МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування
імені адмірала Макарова

Д. М. САМОЙЛЕНКО

СПЕЦІАЛЬНІ РОЗДІЛИ МАТЕМАТИКИ У КІБЕРБЕЗПЕЦІ

Навчальний посібник
для індивідуальної роботи студентів

Рекомендовано Методичною радою НУК

Електронне видання
комбінованого використання на DVD-ROM

МИКОЛАЇВ • НУК • 2017

УДК 004.056.55
ББК 32.81я73
С 17

Автор Д. М. Самойленко, канд. фіз.-мат. наук, доцент
Рецензенти: В. С. Блінцов, доктор технічних наук, професор;
О. О. Гайша, кандидат технічних наук, доцент

Самойленко Д. М.
С 17 Спеціальні розділи математики у кібербезпеці : навчальний посібник для індивідуальної роботи студентів / Д. М. Самойленко. – Миколаїв : НУК, 2017. – 88 с.

Розглянуто основи вибраних розділів математики, що не входять до базового курсу вищої математики та є необхідними для професійної підготовки фахівця з кібербезпеки. У першому розділі наведено основи теорії складності алгоритмів, їх класифікацію та асимптотичні оцінки. У другому розділі розглянуто базисні поняття та співвідношення теорії чисел. У третьому – числові методи оптимізації та пошуку. Наведено приклади програмної реалізації окремих обчислювальних алгоритмів.

Призначено для студентів вищих навчальних закладів, що навчаються за напрямком підготовки "Кібербезпека".

УДК 004.056.55
ББК 32.81я73

Навчальне видання

САМОЙЛЕНКО Денис Миколайович

СПЕЦІАЛЬНІ РОЗДІЛИ МАТЕМАТИКИ У КІБЕРБЕЗПЕЦІ

Навчальний посібник
для індивідуальної роботи студентів

Редактор *С. А. Яременко*
Комп'ютерне верстання *В. В. Москаленко*
Коректор *М. О. Паненко*

Формат 60×84/16. Ум. друк. арк. 5,1. Об'єм даних 1704 кб.
Тираж 15 прим. Вид. № 2. Зам. № 154.
Видавець і виготівник Національний університет кораблебудування
імені адмірала Макарова
просп. Героїв України, 9, м. Миколаїв, 54025
E-mail : publishing@nuos.edu.ua
Свідоцтво суб'єкта видавничої справи ДК № 2506 від 25.05.2006 р.

ВСТУП

Підготовка фахівців потребує уваги до вивчення особливих методів та засобів досліджень, властивих конкретній галузі знань. Фундаментальні дисципліни знайомлять із загальнонауковими підходами, придатними до використання у довільній області діяльності. Проте, спеціалізація навчального процесу вимагатиме від майбутнього фахівця володіння особливим арсеналом наукових інструментів, притаманних лише обмеженому колу дослідників.

Для спеціаліста з інформаційної та кібербезпеки такими інструментами є програмні реалізації математичних методів. Вимоги сучасності свідчать про те, що для питань безпеки недостатньо просто вивести формулу чи сформулювати правило, необхідно також скласти робочий і, бажано, оптимізований алгоритм та комп'ютерну програму, яка його реалізує.

При комп'ютерній реалізації математичних висновків виникає ряд особливостей, спричинених дискретністю та обмеженістю обчислювальних примітивів. Це вимагає корегування вихідних задач щодо бажаної точності чи кількості обчислених точок. З іншого боку, сучасна обчислювальна техніка надає розширені можливості, які не є надто популярними у класичній математиці, – рекурсивні обчислення, паралельні алгоритми, логічні операції (які є швидшими за

арифметичні). Ці можливості також вимагають корегування вихідних задач, але результатом таких корегувань стане суттєве покращення швидкодії.

У даному посібнику розглянуто три спеціальні розділи математики, необхідні для фахівця з інформаційної та кібербезпеки. Перший присвячений загальним способам порівняння алгоритмів та їх класифікації за складністю. У другому розділі подаються основи теорії чисел, необхідні для опанування подальших курсів криптографії та криптоаналізу. Останній розділ розкриває методи багатовимірної оптимізації та пошуку, притаманні системам зі штучним інтелектом.

1. АЛГОРИТМИ ТА ЇХ СКЛАДНІСТЬ

У першій половині XX сторіччя американець Еміль Леон Пост та англієць Алан Матісон Тюрінг незалежно один від одного запропонували абстрактні обчислювальні машини, для яких згодом була доведена алгоритмічна універсальність, тобто можливість реалізації довільного алгоритму засобами запропонованих машин. Звичайно, якщо такий алгоритм взагалі може бути реалізований.

З того часу почала бурхливий розвиток теорія складності алгоритмів, метою якої вважається вивчення алгоритмів як деяких формалізованих об'єктів, пошуку засобів вимірювання та порівняння алгоритмів.

Однією з головних характеристик алгоритму було запропоновано використовувати його трудомісткість – кількість елементарних операцій, що їх виконує абстрактний обчислювач протягом оброблення алгоритму.

Залежність трудомісткості алгоритму від різних його параметрів було названо функцією трудомісткості. Ця функція дозволила аналізувати зростання трудомісткості при ускладненні поставленої задачі – проводити її асимптотичний аналіз. За видом функції трудомісткості алгоритми почали класифікувати, виділили класи P, NP, NPC. Були поставлені задачі, які і до сьогодні не мають ні доведення, ні спростування.

Еволюція тривалістю у півстоліття призвела до реальності, в якій абстрактним машинам була залишена роль суто наукових утворень, що оглядово вивчаються у розділах інформатики і детально – в окремих розділах теорії множин. Сучасний науковець, дослідник чи просто розробник алгоритмів не аналізує належність алгоритмів до класів, не розбирає їх на елементарні операції деякої абстрактної машини. Він має ПК, який, найімовірніше, працює під керуванням багатозадачної операційної системи, що зовсім не призначена для оптимізації роботи дослідника чи науковця.

Під час дослідження задач, побудови їх рішень, як правило, створюється багато різних методів, засобів, програм – алгоритмів розв'язку. У деякий момент постає додаткова проблема: обрати з розроблених алгоритмів найкращий.

Зрозуміло, що для існування можливості вибору слід розробити кількісні характеристики алгоритмів, за якими їх взагалі можна порівнювати. Частіше за все, мова йде не стільки про алгоритми, скільки про їх реалізації, оскільки для різних *виконавців* одного алгоритму можливі принципово різні підходи.

Хоча алгоритми можна порівнювати за багатьма критеріями, найбільш прагматичним критерієм буде їх складність як деяка міра витрат для завершення виконання алгоритму. За видом вказаних витрат розрізняють різні типи складності, зокрема:

• **складність опису** (складність подання) – певна міра довжини інструкцій виконавцю для завершення алгоритму. Частіше за все, мова йде про довжину програми, виражену в текстовому еквіваленті;

• **складність функціонування** – міра кількості функціональних інструкцій, що виконуються до завершення алгоритму. Найбільш поширеним різновидом є складність обчис-

лення, яка визначає кількість математичних операцій. До цього виду складності відносять також час роботи виконавця;

• **конструктивна складність** – міра складності побудови виконавця. Характеризує алгоритми, для яких проводиться конструювання або зміна наявної конструкції виконавця для даного алгоритму. У рамках даної складності можна говорити про створення нових мов програмування (або зміну наявних) для впровадження нових інструкцій або, наприклад, зменшення складності опису;

• **енергетична складність** – міра кількості енергії для виконання алгоритму. Характеризує як різні алгоритми, так і різних виконавців;

• **екологічна складність** – міра шкідливого впливу на природне середовище протягом виконання алгоритму. Зокрема, при інтенсивній роботі центрального процесора персонального комп'ютера відбувається додаткове нагрівання і, як результат, випаровування шкідливих хімічних речовин (таких, як лак плати, ізоляція дротів, термопаста радіатора тощо).

Перелік наведених характеристик можна продовжувати, заглиблюючись у певні галузі, в яких переважне значення може надаватись вузькоспеціалізованим параметрам. Проте, і наведених характеристик цілком достатньо для того, щоб зрозуміти: немає єдиного критерію для порівняння алгоритмів. Один алгоритм може бути кращим з точки зору інформатики та математики (менша складність опису та функціонування), проте поступатись з фізичної (енергетичної) чи екологічної точки зору. Який з алгоритмів обрати – вирішувати особі, що приймає рішення, з урахуванням важливості вказаних (і не тільки) критеріїв.

Зауваживши наявність різних критеріїв порівняння, зосередимось на критеріях із галузі математики та інформатики. Виділення саме цих критеріїв пов'язане з тим, що в рамках поставлених задач розробляються порівняльні оцінки

не стільки самих алгоритмів, скільки комп'ютерних (чи процесорних) програм, що їх реалізують. Тобто, частіше за все, мова йде про однакових (принаймні подібних) виконавців, для яких складність програми визначає рівносильно й енергетичну, й екологічну, й інші складності. Іншими словами, алгоритм, що має більшу складність функціонування, напевне, матиме більшу енергетичну та інші складності. Відповідно, різні види складності набувають певної пропорційності, що дозволяє виділити для розгляду лише один з критеріїв. З точки зору математики, зручнішим є критерій складності функціонування.

Визначальною характеристикою складності функціонування алгоритму є **трудомісткість алгоритму** F_A – кількість "елементарних операцій", що виконується протягом усієї роботи програми, яка реалізує даний алгоритм.

У класичній теорії алгоритмів під головною характеристикою алгоритму – його трудомісткістю – розуміється повна кількість операцій, що виконується алгоритмом для досягнення поставленої задачі. Відповідно, для узгодження теоретичного та практичного визначень трудомісткості вимагається визначення змісту "елементарної операції".

Мало в кого повинні виникати сумніви, що для різних алгоритмічних виконавців набори елементарних операцій відрізняються. Особливо, коли мова йде про програмування засобів керування вимірювальними пристроями (кіберфізичних систем), роботів, персональних мобільних пристроїв тощо. За необхідністю порівняння алгоритмів для різних виконавців на провідну позицію виходить більш практична характеристика алгоритму – час виконання.

Але вимірюючи час виконання алгоритму, потрібно враховувати особливості роботи даного виконавця алгоритмічних дій. Зокрема, якщо говорити про персональні комп'ютери, слід обов'язково зазначати операційне середовище та його вплив на роботу програм.

З переходом від абстрактних машин до реальних, сучасних ПК, програми для яких складаються (у переважній більшості випадків) крос-платформними мовами програмування високого рівня, поняття елементарних операцій переходить від архітектури комп'ютера до семантики мов програмування.

У ролі таких "елементарних" операцій сучасних, найбільш розповсюджених мов програмування (принаймні третьої генерації) типовими навчальними посібниками та монографіями з програмування пропонується використовувати наступні:

- просте (руйнівне) присвоювання;
- одновимірна індексація;
- арифметичні операції, причому як цілочисельні, так і з плаваючою точкою;
- операції порівняння;
- логічні операції.

Спираючись на імперативні ідеї структурного програмування, команду безумовного переходу виключають з елементарних і вважають складовою частиною конструкції розгалуження, циклів довільного типу, операторів складного вибору. Тим не менше, команда безумовного переходу має трудомісткість, пов'язану із завантаженням у реєстри процесора адреси переходу та передаванням керування за зазначеною адресою.

1.1. Класифікація алгоритмів за трудомісткістю

Порівняння двох алгоритмів виключно за їх функціями трудомісткості вносить деякі помилки в отримані результати. Основною причиною цих помилок є різна частота появи елементарних операцій, породжувана різними алгоритмами, і відмінність у часі виконання елементарних операцій на реальному процесорі. Таким чином, задача переходу від функції трудомісткості до оцінки часу роботи алгоритму на конкретному процесорі не є елементарно простою і вимагає додаткових досліджень.

Алгоритми, в переважній більшості випадків, складаються не для однієї задачі, а для класу однакових (подібних) задач. Для визначення залежності трудомісткості алгоритму від параметрів задач, що можуть бути ним розв'язані, вводять **функцію трудомісткості $F_A(X)$**, яка визначає кількість елементарних операцій для вирішення конкретної задачі з сукупністю параметрів X.

Позначимо як Ψ множину задач, що її розв'язує даний алгоритм (чи його програмна реалізація). Нехай $\psi \in \Psi$ конкретна задача з даної множини. Розмірність входу даної задачі позначимо як $N = |\psi|$. Розмірність входу характеризує кількість інформації, що передається в алгоритм (у програму) в рамках конкретної поставленої задачі.

Наприклад, під множиною Ψ можна розуміти сукупність задач щодо обчислення значення полінома. Конкретною задачею ψ виступатиме задача обчислення значення конкретного полінома $2x^2 + x - 1$. Розмірністю входу задачі може виступати величина максимального степеня полінома, тобто $N = |\psi| = 2$.

Слід зауважити, що розмірність входу не є точною мірою вхідної інформації. У наведеному прикладі N, хоч і не безпосередньо, проте надає відомості про кількість доданків у поліномі. Якщо ж уявити, що множина Ψ об'єднує задачі обчислення визначника (детермінанта) матриці, то розмірність входу, очевидно, буде визначати розмірність матриці, хоча кількість елементів у ній визначається квадратом розмірності. Якщо ж алгоритм обробляє сукупність файлів, розмірність входу є кількістю файлів незалежно від їх розмірів. Отже, підсумовуючи, зазначимо, що розмірність не є розміром чи його похідною величиною, а лише виступає певною характеристикою кількості вхідних даних.

Підмножину задач з однаковою розмірністю входу позначимо як Ψ_N, тобто $\Psi_N = \{ \psi \in \Psi, |\psi| = N\}$. Оскільки N лише опосередковано характеризує вхідні дані, трудомісткість алгоритмів для задач з множини Ψ_N може відрізнятись. При цьому прийнято виділяти:

• найгірший випадок $F_A^{\uparrow}(N) = \max\limits_{\psi \in \Psi_N} \{F_A(\psi)\}$ – максимальне значення трудомісткості серед усіх задач з однаковою розмірністю входу;

• найкращий випадок $F_A^{\downarrow}(N) = \min\limits_{\psi \in \Psi_N} \{F_A(\psi)\}$ – мінімальне значення трудомісткості;

• середнє значення $\overline{F}_A(N) = \frac{1}{M_N} \cdot \sum\limits_{\psi \in \Psi_N} F_A(\psi)$, де M_N – кіль-кість задач у множині Ψ_N.

За характером та поведінкою функції трудомісткості алгоритми підлягають наступній класифікації:

• **кількісно-залежні** алгоритми. Для цих алгоритмів функція трудомісткості залежить лише від розмірності входу задачі $F_A(\psi) = F_A(N)$. Прикладом може бути алгоритм множення матриць, трудомісткість якого не залежить від значення елементів, від порядку передавання матриць тощо. У випадку, коли функція трудомісткості є поліномом N, алгоритм називають **поліномно-трудомістким**;

• **параметрично-залежні** алгоритми. Для таких алгоритмів функція трудомісткості залежить лише від значень вхідних даних $F_A(\psi) = F_A(d_1, d_2 \ldots d_N)$. Прикладом може бути програма обчислення числа π з точністю, що є вхідним параметром, або алгоритм перевірки введеного числа на наявність дільників;

• **кількісно-параметричні** алгоритми. Для них функція трудомісткості залежить як від розмірності входу, так і від значень вхідних даних. До цієї групи відноситься більшість алгоритмів чисельних методів, у яких задаються як різні розмірності, так і різні точності кінцевого результату;

• **порядково-залежні** алгоритми. Для алгоритмів даної групи трудомісткість залежить від порядку, в якому передаються вхідні дані. Типовим прикладом є програма, що впорядковує передані числа. Вона виконує мінімум дій, якщо дані передавати у вже впорядкованій послідовності.

Слід зазначити, що розрахунок функції трудомісткості є достатньо складною задачею. Як правило, сучасна програма складається мовами програмування достатньо високого рівня з використанням операцій, що вимагають великої кількості операцій самого процесора (які є елементарними для виконавця-комп'ютера). Додатково ускладнюють задачу побудови функції використання у програмах розбалансованих умовних операторів (з різною трудомісткістю блоків "then" та "else"), рекурсій та засобів програмування неімперативного типу (логічні або об'єктно-орієнтовані конструкції). В такому разі для складання $F_A(N)$ слід детально знати "склад" операцій високого рівня і методику врахування неімперативних конструкцій.

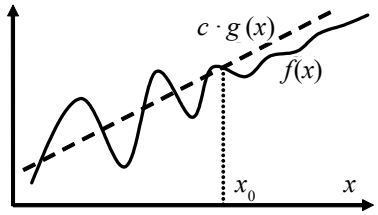
З іншого боку, позитивним є той факт, що точні відомості про функцію трудомісткості не є необхідними. Значення функції трудомісткості корисні, але малоінформативні. Як правило, достатньо знати функцію приблизно, з похибкою 5–10 %. Причому, чим менше значення функції, тим більша похибка допускається, оскільки алгоритми з малою трудомісткістю виконуються практично миттєво і різниця навіть удвічі (1 мкс чи 2 мкс) досить рідко є визначальною на фоні інших складових частин системи.

1.2. Асимптотичні оцінки

У випадку, коли допускається певна похибка встановлення трудомісткості, задача аналізу алгоритмів значно спрощується. Вимагається визначити не стільки саму функцію, скільки її оцінку, яка тим краще описує $F_A(N)$, чим більше її числове значення. Такі наближення носять назву **асимптотичні оцінки**. Вони розробляються і використовуються в рамках математичного аналізу для дослідження швидкості зростання функцій, їх порівняння тощо.

Розглянемо найбільш поширені асимптотичні оцінки.

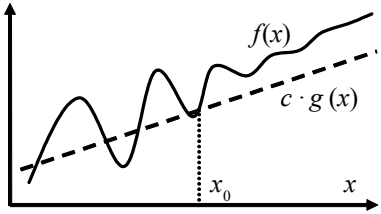
Мажор-оцінка O (оцінка згори, "O" велике)



$$f(x) = \mathrm{O}(g(x)):$$
$$\exists c, x_0 : \forall x > x_0 \qquad (1.1)$$
$$0 \le f(x) \le c \cdot g(x)$$

Говорять, що $g(x)$ мажорує функцію $f(x)$, якщо можна підібрати константу c таку, що праворуч від деякої точки x_0 всі точки функції $f(x)$ лежатимуть нижче, ніж точки $c \cdot g(x)$.

Мінор-оцінка Ω (оцінка знизу)



$$f(x) = \Omega(g(x)):$$
$$\exists c, x_0 : \forall x > x_0 \qquad (1.2)$$
$$0 \le c \cdot g(x) \le f(x)$$

Оцінка знизу протилежна до оцінки згори і свідчить про можливість підбору константи c, щоб праворуч від деякої точки x_0 всі точки функції $f(x)$ лежали вище, ніж точки $c \cdot g(x)$. У такому випадку говорять, що функція $g(x)$ мінорує функцію $f(x)$.

Оцінка o ("o" маленьке)

$$f(x) = \mathrm{o}(g(x)): \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{1.3}$$

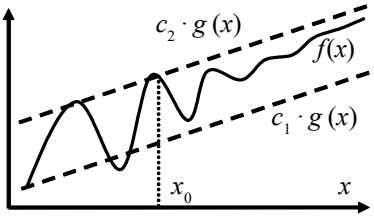
Оцінка "o" маленьке вводиться для функцій, якщо ліміт їх відношення дорівнює нулю. З практичної точки зору це значить, що функція $g(x)$ зростає значно швидше, ніж $f(x)$.

Оцінка ~ (тильда)

$$f(x) = \sim (g(x)): \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 1 \tag{1.4}$$

Оцінка тильда вводиться для функцій, якщо ліміт їх відношення дорівнює одиниці, тобто функція $f(x)$ та функція $g(x)$ збігаються тим краще, чим більше їх аргумент x.

Асимптотично точна оцінка θ (оцінка тета)



$$f(x) = \theta(g(x)): \quad \exists c_1, c_2, x_0 : \forall x > x_0 \quad c_1 \cdot g(x) \le f(x) \le c_2 \cdot g(x) \tag{1.5}$$

Асимптотично точна оцінка свідчить про можливість підбору такої функції $g(x)$, щоб праворуч від деякої точки x_0 функція $f(x)$ проходила між лініями $c_1 \cdot g(x)$ та $c_2 \cdot g(x)$.

Мажор і мінор оцінки (1.1), (1.2) є "однобічними" і використовуються для підтвердження чи спростування можливості виконання алгоритму за певний час. Якщо, наприклад, певна функція $g(x)$ визначає трудомісткість, що вкладається у заданий час і, водночас, $g(x)$ мажорує $f(x)$, то і $f(x)$ вкладеться у заданий час. Аналогічно можна спростовувати подібні можливості за допомогою мінор-оцінки.

Оцінка "о" маленьке (1.3) має скоріше математичний аспект і використовується для підтвердження можливості нехтування певними доданками у складних виразах. Також зазначена оцінка може використовуватись для підтвердження винайдення алгоритму нового типу, який не просто оптимальніший, а й має зовсім іншу асимптотичну оцінку. Дана оцінка задовольняє більш широкі вимоги мінор-оцінки, тобто при вірності оцінки "о" маленьке вірною є і мінор-оцінка.

Оцінка тильда (1.4) є найкращою для використання у задачах трудомісткості алгоритмів, але лише за умови, що існує ліміт відношення функцій. Якщо для певного алгоритму найкраща, найгірша оцінки та середня трудомісткість знаходяться у певній пропорційності, то, по-перше, функція трудомісткості (аналог $f(x)$ у формулі) не має чітко визначеного ліміту та, по-друге, умова оцінки тильда виконуватиметься лише для однієї послідовності, наприклад, для середньої, а для найкращого та найгіршого випадку ліміт відрізнятиметься від одиниці. На практиці для таких задач оцінка тильда не вводиться взагалі.

Асимптотично точна оцінка (1.5) не вимагає існування єдиного ліміту функції трудомісткості та об'єднує у собі оцінки знизу і згори. Якщо $f(x)$ має ліміт, то оцінка тета практично співпадає з оцінкою тильда. Якщо не має – виступає її практичним аналогом.

Отже, для аналізу швидкості зростання функції трудомісткості алгоритмів рекомендується використовувати оцінку тильда й асимптотично точну оцінку. Певною мірою ці оцінки можуть замінювати функцію трудомісткості з міркувань, наведених вище.

Слід зауважити, що для функції може бути зазначено одночасно декілька оцінок. Наприклад, для функції $f(x) = 4x^2 + x \cdot \ln x + 100$ справедливими є наступні оцінки:

• $f(x) = O(5x^2)$ праворуч від точки $x_0 \approx 11{,}285$ скрізь вірною є нерівність $5x^2 > f(x)$;

• $f(x) = \Omega(3x^2)$ очевидно, що для позитивних значень x скрізь виконується нерівність $3x^2 < f(x)$;

• $f(x) = o(x^2\sqrt{x})$, оскільки $\lim\limits_{x\to\infty} \dfrac{f(x)}{x^2\sqrt{x}} = \lim\limits_{x\to\infty} \dfrac{4}{\sqrt{x}} = 0$;

• $f(x) = \sim (4x^2)$ неважко переконатись, що

$$\lim_{x\to\infty} \frac{f(x)}{4x^2} = \lim_{x\to\infty} \frac{4x^2}{4x^2} = 1;$$

• $f(x) = \theta(x^2)$, що випливає з першого та другого пунктів ($c_1 = 3$, $c_2 = 5$).

Вибір тієї чи іншої оцінки як основної визначається умовами конкретної задачі.

1.3. Часові оцінки

У задачах інформаційної безпеки, поруч з функцією трудомісткості, високу актуальність мають оцінки часу виконання алгоритмів – часові оцінки. По-перше, інформаційне забезпечення досить часто передбачає кінцевим споживачем людину. Відповідно, психофізичні обмеження, що стосуються людини та її органів відчуття, можуть виступати критеріями для часових

оцінок. Наприклад, несприйняття слухом людини звукових частот понад 20 кГц (період – 50 мкс) виступає обмеженням для використання криптографічних алгоритмів у телефонії, вони мають спрацьовувати за час, що не перевищує 50 мкс. По-друге, інформація має певний час життя, який також виступає безпосереднім обмеженням часу роботи алгоритмів.

На шляху побудови часової оцінки як для складних алгоритмів, так і для окремих елементарних операцій ми стикаємося з сукупністю різноманітних особливостей, що впливають на перехід від трудомісткості до часу роботи. Зазначимо основні з них:

• неадекватність формальної системи запису алгоритму (операцій) і реальної системи команд процесора;

• наявність архітектурних особливостей, які суттєво впливають на спостережуваний час виконання операцій, таких як конвеєр, кешування пам'яті, передвибірка команд і даних і т. д.;

• різний час виконання однакових машинних команд для різних ПК;

• відмінність у часі виконання однієї команди, в залежності від значень операндів;

• різний час реального виконання однорідних команд у залежності від типів даних;

• неоднозначності компіляції вихідного тексту, обумовлені як версією компілятора, так і його налаштуваннями.

Спроби різного підходу до врахування цих факторів призвели до появи різноманітних методів переходу від функції трудомісткості до часових оцінок.

Операційний аналіз

Ідея операційного аналізу полягає в отриманні функції трудомісткості для кожної із використаних у алгоритмі елементарних операцій з урахуванням типів даних, що обробляються. Наступним кроком є експериментальне визначення середнього часу виконання конкретної елементарної операції

на конкретній обчислювальній машині. Очікуваний час виконання T розраховується як сума добутків операційної трудомісткості F_i на середнє значення часу виконання операцій t_i:

$$T = \sum_i F_i t_i. \tag{1.6}$$

Відносну складність у даному методі може становити сам процес встановлення середнього часу операції (1.6), оскільки прямі вимірювання вимагають граничної точності обчислювальної техніки, а конструкції на зразок циклів додають власної трудомісткості та впливають на результати вимірювань.

Метод Гіббсона

Метод припускає проведення сукупного аналізу трудомісткості різних алгоритмів і перехід до часових оцінок на основі належності поставленої задачі до одного із відомих типів. Серед таких типів, як правило, виділять наступні:

• задачі науково-технічного характеру з переважанням операцій з операндами дійсного типу;

• задачі дискретної математики з переважанням операцій з операндами цілого типу;

• задачі баз даних та пошуку з переважанням операцій з операндами рядкового типу.

Далі на основі аналізу великої кількості реальних програм для розв'язання відповідних типів задач визначається частотний розподіл операцій, створюються відповідні тестові програми і визначається середній час на операцію в даному типі задач t_t.

На основі отриманої інформації про середній час однієї операції для всіх задач, що відносяться до одного з перелічених типів, розраховується загальний час T роботи відповідного алгоритму за відомою трудомісткістю F за формулою:

$$T = F \cdot t_t. \tag{1.7}$$

Метод добре застосовується для "чистих" задач, які напевно відносяться до того чи іншого типу. Для змішаних задач доводиться або розширювати базу даних типів, або проводити безпосереднє дослідження частоти зустрічі відповідних команд з операндами різного типу, виконувати розрахунок усередненого часу, що припадає на одну операцію, і використовувати вираз (1.7).

Метод прямого визначення середнього часу

Цей метод також передбачає проведення сукупного аналізу алгоритмічної складності – визначається функція трудомісткості в залежності від розмірності даних на вході алгоритму $F_A(N)$, після чого на основі прямого експерименту для різних заданих значень N визначається середній час роботи даної програми T і на основі відомих функцій трудомісткості розраховується середній час узагальненої елементарної операції, породжений даним алгоритмом, компілятором і комп'ютером $\bar{t}$.

Далі висловлюється припущення про інваріантність середнього часу однієї операції при змінах розміру входу N. Дані, отримані для заданих значень розмірності входу, інтерполюють або екстраполюють на інші значення розмірності задач

$$T(N) = F(N) \cdot \bar{t}. \tag{1.8}$$

Метод (1.8) є досить дієвим, якщо відомі програмні та апаратні умови функціонування досліджуваних алгоритмів. Саме для цих умов проводяться конкретні вимірювання часу, що значно підвищує довіру до зроблених висновків.

Приклад операційного аналізу часу

У ряді випадків саме операційний аналіз дозволяє виявити тонкі аспекти раціонального застосування того або іншого алгоритму розв'язання задачі. Як приклад розглянемо

дві реалізації різних алгоритмів розв'язання задачі множення двох комплексних чисел:

$$\begin{aligned}&(a+i\cdot b)\cdot(c+i\cdot d)=e+i\cdot f;\\&e=a\cdot c-b\cdot d;\\&f=a\cdot d+b\cdot c.\end{aligned}\qquad(1.9)$$

Розглянемо два алгоритми, що дозволяють одержати наведені вище результати комплексного множення. Алгоритм A_1 передбачає пряме обчислювання e та f за виразами (1.9):

$$\begin{aligned}&e\leftarrow a\cdot c-b\cdot d;\\&f\leftarrow a\cdot d+b\cdot c.\end{aligned}\qquad(1.10)$$

Як нескладно помітити, в алгоритмі використовуються $f_{1\cdot}=4$ операції множення, $f_{1\pm}=2$ операції додавання/віднімання та $f_{1\leftarrow}=2$ операції присвоювання. Всього в алгоритмі A_1 (1.10) нараховується $f_1=8$ операцій.

Як альтернативний розглянемо алгоритм A_2, який передбачає використання проміжних обчислень:

$$\begin{aligned}&z_1\leftarrow c\cdot(a+b);\\&z_2\leftarrow b\cdot(d+c);\\&z_3\leftarrow a\cdot(d-c);\\&e\leftarrow z_1-z_2;\\&f\leftarrow z_1+z_3.\end{aligned}\qquad(1.11)$$

Протягом роботи алгоритму A_2 (1.11) виконується $f_{2\cdot}=3$ операції множення, $f_{2\pm}=5$ операцій додавання чи віднімання та $f_{2\leftarrow}=5$ операцій присвоювання. Всього в алгоритмі A_2 нараховується $f_2=13$ операцій.

За сукупною кількістю елементарних операцій алгоритм A_2 поступається алгоритму A_1 майже у півтора рази, проте в більшості сучасних комп'ютерів операція множення виконується протягом більшого проміжку часу, ніж операції додавання та присвоювання. Засобами операційного аналізу можна дати відповідь на питання: за яких умов алгоритм A_2 буде виконуватися за менший проміжок часу, ніж алгоритм A_1?

Нехай операція додавання/віднімання виконується протягом часу $t_{\pm}$. Введемо параметри q і r, які визначатимуть співвідношення між часом виконання операції множення, додавання і присвоювання для операндів дійсного типу:

$$q = \frac{t_{\bullet}}{t_{\pm}}; \qquad r = \frac{t_{\leftarrow}}{t_{\pm}}. \tag{1.12}$$

З використанням введених параметрів (1.12) ми можемо привести часові оцінки двох алгоритмів, виразивши їх через час виконання операцій додавання і віднімання:

$$T(A_1) = t_{\pm}(4q + 2r + 2); \qquad T(A_2) = t_{\pm}(3q + 5r + 5). \tag{1.13}$$

Неважко переконатись, що рівність часу виконання алгоритмів A_1 та A_2 (1.13) буде досягнута за умови $q = 3 + 3r$. При перевищенні цього значення

$$q > 3 + 3r \tag{1.14}$$

алгоритм A_2 буде працювати більш ефективно з точки зору часу виконання, ніж алгоритм A_1.

Таким чином, якщо середовище реалізації алгоритмів A_1 і A_2 (мова програмування, обслуговуючий її компілятор і комп'ютер, на якому реалізується задача) створює ситуацію, за якою час виконання операції множення двох дійсних чисел більше ніж утричі перевищує час додавання двох дійсних чисел (1.14), то для реалізації множення комплексних чисел кращим стає алгоритм A_2. Слід зауважити, що зазначене співвідношення є цілком реальним для більшості типових обчислювачів.

Звичайно, виграш у часі досить малий, якщо ми перемножуємо тільки два комплексних числа, однак, якщо цей алгоритм є частиною складної обчислювальної задачі з комплексними числами, в якій вимагається суттєва кількість операцій множення, то виграш у часі може бути більш відчутним. Оцінка такого виграшу на одне множення комплексних чисел випливає з проведеного вище аналізу:

$$\Delta t = K \cdot t_{\pm} \cdot (q - 3r - 3), \qquad (1.15)$$

де K – кількість операцій множення комплексних чисел у загальному алгоритмі. Наприклад, у задачі пошуку належності однієї точки до фрактальної множини може виконуватись близько тисячі операцій комплексного множення. Для рисування фрактального образу з відносно невеликою роздільністю 100×100 точок таких операцій буде виконано десятки мільйонів, тобто у формулу (1.15) додався множник 1 млн.

Таким чином, часові оцінки алгоритмів можуть навіть якісно не співпадати з оцінками їх трудомісткості, причому, в окремих випадках із точністю до навпаки – алгоритм з більшою операційною трудомісткістю реально виконується протягом меншого проміжку часу.

2. ЕЛЕМЕНТИ ТЕОРІЇ ЧИСЕЛ

Теорія чисел (вища арифметика) – розділ математики, який вивчає цілі числа (і подібні до них об'єкти), а також функції, пов'язані з цілими числами. У рамках даного розділу всі числа, а також вирази та функції вважаються цілими, якщо не наведено додаткових зауважень.

Якщо для двох чисел a і b може бути знайдене таке число q, при якому

$$b \cdot q = a, \tag{2.1}$$

тоді:

• говорять, що a ділиться на b або a кратне b (позначається $a \vdots b$);

• говорять, що b ділить a або b є дільником a (позначається $b \mid a$);

• число q називають часткою ділення чисел a і b.

Якщо не існує такого числа q, що задовольняє зазначену вище умову (2.1) для пари чисел a і b, тоді число a можна подати єдиним способом у вигляді

$$a = b \cdot q + r, \quad 0 < r < |b|. \tag{2.2}$$

У такому разі число q називають неповною часткою, а число r – залишком від ділення. Якщо для числа a не існує

жодної пари чисел b і q (відмінних від одиниці), для яких $r = 0$, то число a називають простим. Іншими словами, просте число не може бути поділене без залишку на будь-яке інше число.

Поняття простого числа є одним з основних у теорії чисел. Воно дозволяє сформулювати основну теорему арифметики: **будь-яке число може бути єдиним чином подане у вигляді добутку простих чисел**.

Якщо для чисел a і b існує таке число q, яке ділить одночасно a і b ($q \mid a$, $q \mid b$), то таке число q називають спільним дільником чисел a і b. Максимальне серед таких чисел, що ділять одночасно числа a і b, $g = \max\{q : q \mid a, q \mid b\}$ називають найбільшим спільним дільником (НСД, GCD) чисел a і b. У теорії чисел НСД позначають круглими дужками: $g = (a,b)$.

Оскільки число 1 ділить будь-яке інше число, воно є спільним дільником довільної пари чисел. Якщо два числа a і b не мають спільних дільників, окрім одиниці $(a,b) = 1$, то такі числа називають взаємно простими (самі числа a і b при цьому можуть не бути простими).

Поняття НСД є одним з визначальних термінів теорії чисел. Обчислення НСД здійснюється як складова частина довільного практичного алгоритму. Розглянемо основні властивості НСД $g = (a,b)$.

1. g ділиться на всі інші спільні дільники чисел a і b, іншими словами, усі спільні дільники чисел a і b є дільниками g.

2. Властивість масштабу: $(k \cdot a, k \cdot b) = |k| \cdot (a,b)$, зокрема $(k, k \cdot a) = k$; $(1,a) = 1$.

3. Якщо числа поділити на їх НСД, то у них не залишиться спільних дільників: $\left(\frac{a}{g}, \frac{b}{g}\right) = 1$. Це є одним зі способів одержання взаємно простих чисел.

4. Співвідношення Безу (Bèzout's identity): НСД може бути подано у вигляді $g = n \cdot a + m \cdot b$. Числа n і m називають коефіцієнтами Безу.

5. Якщо справедлива формула (2.2), то $(a,b) = (b,r)$.

6. Якщо $(a,b) = 1$, то $(a \cdot c, b) = (c,b)$ – якщо між a і b немає спільних дільників, то їх слід шукати лише між c і b.

Для доведення властивостей використовується формула (2.2) та визначення кратності. Для прикладу доведемо властивість 5 як одну з найбільш використовуваних на практиці.

Для доведення необхідно підтвердити, що залишок від ділення двох чисел ділиться на їх НСД. Якщо $g \mid a$ та $g \mid b$, то справедливе наступне представлення чисел: $a = \alpha g$ та $b = \beta g$. У відповідності до (2.2) $r = a - bq$. Тоді $r = g(\alpha - \beta q)$. Як видно, число g входить множником до складу числа r. Значить, число g є дільником числа r, що і вимагалось довести.

Доведена властивість складає основу алгоритму Евкліда розрахунку НСД двох чисел. Для пошуку НСД двох чисел шукають НСД між одним з чисел та залишком від ділення одного на інше. Оскільки залишок завжди менший за дільник, на кожному кроці алгоритму відбувається гарантоване зменшення чисел, тобто алгоритм є завершуваним.

Для практичної реалізації необхідно передбачити кінцеві етапи зменшення чисел. Можна використати властивості $\gcd(0,a) = a$ (нуль ділиться на будь-яке число) та $\gcd(1,a) = 1$ (див. вл. 2).

Наведемо приклад програмної реалізації алгоритму Евкліда мовою Python 3:

```
def gcd(a,b):
    '''Great common divisor (a,b)'''
    if b==0: return a
    elif b==1: return 1
    else: return gcd(b,a%b)
```

Вибір даної мови обумовлений її високою популярністю серед професійних розробників програмного забезпечення, безкоштовністю дистрибутивів та простотою їх встановлення для різних операційних систем, а також відсутністю принципових обмежень на величини цілих чисел, що можуть бути використані у розрахунках. Останнє є особливо зручним для алгебраїчних задач криптографії, у яких оперують великими числами, більшими за межу мов програмування з жорсткою типізацією даних.

Випробуємо функцію як на малих числах, так і на великих. Неважко перевірити, що gcd (5,15) = 5. Для великих чисел сформуємо два числа з відомим НСД. Візьмемо числа $2^{256} \cdot 11$ та $2^{256} \cdot 17$. Оскільки gcd (11,17) = 1, то $(2^{256} \cdot 11, 2^{256} \cdot 17) = 2^{256}$. Далі наводяться результати виконання програми із зазначеними числами. Як видно, малі та великі числа обробляються вірно:

```
>>> gcd(5,15)
5
>>> 2**256
115792089237316195423570985008687907853269984665640564039457584007913129639936
>>> gcd(2**256*11,2**256*17)
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

Для пошуку коефіцієнтів Безу використовується розширений алгоритм Евкліда. Основу алгоритму складає такий самий рекурсивний перерахунок з використанням залишку від ділення вхідних чисел. Але, оскільки здійснюється розрахунок кількох параметрів, необхідно провести уточнення алгоритму.

Головною задачею алгоритму є пошук НСД пари чисел та коефіцієнтів Безу n і m таких, що

$$g = n \cdot a + m \cdot b, \tag{2.3}$$

де $g = (a,b)$. Нехай на певному етапі було знайдено коефіцієнти n' і m' для чисел b та $r = a–bq$. Тобто

$$g = n' \cdot b + m' \cdot r = n' \cdot b + m' \cdot (a - bq). \qquad (2.4)$$

Після розкриття у формулі (2.4) дужок та зведення подібних можна записати:

$$g = m' \cdot a + (n' - m' \cdot q)b. \qquad (2.5)$$

Порівнюючи останній вираз (2.5) з початковим (2.3), визначимо формули перетворення коефіцієнтів:

$$n = m', \; m = n' - m'q, \qquad (2.6)$$

де $q = [a / b]$ – неповна частка ділення a і b.

Використовуючи одержане співвідношення (2.6), складемо програмну функцію для розширеного алгоритму Евкліда:

```
def gcd_ex(a,b):
  '''Returns list (g,n,m) n,m - Bézout's coefficients'''
   if b==0: return (a,1,0)
   g,n,m = gcd_ex(b,a%b)

   return (g,m,n-(a//b)*m)
```

В останньому рядку використано формули перерахунку та операцію цілочисельного ділення у формалізмі мови програмування (a // b). Випробування алгоритму має вигляд:

```
>>> (g,n,m)=gcd_ex(11,17)
>>> n
-3
>>> m
2
>>> 11*n+17*m
1

>>> (g,n,m)=gcd_ex(11*2**256,17*2**256-1)
>>> n
53685423191846599696382911231300757277425174708615170600112152585486996469
>>> m
-34737626771194858627071295502606372355980995399692169211837275202373938891
>>> 11*2**256*n+(17*2**256-1)*m
1
```

Детальний аналіз трудомісткості алгоритмів Евкліда [14] дозволяє одержати асимптотичну оцінку $O(\log a \cdot \log b)$. Це означає, що складність алгоритму залежить не від самих чисел, а від кількості цифр у їх записі. Такі алгоритми є значно ефективнішими за алгоритми перебору і можуть використовуватись для великих чисел, простий перебір яких неможливий за короткий проміжок часу.

2.1. Конгруентність чисел

Два числа a і b називають порівнянними (конгруентними) за модулем m, якщо їх різниця є кратною до числа m:

$$a \equiv b \pmod{m}: \quad |a-b| \vdots m. \tag{2.7}$$

Альтернативним визначенням порівнянності двох чисел за модулем m є те, що ці два числа мають однаковий залишок при діленні на число m. Обидва визначення є еквівалентними і можуть застосовуватись у тій чи іншій формі в залежності від зручності конкретної ситуації.

Множина чисел, порівняних з даним числом a за модулем m, називається класом лишків числа a за модулем m (позначається $[a]_m$, $\overline{a}_m$ або $K_a^{(m)}$):

$$\overline{a}_m = \{a \pm k \cdot m\}, k \in N. \tag{2.8}$$

Нагадаємо, що класом в алгебрі називається множина, на якій виконуються відношення рефлективності $(a \sim a, \forall a)$, симетричності $(a \sim b \Rightarrow b \sim a)$ та транзитивності $(a \sim b, b \sim c \Rightarrow a \sim c)$.

З урахуванням того, що до класів лишків можуть бути введені операції додавання та множення: $\overline{a}_m + \overline{b}_m = \overline{(a+b)}_m$, $\overline{a}_m \cdot \overline{b}_m = \overline{(a \cdot b)}_m$, то по відношенню до цих операцій розглянута множина утворює кільце лишків. Ця множина (2.8) також позначається як Z_m.

Повною системою лишків (ПСЛ) за модулем m називають довільний набір цілих чисел, попарно непорівнянних між собою за модулем m. Інколи ПСЛ визначають як множину, до якої входять лише по одному елементу з кожного класу лишків Z_m. Іншими словами, ПСЛ містить усі числа, які можуть бути лишками інших чисел при їх діленні на m.

Формальні визначення дозволяють вибір ПСЛ у більшменш довільний спосіб, але на практиці використовують один з двох варіантів таких наборів: числа від 0 до m – 1 або числа 0, ±1, ±2 … ±m/2.

Приведеною системою лишків за модулем m називають максимальний набір взаємно простих з m чисел, попарно непорівнянних за (2.7) між собою за модулем m. Відмінність від повної системи лишків полягає у вилученні з множини елементів тих, що мають спільні дільники з m. Якщо модуль m є простим числом, то вилучається лише елемент 0. В іншому разі кількість вилучених елементів буде більшою. Наприклад, для модуля m = 6 приведена система лишків міститиме лише числа 1 та 5. Усі інші числа матимуть з числом 6 спільні дільники.

Кількість елементів у приведеній системі лишків за модулем m є множиною значень функції Ейлера $\varphi(m)$. Як зазначалось у прикладі вище, для m = 6 приведена система лишків налічує два елементи, значить $\varphi(6) = 2$.

Функція Ейлера має широке використання у теорії чисел, тому розглянемо її властивості більш детально:

• якщо число m просте, то $\varphi(m) = m - 1$, оскільки з простим числом у ПСЛ спільні дільники має лише число 0;

• якщо число m є степенем простого числа $m = p^n$, то

$$\varphi\left(p^n\right) = p^n - p^{n-1} = p^n \cdot \left(1 - \frac{1}{p}\right), \qquad (2.9)$$

оскільки з числом p^n спільні дільники мають лише числа p, $2p$, $3p$, … Неважко підрахувати, що серед p^n послідовних чисел їх виявиться p^{n-1};

• якщо числа m і n взаємно прості, то $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$, оскільки числа приведеної системи лишків формуватимуться саме як добутки чисел, що не мають спільних дільників із m та n.

Розглянутих властивостей достатньо для розрахунку функції Ейлера (2.9) для довільного числа, якщо відомо, як воно розкладається на прості множники:

$$\varphi\left(p_1^{n_1} \cdot p_2^{n_2} \cdots p_k^{n_k}\right) =$$
$$= p_1^{n_1} \cdot p_2^{n_2} \cdots p_k^{n_k} \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right). \qquad (2.10)$$

Зауважимо, що задача розкладання на множники (задача факторизації) сама собою не є простою і на сьогодні не відомі оптимальні алгоритми її розв'язку з логарифмічною трудомісткістю. Відтак, і для обчислення функції Ейлера подібних до (2.10) алгоритмів не існує, якщо розклад числа невідомий.

Для приведеної системи лишків операція додавання не може бути введена. Проте, множина залишається замкненою відносно операції множення (за модулем m). У цьому неважко переконатись, розглянувши наведений приклад з $m = 6$: елементи 1 + 1 чи 1 + 5 не належать до початкової множини,

тоді як $1 \cdot 1$, $1 \cdot 5$, $5 \cdot 5 = 25 \equiv 1 \pmod 6$ є елементами приведеної системи лишків.

Приведена система лишків разом з операцією множення (за модулем) утворює мультиплікативну групу кільця лишків (позначається $Z_m^{\times}$ або F_m).

Властивості конгруентності

Якщо $a \equiv \alpha \pmod m$ та $b \equiv \beta \pmod m$, то

1. $a + b \equiv \alpha + \beta \pmod m$;
2. $a - b \equiv \alpha - \beta \pmod m$;
3. $a \cdot b \equiv \alpha \cdot \beta \pmod m$, зокрема $a^2 \equiv \alpha^2 \pmod m$, $a^n \equiv \alpha^n \pmod m$;
4. $k \cdot b \equiv k \cdot \beta \pmod m$, але обернене вірно не завжди, скорочувати порівняння можна тільки на число, взаємно просте з модулем: $(k,m) = 1$;
5. якщо $c \equiv \alpha \pmod m$, то $a \equiv c \pmod m$.

Розглянемо властивість 4 більш детально. Припустимо, що число k має з модулем m спільний множник. Нехай $k = s \cdot K$, $m = s \cdot M$, тоді порівнянність $k \cdot b \equiv k \cdot \beta \pmod m$ означає, що різниця лівої та правої частин кратна до модуля: $k \cdot (b - \beta) = x \cdot m$. Якщо скорочувати порівняння на k, то кратність модулю виразиться як рівність $(b - \beta) = y \cdot m$. Розкриємо склад чисел та перепишемо одержані рівності:

$$\begin{aligned} s \cdot K \cdot (b - \beta) &= x \cdot s \cdot M \\ (b - \beta) &= y \cdot s \cdot M \end{aligned} \qquad (2.11)$$

Як бачимо, з першої рівності (2.11) друга не виводиться, оскільки множник s у першій рівності може бути скороченим, а значить різниця $(b - \beta)$ може бути вільна від нього у своєму складі. Проте друга рівність вимагає його наявності саме у складі різниці.

Якщо ж число k не має спільних множників з модулем (який у свою чергу залишається складеним числом $m = s \cdot M$), то рівності до і після скорочення матимуть вигляд:

$$\begin{aligned} k \cdot (b - \beta) &= x \cdot s \cdot M \\ (b - \beta) &= y \cdot s \cdot M \end{aligned} \qquad (2.12)$$

Єдина можливість задоволення першої рівності (2.12) – це наявність множників як s, так і M у складі різниці $(b - \beta)$, оскільки число k не має жодного з них, іншими словами, вимагається вірність другого рівняння (2.12). Доведене означає, що ліву і праву частини порівняння можна скорочувати лише на число, взаємно просте до модуля. Простоти самого модуля при цьому не вимагається.

Властивості конгруентності дозволяють, зокрема, встановлювати ознаки ділення чисел. Наприклад, встановимо ознаку ділення на 9. Якщо число у десятковій системі числення записане цифрами ABC, то це число визначає кількість $n = A \cdot 10^2 + B \cdot 10^1 + C$. Використаємо той факт, що $10 \equiv 1 \pmod 9$. Використаємо наслідки третьої властивості конгруентності: $10^2 \equiv 1^2 = 1 \pmod 9$. Узагальнюючи одержаний висновок, зазначимо, що $10^K \equiv 1 \pmod 9$ для усіх K. Значить, $n = A \cdot 10^2 + B \times$ $\times 10^1 + C \equiv A \cdot 1 + B \cdot 1 + C \pmod 9$. Останній вираз свідчить про те, що число ABC має такий самий залишок при діленні на 9, як і сума його цифр $A + B + C$. Зокрема, якщо сума цифр числа ділиться на 9, то і саме число ділиться на 9. Наведений приклад з числом із трьох цифр легко узагальнюється на довільні числа.

2.2. Порівняння першого ступеня

Порівняння першого ступеня, або лінійні порівняння, є аналогом звичайних лінійних рівнянь, але у системі лишків за мо-

дулем m. Лінійне порівняння можна сформулювати як задачу пошуку такого числа x, для якого буде вірним порівняння

$$a \cdot x \equiv b \pmod{m}. \tag{2.13}$$

Якщо числа a та m мають спільні дільники, то порівняння (2.13) матиме розв'язок лише у тому випадку, коли ці ж дільники матиме число b. Це випливає з основної теореми арифметики, у відповідності до якої одне число не можна подати кількома способами у вигляді добутків простих множників.

Нехай $g = (a,m)$ – НСД чисел a і m. Тоді, якщо $g \mid b$, то рівняння може бути скорочене:

$$\frac{a}{g} x \equiv \frac{b}{g} \left(\mathrm{mod}\ \frac{m}{g}\right). \tag{2.14}$$

В силу властивостей НСД, числа a/g та m/g не матимуть більше спільних дільників.

Висловимо твердження: якщо числа a і m взаємно прості, то при перебігу числом x повної системи лишків за модулем m результат $a \cdot x$ також перебігає цю систему.

Довести це твердження простіше від суперечного: припустимо, що існують два числа з ПСЛ x_1 та x_2 такі, що $a - x_1 \equiv \equiv a \cdot x_2 \pmod{m}$. У відповідності до четвертої властивості конгруентності (розд. 2.1) порівняння може бути скорочене на число, взаємно просте до модуля. Це означає, що $x_1 \equiv x_2 \pmod{m}$. Останнє суперечить визначенню ПСЛ – у ній не може бути двох чисел, що є порівнянними за модулем m.

Доведене твердження свідчить про те, що порівняння (2.14) має єдиний розв'язок у ПСЛ за модулем m / g. Це еквівалентне наявності g розв'язків вихідного рівняння (2.13): числа $x,\ x + m / g, x + 2m / g \ldots x + (g - 1)m / g$ є порівнянними за модулем m / g, але різними за модулем m.

Підсумовуючи, зазначимо, що порівняння (2.13)

• за умови $(a,m) = 1$ має один розв'язок у ПСЛ за модулем m;

• за умови $(a,m) = g \neq 1$ порівняння має g розв'язків (за модулем m), порівнянних за модулем m / g, якщо права частина порівняння кратна g, інакше порівняння не має розв'язків.

Як приклад розв'яжемо декілька лінійних порівнянь:

1. $5x \equiv 7 \pmod{11}$. Перевіряємо, чи є простими числа 5 та 11 – шукаємо їх НСД: $(5,11) = 1$. Це означає, що порівняння має єдиний розв'язок (за модулем 11). Його неважко знайти ($x = 8$) та пересвідчитись у тому, що він єдиний, перебором ПСЛ числа 11.

2. $5x \equiv 7 \pmod{10}$. Визначаємо $(5,10) = 5$. Оскільки числа не взаємно прості, перевіряємо праву частину на кратність числу 5 і переконуємось, що 7 на 5 не ділиться. Це означає, що порівняння не має розв'язків, у чому також нескладно пересвідчитись перебором ПСЛ за модулем 10.

3. $4x \equiv 6 \pmod{10}$. Визначаємо $(4,10) = 2$. Перевіряємо кратність правої частини – 6 ділиться на 2. Скорочуємо порівняння за формулою (2.3) – одержуємо $2x = 3 \pmod 5$. Знаходимо його розв'язок: $x = 4 \pmod 5$. Переходимо до початкового модуля додаванням $5 : x = 4, 4 + 5 = 9$. Перевіряємо одержані розв'язки: $4 \cdot 4 = 16 \equiv 6 \pmod{10}$, $4 \cdot 9 = 36 \equiv 6 \pmod{10}$. Також можна пересвідчитись у відсутності інших розв'язків у ПСЛ за модулем 10.

Для простоти подальшого викладання матеріалу будемо вважати, що у порівнянні за формулою (2.13) числа a та m є взаємно простими. По-перше, саме такі випадки реалізуються у практичних застосуваннях теорії чисел (в основному у криптографії). По-друге, вище наведено алгоритм дій за умов невірності вихідного твердження.

Порівняння (2.13) дозволяє ввести визначення дробу за модулем m. За аналогією зі звичайною алгеброю, де розв'я-

зок рівняння, подібного до виразу (2.13), позначається дробом $x = b / a$, у теорії чисел вводиться схоже поняття дробу як розв'язку порівняння.

Дробом b / a за модулем m називається таке число x, яке задовольняє порівняння $a \cdot x \equiv b \pmod m$. У відповідності до наведеного вище прикладу дробом 7/5 (mod 11) є число 8:

$$\frac{7}{5} \equiv 8 \pmod{11}.$$

Одним з частинних випадків дробу з одиничним чисельником є обернений елемент a^{-1}. Це таке число, яке у добутку з даним числом є порівнянним з одиницею за модулем m:

$$a^{-1} \cdot a \equiv 1 \pmod m. \qquad (2.15)$$

Особливість оберненого елемента (2.15) полягає у тому, що він може бути розрахований за розширеним алгоритмом Евкліда. Оскільки $(a,m) = 1$, співвідношення Безу виглядатиме як

$$a \cdot n_1 + m \cdot n_2 = 1, \qquad (2.16)$$

де коефіцієнти Безу позначені як n_1 та n_2, оскільки символ m використано для позначення модуля обчислень. З (2.16) випливає:

$$a \cdot n_1 = 1 - m \cdot n_2 \equiv 1 \pmod m. \qquad (2.17)$$

Це означає, що коефіцієнт Безу при числі a (n_1) є оберненим до числа a елементом за модулем другого числа (m) у співвідношенні Безу.

Відповідно, для пошуку оберненого елемента за (2.17) можна адаптувати розширений алгоритм Евкліда:

```
def inv(a,mod):
    '''Inerse to 'a' element modulo mod'''
    g,n,m = gcd_ex(a,mod)
    if g!=1: return None
    return (n+mod)%mod
```

Відмінності від початкового алгоритму полягають у наявності додаткової перевірки на взаємну простоту чисел a і m (перевірка g на рівність 1) та перерахунок коефіцієнта Безу до ПСЛ за відповідним модулем. У разі наявності спільних дільників чисел a і m обернений елемент за алгоритмом Евкліда не може бути розрахований, відповідно функція повертає значення None як ознаку аномального завершення.

У разі успішного обчислення коефіцієнта n цілком можливо, що він є від'ємним. Для переведення його у позитивно визначену ПСЛ обчислюється вираз $n + m \pmod m$.

Для перевірки дієздатності функції спробуємо обчислити обернені елементи за різних ситуацій. Далі наводимо вигляд програмної консолі:

```
>>> inv(3,17)
6
>>> 6*3%17
1

>>> a1=inv(3,2**256)
>>> a1
77194726158210796949047323339125271902179989777093709359638389338608753093291
>>> a1*3%2**256
1

>>> a2=inv(4,2**256)
>>> a2==None
True
```

Як видно, обчислені елементи також перевіряються на виконання умови $a \cdot a^{-1} \equiv 1 \pmod m$, що додатково свідчить про правильність роботи функції.

Можливість обчислення оберненого елемента за допомогою алгоритму з логарифмічною складністю дозволяє розв'язувати лінійні порівняння значно швидше, ніж простий перебір. Так, якщо необхідно знайти розв'язок порівняння (2.13), то, у відповідності до властивостей конгруентності, можна помножити обидві частини порівняння на елемент, обернений до a:

$$a^{-1} \cdot a \cdot x \equiv a^{-1} \cdot b \pmod m. \qquad (2.18)$$

За властивістю оберненого елемента $a^{-1} \cdot a \equiv 1 \pmod{m}$, а значить розв'язок порівняння відразу випливає з (2.18):

$$x \equiv a^{-1} \cdot b \pmod{m}. \tag{2.19}$$

Відтак, для практичного розв'язування лінійних порівнянь використовується алгоритм пошуку оберненого елемента з наступним його множенням на праву частину порівняння (2.19). У повністю аналогічний спосіб можуть бути розраховані дробові вирази за модулем m.

2.3. Порівняння другого ступеня

Порівняння другого ступеня, або квадратичні порівняння, є аналогом квадратних рівнянь у звичайній алгебрі. Оскільки довільна квадратична форма може бути перетворена до повного квадрату з константами (ця властивість використовується при виділенні квадратичного дискримінанта і тут детально не розглядається), будемо вважати типовим порівнянням другого ступеня задачу пошуку x такого, що

$$x^2 \equiv a \pmod{m}. \tag{2.20}$$

Неважко переконатись звичайним перебором, що не будь-яке квадратичне порівняння може бути розв'язане. Для ілюстрації приведемо ПСЛ числа 11 та квадрати її елементів (табл. 1).

Таблиця 1. **Квадрати елементів ПСЛ за модулем 11**

$x \pmod{11}$	0	1	2	3	4	5
$x^2 \pmod{11}$	0	1	4	9	$16 \equiv 5$	$25 \equiv 3$
$x \pmod{11}$	6	7	8	9	10	
$x^2 \pmod{11}$	$36 \equiv 3$	$49 \equiv 5$	$64 \equiv 9$	$81 \equiv 4$	$100 \equiv 1$	

Як видно з табл. 1, квадрат числа за модулем 11 може набувати значень 0, 1, 3, 4, 5, 9 і не може – значень 2, 6, 7, 8, 10.

Число a, що може бути розв'язком квадратичного порівняння (2.20), називають квадратичним лишком за модулем m. Інші числа з ПСЛ називають квадратичними нелишками за модулем m.

Розв'язок рівняння (2.20) дозволяє ввести поняття квадратного кореня за модулем як операції, оберненої піднесенню до квадрата.

Так, наприклад, корінь із числа 5 має два значення: $\sqrt{5} \equiv 4; 7 \pmod{11}$, що відповідає порівнянням $4^2 \equiv 7^2 \equiv 5 \pmod{11}$. У той же час корінь з числа 6 не має жодного значення, оскільки немає чисел, що у квадраті порівнянні з числом 6 за модулем 11.

Для позначення квадратичної властивості чисел використовується символ Лежандра:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & a \vdots p \\ 1, \exists x : x^2 \equiv a \pmod{p} \\ -1, \text{інакше.} \end{cases} \qquad (2.21)$$

Слід відзначити, що символ Лежандра (2.21) було введено лише для модулів m, що є простими числами. Цей факт підкреслено зміною у формулі символу m на p.

Для прикладу з табл. 1 можна визначити, що символ Лежандра рівний нулю для числа 0 (оскільки нуль ділиться на довільне число), одиниці – для чисел 1, 3, 4, 5, 9 та –1 – для 2, 6, 7, 8, 10. Можна зауважити, що кількість квадратичних лишків та нелишків є однаковою.

Леонардом Ейлером у XVIII столітті було доведено властивість, відому на сьогодні як критерій Ейлера: якщо $p > 2$ – просте число, a – взаємно просте до p число, то число a є квадратичним лишком за модулем p тоді і тільки тоді, коли

$$a^{\frac{p-1}{2}} \equiv 1 \pmod{p}, \qquad (2.22а)$$

і є квадратичним нелишком за модулем p, якщо

$$a^{\frac{p-1}{2}} \equiv -1 \pmod{p}. \qquad (2.22б)$$

Критерій Ейлера навіть для невеликих чисел перевіряти досить складно, оскільки доводиться мати справу зі значними степенями чисел. Так, для модуля 11 показником степеня буде число $(11 - 1)/2 = 5$. Для окремих елементів ПСЛ – чисел 5 і 6 критерії Ейлера виглядатимуть як

$5^5 = 3125 \equiv 1 \pmod{11}$, тобто вірним є вираз (2.22, а),

$6^5 = 7776 \equiv 10 \equiv -1 \pmod{11}$, тобто вірним є (2.22, б).

Числа, що використовуються у практичному застосуванні вищої арифметики (криптографії), як правило, є досить великими для безпосереднього перебору (близько 2^{1000}). Відповідно, як основа виразу, так і показник степеня мають досить великі значення.

Очевидно, що послідовне множення основи саму на себе $(p - 1)/2$ разів не може використовуватись як практичний обчислювальний алгоритм, оскільки навіть поділене навпіл криптографічне число залишиться таким, що не дозволить виконати перебір за прийнятний час засобами сучасної обчислювальної техніки.

Для практичних потреб було розроблено алгоритми швидкого піднесення до степеня, пов'язані з каскадним обчисленням невеликого степеня від числа (квадрата чи куба). Існує певна кількість модифікацій зазначених алгоритмів, пов'язаних з різними способами аналізу показника степеня та вибору різних числових базисів.

З метою дотримання рекурсивного стилю складання обчислювальних функцій, використаного у попередніх алгоритмах, реалізуємо наступний спосіб швидкого піднесення до степеня:

$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2, & n = 2k, k \in N \\ \left(x^{\frac{n}{2}}\right)^2 \cdot x, & n = 2k+1, k \in N. \end{cases} \tag{2.23}$$

На кожному кроці алгоритму (2.24) відбувається зменшення степеня удвічі, обчислення квадрату та повторні дії зі зменшеним степенем у залежності від його парності. Додавши до алгоритму умови виходу для малих степенів: $x^0 = 1$ та $x^1 = x$, можемо скласти обчислювальну функцію. Додатковим параметром функції буде величина m, оскільки усі обчислення здійснюються за модулем:

```
def pow_mod(base,exp,mod):
    '''Evaluation 'base' powered 'exp' modulo 'mod' '''
    if exp == 0: return 1
    if exp == 1: return base
    if exp & 1:
    return (((pow_mod(base,exp//2,mod)**2)%mod)*base)%mod
   return (pow_mod(base,exp//2,mod)**2)%mod
```

Для випробування алгоритму знайдемо останні 3 цифри числа $\left(2^{256}-1\right)^{2^{128}-1}$, тобто величину цього виразу за модулем 1000:

```
>>> pow_mod(2**256-1,2**128-1,1000)
375
```

Можна переконатись, що використання калькуляторів і навіть спеціалізованих математичних пакетів далеко не завжди дозволить провести наведені обчислення.

З використанням алгоритму швидкого піднесення до степеня символ Лежандра може бути обчислений через безпосереднє використання критерію Ейлера.

Узагальненням символу Лежандра на всі непарні числа (не лише прості) став символ Якобі. Символ Якобі дорівнює добуткові символів Лежандра для всіх множників складеного числа. Для простих чисел ці символи збігаються. Позначаються символи однаково.

Символ Якобі має певні рекурентні ознаки, що дозволяють здійснювати його обчислення безпосередньо. Серед таких ознак можна виділити наступні:

1. $\left(\frac{0}{n}\right) = 0$

2. $\left(\frac{1}{n}\right) = 1, \quad \left(\frac{n}{1}\right) = 1$

3. $\left(\frac{-1}{n}\right) = (-1)^{\frac{n-1}{2}} = \begin{cases} 1, & n \equiv 1 \pmod 4 \\ -1, & n \equiv 3 \pmod 4 \end{cases}$ (2.24)

4. $\left(\frac{2}{n}\right) = (-1)^{\frac{n^2-1}{8}} = \begin{cases} 1, & n \equiv 1 \pmod 8 \vee n \equiv 7 \pmod 8 \\ -1, & n \equiv 3 \pmod 8 \vee n \equiv 5 \pmod 8 \end{cases}$

5. $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)(-1)^{\frac{(n-1)(m-1)}{4}} = \begin{cases} -\left(\frac{n}{m}\right), & n \equiv m \equiv 3 \pmod 4 \\ \left(\frac{n}{m}\right), & \text{інакше} \end{cases}$

У відповідності до наведених властивостей (2.24) може бути складений рекурсивний алгоритм обчислення символу Якобі чисел a і b. Основою алгоритму є постійне "перевертання" чисел a і b з корегуванням знаку (властивість 5) та перерахунку чисельника за модулем знаменника, тобто з постійним зменшенням чисел до величин 0–2, для яких значення встановлюється безпосередньо властивостями 1–4. Дії алгоритму виглядатимуть наступним чином:

1. Якщо $a < 0$, якщо $b \equiv 3 \pmod 4$, змінюємо знак відповіді. Перезапускаємо алгоритм з числами a, b.

2. Якщо $a = 0$ – повернути нуль.

3. Якщо $a = 1$ або $b = 1$ – повернути одиницю.

4. Якщо a парне, виділяємо степінь двійки: $a = 2^n \cdot a_2$, де a_2 – непарне число. Якщо n непарне та $b \equiv 3$ чи $b \equiv 5 \pmod 8$ – змінюємо знак відповіді. Перезапускаємо алгоритм з числами a_2, b.

5. Якщо $a \equiv b \equiv 3 \pmod 4$, змінюємо знак відповіді. Перезапускаємо алгоритм з числами r та a, де r – залишок від ділення b на a: $b = aq + r$.

За такого алгоритму зникає необхідність перевірки чисел на простоту чи розкладання числа на множники, які самі по собі є достатньо складними задачами. Складність наведеного алгоритму розрахунку символу Якобі є порівняною зі складністю алгоритмів Евкліда. Для програмної реалізації алгоритму розглянемо альтернативу досить трудомістким операціям ділення та пошуку залишків.

По-перше, операція порівняння числа за модулем 4 (2^2) еквівалентна виявленню останніх двох цифр числа у бінарному записі, аналогічно тому, як порівняння за модулем 100 (10^2) відповідає останнім двом цифрам числа у десятичній формі. Для одержання бітових цифр можна скористатись бітовою маскою – операцію логічного AND з числом 3, що має дві оди-

ниці на останній позиції ($3 = 0 \ldots 0011_2$). Як результат, замість конструкції $b\%4$, яка безпосередньо відповідає за обчислення залишку через ділення, може бути використана бітова операція $b\&3$, яка дасть той самий результат, але значно швидше. Повністю аналогічно, операція $b\%8$ може бути замінена на $b\&7$ ($7 = 0 \ldots 0111_2$). Слід зауважити, що така заміна можлива лише для модулів виду 2^n, якими є числа у даному алгоритмі.

По-друге, операція ділення на 2 може бути замінена на операцію бітового зсуву, яка також є базовою операцією процесора і виконується максимально швидко. У десятковій системі це аналогічно діленню на 10, що відповідає "прибиранню" останнього нуля, тобто зсуву цифр числа на одну позицію праворуч. Отже, замість запису програмної інструкції ділення $a = a / 2$ можна записати $a = a >> 1$ (бітовий зсув на 1 позицію праворуч), що є цілковитим еквівалентом, проте, значно швидшим.

Із використанням операцій з високою швидкодією, функцію для обчислення символу Якобі можна подати наступним способом:

```
def Jacobi(a,b):
    '''Evaluation of Jacobi symbol (a/b)'''
    if a<0:
        if b&3 == 3: return -Jacobi(-a,b)
        else: return Jacobi(-a,b)
    if a == 0:
        return 0
    if a == 1 or b == 1:
        return 1
    n = 0
    while a&1 == 0:
        n += 1
        a = a>>1
```

```
if n > 0:
    if n&1 == 1 and (b&7 == 3 or b&7 == 5):
        return -Jacobi(a,b)
    else: return Jacobi(a,b)
if a&3 == 3 and b&3 == 3:
    return -Jacobi(b%a,a)
else:
    return Jacobi(b%a,a)
```

2.4. Ступеневі порівняння

Ступеневе порівняння може бути сформульоване як задача пошуку числа x такого, що

$$a^x \equiv b \pmod{m} \tag{2.25}$$

На відміну від звичайної алгебри, у теорії чисел ступенева послідовність є періодичною. Оскільки ПСЛ за модулем m є обмеженою множиною, кількість значень для виразу $a^x \pmod{m}$ також є обмеженою. Значить, для певного x стане вірним порівняння $a^x \equiv 1 \pmod{m}$. Наступний член послідовності $a^{x+1} \equiv a \pmod{m}$ і так далі до наступного $a^{x+h} \equiv 1 \pmod{m}$. Далі послідовність буде повторюватись з періодом h.

Найменше число x (окрім нуля), для якого $a^x \equiv 1 \pmod{m}$, називається порядком (мультиплікативним періодом) числа a за модулем m. Як приклад у табл. 2 наведені ступені чисел 2, 3 та 10 для елементів ПСЛ за модулем 11.

Таблиця 2. **Ступені окремих елементів ПСЛ за модулем**

$x \pmod{11}$	1	2	3	4	5
$2^x \pmod{11}$	2	4	8	$16 \equiv 5$	$32 \equiv 10$
$3^x \pmod{11}$	3	9	$27 \equiv 5$	$81 \equiv 4$	$243 \equiv 1$
$10^x \pmod{11}$	10	1	10	1	10
$x \pmod{11}$	6	7	8	9	10
$2^x \pmod{11}$	$64 \equiv 9$	$128 \equiv 7$	$256 \equiv 3$	$512 \equiv 6$	$1024 \equiv 1$
$3^x \pmod{11}$	$729 \equiv 3$	$2187 \equiv 9$	$6561 \equiv 5$	$19683 \equiv 4$	$59049 \equiv 1$
$10^x \pmod{11}$	1	10	1	10	1

Як видно з табл. 2, вираз $2^x \pmod{11}$ набуває одиничного значення при $x = 10$. Тобто, порядок числа 2 за модулем 11 дорівнює 10.

Для числа 3 рівність виконується двічі – для чисел 5 та 10. Відповідно, найменше з цих чисел визначає порядок числа 3. Також неважко помітити, що послідовність $3^x \pmod{11}$ повторюється двічі при перебігу x ПСЛ.

Для числа 10 порядок рівний 2 і послідовність $10^x \pmod{11}$ 5 разів повторюється у ПСЛ.

Закономірності ступеневих порівнянь визначаються кількома теоремами. Теорема Ферма стверджує, що для простого числа p та довільного числа x, що не є порівнянним з нулем (за модулем p), справедливим є порівняння

$$x^{p-1} \equiv 1 \pmod{p}. \tag{2.26}$$

Ейлер узагальнив теорему Ферма на довільні числа:

$$x^{\varphi(m)} \equiv 1 \pmod{m}. \tag{2.27}$$

Теорема Ейлера (2.27) та її частинний випадок – теорема Ферма (2.26) – визначають максимальну оцінку для порядків елементів. Як нескладно помітити з табл. 2, усі наведені числа з різними періодами задовольняють теоремам. Оскільки $\varphi(11) = 10$, теореми Ферма та Ейлера збігаються. Для всіх чисел у табл. 2 вирази зі ступенем 10 є порівнянними з одиницею $2^{10} \equiv 3^{10} \equiv 10^{10} \equiv 1 \pmod{11}$.

Тим не менше, порядки чисел не завжди рівні $\varphi(m)$. Але ці порядки мають бути кратними до $\varphi(m)$, щоб задовольнити умові теореми Ейлера. Наприклад, для табл. 2 величина функції Ейлера розкладається на два множники $\varphi(11) = 10 = 5 \cdot 2$. Відповідно, мультиплікативними періодами елементів ПСЛ за модулем 11 можуть, окрім 10, бути числа 2 і 5, що і простежується у табл. 2.

З порядком елементів пов'язана можливість швидкого розкладання числа на множники – факторизації числа. Одним з наслідків квантових обчислень [7] є можливість визначення мультиплікативного періоду числа засобами керованої зміни фази хвильової функції. За відомим порядком деякого числа за модулем m існує можливість обчислення його множників.

Якщо порядок елемента x є парним числом $h = 2n$, то справедливим є порівняння

$$x^{2n} \equiv 1 \,(\mathrm{mod}\, m) \Rightarrow y^2 \equiv 1 \,(\mathrm{mod}\, m) \Rightarrow y^2 - 1 \equiv 0 \,(\mathrm{mod}\, m),$$

де $y \equiv x^n \pmod m$. Останнє порівняння свідчить про те, що вираз $y^2 - 1$ є кратним до m. Оскільки $y^2 - 1 = (y - 1)(y + 1)$, то спільні дільники з m слід шукати саме серед чисел $(y - 1)$ та $(y + 1)$. Єдине, що слід зауважити, це вилучення з розгляду очевидних розв'язків $y = 1$ та $y = -1$. Ситуація $y = 1$ не є можливою, оскільки в такому разі порядком числа x має бути удвічі менше число, що протирічить початковій умові. Залишається здійснити перевірку на відповідність $y \equiv -1 \pmod m$.

Кінцевий алгоритм дій можна подати наступною схемою: (на вході алгоритму – число m, що має бути розкладене на множники):

1. Генерується випадкове число $x \pmod m$.

2. Встановлюється порядок h цього числа за модулем m (за допомогою квантових обчислень).

3. Якщо число h непарне, переходимо до п. 1, інакше – до п. 4.

4. Обчислюємо $y \equiv x^{\frac{h}{2}} \,(\mathrm{mod}\, m)$.

5. Перевіряємо, чи $y \equiv -1 \pmod m$. Якщо так, множники m не можуть бути знайдені, перехід до п. 6. Якщо ні – обчислюються НСД $(y - 1, m)$ та НСД $(y + 1, m)$, їх результати будуть множниками числа m, перехід до п. 7.

6. Повторюємо пп.1–5 до досягнення заданої межі імовірності перевірки m на простоту.

7. Повторюємо алгоритм для одержаних дільників числа m до отримання у якості кінцевих дільників простих чисел.

Оцінка, виконана у роботі [7], засвідчує, що у ПСЛ за модулем m з імовірністю не менше 1/2 знаходяться числа x з парним мультиплікативним періодом та такі, що $y \equiv x^{h/2} \neq -1 \pmod{m}$. Відповідно, у разі неуспішної перевірки та переходу до п. 6 імовірність розкладання числа m зменшується удвічі. Це може бути використано для встановлення граничної кількості повторення алгоритму у разі відсутності виявлених множників.

Наведений алгоритм є частиною більш загального алгоритму Шора, який передбачає додаткові перевірки числа m на наявність дільників. На даний момент цей алгоритм становить швидше науковий інтерес, оскільки доступність квантових обчислень зараз ще не є достатньою. Тим не менше, системи, побудовані на складності задачі факторизації (на зразок криптосистеми RSA), потенційно можуть втратити надійність та не рекомендуються для подальшого використання.

3. ОПТИМІЗАЦІЯ ТА ПОШУК

3.1. Методи одновимірного пошуку

Основною задачею методів одновимірного пошуку є знаходження екстремального (мінімального або максимального) значення функції однієї змінної $y = f(x)$, тобто встановлення такого значення x, за якого величина y буде мінімальною або максимальною.

Методи одновимірного пошуку можуть бути зведені до задачі пошуку коренів для похідної від заданої функції $f'(x) = 0$ та перевірку знайдених коренів на відповідність умовам задачі (похідна дорівнює нулю як у точках максимуму чи мінімуму, так і у точках перегину). Відповідно, методи одновимірного пошуку можна поділити на прямі (в яких екстремум шукається безпосередньо) та непрямі, в яких розраховуються та досліджуються корені похідної.

За повнотою відомостей про функцію в окрему групу виділяють класичні методи оптимізації, в яких вимагаються повні математичні дані щодо функції, яка оптимізується: чи є дана функція диференційованою (та скільки разів), монотонною, гладкою, унімодальною тощо. Ці методи використовують результати та методи математичного аналізу і застосовуються, як правило, при аналітичному дослідженні проблем. Для випадків, коли функція задається складним непрямим виразом,

таблицею або алгоритмом (підпрограмою) чи надходить від зовнішнього пристрою, дані методи перестають бути дієвими.

Задачі пошуку максимального та мінімального значення повністю еквівалентні. Якщо вимагається знайти максимум функції $f(x)$, то це повністю еквівалентно знаходженню мінімального значення функції $-f(x)$. У подальшому, якщо це не зазначено окремо, будемо вважати, що за задачу висувається знаходження мінімального значення функції.

У переважній більшості практичних задач самими умовами задаються додаткові обмеження на значення аргументу x. По-перше, у таких задачах мова іде про деякі вимірювані величини або характеристики, які одержуються пристроями зі скінченними діапазонами вимірювань. Іншими словами, відомим є діапазон, у якому слід шукати мінімальне значення функції.

По-друге, практичні результати одержуються, оцифровуються та передаються до обчислювального пристрою з певною похибкою. Значить, не вимагається надзвичайна математична точність і для встановлення мінімального значення функції. Додатковим параметром задачі встановлюється допустима похибка обчислень ε.

З урахуванням зазначених особливостей, практична задача пошуку може бути сформульована наступним чином:

З похибкою Δx не гірше ніж ε встановити значення $x_{\min}$ з діапазону $x_{\min} \in [a; b]$, за якого задана функція $y = f(x)$ набуває мінімального значення.

Більшість методів пошуку, що використовуються на практиці, розвинені за умови відомих величин a та b (меж змін аргументу). Але трапляються випадки, коли діапазон зміни x невідомий або вимагається відшукати всі локальні мінімуми функції (для кожного свій діапазон, але невідомо який). Проте такі випадки є швидше виключенням, ніж правилом.

Для таких задач використовують непрямі методи, тобто зводять задачу пошуку до задачі виявлення коренів похідної.

Оскільки задача знаходження всіх коренів рівняння може бути розв'язана математичними способами, то можливе знаходження всіх екстремумів без відомостей про діапазони їх положення. Методи пошуку коренів рівнянь належать до окремого напряму математичних досліджень і в даному розділі не розкриваються. У подальшому будуть розглянуті лише ті методи, які можна віднести до прямих методів пошуку.

Методи покриття

Методи покриття (або методи перебору) передбачають виділення в діапазоні пошуку $[a; b]$ множини окремих точок (покриття діапазону точками), розрахунок значень функції в цих точках та порівняння цих значень з виділенням мінімального.

Методи покриття поділяють на пасивні та оптимальні. В пасивних методах положення точок обираються безвідносно до властивостей функції, частіше за все, рівномірно. В такому разі метод називають методом рівномірного перебору. В оптимальних методах для розміщення точок використовують значення функції та додаткові відомості про неї.

Алгоритм методу рівномірного перебору можна подати наступним чином:

1. Задаємось бажаною точністю знаходження мінімального значення ε, розбиваємо діапазон $[a; b]$ на точки x_i, відстань між якими становить ε.

2. Розраховуємо значення функції $y_i = f(x_i)$ у вказаних точках.

3. Порівнюємо розраховані значення y_i та знаходимо найменше з них $y_{\min}$.

4. Те значення x, що відповідає $y_{\min}$, є відповіддю на поставлену задачу.

Пасивні методи покриття досить прості для програмної реалізації. Бажану точність досить просто змінювати, добиваючись бажаних результатів. Проте кількість кроків (розрахунків функції) n при високій точності стає достатньо великою і зростає обернено пропорційно до ε:

$$\frac{b-a}{n} \le \varepsilon. \tag{3.1}$$

Проблеми з використанням методу рівномірного покриття можуть виникнути, коли мінімальне значення функції сильно локалізоване (рис. 1). У такому разі бажано при використанні методу мати додаткові відомості щодо "гостроти" екстремумів функції.

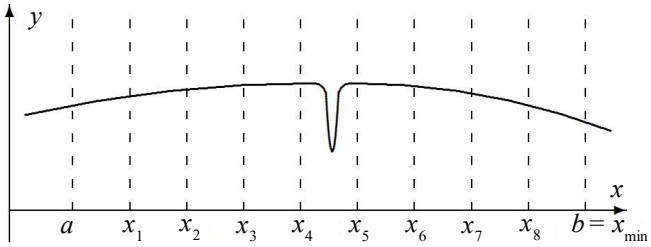


Рис. 1. Пропуск локалізованого екстремуму при рівномірному покритті діапазону пошуку

Оптимальні методи покриття дозволяють вирішувати зазначену проблему. В таких методах використовуються відомості щодо максимального значення першої або другої похідної функції, точніше, чисел L та M, які є не меншими за максимальні значення першої чи другої похідної відповідно (відомості про самі похідні не вимагаються). В такому разі точки наносяться нерівномірно, з урахуванням, окрім самого значення функції, величини чисел L та M у даній точці. Ці відомості

дозволяють обирати такі покриття, за яких гострі мінімуми не пропускаються, оскільки в їх околах значення L та M суттєво зростають і щільність покриття збільшується.

Використання оптимальних методів у більшості випадків можна вважати недоцільним, оскільки вони мають приблизно таку ж кількість кроків, як і дихотомні, що будуть розглянуті далі, проте, окрім багаторазового обчислення значень функції (що вимагає найбільше операцій) як для порівняння, так і для встановлення наступної точки, ще й вимагають додаткових відомостей про похідні від заданої функції, непотрібні для методів дихотомії. Детальніше про оптимальні методи покриття можна дізнатись з роботи [5 с. 35–38].

Дихотомні методи

Дихотомні методи передбачають послідовне ділення відрізку $[a; b]$ на частини, одна з яких відкидається з подальшого розгляду. В процесі розрахунку відбувається зменшення початкового відрізку – положення мінімуму "затискається" з боків. Процес зупиняється, коли довжина відрізку стає меншою (або рівною) за висунуту умовою точність ε.

Загальна схема алгоритму дихотомних методів виглядає наступним чином:

1. На заданому відрізку $[a; b]$ ставлять дві точки λ та μ (рис. 2).

2. Розраховують та порівнюють значення функції в цих точках.

3. Якщо $f(\lambda) < f(\mu)$, то відкидають з розгляду частину відрізку $[\mu; b]$, тобто покладають $b = \mu$, перевіряють умову зупинки пошуку і, якщо вона не виконується, повертаються до п. 1.

4. Якщо $f(\lambda) > f(\mu)$, то відкидають з розгляду частину відрізку $[a; \lambda]$, тобто покладають $a = \lambda$, перевіряють умову зупинки пошуку і, якщо вона не виконується, повертаються до п. 1.

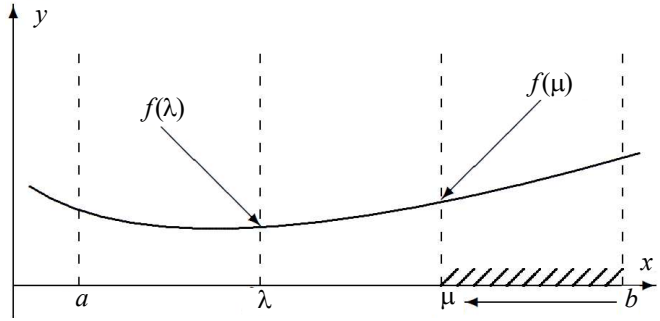


Рис. 2. Загальна схема дихотомних методів

Різні модифікації дихотомних методів відрізняються способами розрахунку значень λ та μ.

Також слід відзначити, що гострий мінімум функції при пошуку, аналогічно до методу рівномірного покриття, може бути пропущений, якщо його ширина набагато менша за довжину відрізка [*a*; *b*]. Для пошуку таких мінімумів потрібне попереднє дослідження функції класичними методами аналізу та локалізація її екстремумів – вибору відрізка [*a*; *b*], порівнянного за довжиною з шириною мінімуму.

Метод ділення навпіл

Метод ділення навпіл практично збігається з таким самим методом пошуку коренів рівнянь. Єдина відмінність полягає у необхідності встановлення напрямку зменшення функції, для чого слід розраховувати два значення функції на кожному кроці. Відповідно до методу ділення навпіл точки λ та μ обираються якомога ближче до центру відрізка [*a*; *b*]:

$$\lambda = \frac{b+a}{2} - \delta, \quad \mu = \frac{b+a}{2} + \delta, \tag{3.2}$$

де величина δ обирається якомога меншою, але з міркувань відмінностей значень $f(\lambda)$ та $f(\mu)$, розрахованих на даній ЕОМ,

для можливості їх порівняння та встановлення напрямку зменшення функції. З практичних міркувань, можна порадити обирати δ у 5–10 разів меншою за граничну похибку встановлення аргументу ε.

При достатньо малому значенні δ можна говороти, що за один крок пошуку довжина відрізка $[a; b]$ скорочується удвічі. Таким чином, за n кроків довжина скоротиться у 2^n разів. Гранична кінцева довжина відрізку задається умовою задачі та становить ε. Отже, необхідна для пошуку кількість кроків може бути визначена із відношення

$$\frac{b-a}{2^n} \le \varepsilon. \tag{3.3}$$

Для демонстрації використання методу ділення навпіл розв'яжемо наступну задачу.

З похибкою, не гіршою 0,1, встановити мінімальне значення заданої функції $f(x) = x + \frac{2}{x}$, *якщо відомо, що воно знаходиться в проміжку* $[1;2]$.

Встановимо необхідну кількість кроків пошуку. Згідно з формулою (3.3) маємо

$$\frac{2-1}{2^n} \le 0{,}1 \Rightarrow 2^n \ge 10 \Rightarrow n = 4.$$

Отже, достатньо зробити чотири кроки. Покладемо $\delta = 0{,}01$, тобто у десять разів меншою за похибку аргументу. Процес розв'язання будемо подавати у вигляді таблиці.

n	a	b	λ	μ	$f(\lambda)$	$f(\mu)$
1	1	2	1,49	1,51	2,832281879	2,834503311
2	1	1,51	1,245	1,265	2,851425703	2,846027668
3	1,245	1,51	1,3675	1,3875	2,830022852	2,828941441
4	1,3675	1,51	1,42875	1,44875	2,828575022	2,829250431
5	1,3675	1,44875				

Початкові значення $a = 1$ та $b = 2$ заносимо першим рядком таблиці. Розраховуємо λ та μ за формулою (3.2) та визначаємо значення функції в цих точках. Перевіряємо, яке з них менше та відкидаємо з розгляду, в даному випадку, праву частину відрізку (див. рис. 1). Отже, в наступний рядок записуємо $a = 1$ та $b = \mu = 1{,}51$. Описані дії складають один крок пошуку. Повторюємо ці дії ще тричі.

Після четвертого кроку переконуємось, що різниця між a та b дійсно не перевищує граничної точності: $1{,}44875 - 1{,}3675 = 0{,}08125 < 0{,}1$. Отже, вимоги умови дотримані. Обираємо середнє значення діапазону, округлюємо його (велика кількість значущих цифр за такої точності не потрібна) та подаємо відповідь.

Відповідь: мінімального значення $f_{\min} \approx 2{,}83$ функція набуває при $x = 1{,}4 \pm 0{,}1$.

Формально процес пошуку можна продовжувати і далі, але обране значення δ не дозволить суттєво покращити досягнуту точність. До того ж, можлива ситуація, коли положення мінімуму залишилось поза відрізком, на відстані δ від якогось краю (що дозволено умовами даної задачі). Для підвищення точності пошуку слід зменшити δ і почати пошук з початку. Хоча, для задач, в яких може вимагатися оперативна зміна точності результату, можна відразу обрати δ достатньо малим, щоб не було необхідності перезапускати обчислення. В такому разі для підвищення точності достатньо буде виконати додаткові кроки.

Метод ділення навпіл вимагає найменшої кількості кроків при заданій точності серед усіх дихотомних методів. Проте, на кожному кроці відбувається два розрахунки значення функції.

Слід відзначити, що частіше за все, основну трудомісткість пошуку складає саме кількість обчислень функції, яка аналізується, а не кількість алгоритмічних кроків. Тому подальші

методи будуються на ідеї зменшення кількості розрахунків функції при неодмінному збільшенні кількості кроків.

Метод "золотого перерізу"

Ідея методу "золотого перерізу" полягає у підборі таких виразів для розрахунку точок λ та μ, щоб після відкидання однієї з частин відрізку, точку, що залишилась, можна було використати на новому кроці. В такому разі на кожному кроці (окрім першого) доведеться обчислювати функцію лише один раз, а друге розраховане значення перенесеться з попереднього кроку і не вимагатиме повторного обчислення.

Математичний аналіз сказаного вище призводить до виразів для λ та μ, відомих з геометрії як формули "золотого перерізу" α. Властивість такого перерізу полягає якраз в тому, що подвійне ділення відрізку у відношенні α дорівнює залишку від першого ділення $1 - \alpha$ (тобто $\alpha^2 = 1 - \alpha$):

$$\begin{gathered} \alpha = \frac{\sqrt{5}-1}{2} \approx 0{,}618034, \\ \lambda = a + (b-a)\cdot(1-\alpha), \quad \mu = a + (b-a)\cdot\alpha. \end{gathered} \tag{3.4}$$

За такого ділення відрізок скорочується в α разів за кожен крок, тобто дещо менше, ніж удвічі. Загальна кількість кроків для досягнення точності ε буде визначатися формулою

$$(b-a)\cdot\alpha^n \le \varepsilon. \tag{3.5}$$

Можна помітити, що положення точок λ та μ симетричне відносно центру відрізка. Відтак, достатньо розрахувати одне із цих значень, а інше встановити з принципів симетрії.

На відміну від методу ділення навпіл в методі "золотого перерізу" немає потреби у введенні додаткових параметрів на зразок δ. Тому метод можна продовжувати циклічно до

досягнення бажаної точності без попереднього встановлення необхідної кількості кроків. Це також дозволяє поновлювати процес оптимізації для підвищення точності результату без перезапусків алгоритму від початку.

Для демонстрації використання методу "золотого перерізу" розв'яжемо задачу з попереднього підрозділу:

З похибкою, не гіршою 0,1, встановити мінімальне значення заданої функції $f(x) = x + \frac{2}{x}$, *якщо відомо, що воно знаходиться в проміжку* [*1;2*].

Як було сказано вище, попереднє встановлення кількості кроків не є необхідним. Будемо контролювати довжину відрізку на кожному кроці. Розв'язок подаємо у вигляді таблиці.

n	*a*	*b*	*b* – *a*	λ	μ	*f*(λ)	*f*(μ)
1	1	2	1	1,3820	1,6180	2,8292	2,8541
2	1	1,6180	0,6180	1,2361	1,3820	2,8541	2,8292
3	1,2361	1,6180	0,3820	1,3820	1,4721	2,8292	2,8307
4	1,2361	1,4721	0,2361	1,3262	1,3820	2,8343	2,8292
5	1,3262	1,4721	0,1459	1,3820	1,4164	2,8292	2,8284
6	1,3820	1,4721	0,0902				

На першому кроці розраховуємо положення точок λ та μ за формулою (3.4), а також значення функції в цих точках. Встановлюємо, що $f(\lambda) < f(\mu)$. Це значить, що в наступному кроці *b* замінюємо на μ. А за властивістю "золотого перерізу" значення λ та $f(\lambda)$ на наступному кроці можна використати як значення попередніх μ та $f(\mu)$ відповідно. Повторно ці значення не розраховуються (процес перенесення вказується в таблиці стрілками). З урахуванням зазначеної вище симетрії точок λ та μ, точку λ на другому кроці можна розрахувати не за формулою (3.4), а обрати симетрично до точки μ. Це дозволить замінити одну операцію множення на одну операцію додавання.

Після п'ятого кроку помічаємо, що різниця між b та a стає меншою за допустиму похибку 0,1. Значить, процес розрахунку зупиняється, як відповідь подаємо округлене середнє фінальних значень b та a.

Відповідь: мінімального значення $f_{\min} \approx 2{,}83$ функція набуває при $x = 1{,}4 \pm 0.1$.

Відзначимо, що хоча методом "золотого перерізу" зроблено 5 кроків замість 4, достатніх для методу ділення навпіл, проте розрахунок функції виконаний 6 разів замість 8. Якщо розрахунок функції є найскладнішим програмним блоком, то використання методу "золотого перерізу" дозволить впоратися із задачею швидше.

Метод "золотого перерізу" є одним з оптимальніших за швидкодією і практично єдиним його недоліком можна вважати використання ірраціонального числа α, яке розраховується лише наближено і з зростанням кількості кроків збільшує похибку відповіді. Проте, взявши до уваги, що сучасні ЕОМ вільно оперують із 15 значущими цифрами після коми на рівні апаратного забезпечення, похибка α в переважній більшості задач є цілком прийнятною.

Метод Фібоначчі

Метод Фібоначчі ґрунтується на такій самій ідеї, як і метод "золотого перерізу". Проте замість використання ірраціональних множників використовуються цілі числа Фібоначчі F_i.

Нагадаємо, що числа Фібоначчі визначаються сумою двох попередніх значень за рекурентним співвідношенням

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_i = F_{i-1} + F_{i-2} \end{cases} \tag{3.6}$$

Положення точок λ та μ згідно з методом Фібоначчі змінюється від кроку до кроку. На *k*-му кроці точки λ_k та μ_k визначаються за виразами:

$$\begin{aligned} \lambda_k &= a + \frac{F_{n-k+1}}{F_{n-k+3}}(b-a), \\ \mu_k &= a + \frac{F_{n-k+2}}{F_{n-k+3}}(b-a), \end{aligned} \tag{3.7}$$

де *n* – кількість кроків пошуку, яка визначається з нерівності

$$\frac{b-a}{F_{n+2}} \le \varepsilon. \tag{3.8}$$

У формулі фігурує $n + 2$, оскільки один крок методу формується відразу трьома числами Фібоначчі, тобто два перших числа потрібні для "старту" методу.

Для використання методу Фібоначчі кількість кроків пошуку чітко детермінована. Можливість виконання додаткових кроків для покращення точності відповіді (без поновлення розрахунків із початку) принципово відсутня. Відповідно, метод Фібоначчі підходить лише для чітко сформульованих задач і не використовується на стадіях дослідження, коли бажано мати змогу оперативно змінювати граничну точність кінцевого результату обчислень.

Доведемо, що точки λ_k та μ_k, що були розраховані для скорочення відрізку $[a_k; b_k]$, дійсно можна використовувати для наступного кроку обчислень за методом Фібоначчі.

Нехай на *k*-му кроці відбулася заміна $b \leftarrow \mu$, тобто

$$b_{k+1} = a_k + \frac{F_{n-k+2}}{F_{n-k+3}}(b_k - a_k), \quad a_{k+1} = a_k.$$

На $(k+1)$-му кроці положення нової точки μ буде визначатись формулою

$$\mu_{k+1} = a_{k+1} + \frac{F_{n-k+1}}{F_{n-k+2}}\left(b_{k+1} - a_{k+1}\right) =$$

$$= a_k + \frac{F_{n-k+1}}{F_{n-k+2}}\left(a_k + \frac{F_{n-k+2}}{F_{n-k+3}}\left(b_k - a_k\right) - a_k \right).$$

Після скорочень отримаємо

$$\mu_{k+1} = a_k + \frac{F_{n-k+1}}{F_{n-k+3}}\left(b_k - a_k\right) = \lambda_k .$$

Тобто на $(k+1)$-му кроці значення λ_k з попереднього кроку можна буде використати в якості μ_{k+1}. Відповідно, і розраховане вже значення функції можна не перераховувати. В аналогічний спосіб можна переконатись, що і при заміні $a \leftarrow \lambda$ точка μ_k з попереднього кроку буде збігатись з λ_{k+1} на $(k+1)$-му кроці.

Аналізуючи вирази (3.6) та (3.7), можна стверджувати, що останній крок методу Фібоначчі фактично збігається з методом ділення навпіл. Оскільки $F_1/F_3 = F_2/F_3 = 1/2$, то значення λ та μ збігаються між собою і становлять $(a+b)/2$. У той же час перші кроки методу, особливо при великих кількостях кроків n, близькі до методу "золотого перерізу" (відомо, що при великих числах послідовність Фібоначчі наближається до прогресії з коефіцієнтом "золотого перерізу").

У поєднанні з відсутністю ірраціональних чисел при розрахунках метод Фібоначчі має найкращі показники щодо швидкодії серед дихотомних методів. Єдиним недоліком методу залишається його "інерційність" щодо впливу на точність кінцевого результату, тобто неможливість продовження уточнюючих розрахунків після зупинки методу. Для покращення точності

відповіді метод має бути виконано наново з повторним визначенням кількості кроків та потрібних чисел Фібоначчі.

Ілюстрацію роботи методу проведемо на тій самій задачі з попередніх розділів.

З похибкою, не гіршою 0,1, встановити мінімальне значення заданої функції $f(x) = x + \frac{2}{x}$, *якщо відомо, що воно знаходиться в проміжку* [*1;2*].

Перш за все, слід визначити необхідну кількість кроків для досягнення заданої точності. Будемо генерувати числа Фібоначчі та контролювати виконання нерівності (3.8). Результати зведемо в таблицю.

i	1	2	3	4	5	6	7
F_i	1	1	2	3	5	8	13
$\frac{b-a}{F_i}$	1,00	1,00	0,50	0,33	0,20	0,13	0,08

Як видно з таблиці, сьоме число Фібоначчі задовольняє нерівність (3.8), значить, для пошуку достатньо виконати $n = 7 - 2 = 5$ кроків.

Розраховуємо λ та μ за формулами (3.7). Наприклад, для першого кроку ($k = 1$)

$$\lambda_1 = 1 + \frac{F_{5-1+1}}{F_{5-1+3}}(2-1) = 1 + \frac{F_5}{F_7} \approx 1{,}3846;$$

$$\mu_1 = 1 + \frac{F_{5-1+2}}{F_{5-1+3}}(2-1) = 1 + \frac{F_6}{F_7} \approx 1{,}6154.$$

Порівнюючи розраховані значення з величинами λ та μ, одержаними на першому кроці методу "золотого перерізу", можемо помітити досить близькі величини.

Далі діємо за типовим алгоритмом дихотомних методів: розраховуємо значення функції у знайдених точках, порівнюємо

їх та відкидаємо з розгляду частину відрізку. Пам'ятаємо, що одне із значень точки та значення функції в ній переноситься на наступний крок. Процес пошуку зведений у таблиці.

n	a	b	λ	μ	$f(\lambda)$	$f(\mu)$
1	1	2	1,3846	1,6154	2,8291	2,8535
2	1	1,6154	1,2308	1,3846	2,8541	2,8291
3	1,2308	1,6154	1,3846	1,4615	2,8291	2,8307
4	1,2308	1,4615	1,3077	1,3846	2,8343	2,8291
5	1,3077	1,4615	1,3846	1,3846	2,8291	2,8291

По закінченні розрахунків можна переконатись, що значення λ та μ на останньому кроці збіглися і відповідають середині залишкового інтервалу пошуку. Їх спільне значення визначає відповідь на поставлену задачу. Також переконуємось, що нерівність $b - a < 0{,}1$ виконується незалежно від того, замінити a на λ чи b на μ. Остаточно, округлюємо відповідь і стверджуємо:

Відповідь: мінімального значення $f_{\min} \approx 2{,}83$ функція набуває при $x = 1{,}4 \pm 0{,}1$.

Провівши нескладні підрахунки, можна встановити, що в процесі пошуку мінімуму значення функції $f(x)$ розраховувались 5 разів. Це менше, ніж у попередніх розглянутих методах, і ще раз свідчить про перевагу методу Фібоначчі над іншими дихотомними методами.

3.2. Методи багатовимірного пошуку

Якщо функція, що оптимізується, залежить від декількох змінних, то для роботи з нею описані вище методи безпосередньо застосовані бути не можуть. Загальну задачу багатовимірного пошуку можна сформулювати у наступний спосіб:

Знайти мінімальне значення функції $y = f(\vec{x}) = f(x_1, x_2, \ldots x_N)$, *тобто такі значення координат* x_1, x_2, ... x_N (*або* $\vec{x}$), *за яких функція* y *набуває мінімального значення.*

Методи багатовимірного пошуку класифікуються, перш за все, за порядком методу. Порядок методу як число дорівнює максимальному порядку похідної від функції, що оптимізується, необхідної для складання алгоритму роботи методу.

Так, методи нульового порядку (вони ж методи прямого пошуку) для встановлення екстремального значення функції вимагають розрахунку лише значень функції, що оптимізується. Методи першого порядку (або градієнтні методи) вимагають також розрахунку першої похідної, методи другого порядку (або ньютонівські методи) вимагають відомостей щодо першої та другої похідної.

Методи прямого пошуку

Методи прямого пошуку (або методи нульового порядку) є найпростішими з точки зору "ручного" використання. Методи відносно просто пояснюються наочно, принципи, закладені в них, є інтуїтивно зрозумілими, хоч і не завжди найкращими за швидкодією.

Якщо задача оптимізації має одноразовий характер, тобто якщо для її розв'язання не доцільно складати складні програми чи програмні комплекси, то методи прямого пошуку є оптимальними відносно вкладених зусиль для досягнення мети. Слід також зазначити, що методи прямого пошуку в ряді особливих задач дозволяють досягти мети навіть швидше, ніж методи вищих порядків. Інколи інтуїція оператора методу, що виконує його "вручну", дозволяє відшукати відповідь швидше, ніж формалізовані алгоритми вищих порядків. Проте, це не завжди так.

Метод випадкового пошуку

Метод випадкового пошуку (або метод випадкової стрільби) є одним з найпростіших для швидкої програмної реалізації, оскільки передбачає лише розрахунок значень функції та їх порівняння. Але при цьому гарантується як завгодно тісне

наближення одержаної відповіді до дійсного екстремального значення функції в процесі розрахунку за умови проведення достатньої кількості порівнянь. Безпосередньо "на папері" даний метод не використовується, оскільки вимагається побудова випадкових чисел та запам'ятовування значень функції для порівнянь.

Зміст методу полягає у генеруванні випадкових чисел, складанні з них випадкових векторів, розрахунку значень функції на цих векторах та їх порівняння між собою з відбором екстремального значення. Вказаний процес продовжується, як правило, до зупинки користувачем. Чим довше відбувається пошук, тим ближче знайдена відповідь до реального значення.

Для роботи методу необхідні дані про область, в якій знаходиться екстремальне значення функції, інакше генерування випадкових чисел без обмежень на їх значення може перетворитися у нескінченний процес.

Широке використання має метод випадкового пошуку зі змінним радіусом (мабуть, саме для цього методу доцільніша аналогія з прицільними стрільбами). Для конкретики пояснень будемо вважати, що вимагається відшукати мінімальне значення заданої функції $y = f(\vec{x}) = f(x_1, x_2, \ldots x_N)$ в області, що являє собою гіперсферу $\left|\vec{x} - \vec{X}\right| < R$ радіуса R з центром у точці, яка задається вектором $\vec{X}$. Алгоритм методу можна подати наступним чином.

1. Генеруємо N випадкових чисел з області визначення, складаємо з них вектор $\vec{x}_0$.

2. Розраховуємо значення функції на цьому векторі $y_0 = f(\vec{x}_0)$, запам'ятовуємо це значення.

3. Генеруємо N випадкових чисел, складаємо з них вектор $\vec{x}_1$.

4. Розраховуємо $y_1 = f(\bar{x}_1)$, порівнюємо, чи $y_1 < y_0$. Якщо так, виконуємо п. 5, якщо ні – переходимо до п. 3.

5. Знайдена точка $\bar{x}_1$ відповідає меншому значенню функції. Замінюємо збережену величину y_0 новим значенням y_1, зменшуємо область пошуку на гіперсферу $|\vec{x} - \vec{x}_1| < R'$ з центром у знайденій точці та з новим радіусом R', який обирається меншим за попередній, скажімо, з міркувань торкання гіперсфер попередньої та нової областей, або просто меншим за попередній радіус на деякий відсоток, який підбирається експериментально.

6. Перевіряємо, чи відповідає знайдене значення умовам задачі (наприклад, за величиною радіусу області пошуку). Якщо ні, переходимо до п. 3.

Метод координатного спуску

Метод координатного спуску, по суті, являє собою послідовність одновимірних пошуків уздовж різних координат.

На певному кроці методу фіксується значення всіх координат, окрім однієї (x_1) – утворюється функція f_1, що залежить лише від x_1. Довільним методом одновимірного пошуку встановлюють значення x_1 за якого f_1 набуває мінімального значення. Фіксують це значення координати x_1 та переходять до іншої координати x_2. Описану процедуру повторюють N разів, тобто послідовно проводять пошук уздовж усіх координатних осей.

Пошук уздовж усіх координат становить один загальний крок пошуку мінімуму функції кількох аргументів. Кількість кроків, тобто серій пошуків, підбирають за критеріями, заданими умовами початкової задачі.

Метод координатного спуску є досить простим для розуміння та програмної реалізації. При вдалому виборі початкової області пошуку, метод дозволяє отримати відповідь на

поставлену задачу достатньо швидко. Проте, для складних функцій або за несприятливих умов вибору початкової точки доводиться робити багато кроків пошуку.

Для демонстрації роботи методів багатовимірного пошуку сформулюємо задачу пошуку мінімуму функції двох аргументів, що задана табличним способом. Більша кількість аргументів не дозволить проводити наочні маніпуляції, проте алгоритмічно нічим не відрізнятиметься від двовимірного прикладу.

Схема використання методу координатного спуску наведена на рис. 3. Починаючи з точки (0,0; 0,0), здійснюється одновимірний пошук вздовж координати x_1. Пошук зупиняється на мінімальному значенні у точці (0,7; 0,0). За наявності кількох комірок (точок) з однаковими значеннями зупинитись слід на їх середині.

Після закінчення пошуку за координатою x_1 здійснюється пошук за іншою координатою (x_2) з початком у точці зупинки попереднього пошуку. Як видно з рис. 3, пошук зупиняється у точці (0,7; 2,0). За наявності інших координат здійснюється послідовний пошук за кожною з них.

Далі повторюється одновимірний пошук послідовно за координатами x_1 та x_2, кожного разу починаючи наступний пошук з точки зупинки попереднього. Пошук зупиняється, коли жоден з одновимірних пошуків не змінює початкової точки. На рис. 3 пошук зупиняється у точці (0,2; 2,2).

Якщо стартувати метод координатного спуску з іншого кута області пошуку – точки (2,0; 0,0) та використовувати інший порядок одновимірних пошуків (спочатку x_2, потім x_1), то послідовне виконання кроків методу приведе до фінальної точки (1,0; 1,2) (показано на рис. 3 пунктирними лініями). Як неважко переконатись, ця точка також є локальним мінімумом табульованої функції.

$x_2 \backslash x_1$	**0.0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**	**1.8**	**1.9**	**2.0**
0.0	78	77	77	76	76	75	75	75	75	75	75	76	76	77	77	78	79	80	80	81	82
0.1	76	76	75	74	74	74	73	73	73	73	73	74	74	75	75	76	77	78	79	80	81
0.2	75	74	73	73	72	72	71	71	71	71	71	72	72	73	74	75	76	77	78	79	80
0.3	73	72	71	71	70	70	69	69	69	69	69	69	70	71	72	73	74	75	76	78	79
0.4	71	70	69	69	68	67	67	66	66	66	66	67	68	68	70	71	72	74	75	76	78
0.5	69	68	67	67	66	65	64	64	63	63	64	64	65	66	67	69	70	72	73	75	76
0.6	67	66	65	64	63	63	62	61	61	60	61	61	62	63	65	67	68	70	72	74	75
0.7	66	64	63	62	61	60	59	58	57	57	57	58	59	61	63	65	67	69	71	72	74
0.8	64	62	61	60	59	58	56	55	54	54	53	54	56	58	60	62	65	67	69	71	73
0.9	62	60	59	58	56	55	54	52	51	50	49	50	52	55	58	61	63	66	68	70	72
1.0	59	58	57	55	54	53	51	49	48	46	45	46	49	52	56	59	62	65	67	69	72
1.1	57	56	54	53	52	50	49	47	45	43	40	41	46	50	54	58	61	64	66	69	71
1.2	55	54	52	51	50	48	47	45	43	41	39	40	45	49	53	57	60	63	66	68	70
1.3	53	51	50	49	48	47	45	44	43	42	41	43	46	49	53	56	60	63	65	68	70
1.4	50	49	48	46	46	45	44	43	43	43	43	45	47	50	53	57	60	62	65	68	70
1.5	48	46	45	44	43	43	43	43	43	44	44	46	48	51	54	57	60	63	65	68	70
1.6	45	43	42	41	41	41	42	42	43	44	46	47	50	52	55	58	60	63	65	68	70
1.7	41	40	39	38	39	39	40	42	43	45	46	48	51	53	56	58	61	63	66	68	70
1.8	38	36	35	35	36	37	39	41	43	45	47	49	52	54	56	59	61	64	66	69	70
1.9	34	32	31	31	33	35	38	40	43	45	48	50	52	55	57	60	62	64	67	69	71
2.0	29	26	26	27	30	33	37	40	43	45	48	51	53	56	58	60	63	65	67	70	71
2.1	25	20	20	23	27	32	36	40	43	46	49	52	54	57	59	61	64	66	68	70	72
2.2	22	14	13	20	26	31	36	40	44	47	50	52	55	58	60	62	64	67	69	70	72
2.3	22	15	14	21	27	32	37	41	45	48	51	54	56	59	61	63	65	67	69	71	73
2.4	27	22	22	25	30	35	39	43	46	49	52	55	57	60	62	64	66	68	70	72	74
2.5	32	30	29	31	34	38	41	45	48	51	54	56	59	61	63	65	67	69	71	73	74
2.6	38	36	36	37	39	42	45	47	50	53	56	58	60	62	64	66	68	70	72	73	76
2.7	43	41	41	42	46	45	48	50	53	55	58	60	62	64	66	68	69	71	73	74	76
2.8	47	46	46	47	48	49	51	53	55	57	60	62	63	65	67	69	71	72	74	75	77
2.9	52	51	51	51	52	53	54	56	58	60	62	63	65	67	69	70	72	73	75	76	78
3.0	55	55	54	55	55	56	57	59	90	62	64	65	67	68	70	72	73	74	76	77	78

Рис. 3. Демонстрація методу координатного спуску

Даний факт свідчить про можливу відмінність результатів пошуку в залежності від початкових умов. Подібна ситуація є типовою для функцій з кількома локальними екстремумами. Чим ближче початкова точка до якогось з екстремумів, тим з більшою імовірністю знайдено буде саме цей екстремум. Особливості використання методів пошуку слід досліджувати окремо для кожної задачі, загальних рекомендацій надавати не доцільно.

Метод Хука–Дживса

Метод Хука–Дживса можна віднести до квазіградієнтних методів, ідея яких схожа з градієнтними методами, проте саме значення градієнта (ані величина, ані напрямок) безпосередньо не розраховується.

Метод починається з однієї початкової точки і не вимагає відомостей про область, що містить екстремальне значення. Один крок пошуку даним методом складається з двох етапів:

1. Знаходячись у початковій точці $\vec{x}_0 = (x_1, x_2, \dots x_N)_0$, визначають мінімальне із "сусідніх" значень функції за першою координатою: $f_{-1} = f(x_1 - h, x_2, \dots x_N)$, $f_0 = f(\vec{x}_0)$ та $f_{+1} = f(x_1 + h, x_2 \dots x_N)$, де h – крок пошуку. Якщо мінімальним є f_{-1}, то значення координати x_1 змінюють на $x_1 - h$, якщо f_{+1} – на $x_1 + h$, якщо f_0 – не змінюють. Повторюють описану процедуру для всіх координат, від яких залежить функція. Внаслідок зміни координат опиняємось у точці $\vec{x}_1$. Якщо різниця між точками $\vec{x}_0$ та $\vec{x}_1$ відсутня, процес пошуку зупиняється, якщо ні – переходять до другого кроку.

2. Проводять одновимірний пошук у напрямку $\vec{x}_1 - \vec{x}_0$, тобто вздовж лінії, що сполучає початкову та кінцеву точки першого кроку. Даний напрямок є близьким до напрямку градієнта, за що метод і називають квазіградієнтним. Знайдену точку мінімуму вважають новою початковою точкою $\vec{x}_0$ та повертаються до першого кроку.

Метод Хука–Дживса може використовуватись і без другого кроку. Для функцій, що залежать від багатьох координат поняття напрямку $\vec{x}_1 - \vec{x}_0$, важко зобразити для аналізу чи перевірки обчислювальної програми, а відтак другий крок може становити проблему для програміста, що не має достатнього досвіду в багатовимірній геометрії. Тому інколи опис

даного методу у літературі можна зустріти без другого кроку. У такій варіації схема методу являє собою певні "сходи" від початкової до кінцевої точки.

Для ілюстрації принципів використання методу Хука–Дживса розв'яжемо задачу для табульованої функції з попереднього підрозділу. Використаємо обидві модифікації методу (з другим кроком та без нього).

1. Виберемо за початкову кутову точку (0,0; 2,0). Вздовж координати x_1 знаходимо найменше з сусідніх значень – переходимо до точки (0,1; 2,0). Аналогічний крок виконуємо вздовж координати x_2 – переходимо до точки (0,1; 1,9).

2. Сполучаємо точки (0,0; 2,0) та (0,1; 1,9) і проводимо напрям одновимірного пошуку (пунктирна лінія на рис. 4). Проводимо одновимірний пошук вздовж цієї лінії – знаходимо комірку з найменшим значенням серед тих, що їх перетинає пунктирна лінія. Знаходимо точку (2,3; 0,3), для якої повторюємо алгоритм.

1') Послідовною перевіркою "сусідів" уздовж координат x_1 та x_2 переходимо до точки (2,2; 0,2). Візуально можна переконатись, що ця точка відповідає мінімальному значенню, проте формально слід виконати пошук вздовж напряму, що сполучає точки (2,3; 0,3) та (2,2; 0,2), хоча результат цього пошуку є передбачуваним.

Скорочений варіант методу Хука–Дживса почнемо з іншої кутової точки (0; 0). Послідовно перевіряємо сусідні комірки то за однією, то за іншою координатами та переходимо до комірок з найменшим значенням. Траєкторія "руху" наведена жирним шрифтом. Слід зауважити, що поблизу локального мінімуму траєкторія робить певний крюк, оскільки необхідно дотримуватись саме послідовного оброблення координат x_1 та x_2, не повторюючи двічі крок за однією і тією ж координатою.

Як видно з рис. 4, повністю аналогічно до методу координатного спуску, різні стартові точки та різні модифікації

методу призводять до різних кінцевих результатів пошуку, що відповідають різним локальним екстремумам.

$x_1 \backslash x_2$	**0,0**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1,0**	**1,1**	**1,2**	**1,3**	**1,4**	**1,5**	**1,6**	**1,7**	**1,8**	**1,9**	**2,0**
0,0	**78**	**77**	77	76	76	75	75	75	75	75	75	76	76	77	77	78	79	80	80	81	82
0,1	76	**76**	**75**	74	74	74	73	73	73	73	73	74	74	75	75	76	77	78	79	80	81
0,2	75	74	**73**	73	72	72	71	71	71	71	71	72	72	73	74	75	76	77	78	79	80
0,3	73	72	**71**	71	70	70	69	69	69	69	69	69	70	71	72	73	74	75	76	78	79
0,4	71	70	**69**	69	68	67	67	66	66	66	66	67	68	68	70	71	72	74	75	76	78
0,5	69	68	**67**	67	66	65	64	64	63	63	64	64	65	66	67	69	70	72	73	75	76
0,6	67	66	**65**	**64**	63	63	62	61	61	60	61	61	62	63	65	67	68	70	72	74	75
0,7	66	64	63	**62**	**61**	60	59	58	57	57	57	58	59	61	63	65	67	69	71	72	74
0,8	64	62	61	60	**59**	**58**	56	55	54	54	53	54	56	58	60	62	65	67	69	71	73
0,9	62	60	59	58	56	**55**	**54**	52	51	50	49	50	52	55	58	61	63	66	68	70	72
1,0	59	58	57	55	54	53	**51**	**49**	48	46	45	46	49	52	56	59	62	65	67	69	72
1,1	57	56	54	53	52	50	49	**47**	**45**	43	40	41	46	50	54	58	61	64	66	69	71
1,2	55	54	52	51	50	48	47	45	**43**	**41**	**39**	40	45	49	53	57	60	63	66	68	70
1,3	53	51	50	49	48	47	45	44	43	**42**	**41**	43	46	49	53	56	60	63	65	68	70
1,4	50	49	48	46	46	45	44	43	43	43	43	45	47	50	53	57	60	62	65	68	70
1,5	48	46	45	44	43	43	43	43	43	44	44	46	48	51	54	57	60	63	65	68	70
1,6	45	43	42	41	41	41	42	42	43	44	46	47	50	52	55	58	60	63	65	68	70
1,7	41	40	39	38	39	39	40	42	43	45	46	48	51	53	56	58	61	63	66	68	70
1,8	38	36	35	35	36	37	39	41	43	45	47	49	52	54	56	59	61	64	66	69	70
1,9	34	32	31	31	33	35	38	40	43	45	48	50	52	55	57	60	62	64	67	69	71
2,0	29	26	26	27	30	33	37	40	43	45	48	51	53	56	58	60	63	65	67	70	71
2,1	25	20	20	23	27	32	36	40	43	46	49	52	54	57	59	61	64	66	68	70	72
2,2	22	14	13	20	26	31	36	40	44	47	50	52	55	58	60	62	64	67	69	70	72
2,3	22	15	14	21	27	32	37	41	45	48	51	54	56	59	61	63	65	67	69	71	73
2,4	27	22	22	25	30	35	39	43	46	49	52	55	57	60	62	64	66	68	70	72	74
2,5	32	30	29	31	34	38	41	45	48	51	54	56	59	61	63	65	67	69	71	73	74
2,6	38	36	36	37	39	42	45	47	50	53	56	58	60	62	64	66	68	70	72	73	76
2,7	43	41	41	42	46	45	48	50	53	55	58	60	62	64	66	68	69	71	73	74	76
2,8	47	46	46	47	48	49	51	53	55	57	60	62	63	65	67	69	71	72	74	75	77
2,9	52	51	51	51	52	53	54	56	58	60	62	63	65	67	69	70	72	73	75	76	78
3,0	55	55	54	55	55	56	57	59	90	62	64	65	67	68	70	72	73	74	76	77	78

Рис. 4. Демонстрація методу Хука–Дживса

Метод координат, що обертаються (*Розенброка*)

Метод Розенброка використовує принцип, схожий з принципом метода Хука–Дживса. Ідея полягає у проведенні одного кроку методом координатного спуску, повороті системи

координат та проведенні наступного спуску у нових (повернутих) координатах.

Більш детально алгоритм пошуку виглядає наступним чином:

1. Обирається початкова точка $\vec{x}_0$, проводиться один крок методом координатного спуску. Процес приводить у точку $\vec{x}_1$.

2. Будується нова система координат: одна з осей проводиться у напрямку $\vec{x}_1 - \vec{x}_0$ (тобто вздовж лінії, що сполучає точки $\vec{x}_0$ та $\vec{x}_1$), інші осі проводяться перпендикулярно до неї.

3. Новою початковою точкою вважається $\vec{x}_1$, новою системою координат – щойно побудована. Повторюються кроки 1–3 до задоволення критеріїв пошуку мінімуму.

Ілюстрація роботи методу Розенброка наведена на рис. 5. Як початкову точку оберемо точку (0,0; 0,0). Координатні осі збігаються з лініями сітки числової таблиці.

Проводимо координатний спуск: вздовж горизонтальної осі – у точку (0,0; 0,8), вдовж вертикальної – у точку (1,6; 0,8). Через початкову та кінцеву точки проводимо нову координатну вісь, іншу проводимо перпендикулярно до неї.

Проводимо нову серію одновимірних пошуків у новій системі координат. Як видно, після пошуку вздовж однієї з осей (ближчої за нахилом до горизонтальної) ми відразу потрапляємо у точку (2,2; 0,2), яка відповідає мінімальному значенню функції. Процеси одновимірного пошуку за методом координатних спусків у різних системах координат показані на рис. 5 штриховими лініями.

Хоча метод Розенброка є схожим на метод Хука–Дживса, до квазіградієнтних цей метод не відноситься. Причиною цьому є те, що в методі Хука–Дживса зміна напрямку пошуку відбувається з аналізу найближчих до початкової точки значень (що є аналогом диференціальної зміни *dx*), тоді як

у методі Розенброка проводиться пошук достатньо віддаленої точки, що може значно відхилятись від напряму градієнта.

$x_1 \backslash x_2$	**0,0**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1,0**	**1,1**	**1,2**	**1,3**	**1,4**	**1,5**	**1,6**	**1,7**	**1,8**	**1,9**	**2,0**
0,0	78	77	77	76	76	75	75	75	75	75	75	76	76	77	77	78	79	80	80	81	82
0,1	76	76	75	74	74	74	73	73	73	73	73	74	74	75	75	76	77	78	79	80	81
0,2	75	74	73	73	72	72	71	71	71	71	71	72	72	73	74	75	76	77	78	79	80
0,3	73	72	71	71	70	70	69	69	69	69	69	69	70	71	72	73	74	75	76	78	79
0,4	71	70	69	69	68	67	67	66	66	66	66	67	68	68	70	71	72	74	75	76	78
0,5	69	68	67	67	66	65	64	64	63	63	64	64	65	66	67	69	70	72	73	75	76
0,6	67	66	65	64	63	63	62	61	61	60	61	61	62	63	65	67	68	70	72	74	75
0,7	66	64	63	62	61	60	59	58	57	57	57	58	59	61	63	65	67	69	71	72	74
0,8	64	62	61	60	59	58	56	55	54	54	53	54	56	58	60	62	65	67	69	71	73
0,9	62	60	59	58	56	55	54	52	51	50	49	50	52	55	58	61	63	66	68	70	72
1,0	59	58	57	55	54	53	51	49	48	46	45	46	49	52	56	59	62	65	67	69	72
1,1	57	56	54	53	52	50	49	47	45	43	40	41	46	50	54	58	61	64	66	69	71
1,2	55	54	52	51	50	48	47	45	43	41	39	40	45	49	53	57	60	63	66	68	70
1,3	53	51	50	49	48	47	45	44	43	42	41	43	46	49	53	56	60	63	65	68	70
1,4	50	49	48	46	46	45	44	43	43	43	43	45	47	50	53	57	60	62	65	68	70
1,5	48	46	45	44	43	43	43	43	43	44	44	46	48	51	54	57	60	63	65	68	70
1,6	45	43	42	41	41	41	42	42	43	44	46	47	50	52	55	58	60	63	65	68	70
1,7	41	40	39	38	39	39	40	42	43	45	46	48	51	53	56	58	61	63	66	68	70
1,8	38	36	35	35	36	37	39	41	43	45	47	49	52	54	56	59	61	64	66	69	70
1,9	34	32	31	31	33	35	38	40	43	45	48	50	52	55	57	60	62	64	67	69	71
2,0	29	26	26	27	30	33	37	40	43	45	48	51	53	56	58	60	63	65	67	70	71
2,1	25	20	20	23	27	32	36	40	43	46	49	52	54	57	59	61	64	66	68	70	72
2,2	22	14	13	20	26	31	36	40	44	47	50	52	55	58	60	62	64	67	69	70	72
2,3	22	15	14	21	27	32	37	41	45	48	51	54	56	59	61	63	65	67	69	71	73
2,4	27	22	22	25	30	35	39	43	46	49	52	55	57	60	62	64	66	68	70	72	74
2,5	32	30	29	31	34	38	41	45	48	51	54	56	59	61	63	65	67	69	71	73	74
2,6	38	36	36	37	39	42	45	47	50	53	56	58	60	62	64	66	68	70	72	73	76
2,7	43	41	41	42	46	45	48	50	53	55	58	60	62	64	66	68	69	71	73	74	76
2,8	47	46	46	47	48	49	51	53	55	57	60	62	63	65	67	69	71	72	74	75	77
2,9	52	51	51	51	52	53	54	56	58	60	62	63	65	67	69	70	72	73	75	76	78
3,0	55	55	54	55	55	56	57	59	90	62	64	65	67	68	70	72	73	74	76	77	78

Рис. 5. Демонстрація методу Розенброка

Метод паралельних дотичних (*Пауелла*)

Метод Пауелла також можна віднести до квазіградієнтних методів, які в непрямому вигляді використовують властивості градієнта. Ідея методу полягає в наступному:

1. В області визначення функції довільним чином проводяться дві паралельні прямі лінії.

2. Вздовж кожної з ліній проводиться одномірний пошук – визначаються точки $\vec{x}_1$ та $\vec{x}_2$, що відповідають мінімальним значенням функції вздовж ліній 1 та 2 відповідно.

3. По точках $\vec{x}_1$ та $\vec{x}_2$ проводиться пряма лінія. Паралельно до неї проводиться друга пряма лінія та повторюються кроки 2–3 до задоволення критеріїв пошуку мінімуму.

$x_1 \backslash x_2$	**0,0**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1,0**	**1,1**	**1,2**	**1,3**	**1,4**	**1,5**	**1,6**	**1,7**	**1,8**	**1,9**	**2,0**
0,0	78	77	77	76	76	75	75	75	75	75	75	76	76	77	77	78	79	80	80	81	82
0,1	76	76	75	74	74	74	73	73	73	73	73	74	74	75	75	76	77	78	79	80	81
0,2	75	74	73	73	72	72	71	71	71	71	71	72	72	73	74	75	76	77	78	79	80
0,3	73	72	71	71	70	70	69	69	69	69	69	69	70	71	72	73	74	75	76	78	79
0,4	71	70	69	69	68	67	67	66	66	66	66	67	68	68	70	71	72	74	75	76	78
0,5	69	68	67	67	66	65	64	64	63	63	64	64	65	66	67	69	70	72	73	75	76
0,6	67	66	65	64	63	63	62	61	61	60	61	61	62	63	65	67	68	70	72	74	75
0,7	66	64	63	62	61	60	59	58	57	57	57	58	59	61	63	65	67	69	71	72	74
0,8	64	62	61	60	59	58	56	55	54	54	53	54	56	58	60	62	65	67	69	71	73
0,9	62	60	59	58	56	55	54	52	51	50	49	50	52	55	58	61	63	66	68	70	72
1,0	59	58	57	55	54	53	51	49	48	46	45	46	49	52	56	59	62	65	67	69	72
1,1	57	56	54	53	52	50	49	47	45	43	40	41	46	50	54	58	61	64	66	69	71
1,2	55	54	52	51	50	48	47	45	43	41	39	40	45	49	53	57	60	63	66	68	70
1,3	53	51	50	49	48	47	45	44	43	42	41	43	46	49	53	56	60	63	65	68	70
1,4	50	49	48	46	46	45	44	43	43	43	43	45	47	50	53	57	60	62	65	68	70
1,5	48	46	45	44	43	43	43	43	43	44	44	46	48	51	54	57	60	63	65	68	70
1,6	45	43	42	41	41	41	42	42	43	44	46	47	50	52	55	58	60	63	65	68	70
1,7	41	40	39	38	39	39	40	42	43	45	46	48	51	53	56	58	61	63	66	68	70
1,8	38	36	35	35	36	37	39	41	43	45	47	49	51	54	56	59	61	64	66	69	70
1,9	34	32	31	31	33	35	38	40	43	45	48	50	52	55	57	60	62	64	67	69	71
2,0	29	26	26	27	30	33	37	40	43	45	48	51	53	56	58	60	63	65	67	70	71
2,1	25	20	20	23	27	32	36	40	43	46	49	52	54	57	59	61	64	66	68	70	72
2,2	22	14	13	20	26	31	36	40	44	47	50	52	55	58	60	62	64	67	69	70	72
2,3	22	15	14	21	27	32	37	41	45	48	51	54	56	59	61	63	65	67	69	71	73
2,4	27	22	22	25	30	35	39	43	46	49	52	55	57	60	62	64	66	68	70	72	74
2,5	32	30	29	31	34	38	41	45	48	51	54	56	59	61	63	65	67	69	71	73	74
2,6	38	36	36	37	39	42	45	47	50	53	56	58	60	62	64	66	68	70	72	73	76
2,7	43	41	41	42	46	45	48	50	53	55	58	60	62	64	66	68	69	71	73	74	76
2,8	47	46	46	47	48	49	51	53	55	57	60	62	63	65	67	69	71	72	74	75	77
2,9	52	51	51	51	52	53	54	56	58	60	62	63	65	67	69	70	72	73	75	76	78
3,0	55	55	54	55	55	56	57	59	90	62	64	65	67	68	70	72	73	74	76	77	78

Рис. 6. Демонстрація методу Пауелла

За сприятливих умов метод дозволяє відкидати з розгляду один з напрямків за один крок. Це значить, що за таких умов повторення алгоритму відбуватиметься N разів (де N – кількість координат, що від них залежить досліджувана функція).

Для ілюстрації процесу використання методу Пауелла здійснимо пошук на табульованій функції (рис. 6). Проведемо дві паралельні лінії вздовж горизонтальної осі, оскільки початковий напрямок ліній може бути обраний довільним чином. На рис. 6 вони наведені суцільними лініями. Встановимо мінімальні значення вздовж кожної з цих ліній – ці значення акцентовані напівжирним шрифтом: точки (0,2; 0,8) та (0,6; 0,9). По цих точках проводимо нову пряму лінію та іншу лінію, паралельну до неї (пунктирні лінії). Встановлюємо мінімальні значення функції вздовж них. Продовжуємо побудову ліній (наступний крок проведено точковою лінією). Як можна помітити, після двох кроків лінія проходить крізь локальний екстремум, що свідчить про завершення роботи методу.

Метод деформованого багатогранника (*Нелдера–Міда*)

Метод Нелдера–Міда є одним з найбільш вживаних методів багатовимірного пошуку завдяки надійності його результатів. Метод можна назвати аналогічним до дихотомних у тому сенсі, що пошук мінімального значення відбувається "затисканням" його з боків певним багатогранником у багатовимірному просторі. Причому, на відміну від дихотомних методів, метод Нелдера–Міда може використовуватись і в тому випадку, коли мінімальне значення функції не міститься у початково обраній області локалізації (вихідному багатограннику).

Алгоритм використання методу можна подати у наступний спосіб:

1. Обираємо довільним чином $N + 1$ точку в області визначення функції. Проводимо по них, як по вершинах, багато-

гранник (N – розмірність простору визначення функції, що досліджується).

2. Розраховуємо значення функції у вершинах багатогранника та встановлюємо вершину з максимальним значенням функції.

3. Серед інших вершин (окрім тієї, де значення функції максимальне) знаходимо геометричний центр.

4. Проводимо лінію, що сполучає вершину з максимальним значенням та центр інших вершин.

5. Проводимо одновимірний пошук уздовж проведеної лінії – визначаємо точку з мінімальним значенням функції.

6. Знайдена точка з мінімальним значенням вважається новою вершиною багатогранника замість тієї, де спостерігалось максимальне значення функції. Будується новий (деформований) багатогранник.

7. Продовжуємо деформувати багатогранник (пп. 2–6) до тих пір, поки його об'єм не задовольнить умовам зупинки щодо пошуку мінімального значення.

Ілюстрація використання методу наведена на рис. 7 на прикладі пошуку мінімального значення функції, заданої таблицею. Оскільки функція залежить від двох координат, мова йтиме про деформовані трикутники. Оберемо як початкові точки (0,2; 0,1), (0,6; 0,5), (1,4; 0,3).

Вершина з максимальним значенням функції відповідає точці (0,2; 0,1). Серед інших двох точок визначаємо геометричний центр, який для двох точок відповідає середині відрізка між точками. Проводимо з точки максимального значення через центр відрізка лінію (пунктир на рис. 7).

Уздовж побудованого напрямку проводимо пошук та знаходимо мінімальне значення у точці (1,8; 0,6), що відзначена на рис. 7 сірим фоном. Використовуючи її як нову вершину, будуємо деформований трикутник (зображений штриховою лінією).

$x_1 \backslash x_2$	**0,0**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1,0**	**1,1**	**1,2**	**1,3**	**1,4**	**1,5**	**1,6**	**1,7**	**1,8**	**1,9**	**2,0**
0,0	78	77	77	76	76	75	75	75	75	75	75	76	76	77	77	78	79	80	80	81	82
0,1	76	76	75	74	74	74	73	73	73	73	73	74	74	75	75	76	77	78	79	80	81
0,2	75	**74**	73	73	72	72	71	71	71	71	71	72	72	73	74	75	76	77	78	79	80
0,3	73	72	71	71	70	70	69	69	69	69	69	69	70	71	72	73	74	75	76	78	79
0,4	71	70	69	69	68	67	67	66	66	66	66	67	68	68	70	71	72	74	75	76	78
0,5	69	68	67	67	66	65	64	64	63	63	64	64	65	66	67	69	70	72	73	75	76
0,6	67	66	65	64	63	**63**	62	61	61	60	61	61	62	63	65	67	68	70	72	74	75
0,7	66	64	63	62	61	60	59	58	57	57	57	58	59	61	63	65	67	69	71	72	74
0,8	64	62	61	60	59	58	56	55	54	54	53	54	56	58	60	62	65	67	69	71	73
0,9	62	60	59	58	56	55	54	52	51	50	49	50	52	55	58	61	63	66	68	70	72
1,0	59	58	57	55	54	53	51	49	48	46	45	46	49	52	56	59	62	65	67	69	72
1,1	57	56	54	53	52	50	49	47	45	43	40	41	46	50	54	58	61	64	66	69	71
1,2	55	54	52	51	50	48	47	45	43	41	39	40	45	49	53	57	60	63	66	68	70
1,3	53	51	50	49	48	47	45	44	43	42	41	43	46	49	53	56	60	63	65	68	70
1,4	50	49	48	46	46	45	44	43	43	43	43	45	47	50	53	57	60	62	65	68	70
1,5	48	46	45	44	43	43	43	43	43	44	44	46	48	51	54	57	60	63	65	68	70
1,6	45	43	42	41	41	41	42	42	43	44	46	47	50	52	55	58	60	63	65	68	70
1,7	41	40	39	38	39	39	40	42	43	45	46	48	51	53	56	58	61	63	66	68	70
1,8	38	36	35	35	36	37	39	41	43	45	47	49	52	54	56	59	61	64	66	69	70
1,9	34	32	31	31	33	35	38	40	43	45	48	50	52	55	57	60	62	64	67	69	71
2,0	29	26	26	27	30	33	37	40	43	45	48	51	53	56	58	60	63	65	67	70	71
2,1	25	20	20	23	27	32	36	40	43	46	49	52	54	57	59	61	64	66	68	70	72
2,2	22	14	13	20	26	31	36	40	44	47	50	52	55	58	60	62	64	67	69	70	72
2,3	22	15	14	21	27	32	37	41	45	48	51	54	56	59	61	63	65	67	69	71	73
2,4	27	22	22	25	30	35	39	43	46	49	52	55	57	60	62	64	66	68	70	72	74
2,5	32	30	29	31	34	38	41	45	48	51	54	56	59	61	63	65	67	69	71	73	74
2,6	38	36	36	37	39	42	45	47	50	53	56	58	60	62	64	66	68	70	72	73	76
2,7	43	41	41	42	46	45	48	50	53	55	58	60	62	64	66	68	69	71	73	74	76
2,8	47	46	46	47	48	49	51	53	55	57	60	62	63	65	67	69	71	72	74	75	77
2,9	52	51	51	51	52	53	54	56	58	60	62	63	65	67	69	70	72	73	75	76	78
3,0	55	55	54	55	55	56	57	59	90	62	64	65	67	68	70	72	73	74	76	77	78

Рис. 7. Демонстрація методу Нелдера–Міда

Повторюємо алгоритм методу. Знаходимо вершину з максимальним значенням – точку (0,6; 0,5). Між іншими точками знаходимо центр відрізка та проводимо напрям, що сполучає цей центр та точку з максимальним значенням (штрихпунктирна лінія на рис. 7).

Уздовж одержаного напряму проводимо пошук – знаходимо мінімальне значення функції у точці (2,2; 0,4). Вилучаємо з розгляду точку (0,6; 0,5) та додаємо нову вершину (2,2; 0,4).

Продовжуємо деформувати трикутник до тих пір, поки його розмір перевищуватиме величину допустимої похибки, або доки усі вершини не опиняться в одній точці (наступні кроки пошуку на рис. 7 не наведені).

3.3. Градієнтні методи

Методи першого порядку, або градієнтні методи, для пошуку мінімального значення функції використовують відомості про її похідні. Оскільки розрахунок частинних похідних складних функцій без використання обчислювальної техніки проводиться рідко, градієнтні методи, частіше за все, використовуються в складі програмних пакетів оптимізації та пошуку.

Градієнтом функції $f(\vec{x}) = f(x_1, x_2, \ldots x_N)$ у точці $\vec{x}_0$ називається вектор, складений з частинних похідних функції по всіх її координатах:

$$\text{grad } f = \begin{pmatrix} \left.\dfrac{\partial f}{\partial x_1}\right|_{\vec{x}=\vec{x}_0} \\ \left.\dfrac{\partial f}{\partial x_2}\right|_{\vec{x}=\vec{x}_0} \\ \vdots \\ \left.\dfrac{\partial f}{\partial x_N}\right|_{\vec{x}=\vec{x}_0} \end{pmatrix}.$$

Вектор градієнта перпендикулярний до поверхні постійного значення (поверхні, на якій значення функції не змінюється), що проходить крізь дану точку та спрямований у бік найбільшого зростання функції в даній точці. Модуль градієнта визначає швидкість зростання функції. Вектор, рівний градієнту за модулем та протилежний за напрямком, називається антиградієнтом. Антиградієнт спрямований у напрямку найшвидшого спадання

функції. В точках, де функція набуває екстремального значення (максимального чи мінімального), градієнт функції (як і антиградієнт) дорівнює нулю.

Метод найшвидшого спуску

У методі найшвидшого спуску безпосередньо використовується властивість градієнта (або антиградієнта) вказувати напрямок найшвидшої зміни функції. Алгоритм використання методу можна подати у наступний спосіб.

1. Обирається довільним чином початкова точка $\vec{x}_0$.

2. У даній точці розраховується значення градієнта.

3. Проводиться одновимірний пошук у напрямку, заданому вектором градієнта. Визначається точка $\vec{x}_1$ з мінімальним значенням фунції на цьому напрямі.

4. Перевіряється, чи задовольняє знайдена точка умовам припинення пошуку; якщо ні, то знайдена точка $\vec{x}_1$ вважається новою початковою точкою і повторюються кроки 2–3.

Градієнтні методи дуже швидко приводять до відповіді на поставлену задачу. За сприятливих умов (якщо функція має вигляд, подібний до квадратичного) градієнт відразу вказує напрямок екстремального значення функції, відповідно, вимагається лише один крок одновимірного пошуку.

Проблеми з використанням методу полягають у необхідності розраховувати всі частинні похідні функції на кожному кроці пошуку. Якщо початкова точка знаходиться достатньо далеко від екстремальної, то похибки розрахунку похідних призведуть до відхилення напрямку розрахованого градієнта від дійсного, внаслідок чого кількість необхідних кроків пошуку збільшиться. Також пряме використання даного методу "вручну" для табульованих функцій ускладнено, оскільки доводиться проводити додаткові розрахунки з обчислення частинних похідних. Відтак, метод найшвидшого спуску частіше за все використовується у вигляді програмної реалізації.

Тим не менше, для табульованих функцій можна використати властивість градієнта бути перпендикулярним до ліній постійного значення. На рис. 8 наводиться ілюстрація такої можливості. Оберемо за початкову точку (0,7; 1,7) зі значенням функції 69. Знайдемо серед сусідніх точок такі, що також мають значення 69, та сполучимо їх плавною лінією. До цієї лінії проведемо перпендикуляр, який відповідатиме приблизному напряму градієнта. Здійснимо пошук вздовж цього напряму – одержимо точку (2,5; 0,6) зі значенням 41.

$x_1 \backslash x_2$	**0,0**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1,0**	**1,1**	**1,2**	**1,3**	**1,4**	**1,5**	**1,6**	**1,7**	**1,8**	**1,9**	**2,0**
0,0	78	77	77	76	76	75	75	75	75	75	75	76	76	77	77	78	79	80	80	81	82
0,1	76	76	75	74	74	74	73	73	73	73	73	74	74	75	75	76	77	78	79	80	81
0,2	75	74	73	73	72	72	71	71	71	71	71	72	72	73	74	75	76	77	78	79	80
0,3	73	72	71	71	70	70	69	69	69	**69**	**69**	**69**	70	71	72	73	74	75	76	78	79
0,4	71	70	69	69	68	67	67	66	66	66	66	67	68	68	70	71	72	74	75	76	78
0,5	69	68	67	67	66	65	64	64	63	63	64	64	65	66	67	**69**	70	72	73	75	76
0,6	67	66	65	64	63	63	62	61	61	60	61	61	62	63	65	67	68	70	72	74	75
0,7	66	64	63	62	61	60	59	58	57	57	57	58	59	61	63	65	67	**69**	71	72	74
0,8	64	62	61	60	59	58	56	55	54	54	53	54	56	58	60	62	65	67	**69**	71	73
0,9	62	60	59	58	56	55	54	52	51	50	49	50	52	55	58	61	63	66	68	70	72
1,0	59	58	57	55	54	53	51	49	48	46	45	46	49	52	56	59	62	65	67	**69**	72
1,1	57	56	54	53	52	50	49	47	45	43	40	41	46	50	54	58	61	64	66	**69**	71
1,2	55	54	52	51	50	48	47	45	43	41	39	40	45	49	53	57	60	63	66	68	70
1,3	53	51	50	49	48	47	45	44	43	42	41	43	46	49	53	56	60	63	65	68	70
1,4	50	49	48	46	46	45	44	43	43	43	43	45	47	50	53	57	60	62	65	68	70
1,5	48	46	45	44	43	43	43	43	43	44	44	46	48	51	54	57	60	63	65	68	70
1,6	45	43	42	41	41	41	42	42	43	44	46	47	50	52	55	58	60	63	65	68	70
1,7	41	40	39	38	39	39	40	42	43	45	46	48	51	53	56	58	61	63	66	68	70
1,8	38	36	35	35	36	37	39	41	43	45	47	49	52	54	56	59	61	64	66	69	70
1,9	34	32	31	31	33	35	38	40	43	45	48	50	52	55	57	60	62	64	67	69	71
2,0	29	26	26	27	30	33	37	40	43	45	48	51	53	56	58	60	63	65	67	70	71
2,1	25	20	20	23	27	32	36	40	43	46	49	52	54	57	59	61	64	66	68	70	72
2,2	22	14	**13**	20	26	31	36	40	44	47	50	52	55	58	60	62	64	67	69	70	72
2,3	22	15	14	21	27	32	37	**41**	45	48	51	54	56	59	61	63	65	67	69	71	73
2,4	27	22	22	25	30	35	39	43	46	49	52	55	57	60	62	64	66	68	70	72	74
2,5	32	30	29	31	34	38	41	45	48	51	54	56	59	61	63	65	67	69	71	73	74
2,6	38	36	36	37	39	42	45	47	50	53	56	58	60	62	64	66	68	70	72	73	76
2,7	43	41	41	42	46	45	48	50	53	55	58	60	62	64	66	68	69	71	73	74	76
2,8	47	46	46	47	48	49	51	53	55	57	60	62	63	65	67	69	71	72	74	75	77
2,9	52	51	51	51	52	53	54	56	58	60	62	63	65	67	69	70	72	73	75	76	78
3,0	55	55	54	55	55	56	57	59	90	62	64	65	67	68	70	72	73	74	76	77	78

Рис. 8. Демонстрація методу найшвидшого спуску

Аналогічно до попереднього, знайдемо сусідів із таким само значенням, встановимо лінію постійного рівня та проведемо перпендикуляр до неї. На рис. 8 друга лінія постійного рівня не зображена (це відрізок між точками зі значенням 41), але зображено перпендикуляр до неї. Як видно, після другого кроку мінімальне значення функції встановлюється.

Підсумовуючи, можна сказати, що метод найшвидшого спуску є одним з найкращих для використання в автоматизованих розрахунках при складанні програм алгоритмічними мовами, але з певними недосконалостями може бути використаний для табличних функцій.

Метод спряжених градієнтів

Однією з особливостей методу найшвидшого спуску є те, що траєкторія спуску являє собою ламану лінію з практично перпендикулярними ланками. Метод спряжених градієнтів полягає у знаходженні нового напрямку пошуку, який не обов'язково буде перпендикулярний до попереднього. При цьому для пошуку нового напрямку використовується поняття спряженості векторів відносно деякої матриці. Відтак, метод оперує зі спряженими напрямками, проте, з історичних обставин, називається методом спряжених градієнтів.

Алгоритм методу подібний до алгоритму методу найшвидшого спуску, змінюється лише процедура встановлення нового напрямку пошуку:

1. Обирається довільним чином точка $\vec{x}_1$.

2. У даній точці розраховується значення антиградієнта $\vec{g}_0$.

3. Проводиться одновимірний пошук у напрямку $\vec{d}_0 = \vec{g}_0$, заданому антиградієнтом. Визначається точка мінімуму $\vec{x}_1$, яка буде початковою для методу спряжених градієнтів.

4. У точці $\vec{x}_1$ розраховується значення антиградієнта $\vec{g}_1$ та визначається новий напрямок пошуку за формулою $\vec{d}_1 = \vec{g}_0 + \beta \cdot \vec{d}_0$ (вираз для β наводиться далі).

5. Проводиться одновимірний пошук у напрямку $\vec{d}_1$ та визначається точка мінімуму $\vec{x}_2$.

6. Перевіряється, чи задовольняє знайдена точка умовам припинення пошуку. Якщо ні, то знайдена точка $\vec{x}_2$ вважається новою початковою (замінюється $\vec{x}_2$ на $\vec{x}_1$, $\vec{d}_1$ на $\vec{d}_0$) і повторюються кроки 4–6.

Для встановлення значення коефіцієнта β користуються одним з двох виразів: Флетчера–Рівса та Полака–Райбера. Прийняття рішення на користь того чи іншого виразу відбивається у зміненій назві методу. Так, згідно з методом Флетчера–Рівса

$$\beta = \frac{(\vec{g}_1 \cdot \vec{g}_1)}{(\vec{g}_0 \cdot \vec{g}_0)},$$

де $(\vec{g}_1 \cdot \vec{g}_1)$ – скалярний добуток вектора $\vec{g}_1$ самого на себе (квадрат модуля антиградієнта).

У методі Полака–Райбера використовуються наступні значення коефіцієнта β:

$$\beta = \max\left(0, \frac{(\vec{g}_1 \cdot (\vec{g}_1 - \vec{g}_0))}{(\vec{g}_0 \cdot \vec{g}_0)}\right),$$

тобто якщо розрахований вираз менший за нуль, β обирається рівною нулю, що відповідає проведенню наступного пошуку у напрямку антиградієнта. Подібна ситуація називається

"перезапуском" методу, оскільки є аналогічною початку алгоритму пошуку з самого початку.

Метод спряжених градієнтів збігається у 4–5 разів швидше, ніж метод найшвидшого спуску, і наближається за швидкодією до методів другого порядку. Оскільки у методі використовуються відомості щодо самого значення градієнта (точніше антиградієнта), подавати метод наочно на прикладі таблиці немає сенсу.

3.3. Методи другого порядку

У методах другого порядку використовуються відомості щодо других похідних від шуканої функції. Загальною характеристикою другої похідної функції кількох змінних є матриця Гессе $\widehat{H}$ (або гесіан) функції

$$\widehat{H}(\vec{x}) = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_N} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_1^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \partial x_N} \\ \vdots & \vdots & \vdots & \vdots \\ \dfrac{\partial^2 f}{\partial x_N \partial x_1} & \dfrac{\partial^2 f}{\partial x_N \partial x_2} & & \dfrac{\partial^2 f}{\partial x_N^2} \end{pmatrix}.$$

Матриця Гессе визначає опуклість функції у даній точці. Знаючи напрямок найшвидшого зростання функції та ступінь її опуклості, можна відразу вказати очікуване положення її екстремального значення.

Відмінність методів другого порядку полягає саме в тому, що при їх використанні не проводяться одновимірні пошуки, а відразу вказується точка з очікуваним екстремальним значенням.

За рахунок складності розрахунку матриці Гессе розвинено ряд методів, які за принципом пошуку відповідають методам другого порядку, проте саме значення гесіана не використовують. За цією ознакою методи поділяють на ньютонівські, в яких відбувається розрахунок гесіана, та квазіньютонівські, в яких такий розрахунок безпосередньо не відбувається (хоча наближений гесіан будується в процесі оптимізації).

Метод Ньютона–Рафсона

Метод Ньютона–Рафсона є багатовимірною модифікацією звичайного методу Ньютона для знаходження коренів алгебраїчних рівнянь. У задачах пошуку таким рівнянням є $\text{grad}\, f(\vec{x}) = 0$.

Згідно з методом Ньютона–Рафсона, розв'язок цього рівняння буде мати вигляд

$$\vec{x}_1 = \vec{x}_0 - \hat{H}^{-1}(\vec{x}_0) \cdot \text{grad}\, f(\vec{x}_0), \qquad (3.11)$$

де $\vec{x}_0$ – деяке початкове наближення.

Повний алгоритм використання методу Ньютона–Рафсона можна подати у наступний спосіб.

1. Обираємо довільним чином початкову точку $\vec{x}_0$.

2. Розраховуємо в початковій точці градієнт функції $\text{grad}\, f(\vec{x}_0)$ та матрицю Гессе $\hat{H}(\vec{x}_0)$.

3. Знаходимо обернену матрицю в точці $\vec{x}_0$: Гессе $\hat{H}^{-1}(\vec{x}_0)$.

4. Розраховуємо точку очікуваного екстремуму $\vec{x}_1$ за формулою (3.11).

5. Перевіряємо, чи задовольняє знайдена точка умовам зупинки пошуку для даної задачі, якщо ні – вважаємо точку $\vec{x}_1$ новою початковою і повторюємо кроки 2–5.

Для квадратичних функцій метод дозволяє встановити відповідь за одне повторення алгоритму. Для менш оптимальних

випадків метод збігається набагато швидше, ніж градієнтні методи, особливо, якщо початкова точка знаходиться достатньо близько до шуканого значення.

Для складних функцій, особливо у випадках, коли положення екстремуму невідоме навіть наближено і рух починається з достатньо віддаленої точки, використовується модифікація методу Ньютона–Рафсона з керуванням кроку. За даною модифікацією методу проводиться одновимірний пошук у напрямку $\widehat{H}^{-1}(\vec{x}_0)\cdot \operatorname{grad} f(\vec{x}_0)$, і його результат використовується як нове початкове наближення.

Іншим варіантом є введення додаткового коефіцієнта α, що керує величиною кроку зміни

$$\vec{x}_1 = \vec{x}_0 - \alpha \cdot \widehat{H}^{-1}(\vec{x}_0)\cdot \operatorname{grad} f(\vec{x}_0).$$

Величина коефіцієнта α обирається тим ближче до одиниці, чим ближче стартова точка до екстремального значення.

Використовувати ньютонівські методи краще за все у випадках, коли відомо, що задана функція є квадратичною, або коли приблизно відоме екстремальне значення функції (поблизу екстремумів усі функції добре апроксимуються квадратичними). В загальному випадку ньютонівські методи краще комбінувати з методами нижчих порядків для пошуку початкового наближення.

Метод змінної метрики Давідона–Флетчера–Пауелла

Методи змінної метрики утворюють групу квазіньютонівських методів, які використовують формалізм виразу (3.11), проте замість розрахунку та обернення гесіана використовують інші матриці $\widehat{G}$, що корегуються на кожному кроці:

$$\vec{x}_{k+1} = \vec{x}_k + \widehat{G}_k \cdot \operatorname{grad} f(\vec{x}_0). \qquad (3.12)$$

Одним з найрозповсюдженіших методів змінної метрики є метод Давідона–Флетчера–Пауелла, алгоритм якого на *k*-му кроці виглядає наступним чином:

1. За виразом (3.12) розраховується наступне значення координати x_{k+1}. Для першого кроку покладають $\widehat{G} = \widehat{I}$ (одинична матриця), значення градієнта розраховують безпосередньо.

2. Якщо відхилення $\vec{x}_{k+1} - \vec{x}_k$ є меншим за граничну точність, то процес пошуку зупиняють.

3. У точці $\vec{x}_{k+1}$ розраховують значення градієнта функції $\vec{g}_{k+1} = \text{grad}\, f(\vec{x}_{k+1})$.

4. За величинами координати $\vec{x}_k$ та градієнта $\vec{g}_k$ розраховують значення $\vec{s}_k = \vec{x}_{k+1} - \vec{x}_k$ та $\vec{q}_k = \vec{g}_{k+1} - \vec{g}_k$.

5. Корегують матрицю для наступного кроку

$$\widehat{G}_{k+1} = \widehat{G}_k + \frac{(\vec{s}_k \cdot \vec{s}_k)}{(\vec{s}_k \cdot \vec{q}_k)} - \frac{\widehat{G}_k \vec{q}_k \vec{q}_k^T \widehat{G}_k^T}{\vec{q}_k^T \widehat{G}_k \vec{q}_k}.$$

6. Збільшують *k* на одиницю та переходять до першого пункту алгоритму.

Метод Давідона–Флетчера–Пауелла має дещо меншу швидкість пошуку, ніж формальні ньютонівські методи, але має меншу розрахункову складність за рахунок відмови від гесіана.

СПИСОК ЛІТЕРАТУРИ

1. **Арсенюк, І. Р.** Теорія алгоритмів [Текст] / І. Р. Арсенюк, В. В. Колодний, А. А. Яровий. – Вінниця : ВНТУ, 2006. – 150 с.

2. **Бахвалов, Н. С.** Численные методы [Текст] / Н. С. Бахвалов, Н. П. Жидков, Г. М. Кобельков. – М. : Наука, 1987. – 600 с.

3. **Бахвалов, Н. С.** Численные методы [Текст] / Н. С. Бахвалов. – М. : Наука, 1973. – 632 с.

4. **Березин, И. С.** Методы вычислений [Текст] : в 2 т. / И. С. Березин, Н. П. Жидков. – М. : ФМЛ, 1962.

5. **Васильев, Ф. П.** Численные методы решения экстремальных задач [Текст] : учеб. пособие для вузов / Ф. П. Васильев. – М. : Наука, 1988. – 552 с.

6. **Вирт, Н.** Алгоритмы и структуры данных [Текст] / Н. Вирт. – М. : Мир, 1989. – 360 с.

7. **Глибовець, М. М.** Основи комп'ютерних алгоритмів [Текст] / М. М. Глибовець. – К. : Видавничий дім "КМ Академія", 2002. – 452 с.

8. **Дэвенпорт, Г.** Высшая арифметика. Введение в теорию чисел [Текст] : [пер. с англ.] / Г. Дэвенпорт. – М. : Наука, 1965. – 176 с.

9. **Калиткин, Н. Н.** Численные методы [Текст] / Н. Н. Калиткин. – М. : Наука, 1978. – 512 с.

10. **Кормен, Т.** Алгоритмы: построение и анализ [Текст] / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М. : МЦНМЩ, 2001. – 955 с.

11. **Марков, А. А.** Теорія алгоритмів [Текст] / А. А. Марков, Н. М. Нагорний. – К., 1984. – 420 с.

12. **Нильсен, М.** Квантовые вычисления и квантовая информация [Текст] : [пер. с англ.] / М. Нильсен, И. Чанг. – М. : Мир, 2006. – 824 с.

13. **Самарский, А. А.** Численные методы [Текст] / А. А. Самарский, А. В. Гулин. – М. : Наука, 1989. – 432 с.

14. **Gathen, Joachim von zur.** Modern Computer Algebra [Text] / Joachim von zur Gathen, Jorgen Gerhard. – New York : Cambridge University Press, 2013. – 812 p.

ЗМІСТ