ЛЕКЦІЯ 1. ВСТУП

Розвинута в XX ст. теорія алгоритмів стала базою для теорії обчислювальних машин, теорії та практики програмування, математичної логіки, інформатики. Предметом вивчення цієї науки є алгоритми.

Алгоритм - це формально описана обчислювальна процедура, що отримує вхідні дані, які називають також входом алгоритму, або його аргументом, і видає результат обчислень на вихід.

Слово "алгоритм" походить від імені узбецького вченого аль-Хорезмі, який у IX ст. розробив правила виконання чотирьох арифметичних дій над числами в десятковій системі числення. Сукупність цих правил у Європі почали називати "алгоризм". Пізніше цей термін перетворився в "алгоритм" (іноді "алгорифм") і ним називали правила розв'язування різних задач. Першим алгоритмом, який дійшов до нас, у його інтуїтивному розумінні як скінченної послідовності елементарних дій, які розв'язують поставлену задачу, вважають запропонований Евклідом у III ст. до нашої ери алгоритм знаходження найбільшого спільного дільника (НСД) двох чисел.

Аж до 30-х років XX ст. поняття алгоритму мало швидше методологічне, ніж математичне значення. Під алгоритмом розуміли скінченну сукупність точно сформульованих правил, які дають змогу розв'язувати задачі певного класу. Таке визначення алгоритму не є строгим, а скоріше інтуїтивним, оскільки спирається на інтуїтивні поняття (правило, клас задач). У 20-х роках XX ст. задача строгого визначення алгоритму стала однією з центральних математичних проблем.

Початковою точкою відліку сучасної теорії алгоритмів можна вважати теорему про неповноту символічних логік, доведену німецьким математиком К. Геделем 1931 р. У цій роботі з'ясовано, що деякі математичні проблеми не можуть бути розв'язані алгоритмами з певного класу. Загальність результатів К. Геделя пов'язана з питанням про те, чи тотожний використовуваний ним клас алгоритмів з класом усіх алгоритмів в інтуїтивному понятті цього терміна.

Зазначена праця дала поштовх до пошуку й аналізу різних формалізацій поняття "алгоритм".

Перші фундаментальні праці з теорії алгоритмів опубліковані в середині 30-х років XX ст. Аланом Тьюрінгом, Алоізом Черчем і Емілем Постом. Запропоновані ними машина Тьюрінга, машина Поста і клас рекурсивних функцій Черча стали першими формальними описами алгоритму, які використовували строго визначені моделі обчислень. Сформульовані гіпотези Поста і Черча-Тьюрінга постулювали еквівалентність запропонованих ними моделей обчислень як формальних систем та інтуїтивного поняття алгоритму.

Важливим розвитком цих праць стало формулювання і доведення існування алгоритмічно нерозв'язних проблем.

У 1950-х роках суттєвий внесок у розвиток теорії алгоритмів зробили праці А. Колмогорова і алгоритмічний формалізм Маркова, що ґрунтується на теорії формальних граматик. Формальні моделі алгоритмів Поста, Тьюрінга і Черча, як і моделі Колмогорова і Маркова, виявились еквівалентними в тому розумінні, що будь-який клас проблем, розв'язний в одній моделі, буде розв'язним і в іншій.

Поява доступних комп'ютерів і суттєве розширення кола задач, які можна розв'язати за їхньою допомогою, привели в 1960-1970-х роках до практично значимих досліджень алгоритмів і обчислювальних задач. На цій підставі тоді оформились такі розділи в теорії алгоритмів:

- класична теорія алгоритмів (формулювання задач у термінах формальних мов, поняття розв'язності задачі, опис класів задач за складністю, формулювання у 1965 р. Ж.Едмондсом проблеми $P = NP$ відкриття класу NP-повних задач і його дослідження; введення нових моделей обчислень - алгебричного дерева обчислень (Бен-Ор), машин з довільним доступом до пам'яті, схем алгоритмів Янова, стандартних схем програм Котова та ін. [??];
- теорія асимптотичного аналізу алгоритмів (поняття складності алгоритму, критерії оцінки алгоритмів, методи отримання асимптотичних оцінок, зокрема для рекурсивних алгоритмів, асимптотичний аналіз

трудомісткості або часу виконання, отримання теоретичних нижніх оцінок складності задач), у розвиток якої суттєвий внесок зробили Д. Кнут, А. Ахо, Дж. Хопкрофт, Дж. Ульман, Р. Карп, Л. Кудрявцев та ін. [??];

- теорія практичного аналізу обчислювальних алгоритмів (отримання явних функцій трудомісткості, інтервальний аналіз функцій, практично значимі критерії якості алгоритмів, методики вибору раціональних алгоритмів); основоположною в цьому напрямі вважають фундаментальну працю Д. Кнута "Мистецтво програмування" [??].

З теорією алгоритмів тісно пов'язані дослідження, що стосуються розробки методів створення ефективних алгоритмів (динамічне програмування, метод гілок і меж, метод декомпозиції, "жадібні" алгоритми, спеціальні структури даних тощо) [??].

Отже, на сучасному рівні теорія алгоритмів є дисципліною теоретичної математики, яка надає їй апарат для дослідження розв'язності проблем, і дисципліною прикладної математики, яка безпосередньо вивчає певні явища реального світу.

Поняття алгоритму є концептуальною основою різноманітних процесів опрацювання інформації, бо власне наявність відповідних алгоритмів і забезпечує можливість автоматизації таких процесів. Разом з математичною логікою теорія алгоритмів утворює теоретичний фундамент сучасних обчислювальних наук.

Теорія алгоритмів тісно пов'язана не лише з інформатикою та програмуванням, а й з лінгвістикою, економікою, психологією та іншими науками. У рамках теорії алгоритмів сформувалось багато нових розділів, які мають яскраво виражений прикладний напрям - теорія алгоритмічних мов, складність алгоритмів і обчислень, аналіз і верифікація алгоритмів та програм тощо. Ці розділи, поряд з іншими теоретичними напрямами інформатики, утворюють теоретичну базу опрацювання даних з використанням комп'ютерів.

У загальній теорії алгоритмів виділяють дві сторони:

- дескриптивну, яка вивчає питання наявності чи відсутності алгоритмів і числень, які приводять до заданої мети (але без оцінки затрат на досягнення цієї мети), і способи задання цих алгоритмів та числень;
- метричну, яка займається оцінюванням складності процесів обчислення. Ця сторона теорії алгоритмів сьогодні ще не склалась у єдину чітку систему.

Необхідність формалізації поняття алгоритму

Інтуїтивне поняття алгоритму, наведене у вступі, є хоча не строгим, проте настільки ясним, що практично завжди очевидно, чи буде алгоритмом той чи інший процес. У загальному вигляді проблему знаходження алгоритму можна сформулювати так: "задано певний клас початкових даних і задача, для якої ці початкові дані допустимі. Треба знайти алгоритм, який розв'язує цю задачу".

Задачу, для якої знайдено алгоритм розв'язування, називають *розв'язною* (такою, яку можна розв'язати). Щоб довести розв'язність задачі, достатньо побудувати відповідний алгоритм, а тому достатньо інтуїтивного поняття алгоритму. Для того ж, щоб довести, що задача є *нерозв'язною* (не існує алгоритму для її розв'язування), такого формулювання не достатньо. Для цього необхідно точно знати, що таке алгоритм, визначити це поняття чітко математично.

Є велика кількість задач, розв'язок яких не знайдено. Наприклад, *проблема Гольдбаха*: знайти алгоритм, який дав би змогу для довільного цілого числа n ($n > 6$) знайти хоча б один розклад на три прості доданки. Для непарних n ця проблема розв'язана 1937 р. І. Виноградовим, однак для парних n вона нерозв'язна й дотепер.

Деякі задачі на знаходження алгоритму, які довго не піддавались розв'язанню, виявились такими, що не можуть бути розв'язані, і це строго доведено. До таких задач, наприклад, належать три стародавні геометричні проблеми:

- *задача про квадратуру круга* (за допомогою циркуля і лінійки знайти метод побудови квадрата, рівновеликого заданому кругу);

- *задача про трисекцію кута* (за допомогою циркуля і лінійки знайти метод ділення довільного кута на три однакові частини);
- *задача про подвоєння куба* (знайти метод, який дає змогу на стороні довільного куба за допомогою циркуля і лінійки побудувати сторону куба, об'єм якого вдвічі більший від об'єму заданого).

Десята проблема Гільберта: побудувати алгоритм, який дає змогу для довільного діофантового рівняння $F(x, y, \ldots) = 0$ визначити, чи має воно цілочисловий розв'язок. Ця проблема сформульована 1901 р., а її нерозв'язність строго доведена 1970 р. Ю. Матіясевичем. Тут $F(x, y, \ldots) = 0$ - поліном з цілими показниками степенів і цілими коефіцієнтами. Наприклад, рівняння $x^2 + y^2 - z^2 = 0$ має цілочисловий розв'язок $x = 3, y = 4, z = 5$, а рівняння $6x^{18} - x + 3 = 0$ не має цілочислового розв'язку.

Насправді алгоритмічна нерозв'язність певного класу задач означає, що алгоритм їхнього розв'язування жодним способом ніколи і ніким не буде побудовано. А це потребує обґрунтування у вигляді математичного доведення. Таке доведення можна побудувати лише тоді, коли поняття алгоритму є об'єктом математичної теорії.

Проблема формалізації поняття алгоритму стала однією із центральних математичних проблем у середині 30-х років XX ст. Її вирішенням займались Д. Гільберт, К. Гедель, А. Черч, С.-К. Кліні, А. Тьюрінг, Е. Пост, пізніше А. Марков, А. Колмогоров та ін. Результати їхніх досліджень започаткували нову математичну теорію - теорію алгоритмів, яка вивчає різні алгоритмічні системи.

Для чого вивчати алгоритми

По-перше, алгоритми є життєво необхідними складовими для рішення будь-яких задач з різноманітних напрямків комп'ютерних наук. Алгоритми відіграють ключову роль у сучасному розвитку технологій. Тут можна згадати такі розповсюджені задачі, як:

- розв'язання математичних рівнянь різної складності, знаходження добутку матриць, обернених матриць;
- знаходження оптимальних шляхів транспортування товарів та людей;
- знаходження оптимальних варіантів розподілення ресурсів між різними вузлами (виробниками, верстатами, працівниками, процесорами тощо);
- знаходження в геномі послідовностей, які співпадають;
- пошук інформації в глобальній мережі Інтернет;
- прийняття фінансових рішень в електронній комерції;
- обробка та аналіз аудіо та відео інформації.

Цей список можна продовжувати й продовжувати і, власно кажучи, майже неможливо знайти таку галузь комп'ютерних наук та інформатики, де б не використовувались ті або інші алгоритми.

По-друге, якісні та ефективні алгоритми можуть бути каталізаторами проривів у галузях, які є на перший погляд далекими від комп'ютерних наук (квантова механіка, економіка та фінанси, теорія еволюції).

І, по-третє, вивчення алгоритмів це також неймовірно цікавий процес, який розвиває наші математичні здібності та логічне мислення.

Ефективність алгоритмів

Припустимо, швидкодія комп'ютеру та об'єм його пам'яті можна збільшувати до нескінченності. Чи була би тоді необхідність у вивченні алгоритмів? Так, але тільки для того, щоб продемонструвати, що метод розв'язку має скінченний час роботи і що він дає правильну відповідь. Якщо б комп'ютери були необмежено швидкими, підійшов би довільний коректний метод рішення задачі. Звісно, тоді найчастіше обирався би метод, який

найлегше реалізувати.

Сьогодні є дуже потужні комп'ютери, але їх швидкодія не є нескінченно великою, як і пам'ять. Таким чином, час обчислення – це такий само обмежений ресурс, як і об'єм необхідної пам'яті. Цими ресурсами слід користуватись розумно, чому й сприяє застосування алгоритмів, які ефективні в плані використання ресурсів часу та пам'яті.

Алгоритми, які розроблені для розв'язання однієї та тієї самої задачі, часто можуть дуже сильно відрізнятись за ефективністю. Ці відмінності можуть бути набагато більше помітними, чим ті, які викликані застосуванням різного апаратного та програмного забезпечення.

Як зазначалось вище, в цьому розділі центральну роль буде присвячено задачі сортування. Перший алгоритм, який буде розглядатись – *сортування включенням,* для своєї роботи вимагає часу, кількість якого оцінюється як $c_1 n^2$, де n – розмір вхідних даних (кількість елементів у послідовності для сортування), c_1 – деяка стала. Цей вираз вказує на те, як залежить час роботи алгоритму від об'єму вхідних даних. У випадку сортування включенням ця залежність є квадратичною. Другий алгоритм – *сортування злиттям* – потребує часу, кількість якого оцінюється як $c_2 n \log_2 n$. Зазвичай константа сортування включенням менше константи сортування злиттям, тобто $c_1 < c_2$, але як ми пересвідчимось у наступних темах, ці константи не відіграють ролі у порівнянні швидкодії різних алгоритмів. Адже зрозуміло, що функція n^2 зростає швидше зі збільшенням n, аніж функція $n \log_2 n$. І для деякого значення $n = n_0$ буде досягнуто такий момент, коли вплив різниці констант перестане мати значення і надалі функція $c_2 n \log_2 n$ буде менша за $c_1 n^2$ для будь-яких $n > n_0$.

Для демонстрації цього розглянемо два комп'ютери – А та В. Комп'ютер А більш швидкий і на ньому працює алгоритм сортування включенням, а комп'ютер В більш повільний і на ньому працює алгоритм сортування методом злиття. Обидва комп'ютери повинні виконати сортування множини, яка складається з мільйона чисел. Припустимо, що комп'ютер А виконує

мільярд операцій в секунду, а комп'ютер B – лише десять мільйонів, тобто A працює в 100 разів швидше за B. Щоб різниця стала більш відчутною, припустимо що код методу включення написаний найкращим програмістом у світі із використанням команд процесору, і для сортування n чисел за цим алгоритмом потрібно виконати $2n^2$ операцій (тобто $c_1 = 2$). Сортування методом злиття на комп'ютері B написано програмістом початківцем із використанням мови високого рівня і отриманий код потребує $50n\log_2 n$ операцій (тобто $c_2 = 50$). Таким чином для сортування мільйона чисел комп'ютеру A буде потрібно

$$\frac{2 \cdot \left(10^6\right)^2 \text{ операцій}}{10^9 \text{ операцій/сек}} = 2000 \text{ сек}$$

а комп'ютеру B

$$\frac{50 \cdot 10^6 \log_2 10^6 \text{ операцій}}{10^7 \text{ операцій/сек}} \approx 100 \text{ сек}$$

Тож, використання коду, час роботи якого зростає повільніше, навіть при поганому комп'ютері та поганому компіляторі потребує на порядок менше процесорного часу! Для сортування 10 мільйонів чисел перевага сортування злиттям стає ще більш відчутною: якщо сортування включенням потребує для такої задачі приблизно 2,3 дня, то для сортування злиттям – менше 20 хвилин. Загальне правило таке: чим більша кількість елементів для сортування, тим помітніше перевага сортування злиттям.

Наведений вище приклад демонструє, що алгоритми, як і програмне забезпечення комп'ютера, являють собою *технологію*. Загальна продуктивність системи настільки ж залежить від ефективності алгоритму, як і від потужності апаратних засобів.

Алфавітні оператори й алгоритми

Абстрактні алфавіти й алфавітні оператори.

Абстрактним алфавітом називають довільну скінченну сукупність елементів.

Природа елементів, які називають *буквами* алфавіту, може бути довільною, проте вони повинні бути попарно різними, а їхня кількість - скінченною. 13.

Наприклад, алфавітами є . $A_1 = \{a, b, c, \ldots x, y, z\}$ та $A_2 = \{0,1\}$.

Слово у заданому алфавіті - це довільна скінченна впорядкована послідовність букв цього алфавіту. *Довжина слова* - кількість букв у слові.

Наприклад, слова в алфавіті A_1 – будь-яке слово латинкою. Будь-яке двійкове число є словом в алфавіті A_2.

Множина слів, яку можна побудувати у скінченному алфавіті, є зліченною. Поряд зі словами додатної довжини розглядають *порожнє слово* (позначимо його Λ - лямбда велике), яке не містить жодної букви і має довжину 0.

Алфавіти можна *розширювати*, додаючи до них нові букви. Якщо до алфавіту української мови додати пропуск і знаки пунктуації, то цілу сторінку можна розглядати як одне слово.

Слова можна впорядковувати в *лексикографічному порядку*, що визначений порядком букв в алфавіті.

На множині слів визначено дві операції: *конкатенація* (приєднання) та *підстановка* (заміна).

Конкатенацією двох слів *A* та *B* називають слово *AB*, отримане приписуванням слова B до слова A. Ця операція не комутативна, проте асоціативна.

Приклад. Конкатенацією слів *A = abc* та *B = xy* буде слово *AB = abcxy*.

Слово *A* називають *підсловом* слова *B*, якщо *B* можна записати в такому вигляді: *B = CAD*, де *C* та *D* - деякі слова, можливо, порожні. Слово *A* є *входженням* у слово *B*.

Для деяких пар слів *A* та *B* можна навести кілька розкладів вигляду *B = CAD*. Наприклад, нехай *B = xyxyxy*, *A = xy*, тоді існує три розклади: *B = Axyxy, B = xyAxy, B = xyxyA.*

Для усунення такої неоднозначності вводять поняття *i*-го входження підслова у задане слово.

Якщо в розкладі *B = CAD* підслово *C* має мінімальну довжину, то таке входження слова *A* в слово *B* називають *першим (канонічним)* входженням.

Нехай задано слова *A, B, F* і *A* є *i*-м входженням у слово *B = CAD*. Тоді слово *C* отримують *операцією заміни (підстановки) i*-го входження слова *A* словом *F*, якщо *G = CFD*. Якщо в операції заміни $i = 1$, то підстановку називають *стандартною* (або *канонічною*).

Алфавітним оператором, або *алфавітним відображенням*, φ називають відповідність між словами в одному або різних алфавітах.

Функцію називають *словниковою*, якщо вона перетворює слово одного алфавіту в слово іншого алфавіту.

Нехай $\{P\}_X$ - деяка множина вхідних слів в алфавіті *X*, $\{Q\}_Y$ - множина вихідних слів в алфавіті *Y*. Тоді алфавітним оператором φ є словникова функція такого вигляду $\varphi : \{P\}_X \to \{Q\}_Y$.

Приклад. Якщо φ_1 - дзеркальне відображення вхідних даних, то $abcd \xrightarrow{\varphi_1} dcba$. Якщо φ_2 - оператор диференціювання, то $\sin x + \cos x \xrightarrow{\varphi_2} \cos x - \sin x$.

Нехай *P* - вхідне слово. Якщо слову *P* оператор φ не ставить у відповідність жодного вихідного слова, то кажуть, що на слові *P* оператор φ *не визначений*.

Сукупність усіх слів, на яких оператор φ визначений, називають областю *визначення (областю застосування)* оператора φ.

Сукупність усіх слів, на яких оператор φ невизначений, називають *областю заборони* φ.

Завжди можна вважати, що вхідний алфавіт X і вихідний алфавіт Y оператора φ є одним і тим же алфавітом, оскільки їх можна об'єднати в один спільний алфавіт $Z = X \cup Y$ і розглядати оператор $\varphi : \{X\}_Z \to \{Q\}_Z$.

Проте розширення вхідного алфавіту не розширює області визначення оператора φ.

Розрізняють *однозначні* та *багатозначні* оператори.

Однозначний оператор кожному вхідному слову ставить у відповідність не більше одного вихідного слова, а *багатозначний* оператор - декілька різних вихідних слів:

$$P \to \begin{cases} Q_1 \\ Q_2 \\ \vdots \\ Q_n \end{cases}$$

У цьому разі вибір вихідного слова відбувається або випадково, або відповідно до ймовірностей, приписаних словам $Q_1, Q_2, \ldots, Q_n$ (ймовірнісний оператор). Зазначимо, що розподіл ймовірностей між $Q_1, Q_2, \ldots, Q_n$ повинен змінюватися в процесі опрацювання інформації. В іншому випадку багатозначний оператор вироджується в однозначний, оскільки за вихідне слово завжди вибирають одне і те ж слово Q_1, яке має найбільшу ймовірність.

Довільні процеси перетворення інформації можуть бути зведені до поняття алфавітного оператора. Наприклад, процеси перекладу з однієї мови іншою, написання рефератів, статей та інші перетворення лексичної інформації можна розглядати як реалізацію багатозначних алфавітних операторів.

Кодувальні алфавітні оператори.

Серед алфавітних операторів особливу роль відіграють *кодувальні оператори*. Кодування застосовують, головно, для того, щоб слова в довільних алфавітах можна було задавати словами в деякому фіксованому, стандартному алфавіті.

Нехай A - деякий алфавіт, який називають *стандартним*, B - довільний алфавіт. Нехай $\{L\}_A$, $\{M\}_B$ - множини слів у відповідних алфавітах. Кажуть, що множина $\{M\}_B$ *закодована* в алфавіті A, якщо задано такий однозначний оператор: $\varphi:\{M\}_B \to \{L\}_A$.

Оператор φ називають *кодувальним*, а слова з $\{L\}_A$ - *кодами об'єктів* з $\{M\}_B$.

Легко довести таке твердження.

Теорема. Кодувальний оператор є взаємно однозначним тоді й тільки тоді, коли:

- коди різних букв алфавіту B різні;
- код довільної букви алфавіту B не може бути першим входженням у коди інших букв цього алфавіту.

Умова 2 виконується автоматично, якщо коди мають однакову довжину.

Кодування об'єктів алфавіту B словами однакової довжини називають *нормальним кодуванням*.

Отже, у разі виконання умови 1 нормальне кодування забезпечує взаємну однозначність оператора кодування φ.

Розглянемо декілька кодувань, які трапляються найчастіше.

Кодування слів у багатобуквеному алфавіті. Нехай B - довільний алфавіт з n букв, A - стандартний алфавіт з m букв ($m > 1$). Для довільної пари (n, m) завжди можна задати таке число l, що $m^l \geq n$. Проте відомо, що величина m^l визначає кількість різних слів довжини l в m-буквеному алфавіті. Отже, дана нерівність засвідчує, що всі букви алфавіту B завжди можна закодувати словами довжини l в алфавіті A так, щоб коди різних букв були різними. Кожне таке кодування є нормальним і забезпечує взаємну однозначність оператора кодування φ.

Приклад. Якщо $n = 16$, $m = 2$ (наприклад, $A = \{0,1\}$), то $l = 4$, $2^4 \geq 16$ і кожну з 16 букв можна закодувати словами довжини 4: 0000, 0001,...,1111.

Власне ці коди використовують для кодування цифр шістнадцяткової системи в комп'ютері.

Кодування безконечних алфавітів.

Згідно з визначенням алфавіту, він повинен бути скінченною множиною. Проте на практиці зручно розглядати безконечні алфавіти, які складаються зі скінченної кількості букв з індексами, які набувають натуральних значень 0, 1, 2,…. Такий алфавіт формально безконечний. Наприклад, якщо алфавіт A складається з символів $a,b,c,g,\ldots,rf_j,d_j^i$ $(i,j=0,1,2,\ldots)$, то від A можна перейти до скінченного алфавіту $B=\{a,b,c,g,\ldots,r,f,d\}\cup\{\alpha,\beta\}$ $\{\alpha,\beta\}\notin A$ і кодувати букви A в алфавіті B так:

1. символи $a,b,\cdots,r$;
2. символи f_j,d_j^i кодують словами $f_j = f\underbrace{a\ldots a}_{j}$, $d_j^i = d\underbrace{\alpha\ldots\alpha}_{j}\underbrace{\beta\ldots\beta}_{i}$.

Тоді скінченний алфавіт B завдяки такому кодуванню дає змогу виписати коди як скінченні слова для довільних $f_jd_j^i\in A$ з конкретними значеннями i, j, які б великі вони не були.

Отже, довільні слова у безконечному алфавіті, де безконечність виникає внаслідок індексації букв, можна закодувати у скінченному алфавіті.

Способи задання алфавітних операторів. Поняття алгоритму.

Важливим моментом для алфавітних операторів є спосіб їхнього задання. Розрізняють два способи задання операторів.

1. Табличний спосіб задання операторів. Такий спосіб застосовують тоді, коли область визначення оператора скінченна. Оператор φ задають *таблицею відповідності*, у якій для кожного вхідного слова P з області визначення φ виписано відповідне вихідне слово Q:

$$\begin{array}{l} P_1 \to Q_1, \\ \ldots \quad . \\ P_n \to Q_n \end{array}$$

У деяких випадках табличний спосіб можна використовувати і для задання операторів з безконечною областю визначення. Наприклад, $\underbrace{xx\ldots x}_{n} \to \underbrace{yy\ldots y}_{n+1}\,(n=1,2,\ldots)$. Проте в загальному випадку для задання операторів з безконечною областю визначення табличний спосіб принципово не підходить.

2. Задання оператора скінченною системою правил, яка дає змогу за скінченну кількість кроків знайти значення φ на довільному вхідному слові, наприклад, система правил для додавання натуральних чисел у системі числення з основою p або система правил знаходження найбільшого спільного дільника двох додатних чисел та ін.

Алфавітний оператор, заданий скінченною системою правил, називають *алгоритмом*.

Очевидно, що оператор, заданий табличним способом, теж є алгоритмом.

Відмінність у поняттях алфавітного оператора й алгоритму полягає в тому, що в понятті оператора суттєвою є лише відповідність, яка усталюється між вхідними і вихідними словами, і не задано правил, які реалізують цю відповідність. У понятті алгоритму головним є спосіб задання відповідності. Тобто оператор визначає, що треба зробити, а алгоритм відображає не лише що, але і як це треба зробити.

Отже, алгоритм - це алфавітний оператор разом із системою правил, яка визначає його дію:

$$A = \langle \varphi, P \rangle, P - \text{система правил}.$$

Два алфавітні оператори називають *рівними* якщо вони мають одну і ту ж область визначення й однаковим вхідним словам з цієї області ставлять у відповідність однакові вихідні слова.

Два алгоритми $A_1 = \langle \varphi_1, P_1 \rangle$, $A_2 = \langle \varphi_2, P_2 \rangle$ називають *рівними*, якщо відповідні їм алфавітні оператори рівні та системи правил збігаються:

$$A_1 = A_2 \Leftrightarrow \varphi_1 = \varphi_2, P_1 = P_2.$$

Два алгоритми A_1, A_2, називають *еквівалентними*, якщо відповідні їм оператори рівні, а системи правил різні:

$$A_1 \cong A_2 \Leftrightarrow \varphi_1 = \varphi_2, P_1 \neq P_2.$$

Еквівалентні алгоритми задають розв'язок однієї й тієї ж задачі, проте різними способами.

Властивості алгоритмів. Способи композиції алгоритмів

Основні властивості алгоритмів.

В алгоритмі $A = \langle \varphi, P \rangle$ система P - це не просто довільний перелік скінченної кількості правил, які визначають послідовність виконання операцій під час розв'язування певної задачі. Система P повинна мати властивості, притаманні кожному алгоритму: дискретність, ефективність, скінченність, результативність, масовість.

Дискретність алгоритму. Алгоритм описує процес послідовної побудови величин, який відбувається в дискретному часі. У початковий момент t_0 задають скінченну систему - σ_0, вхідних величин. Далі в кожен інтервал часу (t_i, t_{i+1}) із системи σ_i; величин, які були в момент t_i, за певним правилом отримують систему величин σ_{i+1}. Перехід від σ_i до σ_{i+1} вважають елементарним кроком алгоритму. Інтервали (t_i, t_{i+1}) є різними для різних елементарних кроків.

Ефективність алгоритму. Елементарні кроки, які необхідно зробити в алгоритмі, повинні бути ефективними, тобто виконуваними точно і за короткий відрізок часу. Цю властивість ще називають *зрозумілістю* алгоритму. Це означає, що кроки алгоритму повинні містити лише операції з набору операцій виконавця. Тобто різні виконавці, згідно з алгоритмом, повинні діяти однаково і прийти до одного й того ж результату.

Скінченність. Алгоритм завжди повинен закінчуватися після скінченної (можливо, дуже великої) кількості кроків.

Результативність. Алгоритм завжди забезпечує отримання певного результату розв'язування задачі, що є наслідком скінченної кількості елементарних кроків та їхньої ефективності. Якщо деякий елементарний крок

не дає результату, то має бути зазначено, що саме треба вважати результатом алгоритму.

Масовість. Алгоритм повинен бути застосовним до цілого класу задач, а не до одної задачі. Це означає, що початкову систему величин σ_0, можна вибирати з деякої потенційно безконечної множини.

Якщо система P з пари $A=\langle \varphi, P\rangle$ задовольняє всі перелічені вище властивості, крім властивості скінченності, то таку пару називають *обчислювальним методом*.

Обчислювальним методом, наприклад, є процес ділення двох натуральних чисел.

Алгоритми, у яких кількість кроків скінченна, але дуже велика, доцільно вважати обчислювальними методами.

Різновиди алгоритмів

Наявність чи відсутність у системі правил P тієї чи іншої властивості розбиває множину всіх алгоритмів на два класи щодо конкретної властивості.

Розглянемо деякі з цих властивостей - *детермінованість, самозмінність, самозастосовність, універсальність*.

1. Нехай система P є такою, що система величин σ_{i+1}, однозначно визначена системою σ_i. Алгоритм $A=\langle \varphi, P\rangle$ називають *детермінованим (визначеним)*, якщо P задовольняє цю властивість, і *недетермінованим* в іншому випадку.

У детермінованому алгоритмі $A=\langle \varphi, P\rangle$ алфавітний оператор φ є однозначним, у недетермінованому - багатозначним.

2. Алгоритм $A=\langle \varphi, P\rangle$ називають *самозмінним*, якщо в процесі переробки алгоритмом вхідних слів система P змінюється, і *несамозмінним* в іншому випадку.

Результат дії самозмінного алгоритму залежить не лише від заданого вхідного слова P_k, а й від того, які саме слова $P_1,\ldots,P_{k-1}$ були опрацьовані

алгоритмом до подання на вхід слова P_k. Кожна послідовність $P_1,\ldots,P_{k-1}$ генерує різні зміни в системі правил P.

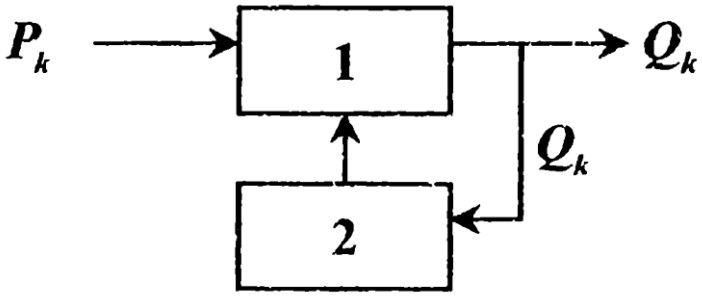

Рис. 1.1.

Самозмінний алгоритм можна розглядати як систему двох алгоритмів, перший з яких (робочий) виконує переробку вхідних слів, а другий (керівний) вносить зміни в робочий алгоритм (рис. 1.1).

Частковим випадком самозмінних алгоритмів є алгоритми, які *самонавчаються*. В таких алгоритмах закладені лише найзагальніші принципи опрацювання інформації, а детальний алгоритм формується в процесі його роботи залежно від характеру інформації, яка надходить.

3. Нехай система правил P алгоритму $A=\langle\varphi,P\rangle$ закодована певним способом у вхідному алфавіті алгоритму A. Позначимо цей код P^{cod}.

Алгоритм $A=\langle\varphi,P\rangle$ називають самозастосовним, якщо слово P^{cod} входить в область визначення A, і несамозастосовним в іншому випадку.

Приклад. Самозастосовним алгоритмом є тотожний алгоритм у довільному алфавіті A, що містить не менше двох букв, який переробляє довільне слово в себе.

Несамозастосовним алгоритмом є нульовий алгоритм, який порожнє слово перетворює в букву з заданого алфавіту: $\Lambda\to a,\ a\in A,\ \Lambda-$порожнє слово. Цей алгоритм не застосовний до жодного слова, і тому й до P^{cod}.

4. Алгоритм називають *універсальним*, якщо він еквівалент- ний довільному наперед заданому алгоритму $A=\langle\varphi,P\rangle$.

Отже, універсальний алгоритм може виконувати роботу довільного алгоритму *A*. Іншими словами, за його допомогою можна реалізувати будь-які алгоритмічні процеси, якими б складними вони не були.

Композиція алгоритмів

Нові алгоритми можуть бути побудовані з уже відомих шляхом застосування різних способів композиції алгоритмів. Є чотири способи композиції алгоритмів: *суперпозиція, об'єднання, розгалуження та ітерація*.

1. Суперпозиція алгоритмів.

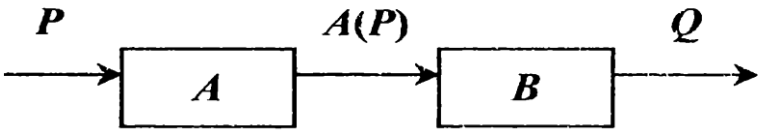

Рис. 1.2.

У випадку суперпозиції двох алгоритмів *A* та *B* вихідне слово одного з них розглядають як вхідне слово іншого (рис. 1.2).

Нехай *D*(*A*) - область визначення алгоритму *A*, а *D*(*B*) - область визначення алгоритму *B*. Результат суперпозиції *A* та *B* можна зобразити у такому вигляді: $C(P) = B(A(P)), P \in D(A), A(P) \in D(B)$.

У цьому випадку область визначення алгоритму *B* повинна включати в себе множину вихідних слів алгоритму *A*.

Суперпозиція може виконуватись для довільної скінченної кількості алгоритмів.

Приклад. Суперпозицією алгоритмів $A(P) = xp$ та $B(P) = Py$ є алгоритм $C(P) = xPy$.

2. Об'єднання алгоритмів.

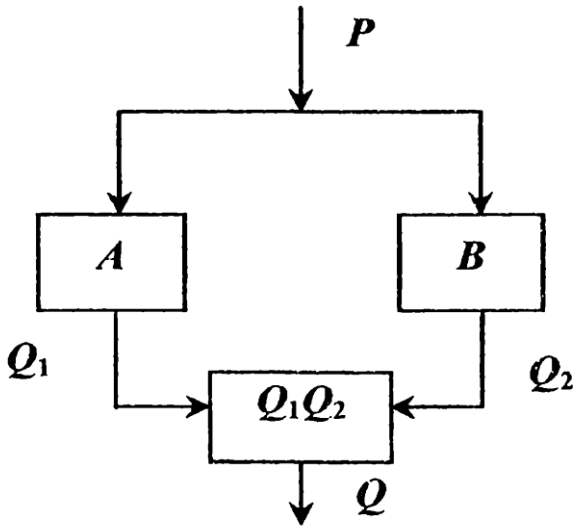

Рис. 1.3.

Розглянемо два алгоритми A та B в одному й тому ж алфавіті V. Нехай $D(A)$ та $D(B)$ - області визначення цих алгоритмів.

Об'єднанням алгоритмів A та B є такий алгоритм C в цьому ж алфавіті, який перетворює довільне вхідне слово $P \in D(\mathrm{A}) \cap \mathrm{D(B)}$ в конкатенацію слів $A(P)$ і $B(P)$ (рис. 1.3): $C(P) = A(P)B(P)$. На всіх інших словах алгоритм C вважають невизначеним.

Приклад. Об'єднанням двох алгоритмів $A(P) = xp$ та $B(P) = Py$ буде алгоритм $C(P) = xPPy$.

3. Розгалуження алгоритмів.

Нехай задано три алгоритми A, B, $\underline{C}$ з областями визначення $D(A)$, $D(B)$, $D(C)$, відповідно.

Розгалуженням алгоритмів називають композицію трьох алгоритмів A, B, C, задану співвідношенням,

$$F(\mathrm{P}) = \begin{cases} A(P), \text{ якщо } C(P) = R \\ B(P), \text{ якщо } C(P) \neq R \end{cases},$$

де R - деяке фіксоване слово, переважно порожнє ($R = \Lambda$) (рис. 1.4). Областю визначення отриманого алгоритму буде переріз областей визначення всіх трьох алгоритмів: $D(F) = D(A) \cap D(B) \cap D(C)$.

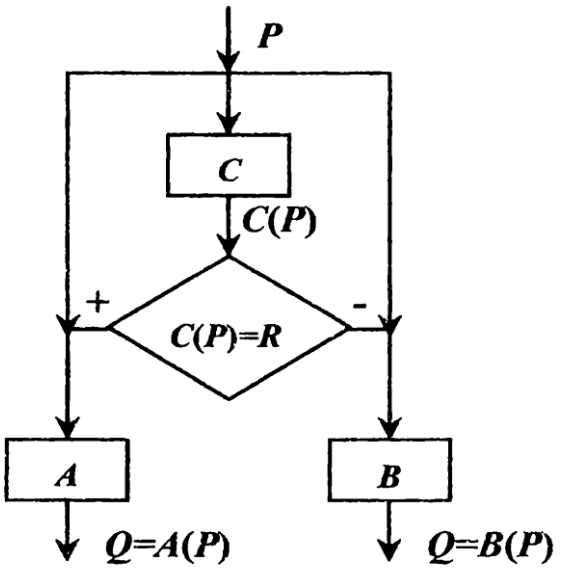

Рис. 1.4.

Приклад. Нехай алгоритми A, B, C задані такими правилами:

$A: ab \to ba; ba \to ab,$

$B: ab \to aab; ba \to aba,$

$C: ab \to a; ba \to \Lambda,$

$R = \Lambda$ (порожнє слово)

Тоді $D(ab) = aab$ і $D(ba) = ab$.

4. Ітерація алгоритмів.

Ітерацією (повторенням) двох алгоритмів A та B називають алгоритм C, який визначають так: для довільного вхідного слова P вихідне слово $C(P)$ отримують як результат послідовного багаторазового застосування алгоритму A доти, доки не буде отримано слово, перетворюване алгоритмом B в деяке фіксоване слово R (рис. 1.5).

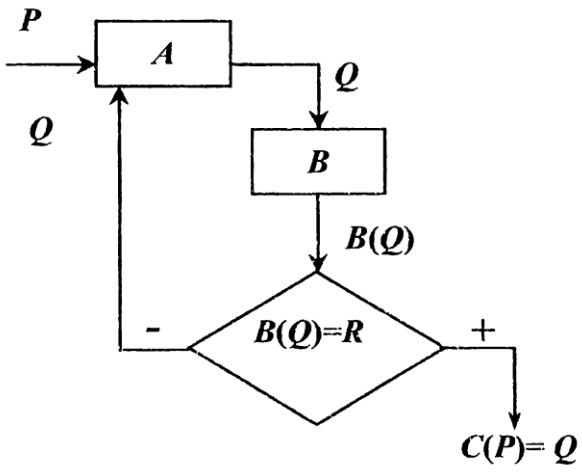

Рис. 1.5.

Отже, у процесі ітерації алгоритму A формується така послідовність слів $Q = Q_0, Q_1, \ldots, Q_n = C(P)$, що $Q_i = A(Q_{i-1})\ (i = 1 \ldots n)$, причому для $i = 1 \ldots n-1$: $B(Q_i) \neq R$, а $B(Q_n) = R$. Області визначення алгоритмів A та B повинні охоплювати множину вихідних слів алгоритму A.

Приклад. Нехай алгоритм A заданий правилом $P_1abP_2 \to P_1baP_2$, де P_1, P_2 - довільні (можливо, порожні) слова в алфавіті $V = \{a, b\}$.

Алгоритм B заданий правилами $bbbaa \to \Lambda, P \to P$, де P - довільне слово в V, $P \neq bbbaa$.

Тоді при $R = \Lambda$: $C(ababb) = bbbaa$.