УДК 004.042+004.043 https://doi.org/10.32515/2664-262X.2025.11(42).2.23-29

М. В. Недьошев, **В. В. Кириченко**, доц., канд. фіз.-мат. наук
Національний університет біоресурсів і природокористування України, м.Київ, Україна
e-mail: m.nedoshev@nubip.edu.ua, v.kyrychenko@nubip.edu.ua

Особливості роботи складних обчислювальних алгоритмів на прикладі перевірки гіпотези Полокка

В роботі проведено аналіз алгоритмів пошуку представлення числа як суми тетраедричних чисел із використанням різних методів оптимізації, таких як жадібний алгоритм, апроксимація, двоє вказівників, мемоізація та багатопотоковість. Розглянуто різні техніки, що сприяють зменшенню часу обчислень, зокрема багатопотокова обробка та використання ефективних структур даних. Описано, як реалізувати ці методи в контексті різних мов програмування. Також проведено експерименти з оцінкою продуктивності різних підходів та порівняння результатів. Результати демонструють суттєве підвищення швидкості виконання алгоритму під час використання багатопотокових підходів та оптимізації коду.
гіпотези Поллока, алгоритми пошуку, складність алгоритмів алгоритм, паралельні складні обчислення, багатопотоковість

Постановка проблеми. Гіпотези Поллока стосуються представлення чисел у вигляді суми фігурних чисел. Зокрема, третя гіпотеза Поллока стверджує, що будь-яке натуральне число можна представити у вигляді суми не більше ніж п’яти тетраедричних чисел. Попри те, що ця гіпотеза було висунуто ще у XIX столітті, її доведення або спростування досі залишається відкритим питанням у теорії чисел.

Ключовою проблемою під час дослідження є побудова ефективного обчислювального алгоритму, який дозволяє знаходити відповідне представлення для довільного натурального числа N. Оскільки тетраедричні числа зростають за кубічною прогресією, наївний повний перебір комбінацій супроводжується високими обчислювальними витратами, особливо при великих значеннях N. Це зумовлює необхідність розробки оптимізованих підходів до перебору, які мінімізують кількість операцій та забезпечують масштабованість.

Метою дослідження є створення продуктивного алгоритму для перевірки третьої гіпотези Поллока, а також його реалізація та експериментальне тестування на практиці з використанням різних мов програмування — JavaScript, Python, Apple Metal (GPU) та C — для порівняння їхньої продуктивності та особливостей реалізації. Результатом дослідження має бути порівняння виконання складних алгоритмів використовуючи поширені скриптові мови програмування, такі як JavaScript та Python, з C та виконанням коду на графічному адаптері з використанням паралелізації.

Аналіз останніх досліджень і публікацій. Гіпотези Поллока одна із проблем адитивної теорії чисел сформована Фредеріком Поллоком у 1850 році. Не дивлячись на простоту формулювання гіпотез, вони не доведені математичним шляхом. Однак гіпотезу було перевірено емпірично через 150 років. Було перевірено до 250 000 000 чисел і виявлено що існує мінімум 241 число, яке формується сумою не менше як 5 тетраедричних чисел, останнє з яких 343 867 [1].

Для виявлення особливостей роботи алгоритмів, алгоритм спочатку тестується використовуючи просту скриптову мову. Автори роботі [2] та [3] провели порівняння актуальних скриптових мов програмування у роботі з великою кількістю даних. Окремо має сенс перевірити виконання обрахунків використовуючи графічний адаптер, як зроблено авторами [4].

Постановка завдання. Перевірка третьої гіпотези Поллока про представлення

будь-якого натурального числа у вигляді суми не більше ніж п’яти тетраедричних чисел є важливим завданням у теорії чисел. Оскільки математичного доведення цієї гіпотези досі немає [1], необхідно розробити ефективний алгоритм для її експериментальної перевірки. Тому постає завдання дослідити властивості тетраедричних чисел та їх застосування для перевірки гіпотези Поллока.

Для досягнення мети необхідно вирішити такі завдання:

1. Дослідити математичні властивості тетраедричних чисел, їхнє зростання та особливості представлення натуральних чисел.
2. Розглянути наявні алгоритмічні підходи до розкладу чисел на суму фігурних чисел.
3. Розробити ефективний для великих чисел алгоритм, що знаходить представлення числа у вигляді суми від одного до п’яти тетраедричних чисел.
4. Встановити метрики для перевірки ефективності алгоритму.
5. Реалізувати алгоритм у вигляді програмного коду та провести тестування використовуючи декілька мов програмування, які досліджуються.
6. Проаналізувати отримані результати, визначити закономірності та можливі винятки.

Виклад основного матеріалу. Гіпотеза Поллока про представлення натуральних чисел у вигляді суми тетраедричних чисел є однією з невирішених задач теорії чисел. Для її експериментальної перевірки необхідно розробити алгоритм, що дозволяє знаходити такі представлення ефективним способом.

Визначимо формулу для обчислення тетраедричних чисел, потім проаналізуємо можливі методи розкладу чисел у їхню суму та розглянемо оптимізаційні стратегії, які дозволяють зменшити обчислювальну складність.

Тетраедричні числа — це особливий вид фігурних чисел, які відповідають кількості куль у тривимірній піраміді (тетраедрі), які можна представити у вигляді формули (1) для будь-якого натурального числа n.

$$T_n = \frac{n(n+1)(n+2)}{6} \tag{1}$$

де T_n – n-те тетраедричне число.

Згідно з третьою гіпотезою Поллока, будь-яке натуральне число (N) може бути подано у вигляді суми не більше ніж п’яти тетраедричних чисел (2).

$$N = T_{i1} + T_{i2} + T_{i3} + T_{i4} + T_{i5} \tag{2}$$

де $i1, i2, i3, i4, i5$ – деякі натуральні числа.

Послідовність перших тетраедричних чисел [5] виглядає наступним чином: 1, 4, 10, 20, 35, 56, 84, 120, 165, 220…

Ці числа ростуть квадратично, що ускладнює пошук їхніх сум для великих значень N.

Для доведення що будь-яке число N можна отримати додавши максимум п’ять тетраедричних чисел було створено алгоритм, який перебирає можливі суми від одного до п’яти тетраедричних чисел поки їх сума не буде дорівнювати N.

Алгоритм має перебирати можливі комбінації із п’яти чисел, що в гіршому випадку має складність $O(n^5)$. Очевидно що можна використовувати більш оптимальне рішення. Для оптимізації алгоритму було використано такі принципи оптимізації: принцип жадібного алгоритму, апроксимацію математичних виразів, пошук з використанням двох показників, мемоізації та паралельні обчислення.

Пошук до п’яти тетраедричних чисел, які утворюють задане число N можна розглядати поетапно: спочатку перевірити чи є число, якого само по собі вже достатньо для утворення числа N (3), потім знайти такі два числа, які утворюють у сумі N і так п’яти чисел. Потрібно знайти оптимальне рішення для пошуку T_i для таких випадків.

$$\begin{gathered} T_1 = N \\ T_1 + T_2 = N \\ T_1 + T_2 + T_3 = N \\ T_1 + T_2 + T_3 + T_4 = N \end{gathered} \tag{3}$$

$$T_1 + T_2 + T_3 + T_4 + T_5 = N$$

Для пошуку першого числа з суми (T_1), потрібно перевірити чи є число N тетраедричним — якщо число N тетраедричне, значить це число $T_1 = N$. Для цього потрібно представити формулу тетраедричного числа у вигляді кубічного рівняння (4) і підставити число N як y, де x буде індексом тетраедричного числа. Таким чином перший елемент можна знайти за $O(1)$. Операція розв'язання кубічного рівняння не швидка сама по собі. Тому було вирішено спростити формулу розв'язання для пошуку числа наближеного до індексу тетраедричного числа (5). Отже, формула пошуку тетраедричного числа виглядає наступним чином:

$$y = \frac{x^3+3x^2+2x}{6} \quad (4)$$

$$x_{aprox} \approx \lfloor \sqrt[3]{6y+3+2} \rfloor \quad (5)$$

Таким чином було відкинуто зайвий пошук дискримінантів і зменшено кількість операцій для пошуку наближеного індексу ближчого тетраедричного числа. Наближене значення має похибку не більше одиниці. Можна отримати індекс числа значно більше за N, в такому випадку індекс потрібно зменшити на один.

Для отримання найближчого тетраедричного числа, яке менше за саме число можна використовувати таку формулу (6).

$$x = max(x_{aprox} - \delta, 0) \quad (6)$$

де $\delta = 1$, якщо $x_{aprox} > \frac{y(y+1)(y+2)}{6}$, інакше $\delta = 0$

Якщо підставити тетраедральне число за знайденим індексом і воно дорівнює N, то перший випадок $N = T_1$, отже для числа N виконується гіпотеза Поллока.

Для пошуку двох доданків тетраедричних чисел, які формують число N можна використовувати метод пошуку з двома показниками, що має складність $O(n)$ [6]. Ідеально було б використовувати бінарний пошук, але суми двох тетраедричних чисел, на відміну від самої послідовності тетраедричних чисел не є зростаючою.

	1	2	3	4	5	6	7	8	9	10
1	2	5	11	21	36	57	85	121	166	221
2	5	8	14	24	39	60	88	124	169	224
3	11	14	20	30	45	66	94	130	175	230
4	21	24	30	40	55	76	104	140	185	240
5	36	39	45	55	70	91	119	155	200	255
6	57	60	66	76	91	112	140	176	221	276
7	85	88	94	104	119	140	168	204	249	304
8	121	124	130	140	155	176	204	240	285	340
9	166	169	175	185	200	221	249	285	330	385
10	221	224	230	240	255	276	304	340	385	440

Рисунок 1 – Суми двох тетраедричних чисел

Джерело: розроблено авторами

З рисунка 1 видно, що немає чіткої прогресії при додаванні тетраедричних чисел, тобто умова $x_{i1} + x_{i2} \leq f(x_{i1}) + f(x_{i2})$ не виконується, де $f(x) = \frac{x(x+1)(x+2)}{6}$. Також з рисунка 1 видно що суми повторюються, оскільки $T_{i1} + T_{t2} = T_{i2} + T_{i1}$ тоді має сенс виконувати пошук лише для $T_{i1} + T_{i2}$ при $i2 < i1$.

Формулу пошуку найближчого тетраедричного числа (x_{approx}) можна використовувати для знаходження лімітів в рамках яких можна шукати інші числа, які можуть формувати суму. Таким чином можна сильно знизити кількість елементів, які потрібно перевіряти. Для кожної перевірки наступних чисел потрібно знайти верхню та нижню границю індексів тетраедричних чисел, які має сенс перевіряти. Для верхньої границі немає сенсу підставляти в T_i числа більше за N. Для нижньої границі мінімальне число залежить від кількості доданків. Так, наприклад, для пошуку двох

доданків, немає сенсу перевіряти числа, які разом менше за $N/2$. Тоді для пошуку доданків можна встановити такі ліміти за формулою (7).

$$\frac{N}{c} \leq x_i \leq N \quad (7)$$

де, c — кількість доданків, x_i — тетраедричні числа, що перевіряються, N — число для якого шукаються доданки.

Для пошуку трьох чисел які формують число N потрібно перебрати T_{i3} використовуючи ліміти та знайти два числа $T_{i1} + T_{i2}$, які утворюють $N - T_{i3}$, що є алгоритмом складності $O(n^2)$. Те саме стосується для пошуку T_{i4} та T_{i5}. Отже, пошук суми дорівнює $O\left(n/5\right) * O\left(n/4\right) * O\left(n/3\right) * O\left(n/2\right) * O(1) = O\left(\frac{n^4}{120}\right)$ в гіршому випадку, де n - це кількість тетраедричних чисел менших за задане N, що приблизно дорівнює $\sqrt[3]{6N+5}$. Використовуючи принцип жадібного алгоритму, перебір чисел починається з найбільших, таким чином кількість ітерацій значно зменшується.

На рисунку 2 представлено розподіл кількості мінімальних тетраедричних чисел для формування натуральних чисел. Враховуючи що з п'яти чисел більше комбінацій то швидше алгоритм працює коли шукаються одразу від 5 доданків. В такому випадку розподіл результату може виглядати не самим оптимальним у порівнянні з пошуком можливих сум с доданками по черзі від 1 до 5. Але враховуючи що існує лише 241 [1] число, яке потребує суми мінімум п'яти тетраедричних для підтвердження гіпотези Поллока, то швидше буде перевірити 4 числа і в разі невдачі, перевірити 5 доданків.

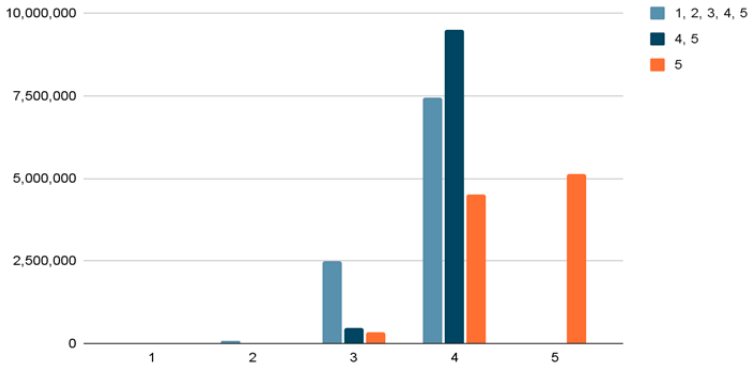

Рисунок 2 – Розподіл мінімально необхідної кількості тетраедричних чисел для формування натуральних чисел

Джерело: розроблено авторами

Використовуючи принцип жадібного алгоритму, якщо починати віднімати від цільового числа N максимально близькі тетраедричні числа, швидко задача зводиться до знаходження суми для маленьких чисел, які вже обчислювались. Тому має сенс використовувати мемоізацію для маленьких чисел, наприклад, перші можливі суми, які утворюють від одного до трьох перших 50 тетраедричних числа, але на дуже великих числах приріст у швидкості виконання вже не так сильно помітно, оскільки числа ростуть квадратично і кількість тетраедричних чисел, які потрібно перевірити значно збільшується.

Для тестів використовувався процесор Apple M1 Pro - 10 ядер, 2 з яких енергоефективні, 16GB LDDR5 оперативної пам'яті, Apple M1 Pro GPU, 16 Cores. Тести проводилися з використанням Node та Bun - платформи, яка може виконувати TypeScript та JavaScript код без браузера, Python та Apple Metal Framework. На рисунку 3 представлений графік, який показує роботу алгоритму виконуючи алгоритм в одному потоці.

Наступний етап оптимізації це використання багатопотоковості. Оскільки алгоритм шукає елементи суми для кожного числа незалежно від інших чисел, то можна поділити перевірку гіпотези Поллока на декілька окремих потоків. Для цього

можна використовувати як окремі потоки центрального процесора, так і графічного процесора. Для виконання на графічному процесорі з алгоритму потрібно прибрати рекурсивні виклики, що доволі просто зробити створивши копії основної функції пошуку до трьох разів для пошуку T_{i1}, T_{i2}, T_{i3}.

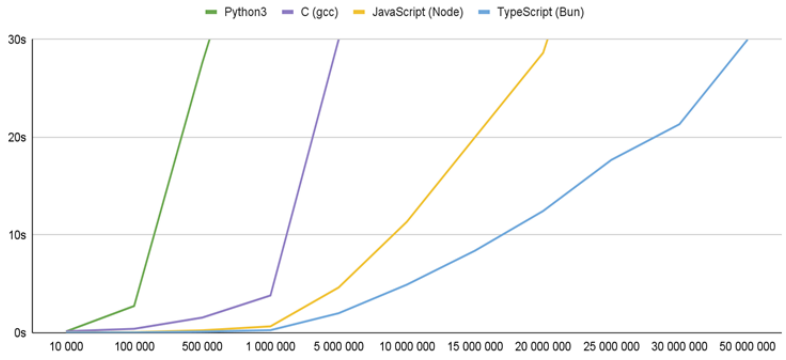

Рисунок 3 – Графік залежності часу від кількості елементів для розрахунку на різних платформах та мовах програмування

Джерело: розроблено авторами

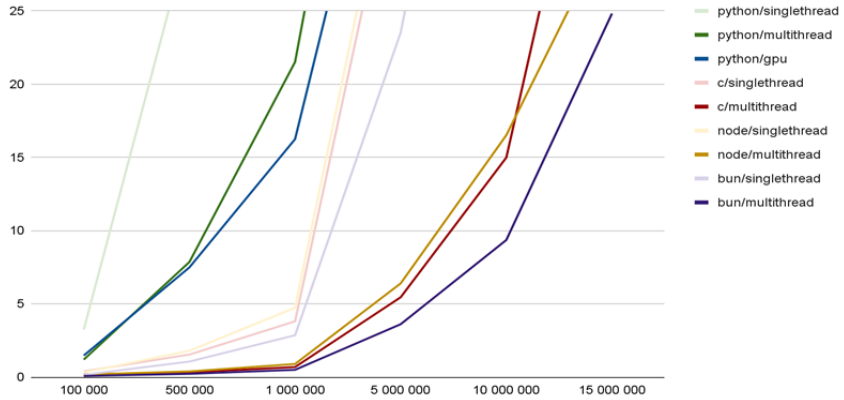

Рисунок 4 – Графік залежності часу від кількості елементів для розрахунку на різних платформах та мовах програмування з використанням багатопотоковості

Джерело: розроблено авторами

З рисунка 4 видно що паралелізм дає значного прискорення для Python, Bun, NodeJS та C.

Для Python було додано варіант паралелізму з використанням графічного адаптера. Використовуючи Apple Metal Framework [7] алгоритм було написано у вигляді шейдера, який перевіряє натуральні числа на відповідність третій гіпотезі Поллока, Python же використовував відображення до фреймворку Apple Metal. Було спеціально мінімізовано дані, які передаються від CPU до GPU. Вхідними даними для шейдера є початок (зсув), а вихідними даними одне число, де 0 означає що сума не була знайдена, а 1 що сума знайдена. Таким чином було мінімізовані дані, які потрібно передавати між CPU і GPU. Завдяки цьому значно зменшилось навантаження на канал передачі даних, що позитивно позначилось на загальній продуктивності. В середньому використовувалось від 10 до 40 мб пам’яті.

Для мови C була використана багатопоточність за допомогою бібліотеки pthread, що дозволило ефективно розподілити обчислювальне навантаження на кілька ядер процесора. Компіляція з оптимізацією за допомогою GCC (-O3) також допомогла забезпечити максимальну продуктивність. Цей підхід дозволив повторити структуру коду, схожу на ті, що використовуються в інших мовах, і забезпечити масштабованість при обробці великих обсягів даних. Використання пам’яті мінімальне – 1мб.

У середовищах Bun і NodeJS використовувалася багатозадачність через Worker-и, що дозволило максимально ефективно використовувати багатоядерні процесори. Ці платформи автоматично оптимізують розподіл потоків через Thread Pool, що додатково підвищує ефективність виконання алгоритмів без необхідності вручну налаштовувати кількість потоків. Використання пам'яті максимальне від 60 до 270 мб.

Було протестовано алгоритм для натуральних чисел від 1 до 1 000 000 000, перевірено для всіх цих чисел гіпотеза Поллока справедлива за 81 хвилин та 22 секунди з середнім споживанням пам'яті 387.563 МБ, використовуючи платформу Bun.

Висновки. У ході дослідження було розроблено і показано ефективність алгоритму для експериментальної перевірки третьої гіпотези Поллока, яка стверджує, що будь-яке натуральне число можна представити у вигляді суми не більше ніж п'яти тетраедричних чисел.

Алгоритм спеціально оптимізовано для роботи з великими числами. Апроксимація, пошук з двома покажчиками та паралелізм позитивно вплинули на роботу алгоритму. Мемоїзація ж не дала значного зростання продуктивності на великих числах.

Найшвидшим з тестів виявився Bun у багатопотоковому режимі навіть швидше за C. Порівнюючи швидкодію C з NodeJS у інших тестах [8], можна дійти висновку, що часто трапляються випадки, коли NodeJS та C мають схожу продуктивність. Розробники Bun на своєму сайті стверджують [9], що швидкість виконання JavaScript коду на платформі Bun швидша за NodeJS, що відповідає проведеному дослідженню та видно на рисунку 4. Платформа Bun виконує JavaScript код і сама додає оптимізації, що значно простіше за написання C.

З графіка на рисунку 4 видно, що виконання на графічному процесорі лише трохи прискорює роботу алгоритму у порівнянні з багатопотоковістю на центральному процесорі, що збігається з дослідженням у [10]. Виконання на GPU не завжди може дати великий приріст продуктивності, але додає складності коду.

В результаті перевірки до 1 000 000 000 було виявлено лише 241 число, яке потребує 5 доданків, і число 343 867 є останнім з таких виняткових чисел, що відповідає припущенню в дослідженні [1]. Використанням пам'яті майже однакове при перевірці великих чисел та менше. Надалі можна протестувати алгоритм використовуючи більш продуктивний процесор або Nvidia CUDA, для перевірки алгоритму на більших числах.

Список літератури

1. Salzer H., Levine N. Table of Integers Not Exceeding 100000 That Are Not Expressible as the Sum of Four Tetrahedral Numbers // *Mathematics of Computation*. 1958. T. 12. C. 141–144. DOI: 10.1090/S0025-5718-1958-0099756-3 (дата звернення: 02.04.2025).
2. Dhalla H. A performance analysis of native JSON parsers in Java, Python, MS.NET Core, JavaScript, and PHP // *2020 16th International Conference on Network and Service Management (CNSM)*, 2–6 листопада 2020 р., Ізмір, Туреччина. С. 1–5. DOI: 10.23919/CNSM50824.2020.9269101 (дата звернення: 04.04.2025).
3. Lei K., Ma Y., Tan Z. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js // *2014 IEEE 17th International Conference on Computational Science and Engineering*, 19–21 грудня 2014 р., Ченду, Китай. DOI: 10.1109/CSE.2014.142 (дата звернення: 08.04.2025).
4. Huang Q., Lu N. Optimized Real-Time MUSIC Algorithm With CPU-GPU Architecture // *IEEE Access*. 2021. T. 9. C. 54067–54077. DOI: 10.1109/ACCESS.2021.3070980 (дата звернення: 08.04.2025).
5. Sloane N. J. A. Послідовність A000797 [Електрон. ресурс] // *The On-Line Encyclopedia of Integer Sequences (OEIS)*. 2024. Режим доступу: https://oeis.org/A000797 (дата звернення: 06.04.2025).
6. Zinnia N., Hanada E. Optimizing Search Strategies: A Study of Two-Pointer Linear Search Implementation // *arXiv preprint*. 2024. June. DOI: 10.48550/arXiv.2406.16729 (дата звернення: 08.04.2025).
7. Performing Calculations on a GPU [Електрон. ресурс] // *Apple Developer Documentation*. Режим доступу: https://developer.apple.com/documentation/metal/performing-calculations-on-a-gpu (дата звернення: 10.04.2025).
8. The Computer Language 25.03 Benchmarks Game [Електрон. ресурс]. Режим доступу: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html (дата звернення: 10.04.2025).

9. Bun — A fast all-in-one JavaScript runtime [Електрон. ресурс]. Режим доступу: https://bun.sh/ (дата звернення: 10.04.2025).
10. Gebraad L., Andreas F. Seamless GPU acceleration for C++ based physics with the Metal Shading Language on Apple's M series unified chips // *Seismological Research Letters*. 2023. T. 94. DOI: 10.1785/0220220241 (дата звернення: 10.04.2025).

References

1. Salzer, H., & Levine, N. (1958). Table of integers not exceeding 100000 that are not expressible as the sum of four tetrahedral numbers. *Mathematics of Computation, 12*, 141–144. https://doi.org/10.1090/S0025-5718-1958-0099756-3.
2. Dhalla, H. (2020). A performance analysis of native JSON parsers in Java, Python, MS.NET Core, JavaScript, and PHP. In *Proceedings of the 16th International Conference on Network and Service Management (CNSM)*, 2–6 November 2020, Izmir, Turkey (pp. 1–5). https://doi.org/10.23919/CNSM50824.2020.9269101.
3. Lei, K., Ma, Y., & Tan, Z. (2014). Performance comparison and evaluation of web development technologies in PHP, Python, and Node.js. In *Proceedings of the 17th IEEE International Conference on Computational Science and Engineering*, 19–21 December 2014, Chengdu, China. https://doi.org/10.1109/CSE.2014.142.
4. Huang, Q., & Lu, N. (2021). Optimized real-time MUSIC algorithm with CPU-GPU architecture. *IEEE Access, 9*, 54067–54077. https://doi.org/10.1109/ACCESS.2021.3070980.
5. Sloane, N. J. A. (2024). Sequence A000797 [Electronic resource]. *The On-Line Encyclopedia of Integer Sequences (OEIS)*. Retrieved April 6, 2025, from https://oeis.org/A000797.
6. Zinnia, N., & Hanada, E. (2024). Optimizing search strategies: A study of two-pointer linear search implementation. *arXiv preprint*. June. https://doi.org/10.48550/arXiv.2406.16729.
7. Performing Calculations on a GPU [Electronic resource]. *Apple Developer Documentation*. Retrieved April 10, 2025, from https://developer.apple.com/documentation/metal/performing-calculations-on-a-gpu.
8. The Computer Language 25.03 Benchmarks Game [Electronic resource]. Retrieved April 10, 2025, from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.
9. Bun — A fast all-in-one JavaScript runtime [Electronic resource]. Retrieved April 10, 2025, from https://bun.sh/.
10. Gebraad, L., & Andreas, F. (2023). Seamless GPU acceleration for C++ based physics with the Metal Shading Language on Apple's M series unified chips. *Seismological Research Letters, 94*. https://doi.org/10.1785/0220220241.

Maksym Nedoshev, **Viktor Kyrychenko**, Assoc. Prof., PhD phys. and math. sci.
National University of Life and Environmental Sciences of Ukraine, Kyiv, Ukraine

Peculiarities of the Operation of Complex Computational Algorithms using the Example of Testing the Polock Hypothesis

The aim of this article is to investigate and experimentally verify Pollock's third hypothesis, which states that any natural number can be represented as the sum of no more than five tetrahedral numbers. The main focus is on developing an efficient algorithm capable of processing large numbers to test the hypothesis on a numerical range up to one billion.

Within the research, an algorithm was designed and implemented that combines several modern optimization techniques: two-pointer search, approximation methods, multithreading, and memorization testing. Particular attention was paid to benchmarking different implementations across platforms — NodeJS, Bun, C, Python and GPU. The test results are presented in the form of graphs for both single-threaded and multi-threaded code execution. Experimental results showed that Bun in multithreaded mode demonstrated the highest performance, even outperforming C in certain cases. This supports the developers' claims regarding the speed of Bun. A comparison between CPU and GPU implementations using Python revealed that GPU acceleration provides only a slight improvement in performance while shader language significantly increasing code complexity compared to Python code. The experimental study covered all natural numbers up to one billion to verify the hypothesis.

As a result of testing, only 241 numbers required five tetrahedral summands, with the last such exceptional number being 343,867. This finding aligns with previous research. The algorithm showed stable memory usage across both small and large numbers. Future studies may focus on utilizing CUDA or high-performance computing clusters to validate the hypothesis over even larger numerical ranges.

Pollock's hypothesis, tetrahedral numbers, algorithm optimization, multithreading, GPU computing

Одержано (Received) 15.04.2025 *Прорецензовано (Reviewed) 28.04.2025*
Прийнято до друку (Approved) 02.05.2025