Лекція 3. ПОНЯТТЯ «АЛГОРИТМ»

Неформальне визначення алгоритму. Найпростіші функціональні можливості числової ЕОМ. Способи зображення алгоритмів. Блок-схеми. Типи алгоритмів: послідовні, з розгалуженням, циклічні. Складність алгоритмів. «Швидкий алгоритм» обчислення степені числа. Трасувальна таблиця.

Неформальне визначення алгоритму, вимоги до алгоритму. Як давно Ви робили ранкову зарядку? Пам'ятаєте, як її роблять? «Поставте ноги на ширину плечей, руки – перед собою...» Чи доводилось Вам самотужки готувати обід? Читали кулінарну книгу? «Візьміть чисту каструлю, наповніть її водою...» Можливо, Ви встановлювали нові програми на свій смартфон і намагались виконати всі побачені на екрані вказівки: «Збережіть отриманий файл. Зайдіть в папку Налаштування і активуйте опцію Змінити налаштування...». Усі згадані описи та інструкції є певними *алгоритмами*.

Справді, нам щодня доводиться мати справу з алгоритмами. Найбільше їх, мабуть, формулювали наші вчителі: як переходити вулицю, як виконати арифметичні дії в стовпчик (до речі, хто знає, як в стовпчик добувати квадратний корінь з числа?), як провести фізичний чи хімічний експеримент. Навіть у цьому курсі лекцій ми вже говорили про алгоритм роботи процесора, про алгоритми переведення чисел у нову систему числення. Але чи ми задумувалися, що воно таке «АЛГОРИТМ»?

До 30-х років минулого століття поняття алгоритму мало швидше методологічне, аніж математичне значення. Алгоритмом називали скінченну сукупність точно сформульованих правил, інструкцій виконавцеві щодо розв'язування задач з певного класу. Перед побудовою будь-якого алгоритму точно відомо, що задано і що треба знайти, обчислити. Тобто, відомо, *яку* задачу треба розв'язати. Алгоритм зазначає *як* це зробити. Для того, щоб чіткіше окреслити інтуїтивне визначення, до алгоритму висувають декілька додаткових вимог:

- *дискретність* – процес побудови розв'язку відбувається в дискретному часі (крок за кроком);
- *зрозумілість* – в описі алгоритму використано набір інструкцій лише з алфавіту виконавця: він вміє виконати кожну з них;
- *формальність* – виконавець може діяти за інструкціями, не вникаючи у суть своїх дій, і це не вплине на результат виконання алгоритму;
- *однозначність* – жодна інструкція не допускає двозначного тлумачення;
- *скінченність* – алгоритм завжди закінчується за скінченну (можливо, велику) кількість кроків;
- *результативність* – виконавець у будь-якому випадку або отримає розв'язок задачі, або доведе його відсутність;
- *масовість* – за допомогою алгоритму можна розв'язати цілу сукупність схожих задач.

Жодна з цих вимог не є зайвою. Повернімося хоча б до перших трьох згаданих прикладів. Алгоритмами виконання фізичних вправ чи приготування їжі користуються дуже багато людей, і кожного цікавить кінцевий результат. Варто зауважити, що не завжди однозначно ми розуміємо інструкції тих алгоритмів, бо «ширина плечей» у кожного своя, слово «каструля» не є терміном, і не зрозуміло, який вона повинна мати об'єм, колір, з якого матеріалу зроблена, як це вплине на смак страви. Така неоднозначність по-різному впливає на результат: доброго фізичного стану можна досягти і без строгого дотримання всіх вимог комплексу вправ, навіть, можна вигадувати власні, обід може вдатися або ні, а от смартфон гарантовано не налаштується, якщо не виконати абсолютно точно хоча б одну інструкцію.

На відміну від речей навколишнього світу, математичні об'єкти строго формалізовані, їхні властивості та операції з ними описані однозначно. Тому впродовж тривалого часу математиків задовольняло інтуїтивне визначення алгоритму, хоча воно не є строго математичним, бо опирається на інтуїтивні поняття «інструкція», «виконавець».

Становище суттєво змінилося, коли на перший план вийшли такі алгоритмічні проблеми, розв'язання яких виявилося сумнівним. Щоб довести існування алгоритму розв'язування задачі, достатньо описати який-небудь процес відшукання розв'язку. Для такого доведення достатньо й інтуїтивного означення алгоритму. Зовсім інша справа – довести неалгоритмізовність задачі. Щоб довести неіснування жодного алгоритму розв'язування задачі, необхідно запровадити строге означення поняття алгоритм.

Нагадаємо класичні приклади алгоритмічно нерозв'язних задач. Задача про *квадратуру круга*: побудувати за допомогою циркуля та лінійки круг, рівновеликий до заданого квадрата. У результаті доведення нерозв'язності цієї задачі було доведено також, що число π – ірраціональне. Задача про *трисекцію кута*: за допомогою циркуля та лінійки поділити заданий плоский кут на три однакові частини – нерозв'язна задача, хоча кожен школяр вміє будувати бісектрису плоского кута. Проблема розпізнавання самозастосовності алгоритму: як визначити, чи будь-який заданий алгоритм можна застосовувати до коду цього ж алгоритму?

У 20-х роках XX ст. завдання строгого визначення алгоритму стало однією з центральних математичних проблем. Її розв'язок отримали в середині 30-х років у працях Д. Гільберта, К. Геделя, А. Чьорча, С.-К. Кліні, Е. Поста, А. Тьюрінга, а трохи пізніше – в працях А. А. Маркова, А. М. Колмогорова та ін. Ці праці стали основою нової галузі математичної науки – теорії алгоритмів.

Згодом ми опишемо алгоритмічну систему Тьюрінга, а поки що користуватимемося інтуїтивним визначенням алгоритму. У програмуванні, як і в математиці, цього вистачає у більшості випадків, адже функціональні можливості комп'ютера досить формалізовані: записати значення в пам'ять (надати змінній значення), виконати арифметичну дію, порівняти два значення, перейти до наступної (чи іншої, зазначеної) команди.

Комп'ютер створювали для автоматизації обчислень, тому в його арсеналі саме такі дії. Розглянемо їх детальніше.

Найпростіші функціональні можливості числової ЕОМ. Сучасні ЕОМ є машинами нейманівського типу з адресованою пам'яттю безпосереднього доступу. Кожен байт пам'яті має власну адресу, проте зручніше працювати не з числовими адресами, а з *іменами*. Тому введено поняття змінної.

Змінна – це іменована ділянка пам'яті комп'ютера, в яку можна поміщати значення, і звідки можна це значення отримувати (копіювати, відтворювати). Ім'я вибирають відповідно до контексту задачі. Ним може бути довільна послідовність букв і цифр, що починається буквою. Кожна змінна має унікальне ім'я, займає один чи більше байт пам'яті, має певний тип – перелік значень, які можна в неї заносити (а інакше кажучи, спосіб трактування її вмісту). Надати змінній адресу і тип – «справа рук» транслятора. Для запису алгоритмів нам, поки що, вистачить тільки імен.

Порядок використання змінної подібний до використання аудіокасети. Щойно куплена касета порожня (можна казати, що вміст її невизначений). Далі на плівку записують, наприклад, улюблену мелодію – тепер касету можна використовувати для багаторазового відтворення мелодії (запис від цього не зникає). Через деякий час на ту саму касету можна записати щось інше. При цьому попередня мелодія витреться, а замість неї буде нова. Так само зі змінною: вміст щойно оголошеної змінної невизначений, у змінну можна записати певне значення і зчитувати його потрібну кількість разів без зміни чи пошкодження цього значення, у змінну можна записати нове значення, витерши попереднє.

Надати змінній нового значення можна за допомогою інструкції *присвоїти* їй це значення, або інструкції *прочитати* це значення (з файла, з клавіатури). Інструкцію присвоєння позначають символом «$\leftarrow$» або «**:=**». Наприклад, «$S \leftarrow 0$», чи «$S := 0$». Ці інструкції (оператори) можна читати так: «записати 0 у змінну S», або «змінній S присвоїти (призначити) значення 0». У кожному разі оператор присвоєння діє «справа наліво»: беремо 0 і поміщаємо його в S, а не навпаки, не «називаємо нуля буквою S». Нагадаємо, що попередній вміст S безповоротно втрачається – це не проблема, а правило.

Інструкція присвоїти число завжди, у всіх випадках використання, записує у змінну те саме значення («0» в «*S*» у попередньому прикладі). Якщо у деяку змінну, наприклад, *x*, треба поміщати різні вхідні дані, використовують інструкцію «Прочитати *x*» (оператор введення). Передбачається, що конкретне значення для *x* надає користувач алгоритму за допомогою клавіатури, запису у файлі, голосу тощо.

Відтворення значення змінної відбувається також у двох випадках: якщо ім'я змінної зазначено *у виразі* (арифметичному чи логічному), або *в інструкції надрукувати* значення змінної. Інструкцію «Надрукувати *x*» (оператор виведення) використовують для того, щоб алгоритм повідомляв обчислені результати (на екрані монітора, папері, у файлі).

Вирази використовують для того, щоб ЕОМ опрацьовувала поточні значення змінних алгоритму. Наприклад, вираз «$x > 0$» дає змогу перевірити, чи поточне значення змінної *x* додатне; вираз «$n + 5$» обчислює суму поточного значення *n* і числа 5. Саме за допомогою виразів комп'ютерові наказують «*виконати арифметичну дію*» чи «*порівняти два значення*».

Розглянемо ще декілька прикладів, які ілюструють використання виразів і операторів присвоєння. Нехай змінній *max* треба надати більше з двох значень, розташованих у змінних *a* і *b*. Послідовність дій для вирішення завдання досить очевидна: якщо $a > b$, то $max \leftarrow a$; якщо $b > a$, то $max \leftarrow b$. Інструкції записано безвідносно до конкретних значень змінних, у виразах порівнюють їхні поточні значення. Хочемо звернути увагу на інструкції присвоєння. У випадку $a > b$ потрібно вміст *a* скопіювати в *max*, тому правильною є саме інструкція «$max \leftarrow a$», а не «$max \rightarrow a$» чи «$a \leftarrow max$», як можна було би подумати. Адже комп'ютер може *скопіювати* значення, а не «*назвати* змінну найбільшим».

Нехай треба поміняти місцями значення змінних *a* і *b*, тобто змінній *a* надати значення, записане в *b*, і навпаки – *b* надати значення *a*. Спроба досягти бажаного результату за допомогою операторів $a := b$; $b := a$ приведе до невдачі: значення змінної *a* буде втрачено вже після першого присвоєння. Не ліпшим є і такий варіант $b := a$; $a := b$, бо тепер втрачено значення *b*. Що ж робити? Як би Ви діяли, коли б треба було перелити молоко з каструлі в банку, яка поки що наповнена водою, а цю воду перелити в каструлю замість молока? Правильно, потрібно третю посудину. Так і в нашому прикладі. Сформульоване завдання вирішує така послідовність операторів: $c := a$; $a := b$; $b := c$.

На завершення розглянемо ще один приклад: «$i := i + 1$» – лише один оператор присвоєння, проте дуже важливий, бо у ньому, без перебільшення, зосереджено «половину всієї премудрості» програмування. (Можемо навести ще приклади подібних операторів: «$P \leftarrow P \times m$», «$S := S + d$», «$k \leftarrow k - 3$» тощо.) Як його розуміти? Адже математичне твердження $i = i + 1$ беззмістовне. Воно не виконується ні для яких значень *i*. Нагадаємо, що інструкція присвоєння не є математичною рівністю. Згаданий оператор передбачає виконання таких дій: взяти поточне значення змінної *i*, обчислити суму цього значення та числа 1, записати отриманий результат у змінну *i*. Так старе значення *i* міняється на нове, значення *i* збільшується на одиницю.

Так само оператор $k \leftarrow k - 3$ зменшує поточне значення *k* на три. Оператор $S := S + d$ збільшує *S* на поточне значення *d* тощо. Ідею такого нарощування значення змінної використовують для обчислення сум і добутків. Уявімо собі, що змінна *d* почергово набуває різних значень, тоді декількаразове виконання попереднього оператора накопичить в *S* суму значень *d*.

Перехід до виконання наступної команди відбувається автоматично згідно з програмним принципом роботи процесора, про який ми вже говорили раніше. У записі алгоритму природним вважається порядок команд зліва направо і зверху вниз. Змінити його можна за допомогою спеціальних інструкцій на кшталт «Перейти до кроку 5», «Повернутися до попереднього кроку», «goto Loop» тощо. Їх використовують для реалізації в алгоритмі структур керування (про які ми детальніше поговоримо). Виконання процесором інструкції «Перейти до...» полягає в заміні значення вказівника команд на адресу зазначеної в інструкції команди.

Способи зображення алгоритмів. Блок-схеми. Є декілька вживаних способів запису алгоритмів: словесний, за допомогою псевдокоду, мовою програмування, графічний.

Словесний спосіб найпоширеніший і найнеформальніший. Немає певних правил побудови словесного опису алгоритму, проте такими описами рясніють інструкції користувача побутової техніки, кулінарні книги, збірки дієт і комплексів оздоровчих вправ тощо. Знайти їх можна в науковій і навчальній літературі, присвяченій, наприклад, числовим методам чи програмуванню. Зрозумілість та однозначність такого опису значною мірою залежить від вправності автора і фантазії читача. Дуже часто словесний опис супроводжують наочними прикладами, ілюстраціями, умовними позначеннями.

Кожен з нас, мабуть, пам'ятає як обчислити корені квадратного рівняння $ax^2+bx+c=0$. Треба діяти за таким алгоритмом. «Обчислити дискримінант рівняння за формулою $D=b^2-4ac$. Перевірити знак D: якщо дискримінант невід'ємний, то корені обчислити за формулами $x_{1,2}=\left(-b\pm\sqrt{D}\right)\Big/2a$, у протилежному випадку рівняння не має дійсних коренів.» Цей словесний опис досить зрозумілий. Вже за ним можна будувати програму, тільки не забути додати інструкцію прочитати коефіцієнти рівняння й інструкцію надрукувати обчислені корені чи повідомлення про їхню відсутність.

Пригадаймо одну з попередніх лекцій, присвячену кодуванню інформації. «Переведення цілого числа до нової системи числення виконують послідовним діленням цього числа й отриманих часток на нову основу аж до отримання в частці нуля і записом отриманих остач цифрами нової системи, де перша остача є наймолодшою цифрою.» Це стислий словесний опис алгоритму переведення цілого додатного числа до нової системи числення. Ми ілюстрували його конкретним прикладом застосування.

Псевдокод можна трактувати як певний компроміс між звичайною мовою і мовою програмування. Його записують за допомогою фіксованого набору «службових» слів (формалізованих і однозначних), окремих інструкцій, які ми згадували раніше, і цілих фраз звичайною мовою. Такий запис подає точну загальну схему алгоритму, полегшує його розуміння. Чим більше фраз використано у псевдокоді, тим загальнішим є опис алгоритму.

Наведемо два приклади з книги Гудман С., Хидетниеми С. «Введение в разработку и анализ алгоритмов» (яку повинен прочитати кожен кваліфікований програміст).

Відшукання найбільшого. Задано натуральне N, і N різних дійсних чисел. Треба знайти значення та порядковий номер найбільшого з них.

Алгоритм MAX.

Крок 0. [Кількість чисел і перше з них] Прочитати N; прочитати a.
Крок 1. [Присвоєння початкових значень, або ініціалізація] **Set** $M\leftarrow a$; $J\leftarrow 1$.
Крок 2. [Задано тільки одне число?] **If** $N=1$ **then** перейти до кроку 5.
Крок 3. [Перевірка решти чисел] **For** $k\leftarrow 2$ **to** N **do** крок 4.
Крок 4. [Порівняння і переприсвоєння] **If** $M<a$ **then Set** $M\leftarrow a$; $J\leftarrow k$.
Крок 5. [Результат] Надрукувати "Max=", M, "номер=", J.

У квадратних дужках записано коментарі (пояснення до кожного кроку), службові слова виділено напівжирним шрифтом, їхнє значення легко зрозуміти з контексту (Set – присвоєння, If – перевірка умови, For – перебір усіх значень k від 2 до N і виконання для кожного з них кроку 4).

Задача комівояжера. Агент з продажу комп'ютерів (комівояжер) повинен регулярно відвідувати клієнтів у 20-ти доручених йому містах деякого регіону. Щоб знизити видатки на транспорт, агент виїжджає з рідного міста, відвідує кожне з інших 19-ти міст лише один раз і повертається додому. Як розташувати міста в списку, що задає послідовність відвідання, так, щоб сумарна вартість переїздів була найменшою?

Алгоритм ETS.

Крок 0. [Кількість міст і матриця вартостей переїзду] Прочитати *N*; прочитати *C*.
Крок 1. [Ініціалізація] **Set** *TOUR*←∅; *MIN*←∞.
Крок 2. [Перевірка всіх перестановок міст] **For** *i*←1 **to** (*N*–1)! **do** кроки 3, 4, 5.
Крок 3. [Нова перестановка] **Set** *P*←(*i*-та перестановка чисел 1, 2, ..., *N*–1).
Крок 4. [Побудова нового туру] Побудувати тур *T*←Build(*P*); Обчислити його вартість *C*←Cost(*T*).
Крок 5. [Порівняння] **If** *C*<*MIN* **then Set** *TOUR*←*T*; *MIN*←*C*.
Крок 6. [Результат] Надрукувати "Маршрут=", *TOUR*, "вартість=", *MIN*.

Алгоритм ETS – непогане перше наближення до точного алгоритму. Він відображає загальну схему вичерпного перебору всіх можливих турів. Йому ще бракує важливих підалгоритмів (на кроці 3 для побудови перестановки, на кроці 4 для побудови туру і обчислення його вартості), він досить далекий від остаточної програми. Ці недоліки усувають в ході подальшої розробки алгоритму.

Запис мовою програмування найточніше визначає алгоритм, але «несе на собі печать» синтаксису конкретної мови програмування. Два записи того самого алгоритму різними мовами можуть бути схожими, або відрізнятися, залежно від того, наскільки близькими є самі мови. Наприклад, програми на Basic і C++ – мовах процедурного програмування – відрізнятимуться окремими словами та символами. Їхній аналог на Smalltalk – мові об'єктно-орієнтованого програмування – матиме іншу структуру та взаємодію окремих структурних елементів. Принципово іншою буде програма мовою логічного програмування Prolog.

Графічне зображення алгоритму акцентує увагу на його структурах керування і є універсальним, бо не залежить від синтаксису конкретної мови. Різноманітні схеми та діаграми використовують переважно на етапі проектування. Ми опишемо дві системи позначень.

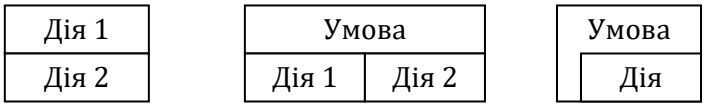


Рис. 3.1. Базові елементи структурних діаграм

Діаграми Нессі-Шнейдерман покликані підтримувати структурне програмування (засади цієї методології викладемо в одній з наступних лекцій). Для базових структур керування використовують прямокутні блоки, зображені на рис. 3.1.

Головні характеристики діаграм: добре визначена функціональна область; заборонено довільне передавання керування; легко визначаються межі локальних і глобальних даних; легко зображаються рекурсивні властивості.

Блок-схеми – найпоширеніша та найзрозуміліша форма графічного подання. Її запропонував фон Нейман як засіб документування програм. (До речі, в колишньому Радянському Союзі діяв державний стандарт оформлення блок-схем.) Для кожної програмної структури існує відповідна їй графічна схема. Найбільша перевага схем – простота і наочність. Проте вони мають і недоліки: неможливе безпосереднє введення в ЕОМ та виведення з неї, мало засобів автоматизованої підтримки; немає ефективного способу керування рівнем деталізації в рамках кожної схеми; нотації недостатні для проектування великомасштабних систем програмного забезпечення.

Для побудови блок-схем використовують графічні елементи, зображені на рис. 3.2.

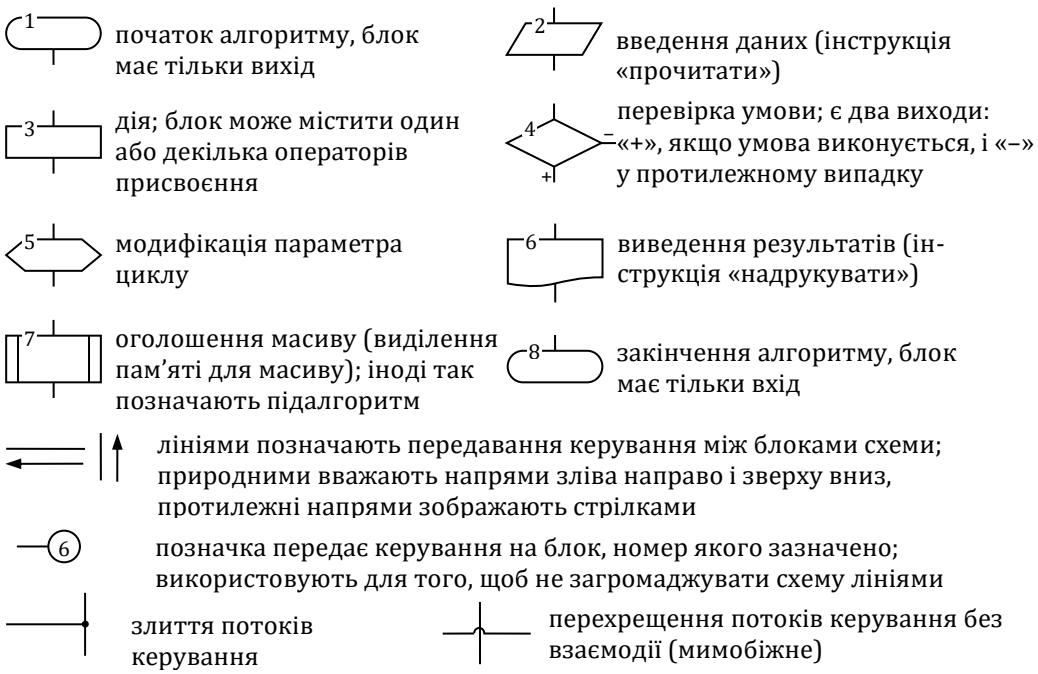


Рис. 3.2. Базові елементи блок-схем

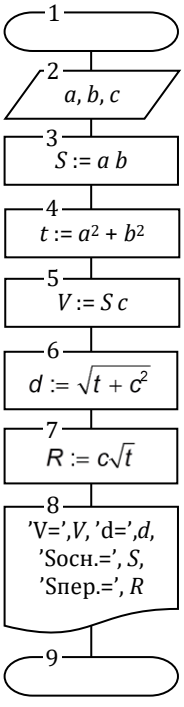


Рис. 3.3. Алгоритм до прикладу 3.1

Типи алгоритмів. Залежно від порядку виконання окремих кроків алгоритми поділяють на послідовні, галужені та циклічні.

Послідовним називають алгоритм, кроки якого виконують один за одним (у природному порядку), цей порядок не залежить ні від вхідних даних, ні від будь-яких інших обставин. У всіх випадках кроки такого алгоритму виконують у незмінній послідовності. Блок-схеми послідовних алгоритмів мають лінійну структуру, тому їх ще називають лінійними. Ми цього робити не будемо, щоб уникнути плутанини з оцінкою складності алгоритмів, про яку йтиметься в наступному пункті.

Приклад 3.1. Задано a, b, c – довжини сторін прямокутного паралелепіпеда. Треба обчислити об'єм паралелепіпеда, його діагональ, площу основи та площу діагонального перерізу.

Пригадаємо відповідні формули. Нехай a і b – сторони основи. Тоді об'єм $V=abc$, діагональ $d=\sqrt{a^2+b^2+c^2}$, площа основи $S_{\text{о}}=ab$, площа діагонального перерізу $S_{\text{п}}=c\sqrt{a^2+b^2}$.

Перш ніж записати алгоритм, потрібно спроектувати склад і призначення його змінних. Досить очевидно, що для зберігання величин a, b, c, V, d можна використати одноіменні змінні. Для значення площі основи використаємо змінну S, а для площі діагонального перерізу, наприклад, змінну R. Зауважимо, що площу основи доцільно обчислювати раніше за об'єм, щоб не множити двічі a на b. Так само, щоб не виконувати зайві обчислення, використаємо додаткову змінну t для зберігання значення виразу a^2+b^2. Блок-схему алгоритму зображено на рис. 3.3.

Галужений алгоритм містить не менше двох можливих шляхів (гілок, альтернатив) виконання. Вибір альтернативи залежить від виконання певних умов. Його неможливо передбачити без знання конкретних вхідних даних.

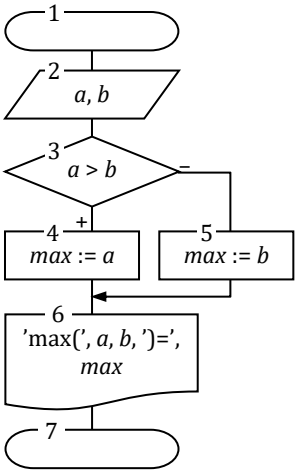


Рис. 3.4. Алгоритм до прикладу 3.2

Приклад 3.2. Задано два різних числа. Відшукати значення більшого з них.

Для зберігання значень заданих чисел використаємо змінні *a* і *b*, для більшого з них – змінну *max*. Щоб знайти більше, використаємо блок перевірки умови. Кожна з альтернатив полягає в копіюванні відповідного значення у змінну *max*. Блок-схему алгоритму зображено на рис. 3.4.

Приклад 3.3. Задано три різних числа. Відшукати значення більшого з них.

Умова задачі майже не відрізняється від попередньої, але не алгоритм відшукання розв'язку. Розглянемо декілька можливих його варіантів і спробуємо вибрати ліпший. У кожному з них для зберігання значень заданих чисел використаємо змінні *a*, *b* і *c*, для більшого з них – змінну *max*. Блок-схеми алгоритмів зображено на рис. 4.5.

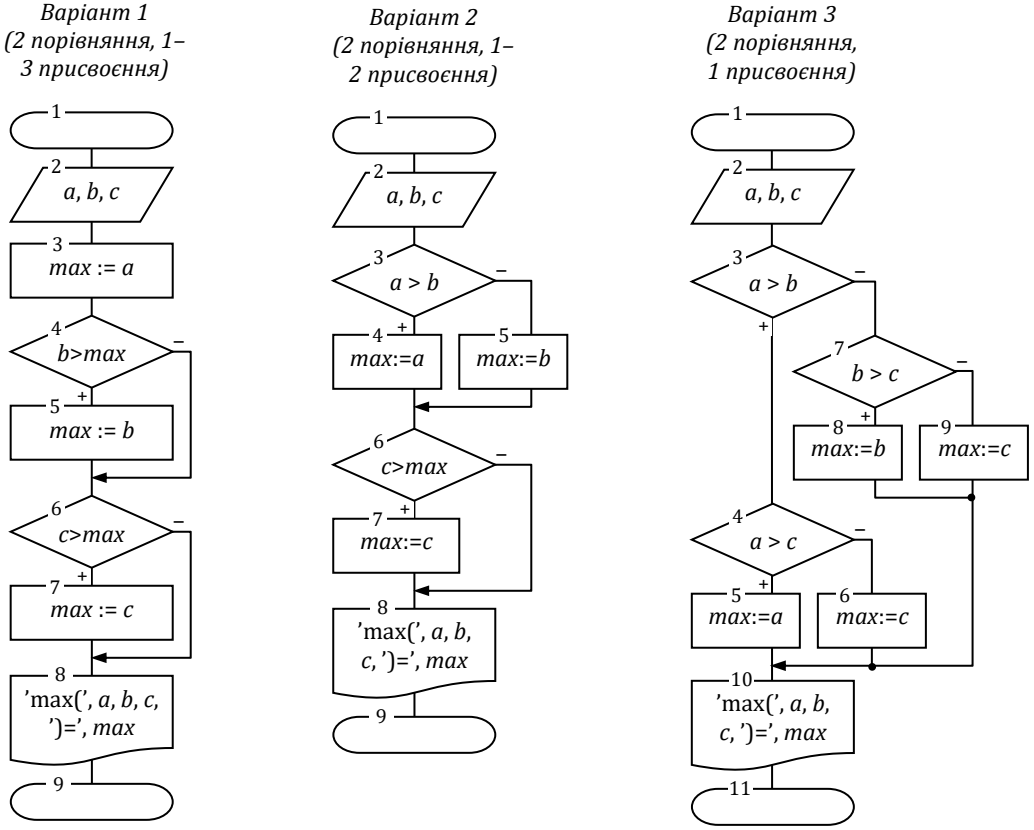
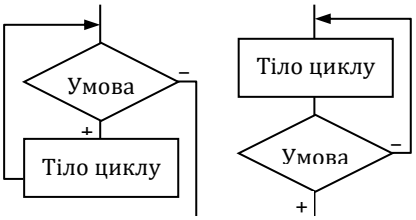


Рис. 3.5. Алгоритми до прикладу 3.3

Алгоритм у першому варіанті припускає спочатку, що найбільшим є перше число, а далі перебирає два інших, якщо знайде більше, змінює значення *max*. Очевидно, що алгоритм завжди виконує два порівняння (блоки 4, 6) і від одного (якщо *a* – найбільше, блок 3) до трьох присвоєнь (якщо вхідні дані впорядковано за зростанням, блоки 3, 5, 7). Другий варіант є пристосуванням до нової задачі алгоритму з прикладу 3.2 і має трохи ліпші показники: він виконує два порівняння і одне або два присвоєння. Третій варіант, незважаючи на громіздкий вигляд, є оптимальним, бо виконує присвоєння тільки після остаточного вибору найбільшого. Як наслідок, за будь-яких вхідних даних спрацьовує два порівняння і одне присвоєння.



Цикл з передумовою *Цикл з постумовою*

Рис. 3.6. Циклічні алгоритми

Для розв'язування багатьох задач доводиться багаторазово повторювати певну послідовність дій. Наприклад, щоб обчислити *n*!, де *n* – натуральне число, нам довелося б послідовно перебирати числа 2, 3, ..., *n* і множити на них значення факторіала. Ми маємо на меті перекласти цю роботу на плечі ЕОМ, але не хотілося б писати комп'ютерові довжелезну послідовність інструкцій вигляду «Візьми число; Помнож; Візьми число; Помнож; ...». Як зробити так, щоб комп'ютер декілька разів виконав те, що написано тільки один раз? Для цього треба змусити ЕОМ у певний момент повертатися до виконання вже виконаних інструкцій: «1. Візьми число; 2. Помнож; 3. Повернись до 1». Звичайно, інструкція повернутись не може бути безумовною, бо виконання алгоритму повинно колись закінчитись. Тому: «3. Повернись до 1, якщо не досягнув кінця».

Алгоритми, які містять групу повторюваних кроків, називають *циклічними*. Розрізняють цикли з передумовою і з постумовою, залежно від того, коли – до чи після виконання тіла циклу – відбувається перевірка умови закінчення циклу. Умова – це довільне твердження, що може бути істинним або хибним.

Передумова «дозволяє» вхід у цикл, а *постумова* – вихід з циклу. Це означає, що тіло циклу з передумовою буде виконано тільки у випадку істинності умови. Можлива ситуація, коли тіло циклу не виконається жодного разу. Цикл з постумовою виконується принаймні один раз. Він завершує роботу, коли умова стає істинною. У тілі обох циклів мали б бути інструкції, виконання яких впливає на умову, змінює її, бо в протилежному випадку такий цикл не закінчиться ніколи – зациклить.

Цикл називають *арифметичним*, якщо наперед відома кількість повторень тіла циклу. Прикладом арифметичного циклу *є цикл з параметром*. Параметром є спеціальна змінна, а її головним призначенням – порахувати кількість повторень.

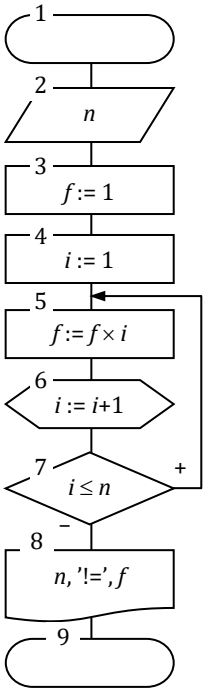


Рис. 3.7. Алгоритм до прикладу 3.4

Приклад 3.4. Задано натуральне *n*. Обчислити *n*!

Алгоритм обчислення зображено на рис. 3.7. Змінна *n* містить значення заданого числа; змінна *i* є параметром циклу й відіграє подвійну роль: рахує кількість повторень циклу та перебирає значення множників, які входять до факторіала; змінна *f* накопичує значення факторіала. Повторювана ділянка алгоритму – це блоки 5, 6, 7. На схемі видно, що потоки керування утворюють кільце. Після виконання блока 7 керування повертається до блока 5, якщо значення *i* не перевищує кінцевого значення.

Зверніть увагу на характерні риси алгоритму, що містить цикл з параметром: блок 3 виконує *підготовку* до циклу (*f* отримує початкове, нейтральне щодо множення, значення); блок 4 – *ініціалізація* циклу, визначає початкове значення параметра; блок 6 – *модифікація* параметра; блок 7 – *перевірка* завершення циклу; блоки між ініціалізацією та модифікацією (у нашому прикладі це блок 5) утворюють *тіло циклу*.

Приклад 3.5. Задано натуральне *n*. Обчислити добуток

$$\frac{\sin 1}{\cos 1}\cdot\frac{\sin 1+\sin 2}{\cos 1+\cos 2}\times\ldots\times\frac{\sin 1+\sin 2+\ldots+\sin n}{\cos 1+\cos 2+\ldots+\cos n}.$$

Добуток складається з *n* множників, тому доцільно для його обчислення використати цикл з параметром. Мусимо запропонувати також спосіб обчислення кожного з множників. Легко бачити, що чисельник і знаменник *i*-го множника відрізняється від чисельника і

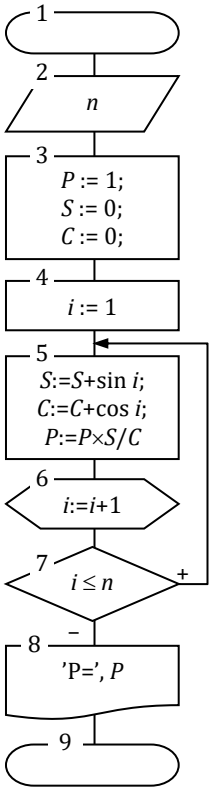


Рис. 3.8. Алгоритм до прикладу 3.5

знаменника попереднього множника відповідно на $\sin i$ та $\cos i$. Отже, їх (чисельник і знаменник) можна накопичувати шляхом додавання відповідних значень.

В алгоритмі, зображеному на рис. 3.8, використано такі змінні: n – кількість множників; i – параметр циклу; P – шуканий добуток; S – сума синусів, чисельник; C – сума косинусів, знаменник. Блоки 3–7 мають таке саме призначення як у попередньому алгоритмі.

Цикл, кількість повторень якого наперед невідома, називають *ітераційним*. Здебільшого на умову закінчення ітераційного циклу впливає результат виконання його тіла.

Приклад 4.6. Задано дійсні x, ε ($x \in [0;\pi/2], \varepsilon > 0$). Обчислити з точністю ε значення $\sin x$.

Для обчислення багатьох математичних функцій застосовують відповідні числові методи, наприклад, наближене обчислення суми ряду Маклорена – розвинення цієї функції.

Пам'ятаємо, що $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - ... = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$.

Ні один комп'ютер не зможе врахувати всіх членів ряду. Відомо, що залишок знакозмінного ряду за абсолютною величиною не перевищує першого відкинутого члена. Тому за наближене значення суми такого ряду приймають часткову суму S_n, де n – номер доданка, модуль якого не перевищує заданої точності.

Тепер залишилось придумати як обчислити ці доданки. Найпростіше це буде зробити за допомогою рекурентних формул, які легко вивести безпосередньо з запису загального члена ряду:

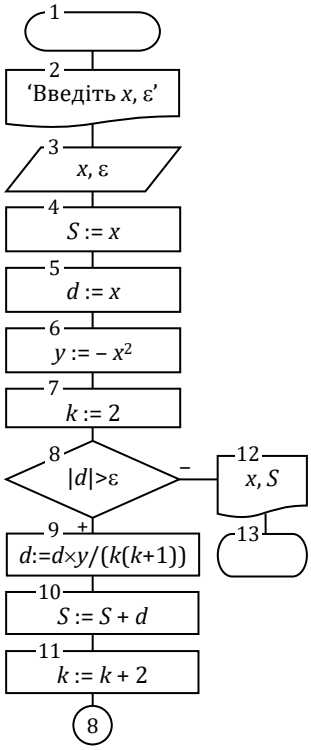


Рис. 3.9. Алгоритм до прикладу 3.6

$d_n = \frac{(-1)^n x^{2n+1}}{(2n+1)!}$, звідки $d_0 = x$; $n = 1, 2, ...$ $d_n = d_{n-1} \frac{-x^2}{2n(2n+1)}$.

Величина $-x^2$ не залежить від номера доданка, тому обчислимо її перед циклом і збережемо у змінній y. Величину $2n$ зберігатимемо в змінній k, суму ряду – в змінній S, а черговий доданок – у змінній d.

Блок-схему алгоритму зображено на рис. 3.9.

Приклад 4.7. Задано дійсні x, ε ($x \neq 0$, $\varepsilon > 0$). Обчислити $\sqrt[3]{x}$ з точністю ε за методом Ньютона: $y_0 = 1$;

$y_n = \left(2y_{n-1} + x / y_{n-1}^2\right)/3, n = 1, 2, ...;$ $\sqrt[3]{x} \approx y_n$, якщо $\left|y_n - y_{n-1}\right| \leq \varepsilon$.

Для відображення в алгоритмі значень y_n, y_{n-1} використаємо змінні u, v. У разі переходу до наступного кроку методу зростає номер n, і значення y_n стає значенням y_{n-1}, тому в тілі циклу треба виконати переприсвоєння $v:=u$ («нове», отримане на попередньому кроці, стає «старим»). Обчислення n необов'язкове для отримання результату, проте в алгоритмі воно враховане для визначення кількості повторень ітераційного циклу.

Блок-схему алгоритму зображено на рис. 3.10.

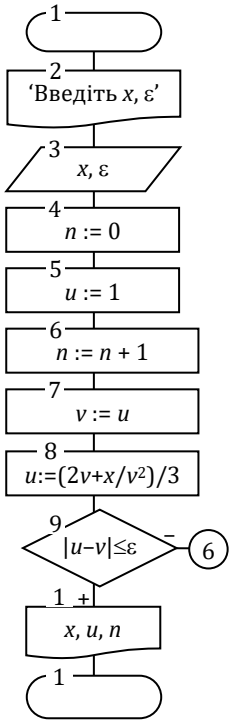


Рис. 3.10. Алгоритм до прикладу 3.7

Наведемо ще один приклад. Алгоритм його розв’язування можна класифікувати як «цикл з галуженням»: він містить повторювані дії, пов’язані з певним вибором.

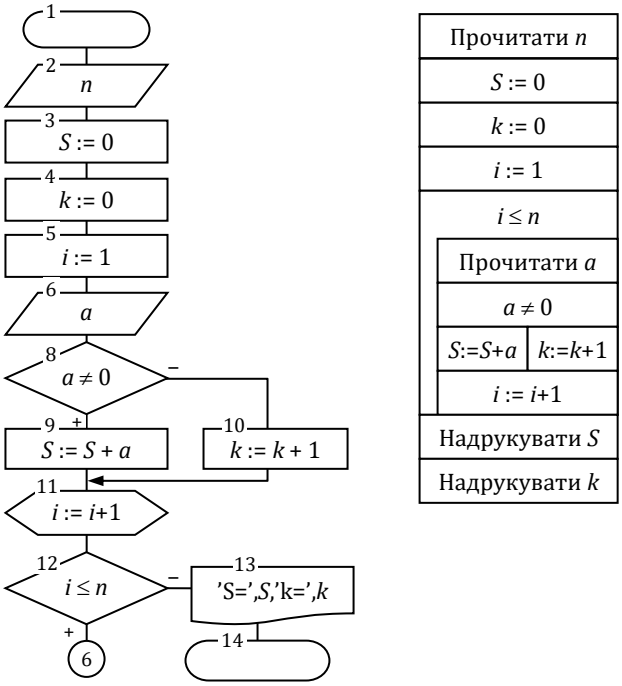


Рис. 3.11. Блок-схема та структурна діаграма алгоритму до прикладу 3.8

Приклад 3.8. Задано натуральне n і послідовність n цілих чисел. Обчислити кількість нульових і суму ненульових членів послідовності.

Для накопичення суми використаємо змінну S, а кількості – k, для перебору членів послідовності – змінну a. Значення кожного члена послідовності треба прочитати і порівняти з нулем. На рис. 3.11 наведено графічне зображення алгоритму у вигляді блок-схеми та, для порівняння, у вигляді діаграми Нессі-Шнейдерман.

Отож, алгоритм з погляду програміста – це послідовність інструкцій комп’ютерові ввести-вивести дані, виконати певні порівняння, обчислення, присвоєння для того, щоб отримати розв’язок певної задачі. Дані зберігаються в іменованих комірках пам’яті комп’ютера (у змінних). Інструкції формулюють у вигляді операторів і зображають їх за допомогою псевдокоду, програми чи графічно. Залежно від послідовності виконання інструкцій алгоритми поділяють на послідовні, галужені, циклічні, циклічні з галуженням, з вкладеними циклами тощо.

Складність алгоритмів. Ми ввели поняття алгоритму, навели приклади алгоритмів розв’язування конкретних задач. Звичайно, ці приклади не дають повного уявлення про способи побудови алгоритмів – тема занадто об’ємна, щоб розкрити її в одній-двох лекціях. Ми повертатимемося до неї ще не раз, щоб вивчити різні методи побудови алгоритмів. Нас завжди буде цікавити питання: «Якщо задано задачу, то як знайти алгоритм її розв’язування?» Але сьогодні обговоримо інші, не менш важливі, питання, що завжди постають перед кожним програмістом.

«Якщо задано задачу, то *як знайти ефективний алгоритм* для її розв’язування? Якщо побудовано новий алгоритм, то *як порівняти його з іншими алгоритмами*, що розв’язують ту саму задачу? Як *оцінити його якість*? Як аналізувати алгоритм і *оцінити його складність*?»

Існує багато різних критеріїв для оцінки складності алгоритмів. Передусім розрізняють синтаксичну, логічну складність, складність за часом (за кількістю операцій), складність за обсягом пам’яті.

Числовими критеріями *синтаксичної складності* можуть бути: обсяг тексту програми, що описує алгоритм; кількість операторів, використаних для побудови алгоритму; кількість структурних операторів (галуження, циклу) тощо.

Важко оцінити кількісно *логічну складність*. Логіку алгоритму різні люди сприймають по-різному. Критерієм може бути кількість переходів (передач керування) в алгоритмі, кількість і глибина вкладення структурних операторів, кількість переприсвоєнь (операторів вигляду $z := f(z)$) тощо.

Завдання дослідження синтаксичної та логічної складності частіше виникають в теорії алгоритмів. Для практики важливішим показником є швидкість зростання потрібного для розв'язування задачі часу (й обсягу пам'яті) за умов збільшення вхідних даних, тобто часова складність (складність за пам'яттю). Щоб оцінити цю складність, вводять поняття розміру задачі. *Розміром задачі* називають числову характеристику вхідних даних, міру їхньої кількості. Для кожної конкретної задачі така характеристика може бути іншою, наприклад, розміром задачі впорядкування послідовності чисел є кількість цих чисел, розміром задачі множення матриць – кількість матриць і найбільший розмір матриці-множника, у задачах на графах розміром може бути кількість ребер графа.

Функцію, яка виражає залежність часу, затраченого алгоритмом на обчислення розв'язку, від розміру задачі, називають *часовою складністю* цього алгоритму. Функцію, яка виражає залежність від розміру задачі загального обсягу пам'яті, потрібної для всіх змінних алгоритму, називають *складністю за пам'яттю*. Поведінку таких функцій у границі зі збільшенням розміру задачі називають *асимптотичною складністю*. Оцінка складності може залежати не тільки від розміру вхідних даних, а й від інших їхніх властивостей. Наприклад, певна сукупність даних може бути несприятливою для алгоритму, або, навпаки, підходити для нього якнайліпше. Тому говорять про *максимальну* складність (або складність у найгіршому випадку – для несприятливих даних), *мінімальну* складність (у найліпшому випадку), *середню складність* та середньоквадратичне відхилення складності (якщо відомий розподіл допустимих вхідних даних).

У попередній лекції ми обговорили три можливі способи відшукання найбільшого серед трьох заданих чисел і порахували, скільки операцій порівняння та присвоєння виконає кожен із запропонованих алгоритмів. Такі оцінки кількості операцій і є простим прикладом визначення складності алгоритму. Якщо відомо, скільки часу затрачає комп'ютер на одне порівняння і одне присвоєння, то оцінку «в операціях» легко перевести в оцінку «в мілісекундах», чи, наприклад, у тактах процесора.

Розмір n задачі, яку можна розв'язати за допомогою алгоритму, визначає його асимптотична складність $f(n)$. За виглядом функції, що описує складність, алгоритми поділяють на логарифмічні ($f(n) = O(\log_2 n)$), лінійні ($f(n) = O(n)$), поліноміальні ($f(n) = O(n^k)$), експотенційні ($f(n) = O(a^n)$). Найшвидше зростає складність експотенційного алгоритму (особливо, для великих a), найповільніше – логарифмічного.

Можна було б подумати, що величезне зростання швидкості обчислень, зумовлене появою сучасних комп'ютерів, зменшить значення ефективних алгоритмів. Проте відбувається все навпаки. Завдяки швидким ЕОМ ми отримуємо змогу розв'язувати все складніші та складніші задачі, тому саме ефективність алгоритму визначає те збільшення розміру задач, якого можна досягти завдяки збільшенню швидкодії машини. Проілюструємо сказане прикладами.

Таблиця 3.1.

Порівняння можливостей алгоритмів з різною часовою складністю

Алгоритм	Часова складність, $f(n)$	Найбільший розмір задачі		
		за 1 с	за 1 хв	за 1 год
A_1	n	1000	6×10^4	$3{,}60\times10^6$
A_2	$n\log_2 n$	140	4895	$2{,}04\times10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Нехай відомо п’ять алгоритмів розв’язування деякої задачі, причому всі вони мають різну часову складність $f(n)$, що виражає кількість одиниць часу, потрібних для опрацювання входу розміру n. Тоді кожен з алгоритмів за той самий проміжок часу зможе опрацювати задачі різних розмірів. У табл. 3.1 наведено ці розміри (вважається, що одиницею часу є мілісекунда).

Таблиця дає добру нагоду пересвідчитися як змінюється розмір задачі зі зростанням часу. Порівняємо, наприклад, третій і п’ятий стовпчики таблиці. Збільшення затраченого часу в 3 600 разів у стільки ж разів збільшує розмір задачі для лінійного алгоритму A_1, у 63 рази – для квадратичного A_3, і лише трохи більше ніж удвічі – для експотенційного A_5.

Припустимо, що майбутнє покоління обчислювальних машин буде швидшим у 100 разів. У табл. 3.2 показано, як зміняться завдяки прискоренню розміри задач, які ми зможемо розв’язувати.

Таблиця 3.2.

Порівняння зростання можливостей алгоритмів

Алгоритм	Часова складність, $f(n)$	Найбільший розмір задачі	
		за швидкодії k	за швидкодії $100k$
A_1	n	s_1	$100s_1$
A_2	$n\log_2 n$	s_2	$\approx 75s_2$ для великих s_2
A_3	n^2	s_3	$10s_3$
A_4	n^3	s_4	$4{,}64s_4$
A_5	2^n	s_5	$s_5+6{,}64$

Зауважимо, що для алгоритму A_5 збільшення швидкодії ЕОМ у сто разів збільшує розмір задачі тільки на шість, для A_3 – у десять разів, а для A_1 – у сто.

Замість ефекту збільшення швидкодії розглянемо ефект застосування дієвішого алгоритму. Повернемось до попередньої таблиці. Якщо за основу для порівняння взяти 1 хв, то бачимо, що заміна алгоритму A_4 на A_3 збільшить розмір задачі у 6 разів, заміна на A_2 – у 125 разів. Ці результати справляють набагато більше враження, ніж десятикратне покращення (A_3 у табл. 3.2) за рахунок стократного збільшення швидкодії. Якщо за основу взяти 1 год, то відмінність стає ще значнішою.

Наведемо ще один приклад. Нехай на потужному комп’ютері, швидкодія якого становить 100 млн операцій за секунду, встановлено програму впорядкування чисел методом обміну з оцінкою $2n^2$ операцій для n чисел. На домашньому комп’ютері, швидкодія якого – 1 млн операцій за секунду, встановлено програму впорядкування методом злиття з оцінкою $50n\log_2 n$. Який з комп’ютерів швидше впорається з завданням впорядкувати мільйон чисел? Не поспішайте з відповіддю. Обчислимо час, потрібний кожному з них:

$$t_{пот} = \frac{2\cdot\left(10^6\right)^2 \text{ оп.}}{10^8 \text{ оп./с}} = 20000 \text{ с} \approx 5{,}56 \text{ год},$$

$$t_{дом} = \frac{50\cdot 10^6 \log_2 10^6 \text{ оп.}}{10^6 \text{ оп./с}} \approx 997 \text{ с} \approx 16{,}6 \text{ хв}.$$

Результати свідчать самі за себе.

Усі наведені приклади і порівняння дають підстави зробити висновок про те, що асимптотична складність алгоритму є важливим мірилом його якості, причому таким мірилом, яке стає важливішим за умов зростання швидкодії.

Дотепер ми звертали увагу на порядок зростання величин і залишили поза розглядом числові коефіцієнти (мультиплікативні сталі), що завжди є в оцінках. Дуже часто трапляється так, що оцінка з великим порядком росту має менший коефіцієнт, ніж оцінка з малим порядком. У цьому випадку алгоритм, складність якого зростає швидше, може виявитися кращим для задач малого розміру, можливо, саме тих задач, які нас цікавлять найбільше.

Наприклад, припустимо, що часові складності п'яти алгоритмів, які ми вже розглядали, насправді рівні: $f_1(n)=1000n$, $f_2(n)=100n\log_2 n$, $f_3(n)=10n^2$, $f_4(n)=n^3$, $f_5(n)=2^n$. Тоді A_5 буде найкращим для задач розміру $2 \le n \le 9$, A_3 – для задач розміру $10 \le n \le 58$, A_2 – при $59 \le n \le 1024$, а A_1 – при $n > 1024$.

«Швидкий алгоритм» обчислення степені числа. Трасувальна таблиця. Розглянемо приклади визначення складності двох цікавих алгоритмів.

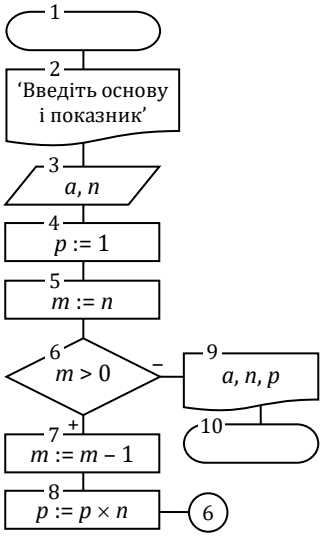


Рис. 3.12. Алгоритм послідовного обчислення a^n

Приклад 3.9. Задано дійсне додатне число a і натуральне число n. Обчислити a^n.

«Знавці» одразу запитають, «що ж тут цікавого, адже $a^n = \exp(n \ln a)$?». Пригадаймо попередню лекцію: обчислення елементарних математичних функцій може означати обчислення суми ряду – купа роботи. То навіщо ж мучити «кремнієвого помічника» експонентою і логарифмом, коли піднесення до натурального степеня можна зробити простим множенням?

На гадку спадає циклічний алгоритм, на кожному кроці якого змінну-добуток домножують на значення a, доки не одержать потрібний степінь. Блок-схему такого алгоритму зображено на рис. 3.12. Тут p використано для накопичення добутку, а m – для обчислення кількості неврахованих множників.

Оцінимо часову складність цього алгоритму. Нехай T_1 – тривалість виконання процесором додавання (віднімання), а T_2 – множення. Конкретні значення легко одержати, дізнавшись тактову частоту процесора і кількість тактів для виконання арифметичної дії. Віднімання виконується в блоці 7 алгоритму, а множення – в блоці 8. Ці блоки становлять тіло циклу і виконуються на кожному його кроці для значень $m = n, n–1, ..., 1$, тобто, n разів. Отож, тривалість роботи алгоритму можна виразити так: $f_1(n) = (T_1+T_2)\,n$. Алгоритм – лінійний.

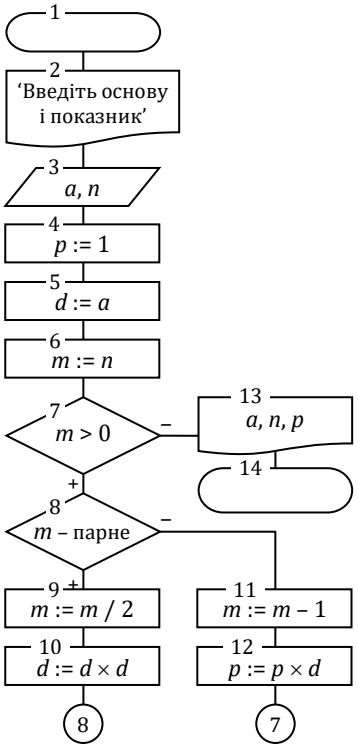


Рис. 3.13. «Швидкий алгоритм» обчислення a^n

Чи є інші, не такі очевидні способи обчислення степеня заданого числа? Задумайтеся, як найшвидше обчислити a^8? Виявляється, для цього зовсім не обов'язково виконувати сім чи вісім множень – вистачить трьох: $a^8 = ((a^2)^2)^2$, оскільки $8=2^3$. Цей підхід можна пристосувати і для тих значень n, які не є точними степенями двійки. Наприклад, $a^5 = a\cdot(a^2)^2$.

Блок-схему алгоритму «швидкого» піднесення до степеня зображено на рис. 3.13. Його ідея полягає в тому, що задане число підносять до квадрата, якщо показник парний, або просто домножують у протилежному випадку. Простіше продемонструвати роботу алгоритму на декількох прикладах, ніж пояснювати його на словах.

Для демонстрації використаємо *трасувальну таблицю*. У шапку трасувальної таблиці записують імена змінних алгоритму та номери блоків галуження. Таблицю заповнюють поступово, по рядках, простежуючи проміжні значення змінних і виконання умов, тобто, виконуючи вручну роботу комп'ютера («принцип кухаря» – скуштуй сам страву, яку приготував для когось). Така перевірка дає змогу краще зрозуміти справжню поведінку алгоритму та,

можливо, виявити помилки. Отже, нехай $n = 8$ ($8_{10}=1000_2$). Бавимося в ЕОМ! Результати трасування показано в табл. 3.3.

Таблиця 3.3.

Трасувальна таблиця «швидкого» алгоритму для n=8

p	d	m	(7) ?	(8) ?
1	a	8	+	+
	a^2	4		+
	a^4	2		+
	a^8	1		–
$\boldsymbol{a^8}$		0	–	

На останньому кроці умова (7) не виконується і алгоритм завершує роботу.

Поекспериментуємо тепер з іншими значеннями n, наприклад, $n = 10$ ($10_{10}=1010_2$) і $n = 15$ ($15_{10}=1111_2$). Трасувальні таблиці наведено в табл. 3.4.

Таблиця 3.4.

Трасувальні таблиці «швидкого» алгоритму для n=10 і n=15

$n = 10$

p	d	m	(7) ?	(8) ?
1	a	10	+	+
	a^2	5		–
a^2		4	+	+
	a^4	2		+
	a^8	1		–
$\boldsymbol{a^{10}}$		0	–	

$n = 15$

p	d	m	(7) ?	(8) ?
1	a	15	+	–
a		14	+	+
	a^2	7		–
a^3		6	+	+
	a^4	3		–
a^7		2	+	+
	a^8	1		–
$\boldsymbol{a^{15}}$		0	–	

Зауважте, зі значенням змінної m відбуваються такі самі зміни, як під час переведення до двійкової системи числення, причому блоки 11, 12 спрацьовують тоді, коли у двійковому записі m трапляється одиниця, а блоки 9, 10 – для кожної двійкової цифри, крім найстаршої. Отож, найліпшим для алгоритму є значення $n = 2^k$, а найгіршим – $n = 2^k - 1$.

Тепер оцінимо складність алгоритму. Ділення на два (у блоці 9) процесор може виконати за допомогою побітного зсуву вправо на один розряд. Тривалість такої дії не перевищує T_1, тому і скористаємося цією величиною. Кількість цифр у двійковому записі числа n дорівнює $[\log_2 n]+1$, де квадратні дужки означають взяття цілої частини. Тому блоки 9, 10 спрацьовують $[\log_2 n]$ разів, а блоки 11, 12 – від 1, у найліпшому випадку, до $[\log_2 n]+1$, у найгіршому. Остаточно отримаємо:

$$(T_1 + T_2)(1 + [\log_2 n]) \le f_2(n) \le (T_1 + T_2)(1 + 2[\log_2 n]).$$

Цей алгоритм – логарифмічний. Права частина нерівності визначає складність у найгіршому випадку.

Який з двох алгоритмів ліпший? Для великих n, без сумніву, другий, бо він має менший порядок зростання. Проте його верхня оцінка має удвічі більшу мультиплікативну сталу, ніж оцінка першого алгоритму. Тому для малих n ліпшим може виявитися лінійний алгоритм. Щоб з'ясувати це питання, розв'яжемо рівняння $2[\log_2 n]+1 = n$. Звідки $n = 5$. Отже, для $n > 5$ ліпшим є «швидкий» алгоритм. Якби всім значенням $n \le 5$ відповідала його верхня оцінка, то для таких n ліпшим би був алгоритм послідовного множення. Безпосередня перевірка засвідчує, що і для малих n другий алгоритм виконує не більше операцій, ніж перший.