**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Факультет комп’ютерних наук та кібернетики

Кафедра математичної інформатики

Звіт

Із дисципліни «Хмарні обчислення»

Лабораторна робота на тему

**ПЕРЕВІРКА НАМЛЕЖНОСТІ ПРЯМОЇ ПЛОЩИНІ З ВИКОРИСТАННЯМ СИСТЕМИ ПАРКС ТА GOOGLE CLOUD PLATFORM**

Виконав студент 4-го курсу

Шевченко Максим Олексійович

КИЇВ 2020

**Постановка задачі**

**Теоретичні відомості**

Бібліотечне сортування, або сортування вставкою з пробілами, являє собою алгоритм сортування , який використовує сортування вставок, але з пробілами в масиві для прискорення наступних вставок. Назва походить від аналогії:

Припустимо, бібліотекар повинен зберігати свої книги в алфавітному порядку на довгій полиці, починаючи з лівого кінця, і продовжуючи праворуч вздовж полиці, без пробілів між книгами, до кінця Z. Якщо бібліотекар придбав нову книгу, яка належить до розділу B, як тільки він знайде правильний простір у розділі B, йому доведеться перенести кожну книгу, починаючи з середини B і закінчуючи Z, щоб звільнити місце для нової книги. Це аналогія сортування вставками. Однак, якби він залишав простір після кожної літери, доки залишався простір після В, йому довелося б перенести лише кілька книг, щоб звільнити місце для нової. Це основний принцип бібліотечного сортування.

Алгоритм був запропонований Майклом А. Бендером, Мартіном Фарах-Колтоном та Мігелем Мостейро в 2004 р. та опублікований у 2006 р.

Подібно до сортування вставкою, на якому воно базується, бібліотечне сортування це сортування з порівнянням; проте було показано, що воно має велику ймовірність запуску за час O(n log n) (порівнянний із швидким сортуванням), проти O(n2) сортування вставками . У статті немає повної реалізації, а також точних алгоритмів важливих частин, таких як вставка та перебалансування. Потрібна буде додаткова інформація, щоб обговорити, як ефективність бібліотечного сортування порівнюється з ефективністю інших методів сортування в реальності.

Порівняно з базовою вставкою, недоліком бібліотечного сортування є те, що воно вимагає додаткового місця для пробілів. Кількість та розподіл цього простору залежить від реалізації. У статті розмір необхідного масиву становить (1 + ε)n, але без подальших рекомендацій щодо вибору ε. Для того, щоб гарантувати обмеження часу виконання з високою ймовірністю, йому потрібно випадково переставити вхідні дані, що змінює відносний порядок рівних елементів і перетасовує будь-який попередній вхід. Якщо його використовувати без такої перетасовки, він може легко отримати квадратичну складність. Крім того, алгоритм використовує двійковий пошук, щоб знайти точку вставки для кожного елемента.

**Алгоритм, його реалізація та підбір параметрів**

**Алгоритм** можна описати наступним чином:

1. Створюємо порожній допоміжний масив, в кілька разів більший за основний.

2. Для чергового елемента шукаємо місце вставки в допоміжному масиві.

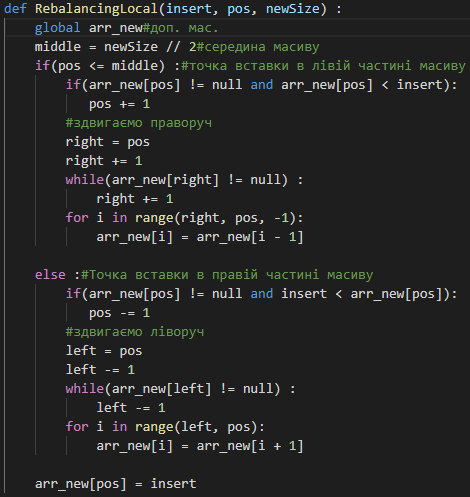
2.1. Якщо знайшли місце для вставки, то переносимо елемент і повертаємося до пункту 2.

2.2. Якщо місця для вставки не знайшлося, виконуємо перебалансування допоміжного масиву і повертаємося до пункту 2.

3. Після обробки всіх елементів переносимо їх назад в основний масив.

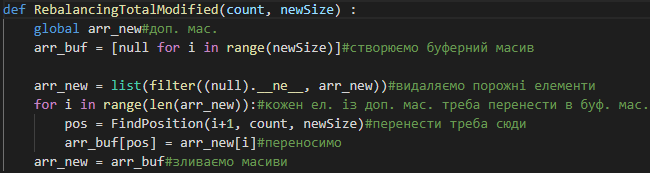
Розглянемо спочатку **пункт 2.2** та варіанти перебалансування: локальне та повне.

При локальному перебалансуванні необхідно змістити декілька елементів ліворуч або праворуч, щоб звільнити місце для вставки чергового елемента. Щоб заповненість допоміжного масиву була більш-менш рівномірна, то виконуватимемо здвиг праворуч, якщо елемент знаходиться в лівій частині масиву, інакше ліворуч. Реалізація:



Тут pos та insert – місце та значення для вставки відповідно, newSize – тут і надалі розмір допоміжного масиву.

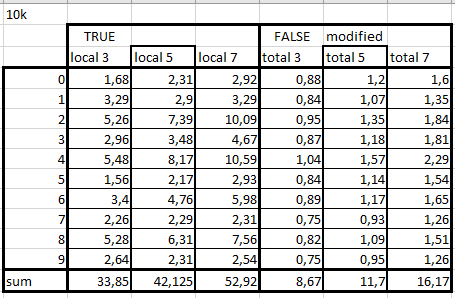
Повне перебалансування являє собою рівномірний перерозподіл елементів допоміжного масиву. Так як розмір допоміжного масиву більший за основний за означенням, то внаслідок такого перебалансування між елементами допоміжного масиву гарантовано з’являться вільні місця та ми зможемо вставити новий елемент. Якщо елементи переставляти саме в допоміжному масиві (як при локальному перебалансуванні), то при здвигу елемента на нове місце ліворуч може виявитися, що воно зайняте іншим елементом, який також треба перенести ліворуч на нове місце. При перенесенні в праву сторону таких проблем не буде (якщо починати переносити елементи від крайнього правого, як у реалізації). Через таку ситуацію перенесення елементів ліворуч треба робити рекурсивно. Як показала практика, таких рекурсивних викликів буде забагато вже при сортуванні більше 3000 рядків та програма видасть помилку про переповнення стеку. Тому було вирішено створити додатковий буферний допоміжний масив, в який перенесемо на відповідні нові місця елементи з допоміжного, а потім присвоїмо значення буферного масиву допоміжному. Реалізація:

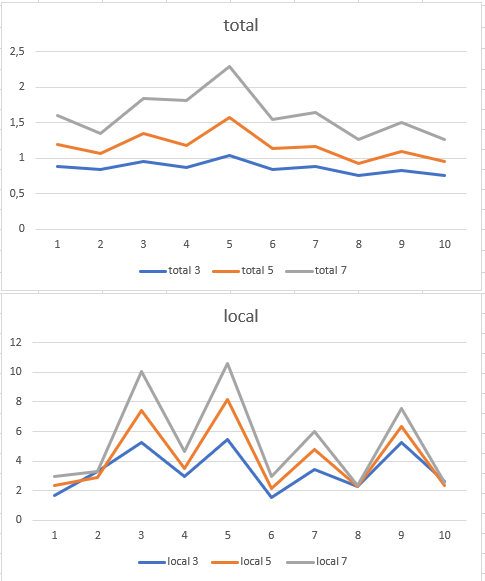


Тут count – теперішня кількість елементів у допоміжному масиві. Її необхідно знати, щоб рівномірно перерозподілити елементи при перебалансуванні. Нова позиція pos кожного елементу залежить від теперішньої кількості елементів у масиві та знаходиться за формулою , де number – який це по рахунку непорожній елемент:



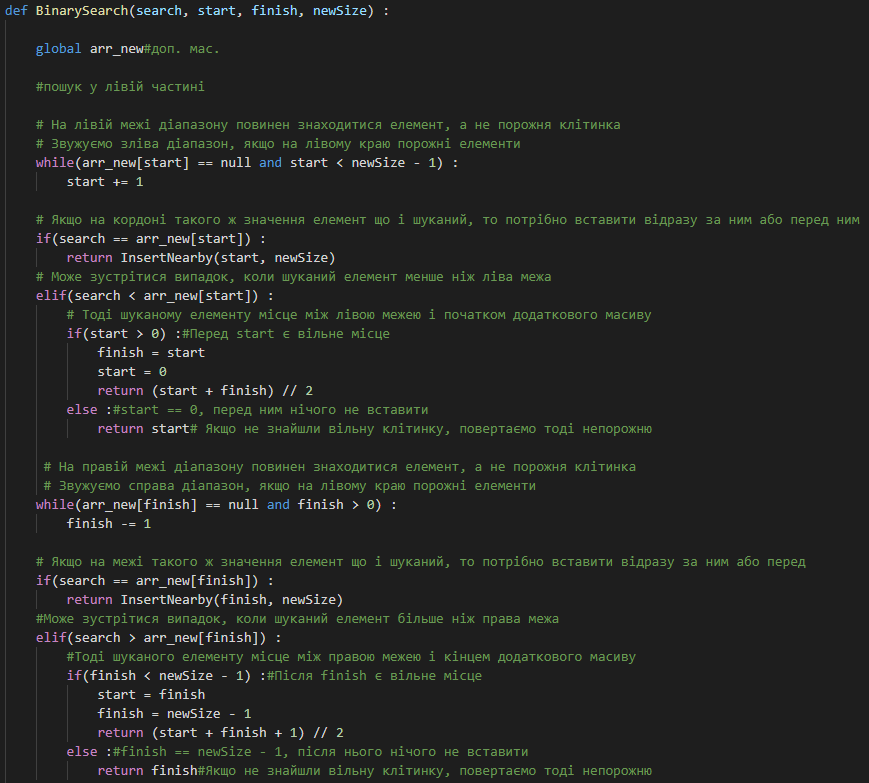
Наразі експериментально визначимо оптимальний тип перебалансування та розмір додаткового масиву. Провівши серію тестів (час сортування десяти згенерованих файлів по 10000 елементів у кожному для допоміжного масиву в 3, 5 та 7 разів більшого за основний), отримав наступні результати:

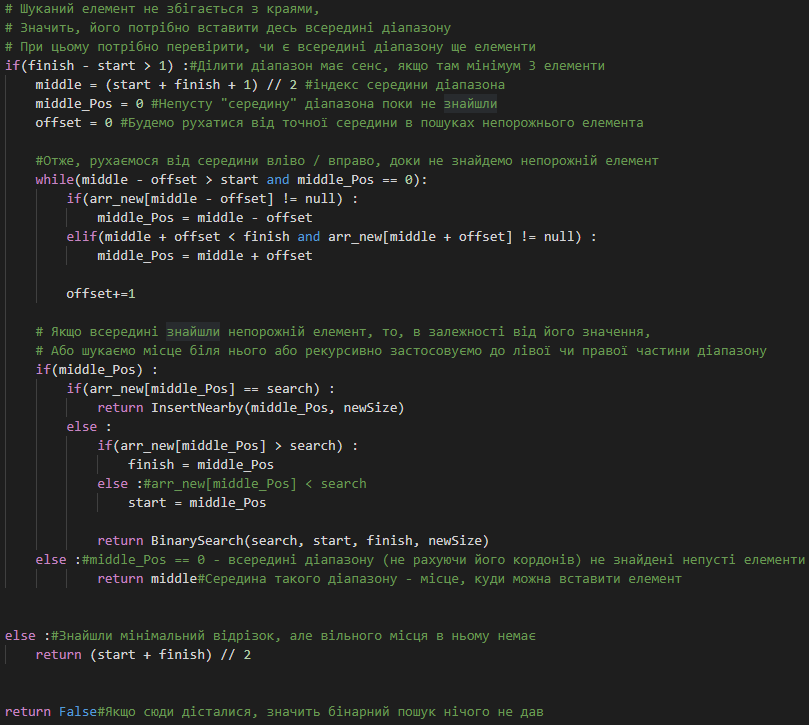


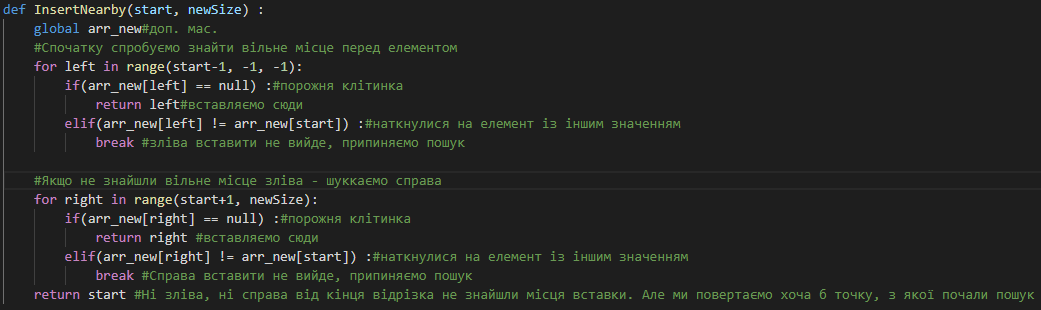


Як можна побачити, використання повного перебалансування значно економить час. Крім того, використання більшого допоміжного масиву не надає особливої переваги та навпаки збільшує час виконання через збільшення кількості ітерацій проходження по масиву (див. решту пунктів реалізації). Ще одним цікавим результатом є те, що алгоритм чутливий до вхідних даних, адже ми спостерігаємо пропорційно різний час виконання сортування тестових файлів.

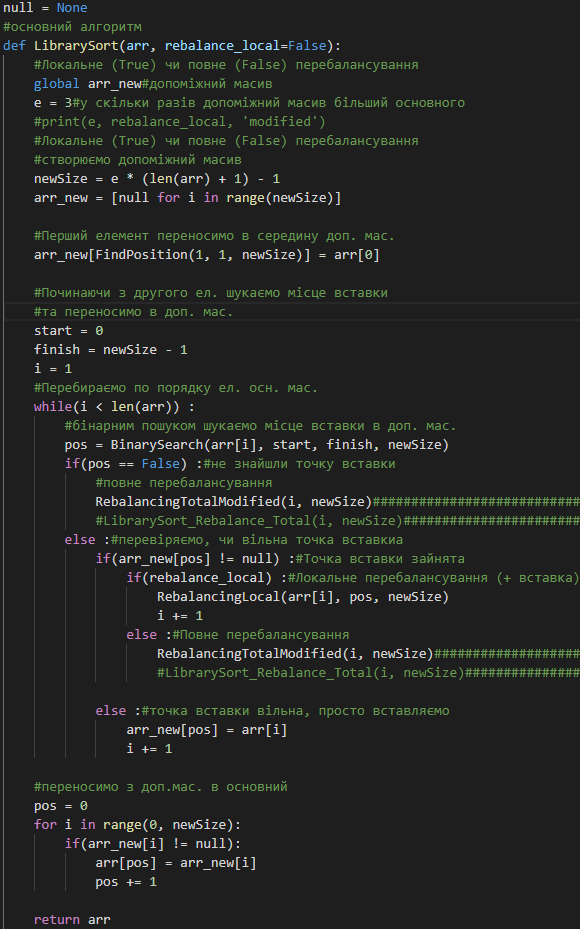
Розглянемо тепер алгоритм пошуку місця вставки елемента (**пункт 2**). Бінарний пошук являє собою пошук двох сусідніх елементів, між якими необхідно вставити черговий елемент. Додатково будуть оброблятися випадки, коли елемент, який вставляємо, менший за наявний або більший за найбільший у допоміжному масиві елемент (тоді його необхідно вставити відповідно зліва або справа), а також коли елемент рівний одному з уже наявних у масиві (тоді його необхідно поставити поруч із ним). Реалізація:



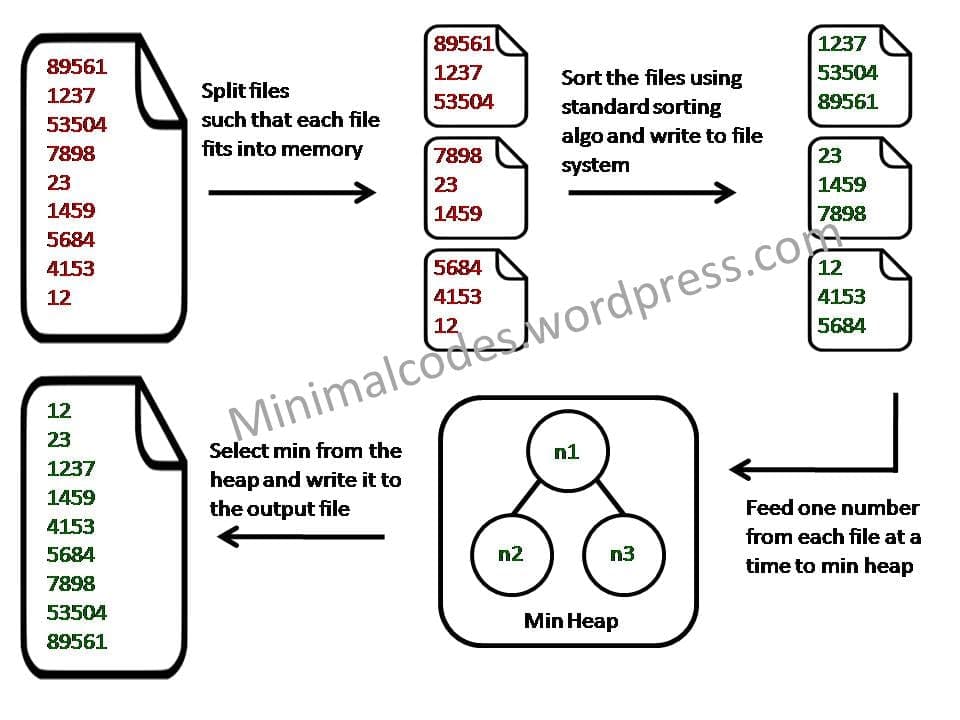




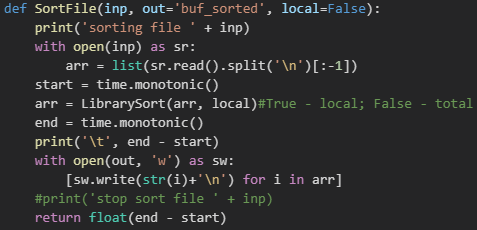
Тепер розглянемо **реалізацію алгоритму сортування**:



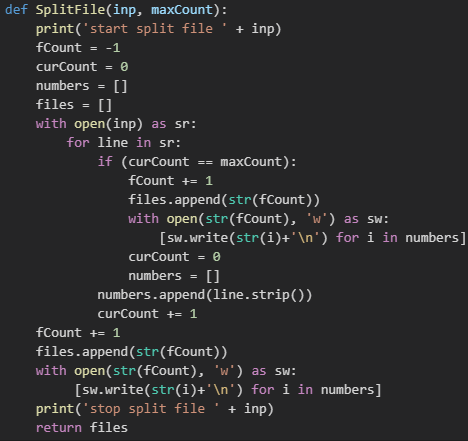
Перейдемо до **сортування даних, які за розміром переважають наявну оперативну пам’ять**. Так як за умовою лабораторної дані зберігаються у файлі, то очевидною є необхідність розбити вхідний файл на менші за розміром файли, відсортувати кожен із них та потім об’єднати результати в один файл. Алгоритм зображено на картинці:



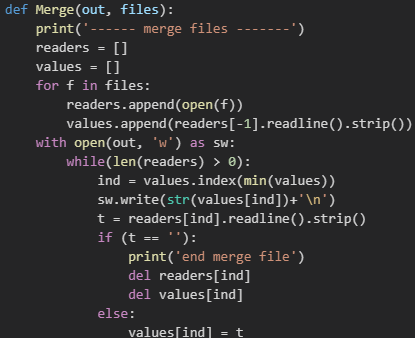
Сортування файлу виконується наступним чином:



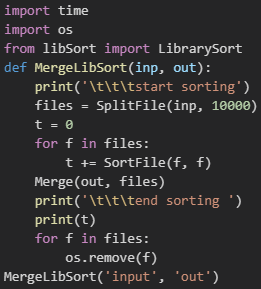
Розбиття файлу на менші за розміром, у кожному з яких кількість елементів не переважає maxCount:



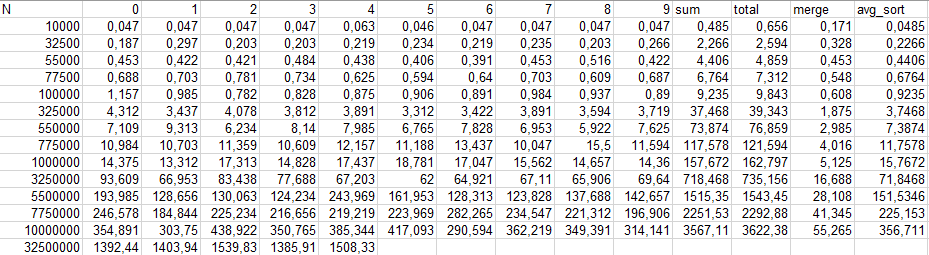
Об’єднання відсортованих файлів в один:



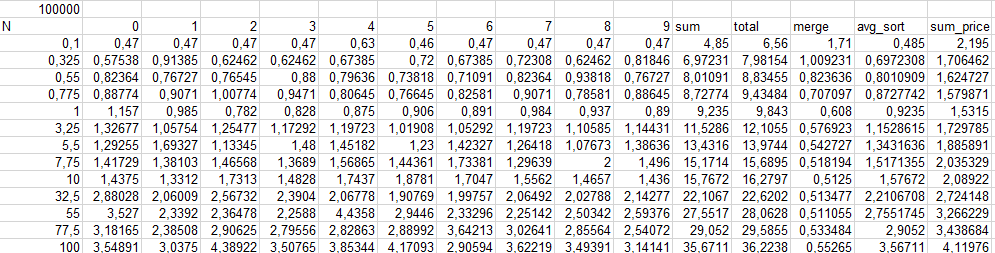
Ну й власне реалізація процедури сортування завеликих файлів:



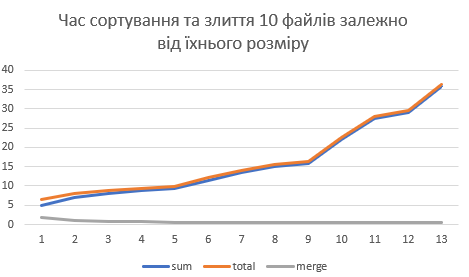
Підбір параметру maxCount відбувався експериментально. Спочатку я провів серію тестів для знаходження часу сортування в залежності від кількості елементів:

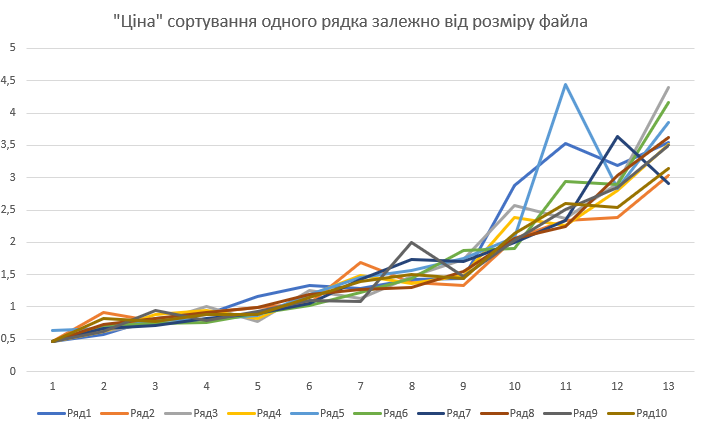


Тут у колонках 0-9 записаний час сортування N/10 елементів, avg\_sort – його середнє значення (бо час залежить від вхідних даних), total – час сортування та злиття N елементів, розділених на 10 файлів, sum – час саме сортування файлів, merge – час злиття файлів. Ці дані складно піддаються аналізу, адже очевидно, що при збільшенні кількості елементів час сортування збільшуватиметься. Тому я ввести поняття ціни сортування – скільки часу витрачається на сортування одного елемента, Price = Time/N:

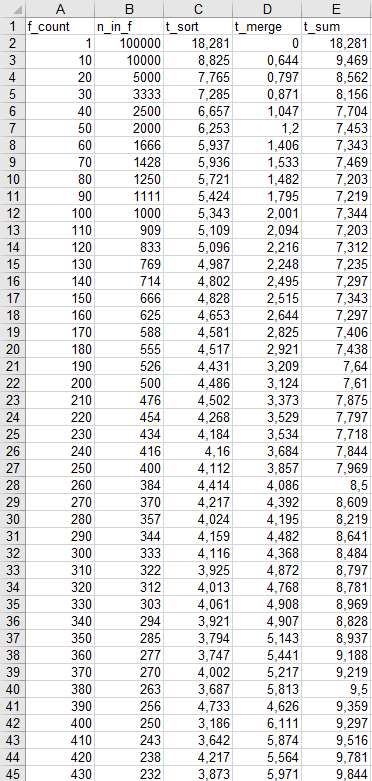


(тут N’= N/10000 для наглядності).



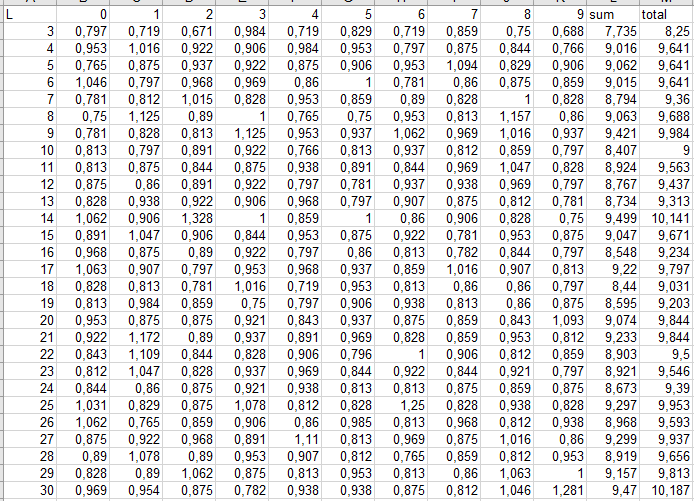


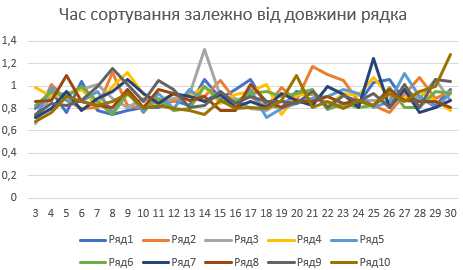
Як бачимо, злиття файлів відбувається набагато швидше, ніж сортування. Користь розбиття на декілька файлів можна побачити також у цій таблиці, де я порівнюю сумарну швидкість сортування 100000 елементів при розбитті на різну кількість файлів:

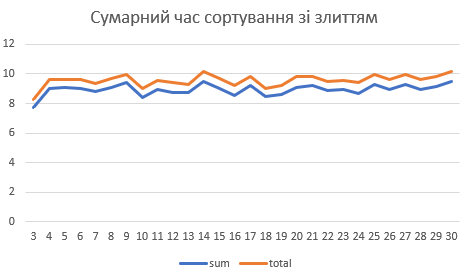


Тут також помітно, що є певне середнє відношення кількості елементів у кожному файлі, за якого сумарний час сортування є найменшим. Повертаючись до таблиці з цінами сортування, бачимо, що найменша ціна сортування та злиття (sum\_price) відповідає 10000 елементам у кожному файлі, тому це число й буде використовуватися як maxCount.

Розглянемо вплив довжини рядка на час сортування:







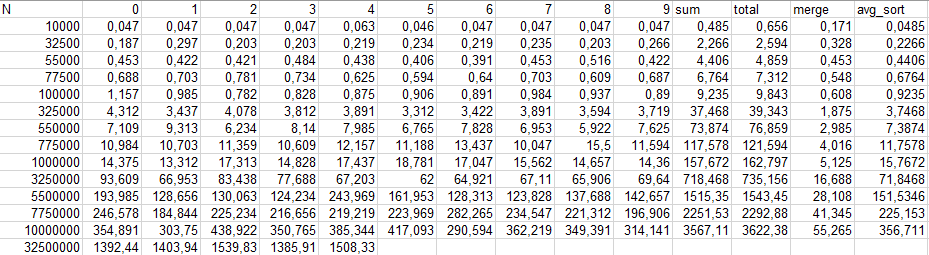
Як бачимо, довжина рядка слабко впливає на час сортування, тому в тестах використовуватимемо його довжину в 10 символів. Зрозуміло, що кількість зайнятої пам’яті пропорційна довжині рядка, адже необхідно зберігати більшу кількість символів при виконанні програми.

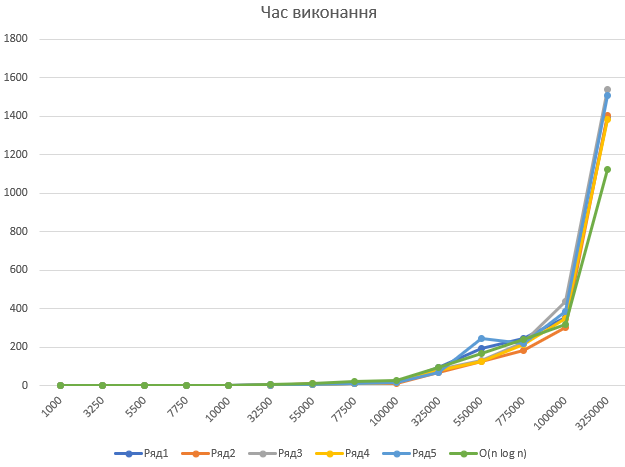
**Теоретична складність алгоритму**

Так як ми сортуємо n чисел, виконуючи пошук місця вставки для кожного бінарним пошуком, то теоретична складність алгоритму рівна O(n log n) із використанням при цьому O(n) пам’яті (адже допоміжний та буферний масиви в е раз більші за n). Проте виконання перебалансування потребу додаткового часу, що призводить до складності в найгіршому випадку O(n2), при цьому не потребуючи додаткових затрат пам’яті.

**Практична складність алгоритму**

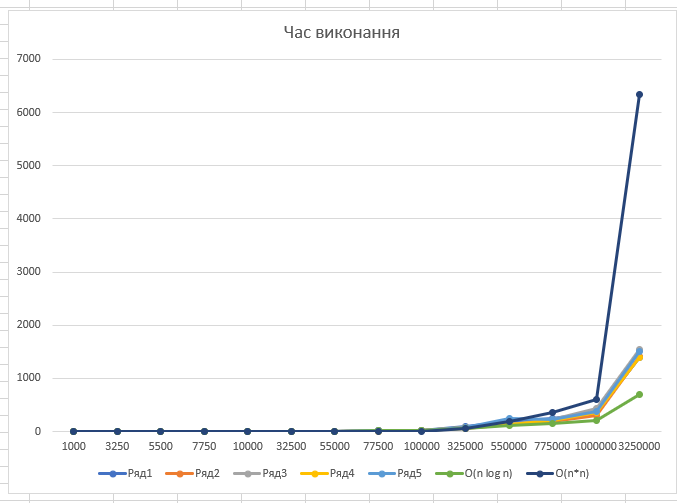
На основі даних, отриманих при експериментальному підборі параметрів, побудуємо графіки часу виконання сортування (використаємо колонки 0-4) та нанесемо на них функцію k\*n\*log n:





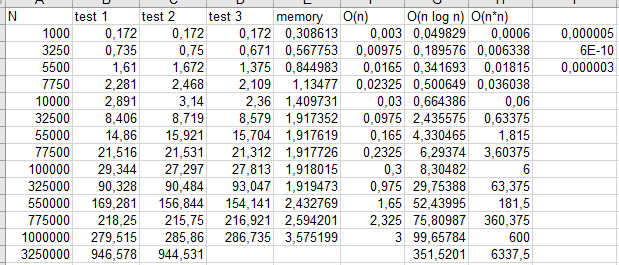
Як бачимо, при k = 0,000016 отримали досить гарне співпадіння графіків, що доводить теоретично обраховану складність виконання. Очевидно, що витрати пам’яті при цьому залежать лінійно від n, адже ми використовуємо масиви довжинами n, е\* n та за потреби(перебалансування) ще е\*n, тобто сумарні затрати пам’яті не перевищать n\*(2е+1)\*L+C, де L – довжина рядка у файлі, C – константна кількість пам’яті, необхідна ітераторам циклів та на інші змінні, кількість яких стала. При цьому е = 3, про що говорилося раніше.

Якщо ж додати до графіку ще й O(n2) та погратися зі значеннями k, то отримаємо такі графіки:



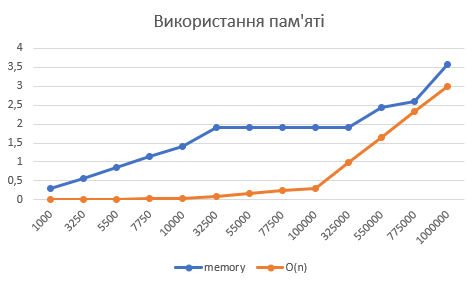
Варто зазначити, що налягання графіків розрахованих оцінок та отриманих тестових пояснюється не ідеальним вибором констант. Проте, як бачимо, теоретичні оцінки підтверджено ще раз.

При сортуванні зі злиттям отримали такі результати:



На графіку:





Навіть при такому сортуванні отримаємо підтвердження теоретичних оцінок. Використання пам’яті зростає не швидко, адже воно більше залежить від кількості утворених нових файлів та необхідності одночасного зчитування з них при записі в один спільний файл (алгоритм був описаний вище), ніж від розміру масиву, який є константним.

**Висновок**: усі завдання лабораторної було виконано, теоретичні оцінки розраховано й підтверджено. Був виконаний експериментальний підбір оптимальних параметрів та проаналізовано отримані дані. Програма підтримує можливість сортування вхідних даних довільного розміру, хоча й доведеться трохи почекати виконання сортування при великій кількості елементів.