DOI: https://doi.org/10.26642/ten-2026-1(97)-217-225
УДК 004.021

А.О. Іванов, здобувач
О.М. Кривонос, к.пед.н., доц.
Житомирський державний університет імені Івана Франка

Порівняльний аналіз часової та просторової складності алгоритмів сортування в сучасних програмних середовищах

Стаття присвячена комплексному дослідженню та порівняльному аналізу найбільш розповсюджених алгоритмів сортування, таких як бульбашкове, вставками, вибором, злиттям, швидке, купами та коміркове сортування. Актуальність роботи зумовлена стрімким розвитком технологій Big Data та Інтернету речей (IoT), що вимагає високої ефективності обробки даних в умовах як необмежених обчислювальних потужностей, так і екстремально обмежених ресурсів пам'яті. Автори детально розглядають теоретичні аспекти кожного алгоритму, проводячи математичне моделювання їхньої часової та просторової складності за допомогою асимптотичного аналізу та методу Акра-Баззі для рекурентних структур.

Методологія дослідження поєднує теоретичний аналіз асимптотик із емпіричним тестуванням продуктивності в середовищі Python 3.14 на сучасній архітектурі процесору Ryzen 7 7445HS. У роботі досліджуються не лише показники швидкодії на випадкових масивах, а й властивість адаптивності алгоритмів до вже впорядкованих або частково впорядкованих даних. Результати експериментів наочно демонструють, як вибір опорного елемента у швидкому сортуванні або специфіка менеджменту стеку викликів впливають на реальний час виконання та обсяг використаної пам'яті.

На основі отриманих даних сформовано практичні рекомендації щодо застосування конкретних методів сортування залежно від апаратних обмежень та характеру вхідних даних. Встановлено, що коміркове сортування є найбільш ефективним для великих масивів за наявності достатнього обсягу оперативної пам'яті, тоді як сортування купами (Heap Sort) виявляється оптимальним за просторовою складністю серед алгоритмів класу O(n \log n). Для специфічних умов впорядкованих даних найкращі результати за швидкістю показало сортування вставками. Стаття має наукову та практичну цінність для розробників програмного забезпечення, які прагнуть оптимізувати процеси індексації та аналізу великих обсягів інформації.

Ключові слова: *алгоритми сортування; часова складність; просторова складність; асимптотичний аналіз; Big Data; адаптивність алгоритмів; Python; метод Акра-Баззі; Big O нотація; порівняльне тестування.*

Актуальність дослідження. Сучасна епоха Big data представила безпрецедентні вимоги до ефективності обробки даних, де сортування є однією з фундаментальних етапів аналізу та індексації. В сьогоденні обсяги даних зростають майже експоненційно, і різниця обчислювальних складностей $O(n \log n)$ і $O(n^2)$ може складати години обчислювального часу на серверних кластерах. Розуміння того, як використання пам'яті і її менеджмент може впливати на швидкість виконання алгоритму, може наблизити нас до розуміння, як можна оптимізувати вже існуючі алгоритми, або навіть до створення концептуально нового підходу до сортування.

Проте це не значить, що ми завжди можемо покладатись на збільшення використання пам'яті задля збільшення ефективності. З постійним розвитком мобільних технологій, в особливості технологій мікроконтролерів як основи «Інтернету речей», виникає необхідність стійкості алгоритмів (і їхньої ефективності) в умовах екстремальних обмежень ресурсів. Поки коміркове сортування та Radix-sort здатні швидко сортувати масиви з використанням відносно великої кількості додаткової пам'яті, сортування купами та швидке сортування можуть відносно швидко сортувати дані без використання додаткової пам'яті взагалі.

Аналіз останніх досліджень та публікацій. Проведено багато досліджень у порівнянні алгоритмів сортування і їх властивостей між собою за такими параметрами, як часова, просторова складність, адаптивність і стабільність (Mohammed, Zhanar, Alaa [5]), (Ashok, Survery [2]). Проте, не так багато з них досліджують фактори, які впливають на ці параметри, виділяють класифікацію і проводять математичне моделювання цих алгоритмів у спробі на більш глибокому рівні зрозуміти, звідки беруться їхні переваги і недоліки.

Сучасні дослідження у сфері алгоритмізації зосереджені на глибокому аналізі продуктивності сортувальних структур, де ключовим аспектом є зіставлення теоретичної асимптотичної складності з реальними показниками часу виконання на різних обчислювальних архітектурах. Зокрема, у працях Pizarro-Vasquez та інші [10] та Santos та інші [3] акцентується увага на емпіричній оцінці швидкодії

© А.О. Іванов, О.М. Кривонос, 2026

алгоритмів, що дозволяє виявити невідповідності між «чистою» нотацією $O(n\log n)$ та практичними витратами пам'яті чи процесорного часу залежно від обсягу вхідних даних. Дослідження Goel та інші [7] доповнює цю картину аналізом методів оптимізації, спрямованих на скорочення часу обробки в специфічних умовах embedded-систем та інтелектуальних мереж. Водночас робота Sakpal та інші [6] розширює контекст порівняння, розглядаючи взаємозв'язок між алгоритмами сортування та пошуку, що є критично важливим для побудови високонавантажених інформаційних систем. Спільним висновком усіх авторів є те, що вибір оптимального методу сортування – від класичних Bubble Sort і Selection Sort до більш досконалих Quick Sort та Merge Sort – вимагає врахування не лише розміру масиву, а й ступеня його попередньої впорядкованості та специфіки апаратної реалізації.

Метою дослідження є комплексний аналіз та порівняння часової та просторової складності класичних алгоритмів (бульбашкове, вставками, вибором) та сучасних методів сортування, таких як злиттям, швидке, купами та коміркове сортування. Робота спрямована на виявлення закономірностей між принципами функціонування цих алгоритмів та їхньою продуктивністю в умовах сучасних обчислювальних архітектур та великих даних.

Методологія дослідження базується на вивченні і аналізі джерел, теоретичному аналізі асимптотичної складності алгоритмів, і емпіричному тестуванні їхньої продуктивності. У ході роботи застосовано метод порівняльного моделювання та класифікація алгоритмів за принципами роботи, також проведено серію контрольованих експериментів у середовищі Python 3.14 на архітектурі процесору Ryzen 7 7445HS. Експериментальний метод передбачав генерацію масивів різного об'єму та ступеня впорядкованості, вимірювання чистого часу виконання, статистичну обробку результатів та їх візуалізацію.

Виклад основного матеріалу.

Бульбашкове сортування. Найпростіший алгоритм сортування, а також прямолінійний, адже його суть полягає у тому, щоб знаходити найбільший/найменший елемент, і «піднімати» його у кінець набору даних. Сортування відбувається шляхом багатократної заміни елементів місцями (операція swap).

Даний алгоритм сортування є адаптивним, тобто, якщо масив вже відсортований, то він лише один раз пройдеться по ньому для порівняння сусідніх елементів, тому нижня асимптотика алгоритму $\Omega(n)$. Інакше, кожен елемент «підніматиметься» в кінець масиву окремо. Якщо, умовно, на переміщення першого елементу буде виконано $n-1$ порівнянь, то на переміщення другого вже піде $n-2$ порівнянь, тобто очікувана асимптотика розраховуватиметься наступним чином:

$$T(n) = \Theta\big((n-1) + (n-2) + \cdots + 1\big) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2), \quad (1)$$

що співпадає з найгіршою асимптотикою, коли масив впорядкований в зворотному порядку, тому $T(n) = O(n^2)$. Також в найгіршому випадку (відсортований з зворотну сторону масив) для «піднімання» елементів виконуватиметься така ж кількість операцій swap $\sim n^2$, що не вплине на загальну асимптотику.

Сортування бульбашкою є алгоритмом з виконанням на місці (in-place), тобто сортування відбувається безпосередньо на вихідному масиві, і використання пам'яті в цьому алгоритмі $Ospace(1)$.

Даний алгоритм є одним з найбільш неефективних, проте він користується популярністю через свою простоту. Він має дещо ефективніші модифікації на кшталт двостороннього сортування бульбашкою, проте, з квадратичною асимптотикою, вони обидва залишаються одними з найгірших варіантів для сортування великих масивів.

Сортування вставками. Ідеєю даного алгоритму є перестановка елементів, але на відміну від бульбашкового сортування (в якому порівнюється та міняється місцями сусідні елементи), за допомогою swap ми одразу ставимо кожен елемент на своє місце. Не дивлячись на цю оптимізацію, кількість порівнянь все ще залишається квадратичною.

Алгоритм також є адаптивним, оскільки сортування також відбувається шляхом порівняння елементів. Проте, кількість порівнянь не змінюється, тому асимптотики по швидкості залишаються такими ж, як і при сортуванні бульбашкою:

$$\mathrm{O}(\mathrm{n}^2),\ \Theta(\mathrm{n}^2),\ \Omega(n).$$

Відповідно, сортування вставками є in-place алгоритмом, тому також не використовує додаткової пам'яті $Ospace(1)$.

Не дивлячись на те, що за асимптотичним аналізом даний алгоритм еквівалентний бульбашковому сортуванню, він є більш ефективним за рахунок меншої кількості перестановок елементів.

Сортування вибіркою. Алгоритм сортування вибіркою полягає у пошуку на кожному етапі максимального/мінімального елементу і його swap з початковим/кінцевим елементом невідсортованої частини, щоб поставити його на своє місце.

Цей алгоритм не є адаптивним, тому при пошуку максимального/мінімального елементу він в будь-якому разі проходитиметься по всій невідсортованій частині масиву. Ідея алгоритму є схожою до сортування вставкою, тобто для пошуку першого елементу ми виконуємо $n-1$ порівнянь, для наступного $n-2$ тощо. Тому асимптотика за часом виконання буде однаковою:

$$\mathrm{O}(\mathrm{n}^2),\ \Theta(\mathrm{n}^2),\ \Omega(n^2).$$

Як і попередні алгоритми, цей виконується на вихідному масиві, тому використання пам'яті константне: $Ospace(1)$.

Сортування злиттям. Це сортування відноситься до класу алгоритмів «розділяй і володарюй» (divide-and-conquer), який полягає у розділенні масиву даних на рівноправні частини, задача для яких може бути розв'язана незалежно, з подальшим об'єднанням цих частин. На етапі об'єднання частин, цей алгоритм за допомогою вказівників на їх елементи формує відсортований об'єднаний масив, що об'єднується з наступною половиною. Таким чином, об'єднання підмасивів відбувається за лінійний час. Підмасиви розділяються навпіл до тих пір, поки не залишаються підмасиви з одним елементом – бази рекурентного співвідношення – оскільки масив з одного елемента завжди є відсортованим.

В ході процесу сортування, першопочатковий масив довжиною n розбивається на два підмасива довжиною $\frac{n}{2}$, після сортування яких, вони об'єднуються за n операцій порівнянь. Ми можемо розрахувати асимптотику, використовуючи теорему Акра-Баззі [1] для розрахунку часової складності узагальнених рекурентних алгоритмів, де часова складність має таку форму:

$$T(n) = \sum a_i T(b_i n) + f(n). \quad (2)$$

В даній формулі a_i та b_i – параметри розбиття даних, а $f(n)$ – складність алгоритму поза рекурсією. Тоді, згідно теореми, часова складність $T(n)$ розраховуватиметься за формулою [1]:

$$T(n) = \Theta\left(n^p \cdot \left(1 + \int_1^n \frac{f(u)}{u^{p+1}} du\right)\right). \quad (3)$$

Зазначимо, що p – параметр, який є розв'язком показникового рівняння [1]:

$$\sum a_i b_i^p = 1. \quad (4)$$

На кожному етапі сортування злиттям алгоритм рекурсивно викликає себе двічі ($a_0 = 2$), а масив ділиться навпіл ($b_0 = \frac{1}{2}$), тому часова складність алгоритму матиме наступну форму:

$$T(n) = 2T\left(\frac{n}{2}\right) + n. \quad (5)$$

Очевидно, що доданок n – складність об'єднання частин (частини обоб'єднуються лінійно, тому складність лінійна). Для подальших розрахунків необхідно знайти p-параметр, розв'язавши рівняння, підставивши значення a_i та b_i у (4): $2 \cdot \left(\frac{1}{2}\right)^p = 1$, з якого отримуємо, що $p = 1$. Тоді можемо знайти часову складність:

$$T(n) = \Theta\left(n \cdot \left(1 + \int_1^n \frac{u}{u^2} du\right)\right) = \Theta\big(n(1 + \ln n)\big) = \Theta(n \ln n) = \Theta(n \log n). \quad (6)$$

Остання рівність вірна, оскільки $\ln n = \log n \ln 2$, а в асимптотичному аналізі константи не грають ролі і просто відкидаються. Така форма є більш зручною в контексті інформатики. Таким чином, отримуємо очікувану асимптотику сортування злиттям $\Theta(n \log n)$.

Подібним чином розраховується складність інших алгоритмів сортування, які базуються на принципі divide-and-conquer. Проте варто зазначити, що за допомогою методу Акра-Баззі ми розраховуємо лише клас швидкості. Тому, не дивлячись на те, що це «сімейство» алгоритмів еквівалентне за часовою складністю, їх реальна швидкість може значно відрізнятись за рахунок їх алгоритмічних відмінностей.

З використанням пам'яті даного алгоритму все стає цікавіше, оскільки функції, що рекурсивно викликаються, заносяться до стеку викликів, де вони зберігаються в пам'яті разом зі своїми локальними змінними. Стек викликів працює за принципом LIFO (останнім прийшов першим вийшов). Припустимо, що в програмі відсутні витоки пам'яті, тобто локальна пам'ять функцій вивільняється по виходу з неї. Таким чином, якщо функція верхнього рівня рекурсії зберігає 2 підмасива розміром $\frac{n}{2}$, що еквівалентно пам'яті розміру n, то функція наступного рівня зберігатиме $2 \cdot \frac{n}{4} = \frac{n}{2}$, і так далі. Таким чином, використання пам'яті буде сумою геометричної прогресії:

$$S(n) = Ospace\left(n + \frac{n}{2} + \frac{n}{4} + \cdots + 1\right) = Ospace(2n) = Ospace(n). \quad (7)$$

Швидке сортування. Один з найпопулярніший алгоритмів сортування, реалізація якого вбудована в велику кількість мов програмування. Як і попередній алгоритм, відноситься до класу divide-and-conquer, проте злиття відбувається інакше. На кожному етапі рекурсії вибирається опорний елемент підмасиву, і всі елементи, які більші або менші за нього, переміщуються по різні його сторони.

Ефективність алгоритму залежить від вибору опорних елементів. В найкращому випадку – при розділі навпіл – ми можемо розрахувати очікувану асимптотику за методом Акра-Баззі, що дасть нам $\Theta(n \log n)$, яка буде рівна також найкращій асимптотиці $\Omega(n \log n)$, оскільки алгоритм не адаптивний [1]. Проте, в найгіршому випадку (коли опорні елементи є максимумами/мінімумами підмасивів), алгоритм перетворюється в аналог сортування вставками, що дасть швидкодію $O(n^2)$.

Даний алгоритм в своїй рекурсивній реалізації виконує всі операції на вихідному масиві, тобто є in-place сортуванням. В найгіршому випадку обираються елементи, які є максимумами/мінімумами підмасиву, тоді стек викликів рекурсії заповниться кількістю викликів, яка рівна довжині масиву (оскільки

не буде його поділу навпіл), і використання пам'яті буде $Ospace(n)$. В найкращому випадку, масив розділятиметься навпіл, і тоді використання пам'яті буде $\Omega space(\log n)$.

Сортування купою (пірамідальне сортування). Купа – структура даних, яка являє собою повне (complete) бінарне дерево, що виконує умову: значення кожного батьківського вузла більше/менше або рівне значенням його дочірніх вузлів. Повні бінарні дерева самі по собі зручні тим, що їх легко представити у вигляді одновимірного масиву, де для елементу a_k дочірніми вузлами будуть a_{2k+1} та a_{2k+2}, звісно, за умов їх існування. Якщо виконується умова, що значення кожного батьківського вузла більше за значення дочірніх вузлів, то купа максимальна, інакше вона мінімальна. Також важливою властивістю куп є те, що всі їхні піддерева також є купами.

Сортування купою полягає у перетворенні вхідного масиву на максимальну/мінімальну купу в залежності від сортування за зростанням/спаданням. Сортування відбувається за наступним алгоритмом (всі подальші розрахунки і роздуми відбуватимуться для сортування за зростанням, але всі твердження вірні і для зворотного):

(I HS) Побудова максимальної купи відбувається за допомогою процедури heapify: рухаючись з середнього вузла (всі наступні вважатимемо вже сформованими купами, адже вони відповідають властивостям) на початок, відбувається «просіювання» елементів, при якому менші елементи переміщуються на низ майбутньої купи. Після цього, максимальний елемент знаходитиметься в корені купи.

(II HS) Міняємо місцями цей елемент з останнім елементом купи. Це зменшує на 1 розмір купи, а в кінці масиву формується відсортований масив.

(III HS) Якщо в купі залишилось більше 1 елементу, застосовуємо процедуру heapify знову на корені купи, щоб перемістити менший елемент на своє місце, і повторюємо крок (2).

Часова складність алгоритму буде сумою асимптотик окремих її частин:

$$T(n) = T_{build_heap}(n) + T_{sort}(n), \tag{8}$$

де build_heap відповідає за крок (I HS) алгоритму, а sort – кроки (II HS)-(III HS). Оцінимо складність T_{build_heap}. Для оцінки меж розглянемо найкращий і найгірший варіант. Нижня межа досягається в тому випадку, коли масив відсортований в зворотному напрямку, оскільки отриманий масив вже буде максимальною купою (в цьому легко переконатися, оскільки всі елементи зліва будуть більшими за елементи справа). Тоді $T_{build_heap}(n) = \Omega(n)$.

Верхня межа – найгірший випадок – досягається, коли масив відсортований за зростанням (з чого випливає, що даний алгоритм не адаптивний), тоді при кожному етапі build_heap витрачатимуться додаткові операції для переміщення елементів в самий низ. Тоді для вершини купи кількість операцій буде $\log n$, для його предків – $\log n - 1$, тощо. Асимптотика build_heap в такому випадку буде

$$T_{build_heap}(n) = O\left(\sum_{1}^{\log n-1} i \cdot 2^{\log n-1-i}\right) = O\left(\frac{n}{2}\sum_{1}^{\log n-1}\frac{i}{2^i}\right). \tag{9}$$

Остання сума є частковою сумою спадного ряду, значення якого можна порахувати шляхом диференціювання геометричного ряду:

$$\begin{cases} \frac{d}{dx}\sum x^i = \sum ix^{i-1} = \frac{1}{\text{x}}\sum ix^i \\ \frac{d}{dx}\sum x^i = \frac{d}{dx}\cdot\frac{1}{1-x} = \frac{1}{(1-x)^2} \end{cases} \Rightarrow \frac{1}{\text{x}}\sum ix^i = \frac{1}{(1-x)^2}, \qquad |x| < 1, i \to \infty. \tag{10}$$

В нашому випадку $x = \frac{1}{2}$, тому ряд збігається до 2, і ми можемо відкинути його з асимптотичної формули, оскільки часткова сума не перевищить цього значення. Результуюча асимптотика буде

$$T_{build_heap}(n) = O(n + n) = O(n). \tag{11}$$

Таким чином, складність формування купи залишається сталою.

Сам процес сортування полягає у заміні місцями кореня і останнього елементу купи, і повторенні операції heapify для кореня. За властивостями купи, новий корінь переміститься в її кінець. Таким чином отримуємо асимптотику сортування:

$$T_{sort}(n) = O\left(\sum_{1}^{n}\log i + n\right) = O\left(\log\prod_{1}^{n} i + n\right) = O(\log n! + n). \tag{12}$$

Використовуючи логарифмічний вигляд формули Стірлінга для апроксимації факторіала, отримуємо, що

$$T_{sort}(n) \approx O(n\log n - n + n) = O(n\log n). \tag{13}$$

Дана асимптотика не залежить від даних, записаних в купі. Підбиваючи підсумки, отримуємо, що загальну асимптотику алгоритму не має найгіршого чи найкращого варіанту, оскільки клас складності для всіх етапів алгоритму не змінюється в залежності від даних:

$$T(n) = O(n + n\log n) = \text{O}(n\log n). \tag{14}$$

Сортування купою є in-place, тому не використовує додаткову пам'ять, тож просторова складність $Ospace(1)$, що робить цей алгоритм просторово найефективнішим серед алгоритмів класу $O(n\log n)$. Даний алгоритм в загальному є дещо повільнішим за швидке сортування і сортування злиттям, проте, на відміну від них, сортування купами не дає гірших випадків.

Коміркове сортування. Даний алгоритм сортування базується на сортуванні підрахунком: коли ми, проходячись по масиву, підраховуємо кількість входжень кожного елементу і вносимо це значення в окремий масив на основі значень цих елементів. Даний алгоритм виконується за час $O(n+k)$, де k – діапазон значень, і вимагає $O(k)$ додаткової пам'яті, що ставить під сумнів ефективність алгоритму при сортуванні великих чисел. Скажімо, при сортуванні масиву 4 байтових чисел, нам необхідно створити додатковий масив з 2^{32} елементів.

Коміркове сортування дозволяє оптимізувати аспект використання пам'яті шляхом обмеження кількості додаткових елементів не діапазоном даних, а розміром масиву, створюючи таку кількість бакетів – «відер» – між якими розподіляються елементи. Нехай для сортування масиву A за зростанням ми використовуємо n бакетів: $B_0, B_1, \dots B_{n-1}$. Тоді справедливими будуть наступні властивості:

$$\left|\bigcup_0^{n-1} B_i\right| = |A| \quad \forall i < j\colon\ \forall x \in B_i, \forall y \in B_j\colon\ x \le y \tag{15}$$

Тобто самі бакети між собою відсортовані. Оскільки не існує властивості, яка б дозволила розподілити всі елементи по бакетам рівномірно, майже завжди існуватимуть бакети з більшою кількістю елементів. Тоді задача зводиться до сортування елементів в середині кожного бакета. Для коміркового сортування для цієї задачі характерним є використання сортування вставками. Покроково алгоритм виглядає наступним чином:

(I BS) Необхідно визначити максимальний елемент a_{max} для визначення діапазону даних, на основі якого відбуватиметься розподіл

(II BS) Кожен елемент a_i масиву помістити в бакет з індексом $\left\lfloor \frac{a_i}{a_{max}} \cdot (n-1) \right\rfloor$. Оскільки множник $\frac{a_i}{a_{max}}$ змінюється в відрізку $[0,1]$, тому $n-1$

(III BS) Кожен бакет окремо відсортувати вставками

(IV BS) Послідовно конкатенувати бакети

Тоді асимптотика алгоритму буде рівна

$$T(n) = T_{(I\ BS)}(n) + T_{(II\ BS)}(n) + T_{(III\ BS)}(n) + T_{(IV\ BS)}(n). \tag{16}$$

Очевидно, що $T_{(I\ BS),(I\ BS),(IV\ BS)} = O(n)$.

Оцінимо час виконання етапу (III BS). Зрозуміло, що нижня межа досягається за умови, коли нема ні одного порожнього бакета. Іншими словами, елементи рівномірно розподілені. Тоді асимптотика етапу сортування складатиме $O(1)$.

Розглянемо верхню межу, в такому випадку, при сортуванні кожного бакета час виконання $T(n) \sim \sum_0^{n-1} n_i^2$, де n_i – кількість елементів в і-ому бакеті. Максимальне значення часу досягається, коли всі елементи зосереджені в одному бакеті, в чому легко переконатись. Виходячи з першої властивості розбиття на бакети: $\sum_0^{n-1} n_i = n$, порівняємо кількість операцій, необхідних для сортування цільного бакету з n елементами, і розбитого, шляхом віднімання:

$$n^2 - \sum_0^{n-1} n_i^2 = \sum_0^{n-1} n_i^2 + 2\sum_{i<j} n_i n_j - \sum_0^{n-1} n_i^2 = \sum_{i<j} n_i n_j > 0 \tag{17}$$

Тому, кількість операцій для сортування одного цільного бакету з n елементів дійсно буде більша. Даний аспект можна оптимізувати, знаходячи разом з a_{max} також мінімальний елемент a_{min}, тому індекс бакета буде розраховуватись як $\left\lfloor \frac{a_i - a_{min}}{a_{max} - a_{min}} \cdot (n-1) \right\rfloor$, що дозволить в окремих випадках зменшити діапазон і краще розподіляти елементи по бакетах. Проте, асимптотика в найгіршому випадку залишиться $O(n^2)$ через сортування вставками. Покращений варіант цього алгоритму Radix-sort дозволяє позбутись найгіршого випадку і необхідності використання додаткових алгоритмів сортування. Очевидно, коміркове сортування не є адаптивним.

З урахуванням випадкового розподілу, яким в більшості випадків і є масиви даних, що підлягають сортуванню, імовірність того, що елементи потраплять в один бакет, експоненційно прямує до нуля. Іншими словами, математичне сподівання кількості елементів в кожному бакеті прямуватиме до 1. Тому середня асимптотика для конкретно розглянутого варіанту буде $\Theta(n)$, або $\Theta(n+k)$ в загальному випадку, де k – кількість бакетів.

Використання пам'яті залежить від конкретної реалізації, теоретично це $Ospace(n+k)$, тобто сума кількості елементів та кількості бакетів. На практиці таке використання пам'яті досягається при використанні динамічних масивів або при завчасному розрахунку кількості елементів в кожному бакеті перед їх створенням, що впливає на швидкодію.

Тести швидкості. Були проведені тести швидкості для порівняння алгоритмів різних класів. Для порівняння були використані засоби мови програмування Python 3.14 і бібліотеки numpy. Для кращої точності вимірювань проводилось декілька тестів алгоритму і бралось середнє значення часу виконання.

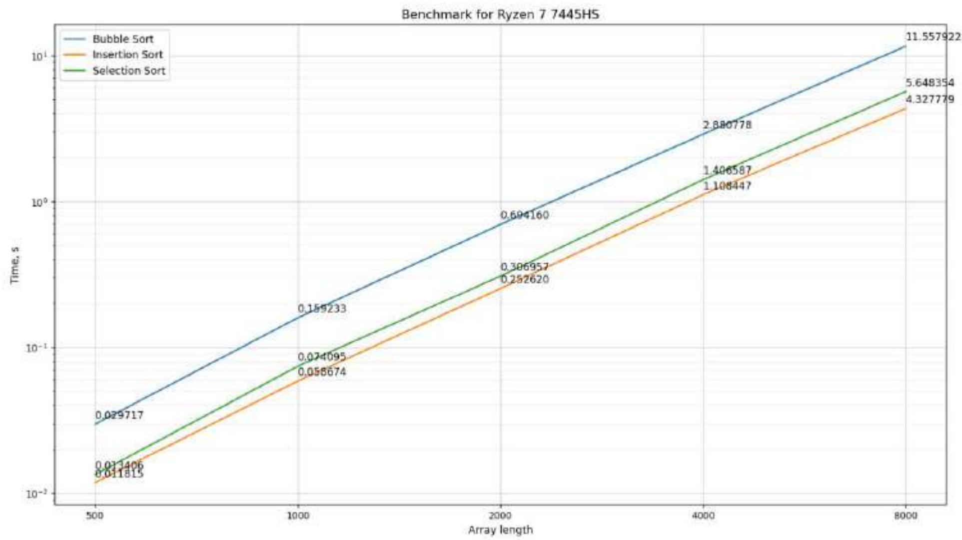

Рис. 1. Тест швидкості алгоритмів сортування асимптотики n^2

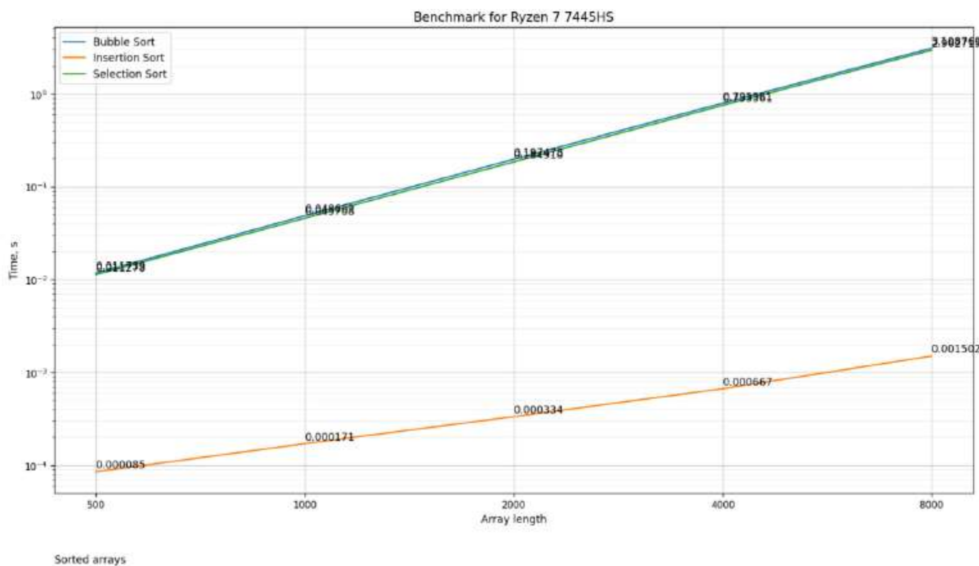

Рис. 2. Тест швидкості алгоритмів сортування асимптотики n^2 на завчасно відсортованих масивах

На рисунках 1 та 2 зображено порівняння часу виконання алгоритмів сортування асимптотик n^2: сортування бульбашкою, вставками та вибіркою. На рисунку 1 сортування відбувається на випадкових масивах, на рисунку 2 – завчасно відсортованих. Як бачимо, на відсортованих масивах час виконання зменшується на порядки, що свідчить про адаптивність алгоритмів. Особливо сильне прискорення спостерігається при сортуванні вставками.

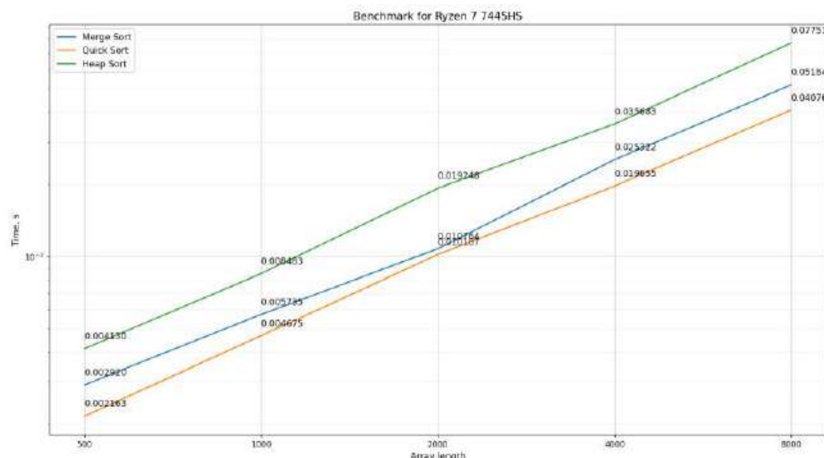

Рис. 3. Тест швидкості алгоритмів асимптотики $n\log n$

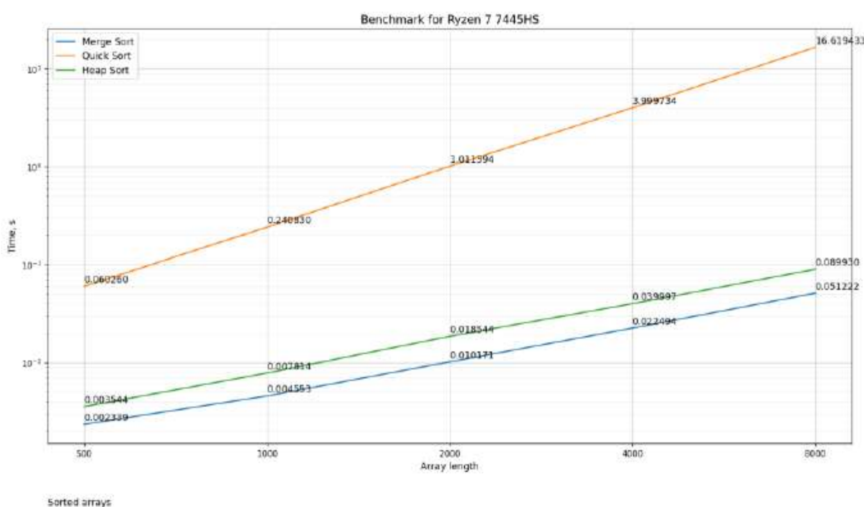

Рис. 4. Тест швидкості алгоритмів сортування асимптотики $n\log n$ *на завчасно відсортованих масивах*

На рисунках 3 та 4 зображено порівняння часу виконання алгоритмів сортування асимптотик $n\log n$: сортування злиттям, швидке сортування та сортування купами. На рисунку 3 сортування відбувається на випадкових масивах, на рисунку 4 – завчасно відсортованих. В основному, ніяких покращень не відбувається при сортуванні завчасно відсортованих масивах, і навпаки – швидке сортування деградувало, що пояснюється неоптимальним вибором опорної точки в нашій реалізації. В оптимізованих реалізаціях та варіаціях швидкого сортування (наприклад, вбудованих функцій сортування багатьох мов програмування) деградація не відбувається.

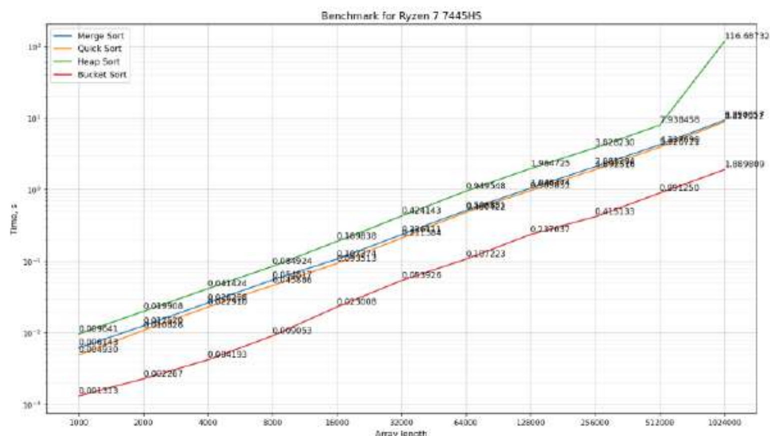

Рис. 5. Тест швидкості для алгоритмів асимптотики $n\log n$ *та коміркового сортування на великих масивах*

Також провели тестування (рис. 5) швидких алгоритмів на великих масивах, щоб зробити висновок про їх поведінку при сортуванні великих даних. Як видно, найшвидшим з алгоритмів є коміркове сортування, хоча воно використовує найбільшу кількість пам'яті. Найповільнішим виявилось сортування купами, хоча його використання пам'яті найменше.

На основі отриманих результатів, можна скласти наступну порівняльну таблицю:

Таблиця 1

Порівняння алгоритмів сортування за класами часової, просторової складності, а також за адаптивністю

Алгоритм	**Час виконання**			**Використання пам'яті**		**Адаптивність**
	Найгірший	Очікуваний	Найкращий	Найгірше	Очікуване	
Bubble	$O(n^2)$		$\Omega(n)$	$O(1)$		+
Insertion	$O(n^2)$		$\Omega(n)$	$O(1)$		+
Selection	$O(n^2)$			$O(1)$		+
Merge	$O(n \log n)$			$O(n)$		-
Quick	$O(n^2)$	$\Theta(n \log n)$		$O(n)$	$\Theta(\log n)$	-
Heap	$O(n \log n)$			$O(1)$		-
Bucket	$O(n^2)$	$\Theta(n+k)$		$O(n+k)$		-

Висновок. В результаті аналізу алгоритмів сортування були виділені основні підходи, які лягають у принципи більшості алгоритмів, і які мають найбільший вплив на швидкодію: порівняння елементів, і присвоєння елементів. Такі алгоритми, як сортування бульбашкою або швидке сортування, цілком засновані на порівнянні елементів, а, наприклад, Radix-sort взагалі не порівнює елементи і базується лише на присвоєннях.

Алгоритми були проаналізовані з точки зору швидкодії. Для більш складних алгоритмів математично були виведені класи швидкостей, до яких вони належать. Також була проаналізована властивість адаптивності алгоритмів сортування, і хоча, на перший погляд, вона не має особливого значення, ми побачили на прикладі тестів швидкості, що різниця може бути в сотні і тисячі разів.

Проведений аналіз та експерименти підтвердили, що вибір оптимального алгоритму залежить від задач і середовища (обмеження даних, обмеження апаратного характеру тощо).

З отриманих результатів можна зробити висновок, що серед розглянутих алгоритмів **коміркове сортування** є найефективнішим алгоритмом сортування, коли мова йде про сортування великих масивів даних за найменший час при наявності достатньої кількості пам'яті (сервери, мультикомп'ютерні кластери тощо). При обмежених апаратних ресурсах (мікроконтроллери, розумні чипи тощо), найкращим алгоритмом може бути **швидке сортування** або **коміркове сортування** в залежності від необхідності адаптивності. Для сортування частково або завчасно впорядкованих даних найкраще показав себе алгоритм **сортування вставками**, який виявився в десятки разів швидше за алгоритми складності $O(n \log n)$.

References:

1. GeeksforGeeks (2025), *Akra-Bazzi method for finding the time complexities*, [Online], available at: https://www.geeksforgeeks.org/dsa/akra-bazzi-method-for-finding-the-time-complexities/
2. Ashok, K.K. (2014), *A Survey, Discussion and Comparison of Sorting Algorithms*, Umeå University, Department of Computing Science, [Online], available at: https://www.diva-portal.org/smash/get/diva2:739880/FULLTEXT01.pdf
3. GeeksforGeeks (2025), *Bucket Sort*, [Online], available at: https://www.geeksforgeeks.org/dsa/bucket-sort-2/
4. Goel, K., Dwivedi, P. and Sharma, O. (2023), «Performance Analysis of Various Sorting Algorithms: Comparison and Optimization», *2023 11th International Conference on Intelligent Systems and Embedded Design (ISED)*, pp. 1–5, doi: 10.1109/ISED59382.2023.10444609.
5. GeeksforGeeks (2026), *Heap Sort*, [Online], available at: https://www.geeksforgeeks.org/dsa/heap-sort/
6. Karuna, S. (2018), *An Introduction to Bucket Sort*, [Online], available at: https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124
7. Mohammed, A., Zhanar, M., Alaa, F.A. et al. (2024), «Comparative Analysis of Sorting Algorithms: A Review», *11th International Conference on Soft Computing & Machine Intelligence*, pp. 1–8.
8. Pizarro-Vasquez, G.O. et al. (2021), «Sorting Algorithms and Their Execution Times an Empirical Evaluation», in Botto-Tobar, M. et al. (ed.), *Advances in Emerging Trends and Technologies*, pp. 329–340, doi: 10.1007/978-3-030-63665-4_27.
9. Sakpal, N. et al. (2024), *Comparative Analysis of Sorting and Searching Algorithms*, SSRN, [Online], available at: https://ssrn.com/abstract=4815467
10. Santos, A.B.G. et al. (2021), «Asymptotic Analysis of the Running Time Performed by Various Sorting Algorithms», *2021 International Conference on Intelligent Technologies (CONIT)*, IEEE, pp. 1–6, doi: 10.1109/CONIT51480.2021.9498490.

Іванов Артем Олександрович – здобувач освіти Житомирського державного університету імені Івана Франка.

https://orcid.org/0009-0006-2704-9906.

Наукові інтереси:

– рефакторинг та алгоритмічна стійкість;
– математичне моделювання обчислювальних процесів.

Кривонос Олександр Миколайович – кандидат педагогічних наук, доцент, доцент кафедри комп'ютерних наук та інформаційних технологій Житомирського державного університету імені Івана Франка.

https://orcid.org/0000-0002-4211-6541.

Наукові інтереси:

– дослідження на стику методів обчислень та обчислювальної геометрії для задач сучасної робототехніки.

E-mail: krypton@zu.edu.ua.

Ivanov A.O., Kryvonos O.M.

Comparative analysis of time and space complexity of sorting algorithms in modern software environments

This article deals with a comprehensive study and comparative analysis of the most common sorting algorithms, such as bubble sort, insertion sort, selection sort, merge sort, quick sort, heap sort, and bucket sort. The relevance of this work stems from the rapid development of Big Data and the Internet of Things (IoT) technologies, which require high data processing efficiency under conditions of both unlimited computing power and extremely limited memory resources. The authors examine in detail the theoretical aspects of each algorithm, performing mathematical modeling of their time and space complexity using asymptotic analysis and the Akra-Bazzi method for recursive structures.

The research methodology combines theoretical asymptotic analysis with empirical performance testing in the Python 3.14 environment on a modern Ryzen 7 7445HS processor architecture. The work investigates not only performance metrics on random arrays but also the algorithms' adaptability to already sorted or partially sorted data. The experimental results clearly demonstrate how the choice of pivot in quick sort or the specifics of call stack management affect actual execution time and memory usage.

Based on the obtained data, practical recommendations have been formulated regarding the application of specific sorting methods depending on hardware constraints and the nature of the input data. Authors found that bucket sort is the most efficient for large arrays when sufficient RAM is available, while heap sort proves to be optimal in terms of space complexity among O(n \log n) algorithms. The insertion sort demonstrates the best speed results for specific conditions of ordered data. The article has scientific and practical value for software developers seeking to optimize the processes of indexing and analyzing large volumes of information.

Keywords: sorting algorithms; time complexity; space complexity; asymptotic analysis; Big Data; algorithm adaptability; Python; Akra-Bazzi method; Big O notation; comparative testing.

Стаття надійшла до редакції 22.12.2025.