це дуже гарно видно на сортуванні кучею та сортуванні злиттям, при рандомізованому масиві, де побудова кучі буде здійснювати перестановки елементів з різних частин масиву, тобто різні частини потрібно буде загружати в кеш, або зливати два підмасиви у сортуванні злиттям, коли вони також не будуть знаходитись повністю у кеші процесору.

3.2.3 Сортування без порівнянь

Це найцікавіші алгоритми сортувань. Їхня суть в тому, що елементи не порівнюються між собою, а тому вони можуть виконуватись швидше. Але зазвичай ці алгоритми можуть сортувати лише цілочисельні значення. Цей клас алгоритмів представлений алгоритмом підрахунку. Теоретична складність даного алгоритму сортування O(N + M), де M – максимальна різниця вхідних значень. А оскільки згідно рисунку 28, M=N, то складність буде порядку O(N). Також, як і в попередньому випадку, зобразимо залежність коефіцієнту C від розміру масиву.

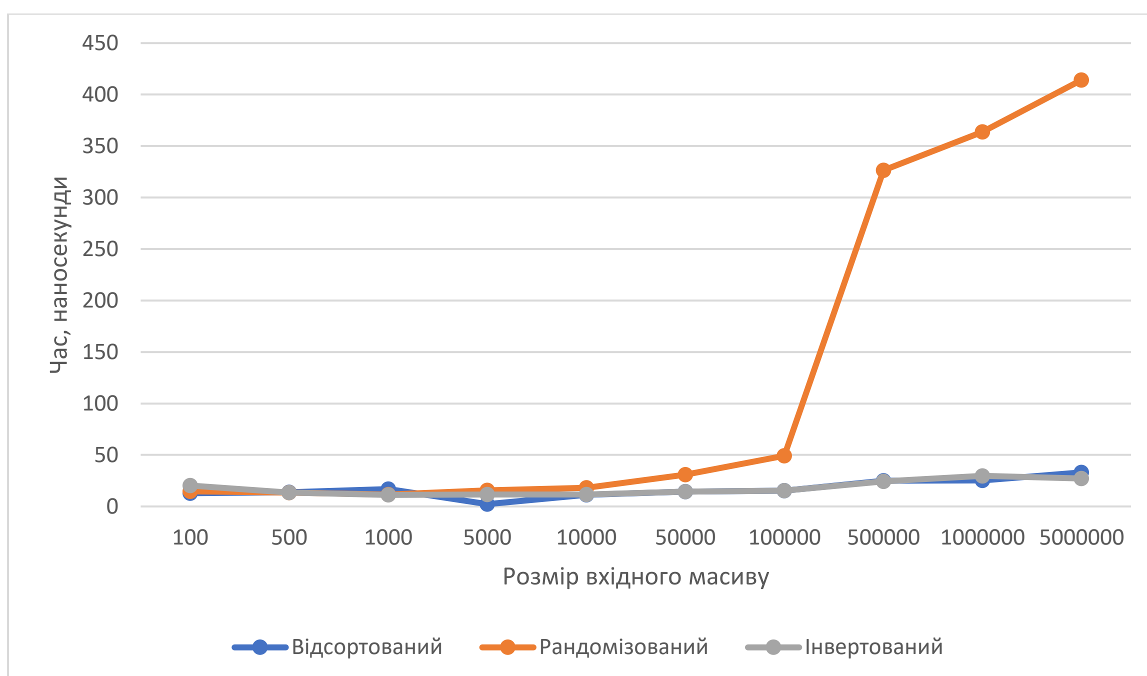


Рисунок 33 – Часова складність елементарної операції в залежності від розміру масиву

Як можна побачити на графіку, вартість елементарної операції при рандомізованому масиві зростає в рази, при великих вхідних масивах. І починає зростати, як і на 32 рисунку, після ста тисяч елементів. Це також пов’язано з кешем процесору, оскільки при рандомізованому вхідному масиві, потрібно збільшувати значення в проміжному масиві в далеких одна від одної клітинках, а отже операційній системі слід з оперативної пам’яті взяти іншу частину масиву в кеш процесору, а оскільки ця операція доволі вартісна, то тому в цьому випадку він буде довше тривати.

Ми вже неодноразово згадували про кеш процесору, і очевидно що на різних машинах він є різним. На моїй машині, кеш процесору 3 рівня має три мегабайти пам’яті, а оскільки зберігається масив цілочисельних значень, то можна порахувати, що він може вмістити одночасно не більше ніж 786432 цілочисельних значень. Але оскільки кеш зайнятий не лише цим масивом, тому і слід зменшити в кілька разів це число, а отже і стає логічно, чому час елементарної операції зростає при розмірі масиву більше ніж сто тисяч елементів.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було:

- проведено дослідження доцільності оптимізації алгоритмів;
- досліджено вплив параметрів на час виконання алгоритмів сортування;
- проведено вимірювання часу роботи підпрограми примітивним вимірюванням, аналіз проблем при цьому вимірюванні;
- структуровано вимоги до коректного вимірювання часу виконання підпрограми;
- виконано дослідження інструментів, які підходять під всі ці вимоги вимірювання часу;
- розроблено програму, яка коректно вимірює час роботи алгоритмів сортування в залежності від параметрів вхідних даних;
- проаналізовано результати роботи програми;
- досліджено вплив апаратних параметрів комп'ютера на час елементарної операції в підпрограмі.

При розробці програмного коду були проаналізовані всі найкращі практики для підрахунку часу виконання підпрограми. В якості інструментарію було обрано комерційне інтегроване середовище розробки (IDE) Intellij IDEA Ultimate Edition, яке є безкоштовним для студентів, мова програмування – Java, інструмент для визначення тривалості роботи підпрограми – Java Microbenchmark Harness.

На сьогоднішній день оптимізація алгоритмів є як ніколи важливою, оскільки зовсім скоро зростання швидкодії процесорів майже повністю зупиниться, і тоді для зростання швидкості роботи програми потрібно буде оптимізовувати її, а не купляти потужніші комплектуючі.

Аналіз складності алгоритмів може застосовуватись для демонстрації необхідності вивчати та використовувати оптимізованіші алгоритми під час опанування студентами курсу «алгоритми та складність». Аналіз складових JRE може використовуватись для глибшого розуміння мови Java під час опанування курсу «Об'єктно орієнтоване програмування». Вплив кешу процесора може використовуватись для більш наочного розуміння теми управління пам'яттю в рамках курсу «Системне програмування та операційні системи».

Для подальшого доопрацювання можливий аналіз більш складних алгоритмів, таких як алгоритми лінійної алгебри, або ж більш детальний аналіз впливу характеристик комп'ютера, таких як розмір кешу процесору, кількість регістрів, затримка пам'яті для різних підпрограм, наприклад тих же сортувань.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Томас Кормен. Алгоритмы: построение и анализ, 3-е изд.: Пер. с англ. СПб.: ООО " И. Д. Вильямс" 2013. — 1328 с.
2. Biography Al-Khwarizmi [Електронний ресурс] – Режим доступу: https://mathshistory.st-andrews.ac.uk/Biographies/Al-Khwarizmi/
3. Ervin Knuth. The Art of Computer Programming, 3-rd edition: Massachusetts: Addison-Wesley 1997. – 650 с.
4. J. Fuegi and J. Francis. Annals of the History of Computing, 4- edition: 2003.
5. Friedrich L. Bauer and H. Wössner. Algorithmic Language and Program Development: Springer Berlin Heidelberg 2011. – 500 с.
6. Конец эпохи закона Мура и как это может повлиять на будущее информационных технологий [Електронний ресурс] – Режим доступу: https://habr.com/ru/company/madrobots/blog/405413/
7. Оценка сложности алгоритмов [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/104219/
8. Java: Everything a Beginner Needs to Know [Електронний ресурс] – Режим доступу: https://www.coursereport.com/blog/what-is-java-programming-used-for
9. JIT in Java [Електронний ресурс] – Режим доступу: https://www.javatpoint.com/jit-in-java
10. Java Garbage Collection [Електронний ресурс] – Режим доступу: https://www.javatpoint.com/Garbage-Collection
11. Java Benchmarking as easy as two timestamps [Електронний ресурс] – Режим доступу: https://shipilev.net/talks/jvmls-July2014-benchmarking.pdf
12. Understanding JIT compiler (just-in-time compiler) [Електронний ресурс] – Режим доступу: https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/

13. Garbage Collection in Java – What is GC and How It Works in the JVM [Електронний ресурс] – Режим доступу: https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm/
14. Microbenchmarking with Java [Електронний ресурс] – Режим доступу: https://www.baeldung.com/java-microbenchmark-harness
15. Understanding Java Microbenchmark Harness or JMH Tool [Електронний ресурс] – Режим доступу: https://medium.com/javarevisited/understanding-java-microbenchmark-harness-or-jmh-tool-5b9b90ccbe8d
16. Sorting Algorithms [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/sorting-algorithms/

ДОДАТОК А

Результат виконання тестового методу

JMH version: 1.29
VM version: JDK 11.0.7, Java HotSpot(TM) 64-Bit Server VM, 11.0.7+8-LTS
VM invoker: C:\Program Files\Java\jdk-11.0.7\bin\java.exe
VM options: -javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\lib\idea_rt.jar=62097:C:\Program Files\JetBrains\IntelliJ IDEA 2019.3.3\bin -Dfile.encoding=UTF-8
Blackhole mode: full + dont-inline hint
Warmup: 5 iterations, 10 s each
Measurement: 5 iterations, 10 s each
Timeout: 10 min per iteration
Threads: 1 thread, will synchronize iterations
Benchmark mode: Throughput, ops/time
Benchmark: benchmark.BenchMark.init

```
# Run progress: 0.00% complete, ETA 00:08:20
# Fork: 1 of 5
# Warmup Iteration   1: 0.991 ops/s
# Warmup Iteration   2: 0.988 ops/s
# Warmup Iteration   3: 0.989 ops/s
# Warmup Iteration   4: 0.989 ops/s
# Warmup Iteration   5: 0.990 ops/s
Iteration   1: 0.990 ops/s
Iteration   2: 0.990 ops/s
Iteration   3: 0.989 ops/s
Iteration   4: 0.990 ops/s
Iteration   5: 0.990 ops/s

# Run progress: 20.00% complete, ETA 00:06:50
# Fork: 2 of 5
# Warmup Iteration   1: 0.990 ops/s
# Warmup Iteration   2: 0.992 ops/s
# Warmup Iteration   3: 0.991 ops/s
# Warmup Iteration   4: 0.991 ops/s
# Warmup Iteration   5: 0.992 ops/s
Iteration   1: 0.989 ops/s
Iteration   2: 0.989 ops/s
Iteration   3: 0.989 ops/s
Iteration   4: 0.990 ops/s
Iteration   5: 0.990 ops/s

# Run progress: 40.00% complete, ETA 00:05:06
# Fork: 3 of 5
# Warmup Iteration   1: 0.991 ops/s
# Warmup Iteration   2: 0.991 ops/s
# Warmup Iteration   3: 0.991 ops/s
```

```
# Warmup Iteration   4: 0.990 ops/s
# Warmup Iteration   5: 0.990 ops/s
Iteration   1: 0.991 ops/s
Iteration   2: 0.991 ops/s
Iteration   3: 0.990 ops/s
Iteration   4: 0.990 ops/s
Iteration   5: 0.989 ops/s

# Run progress: 60.00% complete, ETA 00:03:24
# Fork: 4 of 5
# Warmup Iteration   1: 0.991 ops/s
# Warmup Iteration   2: 0.991 ops/s
# Warmup Iteration   3: 0.989 ops/s
# Warmup Iteration   4: 0.991 ops/s
# Warmup Iteration   5: 0.988 ops/s
Iteration   1: 0.990 ops/s
Iteration   2: 0.988 ops/s
Iteration   3: 0.989 ops/s
Iteration   4: 0.989 ops/s
Iteration   5: 0.989 ops/s

# Run progress: 80.00% complete, ETA 00:01:42
# Fork: 5 of 5
# Warmup Iteration   1: 0.990 ops/s
# Warmup Iteration   2: 0.991 ops/s
# Warmup Iteration   3: 0.988 ops/s
# Warmup Iteration   4: 0.989 ops/s
# Warmup Iteration   5: 0.991 ops/s
Iteration   1: 0.990 ops/s
Iteration   2: 0.991 ops/s
Iteration   3: 0.991 ops/s
Iteration   4: 0.991 ops/s
Iteration   5: 0.990 ops/s


Result "benchmark.BenchMark.init":
  0.990 ±(99.9%) 0.001 ops/s [Average]
  (min, avg, max) = (0.988, 0.990, 0.991), stdev = 0.001
  CI (99.9%): [0.989, 0.990] (assumes normal distribution)


# Run complete. Total time: 00:08:30

REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on
why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial
experiments, perform baseline and negative tests that provide experimental control, make sure
the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.

Benchmark       Mode  Cnt  Score   Error  Units
BenchMark.init  thrpt   25  0.990 ± 0.001  ops/s
```