МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ХАРЧОВИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ

В.о. ректора _____ А. І. Українець
(підпис)
«___» _____2015 р.

Т.М. ГОРЛОВА
К.Є. БОБРІВНИК
Н.В. ЛІМАНСЬКА

ТЕОРІЯ АЛГОРИТМІВ

КОНСПЕКТ ЛЕКЦІЙ
для студентів напряму підготовки 6.050101 «Комп'ютерні науки»
денної та заочної форм навчання

Всі цитати, цифровий та фактичний матеріал, бібліографічні відомості перевірені. Написання одиниць відповідає стандартам

Підпис(и) автора(ів)_____

«____» _____ 2015 р.

СХВАЛЕНО
на засіданні кафедри
інформаційних систем
Протокол № 10
від 17.02.2015 р.

Реєстраційний номер
електронного конспекту лекцій
у НМВ 51.21-12.03.2015

КИЇВ НУХТ 2015

Горлова Т.М. Теорія алгоритмів. [Електронний ресурс]: конспект лекцій для студентів напряму підготовки 6.050101 «Комп'ютерні науки» денної та заочної форм навчання / Т.М. Горлова, К.Є. Бобрівник, Н.В. Ліманська – К.: НУХТ, 2015. – 95 с.

Рецензент: **Маноха Л.Ю.,** канд. техн. наук, доц.

Т.М. ГОРЛОВА, канд. техн. наук, доц.
К.Є. БОБРІВНИК
Н.В. ЛІМАНСЬКА

Подано в авторський редакції

ЗМІСТ

1. ВВЕДЕННЯ В ТЕОРІЮ АЛГОРИТМІВ

1.1 Основні поняття теорії алгоритмів

Теорія алгоритмів – це наука, що вивчає загальні властивості та закономірності алгоритмів, різноманітні формальні моделі їх подання. На основі формалізації поняття алгоритму можливе порівняння алгоритмів за їх ефективністю, перевірка їх еквівалентності, визначення областей застосовності.

В даний час теорія алгоритмів утворює теоретичний фундамент обчислювальних наук. Застосування теорії алгоритмів здійснюється як у використанні самих результатів (особливо це стосується використання розроблених алгоритмів), так і у виявленні нових понять і уточненні старих. З її допомогою пояснюються такі поняття як **доведеність, ефективність, можливість розв'язання** тощо**.**

У техніку термін «алгоритм» прийшов разом з кібернетикою. Поняття алгоритму допомогло, наприклад, точно визначити, що означає ефективно задати послідовність керуючих сигналів. Застосування ПК послужило стимулом розвитку теорії алгоритмів і вивченню алгоритмічних моделей, до самостійного вивчення алгоритмів з метою їх порівняння за робочими характеристиками (числу дій, витраті пам'яті), а також їх оптимізації.

Виник важливий напрямок в теорії алгоритмів – складність алгоритмів і обчислень. Почала складатися так звана метрична теорія алгоритмів, основним змістом якої є класифікація задач за класами складності. Самі алгоритми стали об'єктом точного дослідження як і ті об'єкти, для роботи з якими вони призначені

В цій області природно виділяються завдання отримання верхніх і нижніх оцінок складності алгоритмів. Методи вирішення цих завдань зовсім різні. Для отримання верхніх оцінок досить інтуїтивного поняття алгоритму. Для цього будується неформальний алгоритм вирішення конкретного завдання і потім він формалізується для реалізації на придатній алгоритмічній моделі.

Якщо показується, що складність (час або пам'ять) обчислення для цього алгоритму не перевищує значення підходящої функції при всіх значеннях аргументу, то ця функція оголошується верхньою оцінкою складності рішення розглянутої задачі. В області знаходження верхніх оцінок отримано

багато яскравих результатів для конкретних задач. Серед них розроблені швидкі алгоритми множення цілих чисел, багаточленів, матриць, рішення лінійних систем рівнянь, які вимагають значно менше ресурсів, ніж традиційні алгоритми.

Встановити нижню оцінку – значить довести, що ніякий алгоритм обчислення не має складності меншої, ніж задана межа. Для отримання результатів такого типу необхідна точна фіксація розглянутої алгоритмічної моделі, і такі результати отримані тільки в дуже жорстких обчислювальних моделях. У зв'язку з цим отримала розвиток проблематика отримання «відносних» нижніх оцінок, так звана теорія **NP-повноти**, яка пов'язана з труднощами розв’язку задач перебору.

Розглянемо неформально, що саме в інтуїтивному понятті алгоритму потребує уточнення.

Основні вимоги до алгоритмів.

1. Кожен алгоритм має справу з даними – вхідними, проміжними, вихідними. Для того, щоб уточнити поняття даних, фіксується кінцевий алфавіт вихідних символів (цифри, букви і т.п.) і вказуються правила побудови алгоритмічних об'єктів. Типовим використовуваним засобом є індуктивна побудова. Наприклад, визначення ідентифікатора в Алгол: ідентифікатор – це або буква, або ідентифікатор, до якого приписана праворуч або буква, або цифра. Слова кінцевої довжини в кінцевих алфавітах – найбільш звичайний тип алгоритмічних даних, а число символів в слові – природна міра об'єму даних. Інший випадок алгоритмічних об'єктів – формули. Прикладом можуть служити формули алгебри предикатів і алгебри висловлювань. У цьому випадку не кожне слово в алфавіті буде формулою.

2. Алгоритм для розміщення даних вимагає пам'яті. Пам'ять зазвичай вважається однорідною і дискретною, тобто вона складається з однакових комірок, причому кожна комірка може містити один символ даних, що дозволяє узгодити одиниці виміру обсягу даних і пам'яті.

3. Алгоритм складається з окремих елементарних кроків, причому множина різних кроків, з яких складений алгоритм, кінцеві. Типовий приклад множини елементарних кроків – система команд процесора.

4. Послідовність кроків алгоритму детермінована, тобто після кожного кроку вказується, який крок слід виконувати далі, або вказується, коли слід роботу алгоритму вважати закінченою.

5. Алгоритм повинен бути результативним, тобто зупинятися після кінцевого числа кроків (залежного від вхідних даних) з видачею результату. Дана властивість іноді називають збіжністю алгоритму.

6. Алгоритм передбачає наявність механізму реалізації, який за описом алгоритму породжує процес обчислення на основі вхідних даних. Передбачається, що опис алгоритму та механізм його реалізації кінцеві. Можна помітити аналогію з обчислювальними машинами. Вимога 1 відповідає цифровій природі ПК, вимога 2 – пам'ять ПК, вимога 3 – програмі машини, вимога 4 – її логічній природі, вимоги 5, 6 – обчислювальному пристрою і його можливостям.

Є також деякі риси неформального поняття алгоритму, щодо яких не досягнуто остаточної угоди. Ці риси сформулюються у вигляді запитань і відповідей.

7. Чи слід фіксувати кінцеву границю для розміру вхідних даних?

8. Чи слід фіксувати кінцеву границю для числа елементарних кроків?

9. Чи слід фіксувати кінцеву границю для розміру пам'яті?

10. Чи слід обмежити число кроків обчислення?

На всі ці питання далі приймається відповідь "НІ", хоча можливі й інші варіанти відповідей, оскільки у фізично існуючих ПК відповідні розміри обмежені. Проте теорія, що вивчає алгоритмічні обчислення, здійсненні в принципі, не повинна рахуватися з такого роду обмеженнями, оскільки їх можна подолати принаймні в принципі (наприклад, взагалі кажучи, будь-який фіксований розмір пам'яті завжди можна збільшити на одну клітинку).

Таким чином, уточнення поняття алгоритму пов'язано з уточненням алфавіту даних і форми їх подання, пам'яті і розміщення в ній даних, елементарних кроків алгоритму та механізму реалізації алгоритму. Однак ці поняття самі потребують уточнення. Ясно, що їхні словесні визначення зажадають введення нових понять, для яких у свою чергу, знову будуть потрібні уточнення і т.д. Тому в теорії алгоритмів прийнятий інший підхід, заснований на конкретній алгоритмічної моделі, в якій всі сформульовані вимоги виконуються очевидним чином. При цьому використовувані алгоритмічні моделі універсальні, тобто моделюють будь-які інші розумні алгоритмічні моделі, що дозволяє зняти можливе заперечення проти такого підходу: Чи не приводить жорстка фіксація алгоритмічної моделі до втрати

спільності формалізації алгоритму? Тому дані алгоритмічні моделі ототожнюються з формальним поняттям алгоритму. Надалі будуть розглянуті основні типи алгоритмічних моделей, що розрізняються трактуваннями, що таке алгоритм.

Перший тип трактує алгоритм як деякій детермінований пристрій, здатний виконувати в кожен момент лише строго фіксовану множину операцій. Основною теоретичною моделлю такого типу є машина Тьюринга, запропонована ним у 30-х роках, яка зробила суттєвий вплив на розуміння логічної природи розроблюваних ЕОМ. Іншою теоретичною моделлю даного типу є машина довільного доступу (МДД) – введена досить недавно (у 70-х роках) з метою моделювання реальних обчислювальних машин та отримання оцінок складності обчислень.

Другий тип пов'язує поняття алгоритму з традиційним уявленням – процедурами обчислення значень числових функцій. Основною теоретичною моделлю цього типу є рекурсивні функції – історично перша формалізація поняття алгоритму.

Третій тип алгоритмічних моделей – це перетворення слів у довільних алфавітах, в яких операціями є заміни фрагментів слів іншим словом. Основною теоретичною моделлю цього типу є нормальні алгоритми Маркова.

Теорія алгоритмів має істотний вплив на розвиток ЕОМ і практику програмування. В теорії алгоритмів передбачені основні концепції, які закладені в апаратуру і мови програмування ЕОМ. Згадувані вище основні алгоритмічні моделі математично еквівалентні. Але на практиці вони істотно розрізняються ефектами складності, що виникають при реалізації алгоритмів, і породили різні напрямки в програмуванні. Так, мікропрограмування будується на ідеях машин Тьюринга, структурне програмування запозичило свої конструкції з теорії рекурсивних функцій, мови символьної обробки інформації (РЕФАЛ, ПРОЛОГ) беруть початок від нормальних алгоритмів Маркова та систем Посту.

На основі теорії алгоритмів в даний час отримані практичні рекомендації, що набувають все більшого поширення в області проектування і розробки програмних систем. Результати теорії алгоритмів набувають особливого значення для криптографії.

Першим алгоритмом у вигляді кінцевої послідовності елементарних дій, що вирішують поставлену задачу, вважається запропонований Евклідом в III

столітті до нашої ери **Алгоритм знаходження найбільшого загального дільника двох чисел** (алгоритм Евкліда).

Початковою точкою відліку сучасної теорії алгоритмів вважають роботу німецького математика Курта Геделя [3] (1931 рік – теорема про неповноту символічних логік.

Перші фундаментальні роботи з теорії алгоритмів були опубліковані незалежно в 1936 році роки Аланом Тьюрингом, Алоізом Черчем і Емілем Постом. Запропоновані ними машина Тьюринга, машина Посту і лямбда-числення Черча були еквівалентними формалізмами алгоритму. Важливим розвитком цих робіт стало формулювання і доказ алгоритмічно нерозв'язних проблем.

У 1950-ті роки істотний внесок у теорію алгоритмів внесли роботи Колмогорова і Маркова.

1.2 Історичний огляд

До 1960-70-их років оформилися наступні напрямки в теорії алгоритмів:

• Класична теорія алгоритмів :
- формулювання завдань в термінах формальних мов,
- поняття завдання дозволу,
- введення класів складності,
- формулювання в 1965 році Едмондсі проблеми P = NP,
- відкриття класу NP-повних задач і його дослідження) [5].

• Теорія асимптотичного аналізу алгоритмів:
- поняття складності і трудомісткості алгоритму,
- критерії оцінки алгоритмів,
- асимптотичний аналіз трудомісткості або часу виконання, в розвиток якої внесли істотний внесок Кнут, Ахо, Хопкрофта, Ульман, Карпо [1, 4].

• Теорія практичного аналізу обчислювальних алгоритмів:
- одержання явних функції трудомісткості,
- інтегральний аналіз функцій,
- практичні критерії якості алгоритмів,
- методика вибору раціональних алгоритмів, основоположною роботою в цьому напрямку слід вважати фундаментальну працю Д. Кнута «Мистецтво програмування для ЕОМ» [4].

1.3 Цілі і завдання теорії алгоритмів

Можна виділити наступні цілі і співвіднесені з ними завдання, які вирішуються в теорії алгоритмів:

• формалізація поняття «алгоритм» і дослідження формальних алгоритмічних систем;

• формальний доказ алгоритмічної нерозв'язності ряду задач;

• класифікація завдань, визначення і дослідження класів складності;

• вивчення поняття складності алгоритмів;

• дослідження і аналіз рекурсивних алгоритмів;

• отримання явних функцій трудомісткості в цілях порівняльного аналізу алгоритмів;

• розробка критеріїв порівняльної оцінки якості алгоритмів.

1.4 Практичне застосування результатів теорії алгоритмів

Виділяють наступні два аспекти:

Теоретичний аспект: при дослідженні деякої задачі результати теорії алгоритмів дозволяють відповісти на питання чи є ця задача в принципі алгоритмічно вирішуваною.

У разі алгоритмічної розв'язності задачі – наступне важливе питання про приналежність цього завдання до класу NP-повних задач, при позитивному відповіді на який, можна говорити про істотні тимчасових витратах для отримання точного рішення для великих розмінностей вихідних даних.

Практичний аспект: методи та методики теорії алгоритмів дозволяють здійснити:

• раціональний вибір з відомої безлічі алгоритмів вирішення даного завдання з урахуванням особливостей їх застосування (наприклад, при обмеженнях на розмірність вихідних даних або обсягу додаткової пам'яті);

• отримання тимчасових оцінок вирішення складних завдань;

• отримання достовірних оцінок неможливості вирішення деякої задачі за певний час, що важливо для криптографічних методів;

• розробку і вдосконалення ефективних алгоритмів рішення задач в **області обробки інформації на основі практичного аналізу.**

1.5 Формалізація поняття алгоритму

У всіх сферах своєї діяльності, і зокрема в сфері обробки інформації, людина стикається з різними способами або методиками вирішення завдань. Вони визначають порядок виконання дій для одержання бажаного результату. Їх можна трактувати як початкове або інтуїтивне визначення алгоритму.

Для того щоб дати точне поняття алгоритму, необхідно визначити, як задаються дані, з якими буде працювати виконавець, і як задаються елементарні кроки, з яких складається алгоритм. В якості даних будемо розглядати конструктивні об'єкти.

Будь-який об'єкт може бути описаний деяким набором фраз на деякій мові, інакше кажучи, представлений (закодований) ланцюжком символів. Це досить загальний підхід.

Якщо об'єкт – число, то його можна записати в десятковій або двійковій формі, тобто ланцюжком символів алфавіту {0,1}.

Якщо об'єкт – програма, то вона є ланцюжком символів в алфавіті, що містить букви, цифри і спеціальні символи.

Якщо об'єкт – зображення, то він представляється масивом пікселів, а кожен піксель – трьома числами (інтенсивностями червоного, зеленого і синього кольорів). Тобто зображення також може бути закодовано рядком символів. Сучасні високоякісні системи цифрового запису звуку показують, що і цей об'єкт може бути адекватно описаний рядком символів.

Хоча уявлення даних – самостійна проблема в комп'ютерних науках, а ефективне представлення – це значною мірою мистецтво програміста, тим не менш, можна стверджувати, що існує універсальний спосіб представлення даних – словами в деякому алфавіті.

Таким чином, можна вважати, що алгоритм – це перетворення слів із заданого алфавіту: вихідне слово переробляється (листується) алгоритмом в результуюче слово. Яким би не був кінцевий алфавіт, будь-який його символ може бути закодований за допомогою двійкового алфавіту. Інакше кажучи, вхідні дані алгоритму можуть бути представлені кінцевої ланцюжком бітів. Те ж саме можна сказати і про вихідних даних. Кінцева ланцюжок бітів може інтерпретуватися як ціле невід'ємне число.

З цього можна зробити важливий висновок: будь-якому алгоритму може бути поставлена у відповідність функція, що відображає безліч невід'ємних цілих чисел.

Зворотне не стверджується. Більш того, існують функції, не обчислювані ніяким алгоритмом. Обчислюваною функцією будемо називати функцію, обчислювану деяким алгоритмом.

Деякі додаткові вимоги призводять до неформального визначенням алгоритму.

Визначення 1.1

Алгоритм – це заданий на деякій мові вираз, що задає кінцеву послідовність здійсненних елементарних операцій для вирішення завдання, загального для класу можливих вхідних даних.

Нехай D – область (безліч) вхідних даних завдання, а R – множина можливих результатів, тоді можна говорити, що алгоритм здійснює відображення

$$D \rightarrow R.$$

Оскільки таке відображення може бути не повним, то вводяться наступні поняття.

Визначення 1.2

Алгоритм називається **частковим** алгоритмом, якщо ми отримуємо результат тільки для деяких d є D і **повним** алгоритмом, якщо алгоритм отримує правильний результат для всіх d є D.

Незважаючи на зусилля дослідників відсутнє одне визначення поняття алгоритм. В теорії алгоритмів були введені різні формальні визначення алгоритму і дивним науковим результатом є доказ еквівалентності цих формальних визначень у розумінні їх рівноміцності. Варіанти словесного визначення алгоритму належать російським вченим А.Н. Колмогорову і А.А. Маркову [7].

Визначення 1.3

(Колмогоров): **Алгоритм** – це будь-яка система обчислень, що виконуються за строго визначеними правилами, яка після деякого числа кроків свідомо призводить до вирішення поставленого завдання.

Визначення 1.4 (Марков): **Алгоритм** – це точне розпорядження, що визначає обчислювальний процес, який йде від варійованих вхідних даних до шуканого результату.

Різні визначення алгоритму, в явній або неявній формі, постулюють наступний **ряд вимог**:

- алгоритм повинен містити кінцеву кількість елементарних записів (виконай-експортуй), тобто задовольняти вимогу кінцівки запису;
- алгоритм повинен виконувати кінцеву кількість кроків при вирішенні завдання, тобто задовольняти вимогу кінцівки дій;
- алгоритм повинен бути єдиним для всіх допустимих вхідних даних, тобто задовольняти вимогу універсальності;
- алгоритм повинен призводити до правильного рішення стосовно поставленого завдання, тобто задовольняти вимогу правильності.

Інші формальні визначення поняття алгоритму пов'язані з введенням спеціальних математичних конструкцій (машина Посту, машина Тьюринга, рекурсивно-обчислюваної функції Черча) і постулюванням тези про еквівалентність такого формалізму і поняття «алгоритм».

1.6 Питання для самоконтролю

1) Історичні аспекти створення та розробки теорії алгоритмів.
2) Цілі і завдання класичної теорії алгоритмів.
3) Цілі і задачі теорії асимптотичного аналізу алгоритмів.
4) Цілі і завдання практичного аналізу алгоритмів.
5) Теоретичний і практичний аспекти застосування результатів теорії алгоритмів.
6) Формалізація алгоритму, визначення Колмогорова і Маркова.
7) Основні вимоги до алгоритмів.
8) Три типи алгоритмів.

2. МАШИНА ПОСТА

2.1 Основні поняття та операції

Пост розглядає загальну проблему, що складається з безлічі конкретних проблем. При цьому рішення загальної проблеми це таке рішення, яке доставляє відповідь для кожної конкретної проблеми.

Наприклад, рішення рівняння 3*x +9 = 0 – це одна з конкретних проблем, а рішення рівняння a * x + b = 0 – це загальна проблема.

Алгоритм (сам термін «алгоритм» не використовується Постом) повинен бути універсальним, тобто повинен бути співвіднесений із загальною проблемою.

Основні поняття алгоритмічного формалізму Посту – це простір символів (мова L), в якому задається конкретна проблема. Відповіддю є і набір інструкцій або операцій в просторі символів, які задають як самі операції, так і порядок виконання інструкцій.

	V			V	V	V		V

Рис. 2.1. Постовський простір символів.

Постовський простір символів – це нескінченна стрічка комірок (ящиків). Кожен ящик або комірка можуть мати або не мати позначки див. рис. 2.1.

Конкретна проблема задається «зовнішньою силою» (термін Посту) – позначкою кінцевого кількості комірок, при цьому, очевидно, що будь-яка конфігурація починається і закінчується поміченою коміркою.

Після застосування до конкретної проблеми деякого набору інструкцій рішення представляється у вигляді набору помічених і непомічених комірок, які визначаються тією ж зовнішньою силою.

Пост запропонував набір інструкцій (елементарних операцій), які виконує «працівник» [3]. Відзначимо, що в 1936 році не було ще жодної електронної обчислювальної машини. Цей набір інструкцій є, очевидно, мінімальним набором бітових операцій:

1) Помітити ящик, якщо він порожній.
2) Стерти мітку, якщо вона є.
3) Переміститися вліво на 1 ящик.

4) Переміститися вправо на 1 ящик.
5) Визначити чи має ящик позначку або ні, і за результатом перейти на одну з двох зазначених інструкцій.
6) Зупинитися.

Відзначимо, що 1 і 2 інструкції включають захист від невірних ситуацій.

Програма являє собою нумеровану послідовність інструкцій, причому переходи в інструкції 5 проводяться на вказані в ній номери інших інструкцій.

2.2 Фінітний 1 – процес

Програма (набір інструкцій у термінах Посту) є однією і тією ж для всіх конкретних проблем.

Пост вводить такі поняття:

- набір інструкцій застосовується до загальної проблеми, якщо для кожної конкретної проблеми не виникає колізій в інструкціях 1 і 2, тобто ніколи програма не стирає мітку в порожньому ящику і не встановлює мітку в позначеному ящику;
- набір інструкцій закінчується (за кінцеве кількість інструкцій), якщо виконується інструкція (6);
- набір інструкцій задає **фінітний 1 – процес**, якщо набір може бути використаним, і закінчується для кожної конкретної проблеми;
- **фінітний 1 – процес** для загальної проблеми є 1 – рішення, якщо відповідь для кожної конкретної проблеми є правильною (це визначається зовнішньою силою).

2.3 Спосіб завдання проблеми та формулювання 1

За Постом проблема задається зовнішньою силою шляхом позначки кінцевою кількості ящиків стрічки.

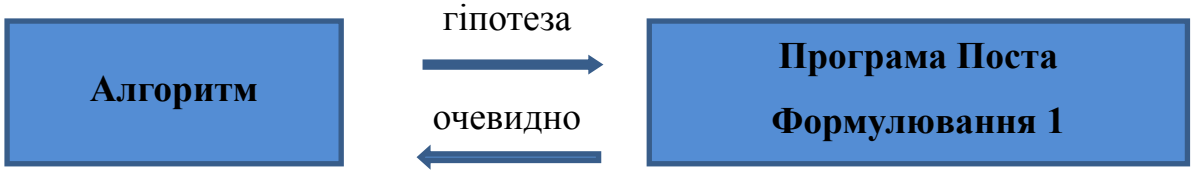
Прийнято вважати, що машина працює в одиничній системі числення (0 = V; 1 = VV; 2 = VVV; 3 = VVVV), тобто нуль представляється одним поміченим ящиком, а ціле позитивне число – поміченими ящиками в кількості на одиницю більше його значення.

Оскільки безліч конкретних проблем, що становлять загальну проблему є рахунковим, то можна встановити взаємно однозначну відповідність (биєктивне відображення) між множиною позитивних цілих чисел N і множиною конкретних проблем.

Загальна проблема називається по Посту **1-заданой**, якщо існує такий **фінітний 1 – процес**, що, будучи, застосованим до **n є N** в якості вхідної конфігурації ящиків, він задає n-ю конкретну проблему у вигляді набору помічених ящиків.

Якщо загальна проблема 1-задана і 1-розв'язана, то, поєднуючи набори інструкцій за завданням проблеми, і її вирішенням отримуємо відповідь за номером проблеми. Це і є в термінах Посту **формулювання 1.**

Таким чином, гіпотеза Посту полягає в тому, що будь-які більш широкі формулювання в сенсі алфавіту символів стрічки, набору інструкцій, подання та інтерпретації конкретних проблем зводяться до **формулювання 1.**



Отже, якщо гіпотеза вірна, то будь-які інші формальні визначення, що задають певний клас алгоритмів, еквівалентні класу алгоритмів, заданих **формулюванням 1** Еміля Посту.

2.4 Принцип роботи

Машина Поста складається з каретки (каретки, яка зчитує або записує) і безмежної в обидві сторони смуги, що поділена на комірки (ящики). Кожна комірка смуги може бути або порожньою – 0, або помічено міткою – 1. За один крок каретка може переміститися на одну позицію вліво або вправо, зчитати, поставити або стерти символ в тому місті, де вона стоїть. Робота машини Посту визначається програмою, що складається з кінцевого числа рядків.

Для роботи машини необхідно задати програму і її початковий стан (тобто стан стрічки і позиції каретки). Кареткою управляє програма, що складається з рядків команд. Кожна команда має наступний синтаксис:

i K j,

де **i** – номер команди,
K – дія каретки,
j – номер наступної команди (перехід).

Всього для машини Поста існує шість типів команд (див. рис. 2.2):

1 – V j – поставити мітку, перейти до j-го рядка програми.
2 – X j – витерти мітку, перейти до j-го рядка програми.
3 – <- j – пересунуться вліво, перейти до j-го рядка програми.
4 – -> j – пересунуться вправо, перейти до j-го рядка програми.
5 – ? j1; j2 – якщо комірка немає мітки, то перейти до j1-го рядка програми, інакше перейти до j2-го рядка програми.
6 – ! – кінець програми (стоп).

У команди «стоп» посилань нема.

Після запуску можливі варіанти:

- робота може закінчитися командою, що не виконується (витирання існуючою мітки або записування в помічене поле);
- робота може закінчитися командою Stop;
- робота ніколи не закінчиться.

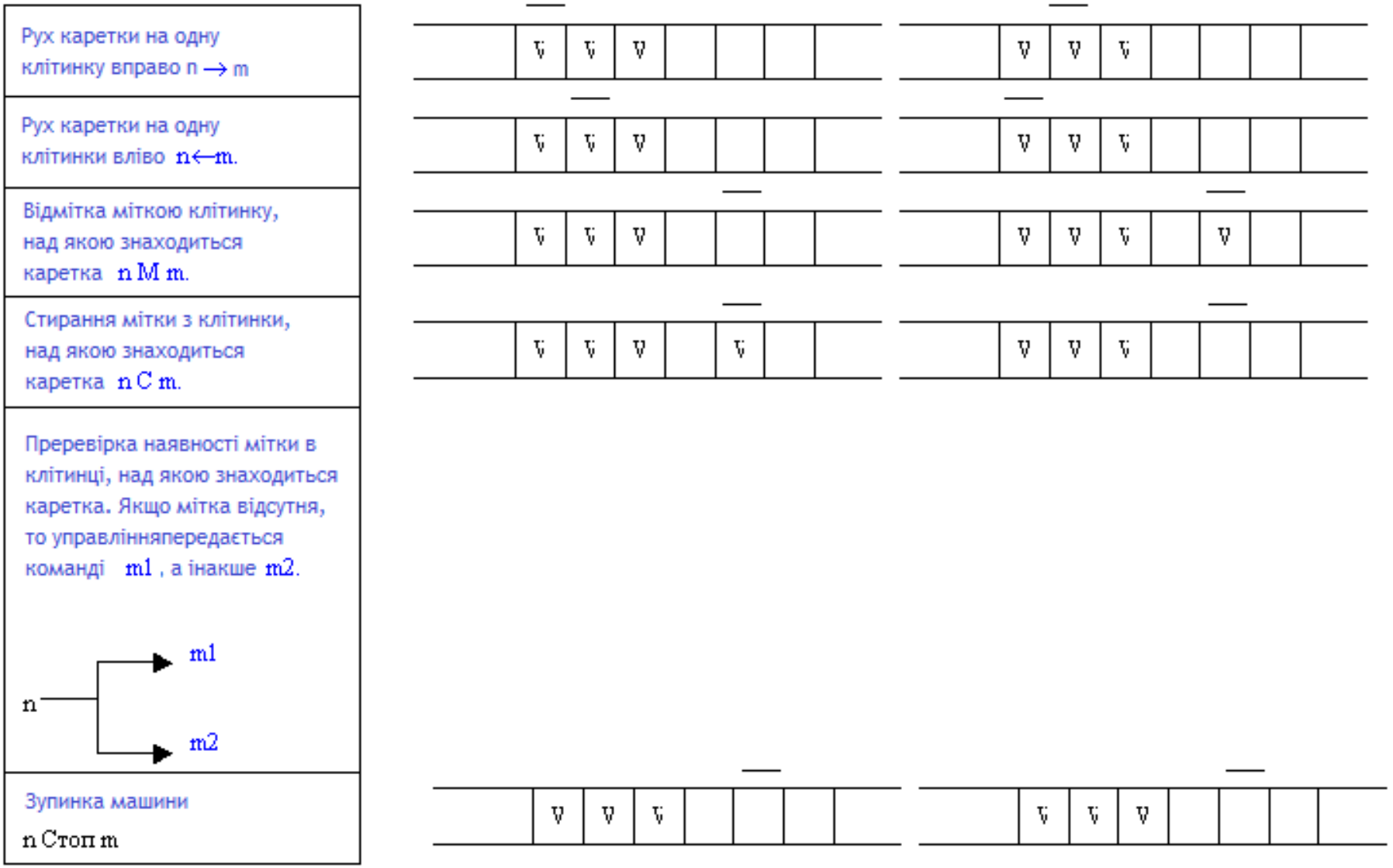


Рис. 2.2. Шість команд машини Поста.

У кожен момент часу каретка («-») знаходиться над однією з клітин стрічки і, як кажуть, обдивляється її. Інформація про місця розташування каретки разом із станом стрічки характеризує стан машини Посту рис. 2.3.

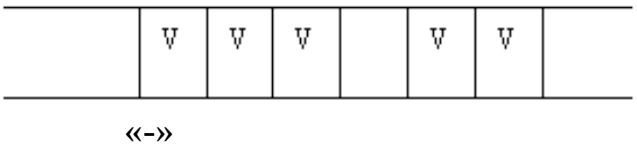


Рис. 2.3. Стан машини Посту.

Ситуації, в яких каретка повинна наносити мітку там, де вона вже є, або навпаки, витирати мітку там, де її немає, є **аварійними** (недопустимими).

Програмою для машини Поста називають не порожній список команд, такій, що :

- на n-м місті команда с номером n;
- номер кожної команди співпадає з номером будь-якої команди списку.

З точки зору властивостей алгоритмів, що вивчаються за допомогою машини Посту, найбільший інтерес представляють причини зупинки машини при виконанні програми:

- **останов по команді "стоп".** Такій останов називається результативним і вказує на коректність алгоритму (програми);
- **останов при виконанні неприпустимою команди.** В цьому випадку останов називається без результативним;
- **машина не зупиняється ніколи.** У цьому і в попередньому випадку ми маємо справу з некоректним алгоритмом (програмою).

Будемо розуміти під початковим станом каретки її становище проти порожньої клітини лівіше найлівішої мітки на стрічці.

Число k представляється на стрічці машини Посту (k + 1) мітками, що йдуть підряд. Одна мітка означає число «0». Між двома числами робиться інтервал як мінімум з однієї порожньої секції на стрічці.

Наприклад, запис чисел 3 і 5 на стрічці машини Посту буде виглядати так:

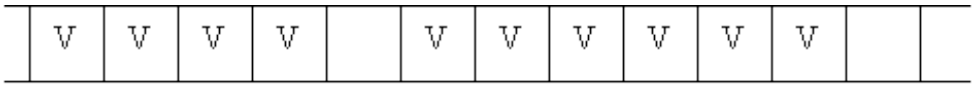


Рис. 2.4. Запис чисел 3 і 5 на стрічці машини Поста.

Система запису чисел, що використовується в машині Посту є непозиційною.

Складемо програму для додавання до довільного числа одиниці. Припустимо, що на стрічці записано тільки одне число і каретка знаходиться над однією з клітин, в якій знаходиться мітка, що належить цьому числу:

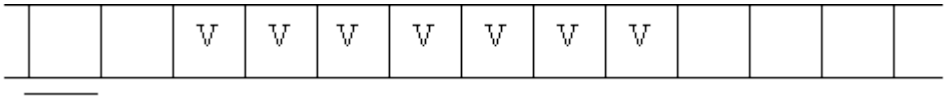


Рис. 2.5. Збільшення числа на одиницю.

Для рішення задачі можна перемістити каретку вліво (або вправо) до першої порожньої клітинки, а потім нанести мітку див. рис. 2.5-2.7.

Програма, що додає до числа мітку справа, має вигляд:

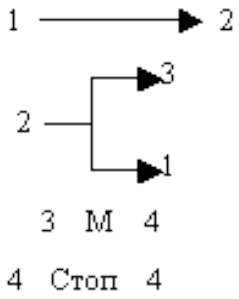


Рис. 2.6. Текст програми, що додає мітку справа.

Програма, що додає до числа мітку зліва, має вигляд:

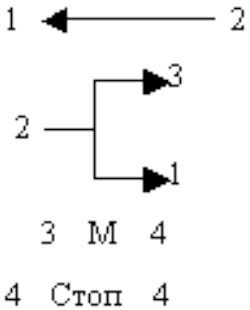


Рис. 2.7. Текст програми, що додає мітку зліва.

Різниця тільки в напрямку руху каретки в першій команді.

Машину Посту можна розглядати як спрощену модель ЕОМ. Справді, як ЕОМ, так і машина Посту мають:

- неподільні носії інформації (клітини – біти), які можуть бути заповненими або незаповненими;
- обмежений набір елементарних дій – команд, кожна з яких виконується за один такт (крок).

Обидві машини працюють на основі програми. Проте в машині Посту інформація розташовується лінійно і читається поспіль, а в ЕОМ можна читати інформацію за адресою; набір команд ЕОМ значно ширше і виразніше, ніж команди машини Посту, і т.д.

Приклад.

Віднімання натуральних чисел P — Q.

Будемо представляти натуральне (ціле невід'ємне) число P набором з P+1 одиниць і розділять числа нулем. Початкове положення каретки помічено символом «v»

```
    v
00111110111000
  P      Q
```

Додавання двох чисел є тривіальним — достатньо поставити 1 між ними і стерти два останніх правих символи у Q.

Програма віднімання складається з послідовного затирання крайніх лівих міток у Q і правих у P:

```
1. 0        – витираємо лівий символ у Q
2. →
3. ? 4, 5
4. Stop     – стоп, якщо затерли Q=0
5. ←
6. ? 5, 7   – цикл пошуку P
7. 0        – витираємо правий символ у P
8. →
9. ? 8, 1   – шукаємо Q
```

Відмітимо, що номер команди переходу не вказується, якщо перехід відбувається на наступний по порядку рядок (для наочності тексту). В 6-ому рядку можливе за циклювання, якщо Q > P (можна добавити перевірку)

Обґрунтування цієї гіпотези відбувається сьогодні не шляхом строго мате-автоматичного доведення, а на шляху експерименту. Дійсно, всякий раз, коли вказується алгоритм, його можна перевести в форму програми машини Посту, яка призводить до того ж результату.

2.5 Питання для самоконтролю

1) Поняття загальної та конкретної проблеми по Посту.
2) Простір символів і примітивні операції в машині Поста.
3) Поняття фінітного 1-процесу в машині Поста.
4) Способи завдання проблем і формулювання 1.
5) Гіпотеза Поста.

3. МАШИНА ТЬЮРІНГА ТА ПРОБЛЕМИ, ЯКІ НЕ РОЗВ'ЯЗУЮТЬСЯ АЛГОРИТМІЧНО

3.1. Машина Тьюринга

Алан Тьюринг (Turing) в 1936 році опублікував у працях Лондонського математичного товариства статтю, яка нарівні з роботами Посту і Черча, лежить в основі сучасної теорії алгоритмів.

Модель обчислень, в якій кожен алгоритм розбивався на послідовність простих, елементарних кроків і була логічною конструкцією, названої згодом машиною «Тьюринга».

Машина Тьюринга – це математична побудова, математичний апарат (аналогічний, наприклад, апарату диференціальних рівнянь), створений для вирішення певних завдань. Цей математичний апарат був названий "машиною" з тієї причини, що за описом його складових частин і функціонуванню він схожий на обчислювальну машину. Принципова відмінність машини Тьюринга від обчислювальних машин полягає в тому, що її запам'ятовуючий пристрій являє собою нескінченну стрічку: у реальних обчислювальних машин запам'ятовуючий пристрій може бути як завгодно великим, але обов'язково кінцевим. Машину Тьюринга не можна реалізувати саме через нескінченності її стрічки. В цьому сенсі вона потужніша будь-якої обчислювальної машини.

У кожній машині Тьюринга є дві частини:

1) необмежена в обидві сторони стрічка, розділена на клітинки;
2) автомат (каретка для зчитування / запису, керована програмою).

З кожною машиною Тьюринга пов'язані два кінцевих алфавіту:

- алфавіт вхідних символів A = {a0, a1, ..., am}
- алфавіт станів Q = {q0, q1, ..., qp}.

(З різними машинами Тьюринга можуть бути пов'язані різні алфавіти A і Q.)

Стан q0 називається пасивним. Вважається, що якщо машина потрапила в цей стан, то вона закінчила свою роботу.

Стан q1 називається початковим. Перебуваючи в цьому стані, машина починає свою роботу.

Вхідне слово розміщується на стрічці по одній літери в розташованих підряд клітинках. Ліворуч і праворуч від вхідного слова знаходяться тільки порожні клітинки (в алфавіт А завжди входить порожня буква а0 – ознака того, що клітинка порожня).

Автомат може рухатися вздовж стрічки вліво або вправо, читати вміст комірок і записувати в клітинку літери. Автомат кожного разу "бачить" тільки одну клітинку.

Залежно від того, яку літеру він бачить, а також залежно від свого стану qj автомат може виконувати наступні дії:

- записати нову букву в клітинку, що оглядається;
- виконати зрушення по стрічці на одну клітинку вправо / вліво або залишитися нерухомим;
- перейти в новий стан.

Тобто у машини Тьюринга є три види операцій. Щоразу для чергової пари (qj, ai) машина Тьюринга виконує команду, що складається з трьох операцій з певними параметрами.

Програми для машин Тьюринга записуються у вигляді таблиці, де перші стовпець і рядок містять літери зовнішнього алфавіту та можливі внутрішні стани автомата (внутрішній алфавіт). Вміст таблиці являє собою команди для машини Тьюринга. Літера, яку зчитує каретка в клітинці (над якою вона знаходиться в даний момент), і внутрішній стан каретки визначають, яку команду потрібно виконати. Команда визначається перетином символів зовнішнього і внутрішнього алфавітів в таблиці.

Щоб задати конкретну машину Тьюринга, потрібно описати для неї наступні складові:

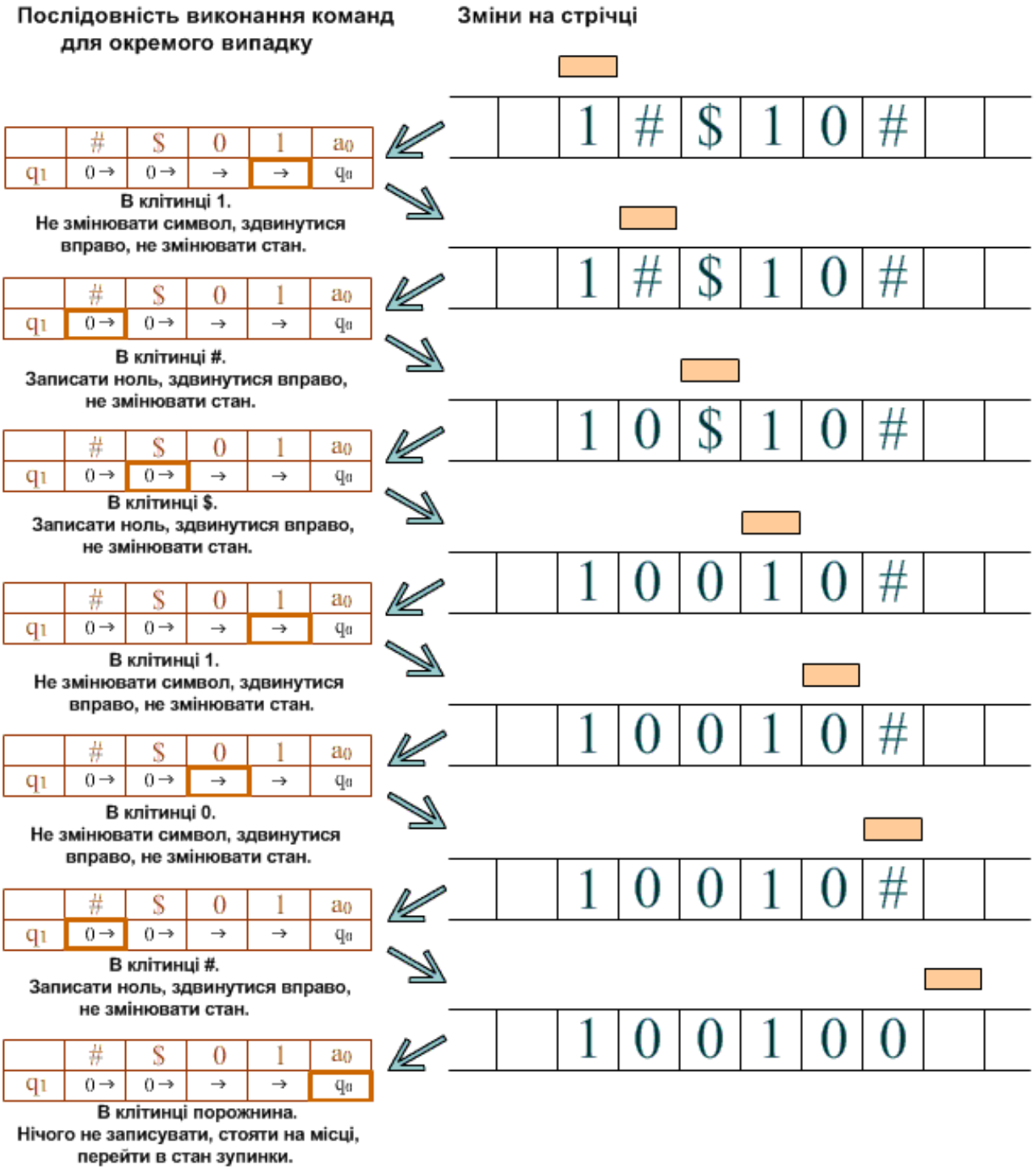
• ***Зовнішній алфавіт.*** *Кінцева безліч (наприклад, А), елементи якої називаються літерами (символами). Одна з літер цього алфавіту (наприклад, а0) повинна являти собою порожній символ.*

• ***Внутрішній алфавіт.*** *Кінцева безліч станів каретки (автомата). Один зі станів (наприклад, q1) повинний бути початковим (що запускає програму). Ще один зі станів (q0) повинний бути кінцевим (завершальним програму) – стан зупинки.*

• ***Таблиця переходів.*** *Опис поведінки автомата (голівки) в залежності від стану і символу, що зчитується.*

Одна команда для машини Тьюринга якраз і являє собою конкретну комбінацію цих трьох складових: вказівок, який символ записати в комірку (над якою стоїть автомат), куди пересунутися і в який стан перейти. Хоча команда може містити і не всі складові (наприклад, не змінювати символ, не пересуватися або не змінювати внутрішній стан).

Приклад роботи машини Тьюринга



Приклад 1.

Припустимо, на стрічці є слово, яке складається з символів ***#, $, 1 і 0*** Потрібно замінити всі символи **#** і ***$*** на нулі. В момент запуску каретка знаходиться над першою літерою слова зліва. Завершується програма тоді, коли каретка знаходиться над порожнім символом після літери слова, що є останньою з права.

Примітка: довжина слова і послідовність символів значення не мають. На рисунку наводиться приклад послідовності виконання команд для конкретного випадку.

Якщо на стрічці буде інше слово, то і послідовність виконання команд буде іншою. Незважаючи на це, дана програма для машини Тьюринга (на рисунку – таблиця зліва) застосовна до будь-яких слів описаного зовнішнього алфавіту (дотримується властивість застосовності алгоритму до всіх однотипних завдань – масовість).

Приклад 2.

Зовнішній алфавіт має вигляд A={0,1,2,3,4,5,6,7,8,9}.

Нехай P – не порожне слово, тобто P – це послідовність з десяткових невід'ємних цифр.

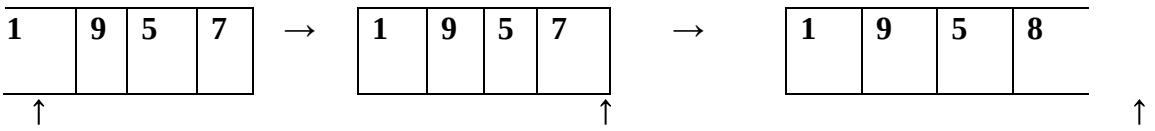
Треба отримати на стрічці запис числа, яке на одиницю більше ніж число P.

Рішення.

Для рішення цієї задачі необхідно виконати наступні дії:

1. Перегнати автомат під останню цифру числа.

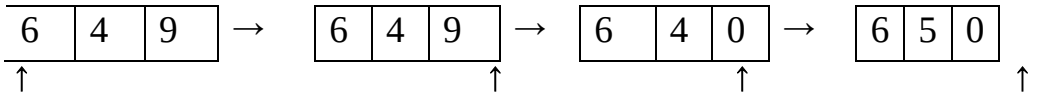
2. Якщо це цифра від 0 до 8, то замінити її цифрою на 1 більше і зупинитися, наприклад:



3. Якщо це цифра 9, тоді замінити її на 0 і перемістити автомат до попередньої цифри.

Після чого збільшити на одиницю передостанню цифру.

Наприклад:

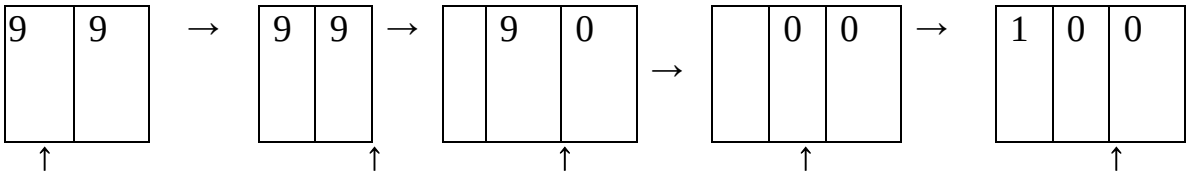


4. Особливий випадок.

В *P* є тільки дев'ятки (наприклад, 99).

Тоді автомат буде переміщуватися вліво, змінюючи дев'ятки на нулі, і в кінці кінців опиниться під порожньою клітинкою.

В цю пустую клітинку треба записати 1 і зупинитися (відповіддю буде 100):



У програми для машини Тюринга ці дії описуються наступним чином:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Λ**
q1	0, R, q1	1, R, q1	2, R, q1	3, R, q1	4, R, q1	5, R, q1	6, R, q1	7, R, q1	8, R, q1	9, R, q1	Λ , R, q2
q2	1, N, !	2, N, !	3, N, !	4, N, !	5, N, !	6, N, !	7, N, !	8, N, !	9, N, !	0, L, q2	1, N, !

Пояснення.

q1 – це стан, в якому автомат «біжить» під останню цифру числа. Для цього він весь час рухається вправо, не змінюючи видимі цифри і залишаючись у тому ж стані.

Але тут є одна особливість: коли автомат перебуває під останньою цифрою, то він ще не знає про це (адже він не бачить, що записано в сусідніх клітинах) і визначить це лише тоді, коли потрапить на порожню клітину.

Тому, дійшовши до першої порожній клітини, автомат повертається назад під останню цифру і переходить в стан q2 (вправо рухатися вже не треба).

q2 – це стан, в якому автомат додає 1 до тієї цифри, яку бачить в даний момент.

Спочатку це остання цифра числа:

- якщо вона – в діапазоні від 0 до 8, то автомат замінює її цифрою, яка на 1 більше, і зупиняється.
- Але якщо це цифра 9, то автомат замінює її на 0 і зсувається вліво, залишаючись в стані q2.

Тобто, тепер він буде додавати 1 до попередньої цифри. Якщо і ця цифра дорівнює 9, то автомат замінює її на 0 і зсувається вліво, залишаючись в стані q2, оскільки повинен виконати те ж саме діяння – збільшить на 1 видиму цифру. Якщо ж автомат зрушився вліво, а у видимій клітці немає цифри (а є «пусто»), то він записує сюди 1 і зупиняється.

Відзначимо, що для порожнього вхідного слова наша задача не визначена, тому на цьому слові машина Тюринга (МТ) може вести себе як завгодно. У нашій програмі, наприклад, при порожньому вхідному слові МТ зупиняється і видає відповідь 1.

Вище наведений запис програми в повному, нескороченому вигляді.

Наведемо запис програми в скороченому, більш наочному вигляді, при цьому праворуч дамо коротке пояснення дій, які реалізуються у відповідних станах автомата:

	0	1	2	3	4	5	6	7	8	9	ʌ	
q1	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,L, q2	Під останню цифру
q2	1,,!	2,,!	3,,!	4,,!	5,,!	6,,!	7,,!	8,,!	9,,!	0,L,	1,,!	Цифра +1, яка є видимою

3.2 Властивості машини Тьюринга як алгоритму

На прикладі машини Тьюринга добре простежуються властивості алгоритмів.

- **Дискретність.** Машина Тьюринга може перейти до (к + 1)-го кроку тільки після виконання к-го кроку, оскільки саме к-й крок визначає, яким буде (до + 1) -й крок.

- **Зрозумілість**. На кожному кроці в комірку пишеться символ з алфавіту, автомат робить один рух (Л, П, Н) і машина Тьюринга переходить в один з описаних станів.

- **Детермінованість.** У кожній клітці таблиці машини Тьюринга записаний лише один варіант дії. На кожному кроці результат визначений однозначно, отже, послідовність кроків розв'язання задачі визначена однозначно, тобто якщо машині Тьюринга на вхід подають одне й те ж слово, то вихідне слово кожен раз буде одним і тим же.

- **Результативність.** Змістовно результати кожного кроку і всієї послідовності кроків визначені однозначно, отже, правильно написана машина Тьюринга за кінцеве число кроків перейде в стан q0, тобто за кінцеве число кроків буде отримана відповідь на питання задачі.

- **Масовість.** Кожна машина Тьюринга визначена над усіма допустимими словами з алфавіту, в цьому і полягає властивість масовості. Кожна машина Тьюринга призначена для вирішення одного класу задач, тобто для кожного.

3.3 Проблеми, які не розв'язуються алгоритмічно

За час свого існування людство придумало безліч алгоритмів для вирішення різноманітних практичних і наукових проблем. ***Виникає питанням – а чи існують які-небудь проблеми, для яких неможливо придумати алгоритми їх вирішення?***

Успіхи математики до кінця XIX століття привели до формування думки, яку висловив Д. Гілберт – «в математиці не може бути проблем, які не розв'язуються алгоритмічно», на конгресі 1900 року в Парижі.

Першою фундаментальною теоретичною роботою, пов'язаною з доказом алгоритмічної нерозв'язності, була робота Курта Геделя – його відома теорема про неповноту символічних логік.

Зусиллями різних дослідників список алгоритмічно нерозв'язних проблем був значно розширений. Сьогодні прийнято при доказі алгоритмічної нерозв'язності деякої задачі зводити її до класичної задачі – «задачі зупину».

Теорема 3.1.

Не існує алгоритму (машини Тьюринга), що дозволяє за описом довільного алгоритму і його вхідних даних (і алгоритм і дані задані символами на стрічці машини Тьюринга) визначити, чи зупиняється цей алгоритм на цих даних або працює нескінченно.

Таким чином, фундаментально алгоритмічна нерозв'язність пов'язана з нескінченністю виконуваних алгоритмом дій, тобто неможливістю передбачити, що для будь-яких вхідних даних рішення буде отримано за кінцеву кількість кроків.

Тим не менш, можна спробувати сформулювати причини, що ведуть до алгоритмічної нерозв'язності. Ці причини достатньо умовні, оскільки всі вони зводяться до проблеми зупину, однак такий підхід дозволяє більш глибоко зрозуміти природу алгоритмічної нерозв'язності:

а) Відсутність загального методу розв'язання задачі

Проблема 1: Розподіл дев'яток в запису числа π

Визначимо функцію f (n) = i, де n – кількість дев'яток поспіль у десятинному запису числа π, а i – номер самої лівої дев'ятки з n дев'яток підряд: π = 3,141592 ... f (1) = 5.

Задача полягає в обчисленні функції f (n) для довільно заданого n.

Оскільки число π є ірраціональним і трансцендентним, то ми не знаємо жодної інформації про розподіл дев'яток (так само як і будь-яких інших цифр) в десятковому запису числа π.

Обчислення f (n) пов'язане з обчисленням наступних цифр в розкладанні π, до тих пір, поки ми не виявимо n дев'яток поспіль, однак у нас немає загального методу обчислення f (n), тому для деяких n обчислення можуть тривати нескінченно.

Не знаємо в принципі (за природою числа π) чи існує рішення для всіх n.

Проблема 2: Обчислення досконалих чисел

Досконалі числа – це числа, які дорівнюють сумі своїх дільників, наприклад:

28 = 1 +2 +4 +7 +14.

Визначимо функцію S(n) = n-ое за рахунком досконале число і сформулюємо задачу обчислення S(n) по довільно заданому n. Немає

загального методу обчислення досконалих чисел, навіть не відома множина досконалих чисел. Тому алгоритм повинен перебирати всі числа поспіль, перевіряючи їх на досконалість. Відсутність загального методу рішення не дозволяє відповісти на питання про умову зупинення алгоритму. Якщо перевірили М чисел при пошуку n-го досконалого числа – чи означає це, що його взагалі не існує?

Проблема 3: Десята проблема Гільберта

Нехай заданий многочлен n-го ступеня з цілими коефіцієнтами – Р, чи існує алгоритм, який визначає, чи має рівняння Р = 0 рішення в цілих числах?

Ю.В. Матіясевіч [3] показав, що такого алгоритму не існує, тобто відсутній загальний метод визначення цілих коренів рівняння Р = 0 за його цілочисловим коефіцієнтами.

б) Інформаційна невизначеність завдання

Проблема 4: Позиціонування машини Посту на останньому позначеному ящику

Нехай на стрічці машини Посту задані набори помічених ящиків (кортежів) довільної довжини з довільними відстанями між кортежами і каретка знаходиться у самого лівого поміченого ящика. Задача полягає в установці каретки на самий правий позначений ящик останнього кортежу.

Спроба побудови алгоритму, що вирішує це завдання, призводить до необхідності відповіді на питання – чи немає більше на стрічці кортежів або вони є десь правіше, коли після виявлення кінця кортежу ми зрушили вправо по порожніх ящиках на М позицій і не виявили початок наступного кортежу?

Інформаційна невизначеність задачі полягає у відсутності інформації або про кількість кортежів на стрічці, або про максимальні відстані між кортежами. За наявності такої інформації (при дозволі інформаційної невизначеності) задача стає алгоритмічно розв’язаною.

в) Логічна нерозв'язність

Проблема 5: Проблема еквівалентності алгоритмів

За двома довільно заданими алгоритмами (наприклад, для двох машин Тьюринга) визначити, чи будуть вони видавати однакові вихідні результати на будь-яких вхідних даних.

Проблема 6: Проблема тотальності

За довільно заданим алгоритмом визначити, чи буде він зупинятися на всіх можливих наборах вхідних даних. Інше формулювання цього завдання – чи є частковий алгоритм P усюди визначеним?

3.4 Питання для самоконтролю

1) Формальний опис машини Тьюринга.
2) Функція переходів в машині Тьюринга.
3) Поняття про алгоритмічно нерозв'язні проблеми.
4) Проблема позиціонування в машині Тьюринга.

4. ВСТУП ДО АНАЛІЗУ АЛГОРИТМІВ

4.1 Порівняльні оцінки алгоритмів

При використанні алгоритмів для вирішення практичних завдань виникає проблема раціонального вибору алгоритму розв'язання задачі. Рішення проблеми вибору пов'язане з побудовою системи порівняльних оцінок, яка спирається на формальну модель алгоритму

Будемо розглядати в подальшому правильні і фінітні алгоритми, тобто алгоритми, що дають 1-рішення загальної проблеми.

В якості формальної системи будемо розглядати абстрактну машину, що включає процесор з фон-Нейманівською архітектурою, яка підтримує адресну пам'ять і набір «елементарних» операцій співвіднесених з мовою високого рівня.

Припущення:

- кожна команда виконується не більше ніж за фіксований час;
- вхідні дані алгоритму представляються машинними словами по β бітів кожне.

Конкретна проблема задається N словами пам'яті, таким чином, на вході алгоритму – Nβ = N * β біт інформації. Відзначимо, що в ряді випадків, особливо при розгляді матричних задач, N є мірою довжини входу алгоритму, що вказує на лінійну розмірність.

Програма, що реалізує алгоритм для вирішення загальної проблеми складається з M машинних інструкцій з βм бітів – Mβ = M * βм біт інформації.

Крім того, алгоритм може вимагати таких додаткових ресурсів абстрактної машини:

- S_d – пам'ять для зберігання проміжних результатів;
- S_r – пам'ять для організації обчислювального процесу (пам'ять, необхідна для реалізації рекурсивних викликів і повернень).

При вирішенні конкретної проблеми, заданої N словами пам'яті алгоритм виконує не більше, ніж кінцеву кількість «елементарних» операцій абстрактної машини за умови розгляду тільки фінітних алгоритмів. У зв'язку з цим введемо таке визначення.

Визначення 4.1 Трудомісткість алгоритму.

Під трудомісткістю алгоритму для даного конкретного входу – Fa (N), розуміється кількість «елементарних» операцій, виконаних алгоритмом для вирішення конкретної проблеми в даній формальній системі [5].

Комплексний аналіз алгоритму може бути виконаний на основі комплексної оцінки ресурсів формальної системи, необхідних алгоритмом для вирішення конкретних проблем. Очевидно, що для різних областей застосування ваги ресурсів будуть різні, що призводить до наступної комплексної оцінки алгоритму:

$$\psi_A = c_1 * F_a(N) + c_2 * M + c_3 * S_d + c_4 * S_r,$$

де c_i – вага ресурсів різного виду.

4.2. Система позначень в аналізі алгоритмів

При більш детальному аналізі трудомісткості алгоритму виявляється, що не завжди кількість елементарних операцій, виконуваних алгоритмом на одному вході довжини N, збігається з кількістю операцій на іншому вході такої ж довжини. Це призводить до необхідності введення спеціальних позначень, що відображають поведінку функції трудомісткості даного алгоритму на вхідних даних фіксованої довжини.

Нехай DA – безліч конкретних проблем даної задачі, заданий у формальній системі. Нехай D ∈ DA – завдання конкретної проблеми і | D | = N [5].

У загальному випадку існує власна підмножина DA, що включає всі конкретні проблеми, які мають потужність N:

- позначимо цю підмножину через DN:DN ={D∈D_N,: |D|=N};
- позначимо потужність множини DN через MDN → MDN = |DN|.

Тоді даний алгоритм, вирішуючи різні завдання розмірності N, буде виконувати в якомусь випадку найбільшу кількість операцій, а в якомусь випадку найменшу кількість операцій.

Введемо такі позначення:

1. $F_a^{\wedge}(N)$ – найгірший випадок – найбільша кількість операцій, що здійснюються алгоритмом А для вирішення конкретних проблем розмірністю N:

$$F_a^{\wedge}(N) = \max_{D \in D_N} \{F_a(D)\}$$ – найгірший випадок на D_N

2. $F_a^{\vee}(N)$ – кращий випадок – найменша кількість операцій, що здійснюються алгоритмом А для вирішення конкретних проблем розмірністю N:

$$F_a^{\vee}(N) = \min_{D \in D_N} \{F_a(D)\}$$ – кращий випадок на D_N

3. $F_a(N)$ – середній випадок – середня кількість операцій, що здійснюються алгоритмом А для вирішення конкретних проблем розмірністю N:

$F_a(N) = (1 / M_{DN}) * \sum \{F_a(D)\}$ – середній випадок на D_N

4.3 Класифікація алгоритмів по виду функції трудомісткості

Залежно від впливу вхідних даних на функцію трудомісткості алгоритму запропонована наступна класифікація, що має практичне значення для аналізу алгоритмів.

1. Кількісно-залежні алгоритми за трудомісткістю

Це алгоритми, функція трудомісткості яких залежить тільки від розмірності конкретного входу, і не залежить від конкретних значень:

$F_a(D) = F_a(|D|) = F_a(N)$

Прикладами алгоритмів з кількісно-залежною функцією трудомісткості можуть служити алгоритми для стандартних операцій з масивами і матрицями – множення матриць, множення матриці на вектор тощо.

2. Параметрично-залежні алгоритми за трудомісткістю.

Це алгоритми, трудомісткість яких визначається не розмірністю входу (як правило, для цієї групи розмірність входу зазвичай фіксована), а конкретними значеннями оброблюваних слів пам'яті:

$$F_a(D) = F_a(d_1,\ldots,d_n) = F_a(P_1,\ldots,P_m),\ m \le n$$

Прикладами алгоритмів з параметрично-залежної трудомісткістю є алгоритми обчислення стандартних функцій із заданою точністю шляхом обчислення відповідних ступеневих рядів. Очевидно, що такі алгоритми, маючи на вході два числових значення – аргумент функції і точність виконують істотно залежне від значень кількість операцій.

а) Обчислення x^k послідовним множенням $\Rightarrow F_a(x, k) = F_a(k)$.

б) Обчислення $e^x = \sum (x^n/n!)$, з точністю до $\xi \Rightarrow F_a = F_a(x, \xi)$

3. Кількісно-параметричні за трудомісткістю алгоритми

У більшості практичних випадків функція трудомісткості залежить як від кількості даних на вході, так і від значень вхідних даних, в цьому випадку:

$F_a(D) = F_a(\ ||D||, P_1,\ldots,P_m) = F_a(\ N, P_1,\ldots,P_m)$

Як приклад можна навести алгоритми чисельних методів, в яких параметрично-залежний зовнішній цикл з точності включає в себе кількісно-залежний фрагмент з розмірності.

4. Порядково-залежні за трудомісткістю алгоритми

Серед розмаїття параметрично-залежних алгоритмів виділимо групу, для якої кількість операцій залежить від порядку розташування вхідних об'єктів.

Нехай множина D складається з елементів $(d_1,\ldots,d_n)$, i $||D||=N$,

Визначимо $D_p = \{(d_1,\ldots,d_n)\}$ – множину всіх впорядкованих N-*ок* з $d_1,\ldots,d_n$, відмітимо, що $|D_p|=n!$.

Якщо $F_a({}_iD_p) \neq F_a({}_jD_p)$, де ${}_iD_p,\ {}_jD_p \in D_p$, то алгоритм називають порядково-залежним за трудомісткістю.

Прикладами таких алгоритмів можуть служити ряд алгоритмів сортування, алгоритми пошуку мінімуму і максимуму в масиві.

Розглянемо більш докладно алгоритм пошуку максимуму в масиві S, що складається з n елементів:

MaxS (S,n; Max)
Max $\leftarrow S_1$
For i$\leftarrow$*2 to n*
if Max $< S_i$
then Max $\leftarrow S_i$

(кількість виконаних операцій присвоювання залежить від порядку розташування елементів масиву).

4.4 Асимптотичний аналіз функцій

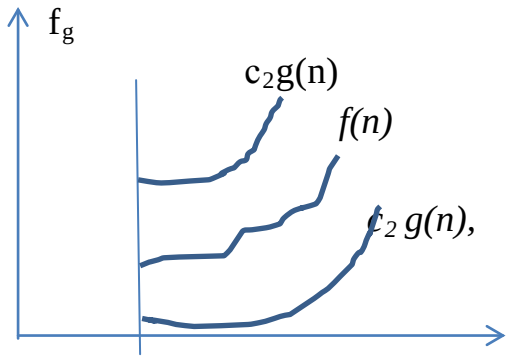
При аналізі поведінки функції трудомісткості алгоритму часто використовують прийняті в математиці асимптотичні позначення, що дозволяють показати швидкість росту функції, маскуючи при цьому конкретні коефіцієнти.

Така оцінка функції трудомісткості алгоритму називається **складністю алгоритму** і дозволяє визначити переваги у використанні того чи іншого алгоритму для більших значень розмірності вхідних даних.

У асимптотичному аналізі прийняті наступні позначення [5]:

1. Оцінка Θ (тетта)

Нехай $f(n)$ і $g(n)$ – додатні функції додатного аргументу, n ≥ 1 (кількість об'єктів на вході і кількість операцій – додатні числа), тоді:



$f(n) = \Theta(g(n))$, якщо існують додатні c_1, c_2, n_0, такі, що: $c_1 * g(n) \le f(n) \le c_2 * g(n)$, *при* $n > n_0$.

Зазвичай кажуть, що при цьому функція g (n) є асимптотично точною оцінкою функції f (n), тому що, за визначенням, функція f (n) не відрізняється від функції g (n) з точністю до постійного множника.

Відзначимо, що з $f(n) = \Theta(g(n))$ випливає наступне:

$$g(n) = \Theta(f(n)).$$

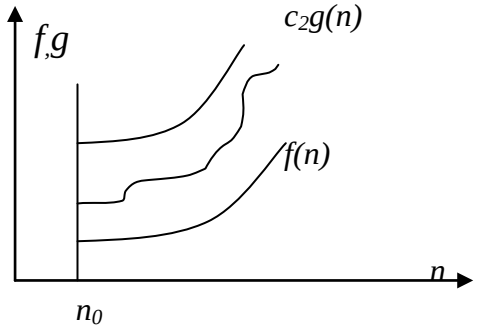
Приклади:

1) $f(n)=4n^2+nlnN+174 – f(n)= \Theta(n^2);$

2) $f(n)=\Theta(1)$ – запис означає, что $f(n)$ або дорівнює константі, яка не дорівнює нулю, або $f(n)$ обмежена константою на ∞: $f(n) = 7+1/n = \Theta(1)$.

2. Оцінка *O* (О велике)

На відміну від оцінки Θ, оцінка O потребує тільки, що б функція $f(n)$ не перевищувала $g(n)$ починаючи з $n > n0$, з точністю до постійного множника:



$$\exists\, c > 0, n0 > 0 :$$
$$0 \le f(n) \le c * g(n), \ \forall n > n0$$

Взагалі, запис $O(g(n))$ означає клас функцій, які зростають не швидше, ніж функція $g(n)$ з точністю до постійного множника. Тому іноді говорять, що $g(n)$ мажорірує функцію $f(n)$.

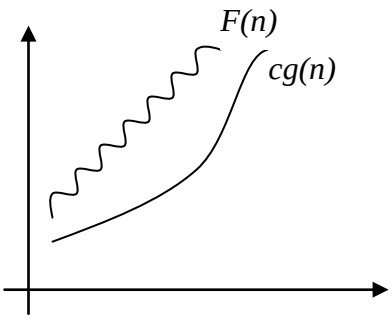
Наприклад, для всіх функцій:

$f(n)=1/n,$
$f(n)= 12,$
$f(n)=3*n+17,$
$f(n)=n*Ln(n),$
$f(n)=6*n^2 +24*n+77$

буде вірною оцінка $O(n^2)$. Вказуючи оцінку О є сенс вказувати найбільш «близьку» функцію, оскільки, наприклад, для f (n) = n2 є справедливою оцінка О (2n), проте вона не має практичного сенсу.

3. Оцінка Ω (Омега)

На відміну від оцінки *O*, оцінка Ω є оцінкою знизу – тобто визначає клас функцій, які зростають не повільніше, ніж *g(n)* з точність до постійного множника:



$$\exists\, c > 0,\ n0 > 0 :$$
$$0 \le c * g(n) \le f(n)$$

Наприклад, запис Ω (n * Ln(n)) позначає клас функцій, які зростають не повільніше, ніж g (n) = n * Ln(n). В цей клас потрапляють всі поліноми зі ступенем більшої одиниці, так само як і всі степеневі функції з основою більш ніж одиниця.

Відзначимо, що не завжди для пари функцій справедливо одне з асимптотичних співвідношень, наприклад для f (n) = n1 + sin(n) і g(n) = n не виконується жодна з асимптотичних співвідношень.

У асимптотичному аналізі алгоритмів розроблені спеціальні методи отримання асимптотичних оцінок, особливо для класу рекурсивних алгоритмів.

Очевидно, що Θ є переважною оцінкою ніж оцінка O. Знання асимптотики поведінки функції трудомісткості алгоритму – його складності, дає можливість робити прогнози щодо вибору більш раціонального з точки зору трудомісткості алгоритму для великих розмірностей вхідних даних.

4.5 Питання для самоконтролю

1) Формальна система мови високого рівня.
2) Поняття трудомісткості алгоритму у формальному базисі.
3) Узагальнений критерій оцінки якості алгоритму.
4) Система позначень в аналізі алгоритмів – найгірший, кращий і середній випадки.
5) Класифікація алгоритмів по виду функції трудомісткості.
6) Приклади кількісних і параметрично-залежних алгоритмів.

7) Позначення в асимптотичному аналізі функцій.
8) Приклади функцій, не пов'язаних асимптотичними.

5. ТРУДОМІСТЬ АЛГОРИТМІВ ТА ЇХ ЧАСОВІ ОЦІНКИ

5.1. Елементарні операції в мові запису алгоритмів

Для отримання функції трудомісткості алгоритму, представленого у формальній системі введеної абстрактної машини необхідно уточнити поняття «елементарних» операцій, співвіднесених з мовою високого рівня.

В якості таких «елементарних» операцій пропонується використовувати наступні:

1) Просте присвоювання:

$$a \leftarrow b.$$

2) Одновимірна індексація a [i]: (адреса (a) + i * довжина елементу).

3) Арифметичні операції: (*, /, -, +).

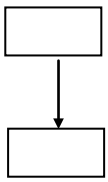
4) Операції порівняння: a < b.

5) Логічні операції (l1) {or, and, not} (l2).

Спираючись на ідеї структурного програмування, виключимо команду переходу за адресою, вважаючи її пов'язаною з операцією порівняння в конструкції розгалуження.

Після введення елементарних операцій аналіз трудомісткості основних алгоритмічних конструкцій у загальному вигляді зводиться до наступних положень:

а) конструкція «Послідовного переходу»

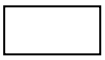


Трудомісткість конструкції є сума трудомісткості блоків, які виконуються послідовно друг за другом.

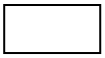
$F_{«Послідовного\ переходу»} = f_1 + \ldots + f_k$,
де k – кількість блоків.

б) конструкція «Розгалуження»

if (l) then

 f_{then} з ймовірністю p

else

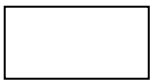
 f_{else} з ймовірністю $(1-p)$

Загальна трудомісткість конструкції «Розгалуження» вимагає аналізу ймовірності виконання переходів на блоки «Then» і «Else» і визначається як:

$$F_{«Розгалуження»} = f_{then} * p + f_{else} * (1-p).$$

в) конструкція «Цикл»

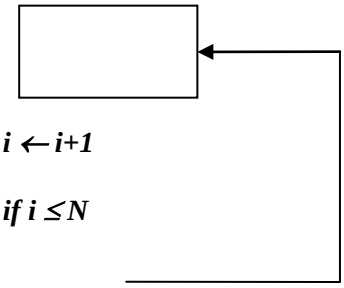
for i ← 1 to N



end

⇔

i ←1



i ← i+1

if i ≤ N

Після зведення конструкції до елементарних операцій її трудомісткість визначається як:

$$F_{«цикл»} = 1+3*N+N*f_{«тіло циклу»}$$

5.2 Приклади аналізу простих алгоритмів

Приклад 1.

Задача підсумовування елементів квадратної матриці

SumM (A, n; Sum)
Sum ← 0

```
For i ← 1 to n
        For j ← 1 to n
                Sum ← Sum + A[i,j]
end for
Return (Sum)
End
```

Алгоритм виконує однакову кількість операцій при фіксованому значенні n, отже є кількісно-залежним. Застосування методики аналізу конструкції «Цикл» дає:

FA (n) = 1 +1 + n * (3 +1 + n * (3 +4)) = 7 n2 +4 * n +2 = Q (n2),

зауважимо, що під n розуміється лінійна розмірність матриці, в той час як на вхід алгоритму подається *n2* значень.

Приклад 2.

Задача пошуку максимуму в масиві

```
MaxS (S,n; Max)
Max ← S[1]
For i ← 2 to n
  if Max < S[i]
        then   Max ← S[i]
end for
return Max
End
```

Даний алгоритм є кількісно-параметричним, тому для фіксованої розмірності вхідних даних необхідно проводити аналіз для гіршого, кращого і середнього випадків.

а) найгірший випадок

Максимальна кількість переприсвоєння максимуму (на кожному проході циклу) буде в тому випадку, якщо елементи масиву відсортовані за зростанням. Трудомісткість алгоритму в цьому випадку дорівнює:

$$F_A^{\wedge}(n)=1+1+1+ (n-1)\ (3+2+2)=7\ n-4 = \Theta(n).$$

б) кращий випадок

Мінімальна кількість переприсвоєння максимуму (жодного на кожному проході циклу) буде в тому випадку, якщо максимальний елемент розміщено на першому місці в масиві. Трудомісткість алгоритму в цьому випадку дорівнює:

$$F_A^{\vee}(n)=1+1+1+ (n-1)\ (3+2)=5\ n-2 = \Theta(n).$$

в) середній випадок

Алгоритм пошуку максимуму послідовно перебирає елементи масиву, порівнюючи поточний елемент масиву з поточним значенням максимуму.

На черговому кроці, коли переглядається к-тий елемент масиву, переприсвоєння максимуму відбудеться, якщо в підмасиві з перших к елементів максимальним елементом є останній. Очевидно, що у випадку даних рівномірного розподілу вхідних, ймовірність того, що максимальний з к елементів, розташований у деякій (останній) позиції, дорівнює 1/к. Тоді в масиві з n елементів загальна кількість операцій переприсвоєння максимуму визначається як:

$$\sum_{i=1}^{N} 1/i = Hn \approx Ln(N) + \gamma, \quad \gamma \approx 0{,}57$$

Величина Hn називається n-им гармонійним числом. Таким чином, точне значення (математичне очікування) середньої кількості операцій присвоювання в алгоритмі пошуку максимуму в масиві з n елементів визначається величиною Hn (на нескінченній кількості випробувань), тоді:

$$F_A(n)=1 + (n-1)\ (3+2) + 2\ (Ln(n) + \gamma) = 5\ n + 2\ Ln(n) - 4 + 2\ \gamma = \Theta(n).$$

5.3 Перехід до часових оцінок

Порівняння двох алгоритмів за їх функцією трудомісткості вносить помилку в одержувані результати. Основною причиною цієї помилки є різна частотна елементарних операцій, що зустрічаються, породжувана різними

алгоритмами і відмінність в часах виконання елементарних операцій на реальному процесорі.

Таким чином, виникає задача переходу від функції трудомісткості до оцінки часу роботи алгоритму на конкретному процесорі:

Дано: $F_A(D_A)$ – трудомісткість алгоритму. Потрібно визначити час роботи програмної реалізації алгоритму – $T_A(D_A)$.

На шляху побудови часових оцінок стикаються з цілим набором різних проблем, пофакторний облік яких викликає суттєві труднощі.

Зазначимо основні з цих проблем:

- неадекватність формальної системи запису алгоритму і реальної системи команд процесора;
- наявність архітектурних особливостей процесора істотно впливають на час виконання програми, таких як конвеєр, кешування пам'яті, предвибірки команд і даних тощо;
- різні часи виконання реальних машинних команд;
- відмінність у часі виконання однієї команди, залежно від значень операндів;
- різні часи реального виконання однорідних команд в залежності від типів даних;
- неоднозначності компіляції вхідного тексту, зумовлені як компілятором, так і його налаштуванням.

Спроби різного підходу до обліку цих факторів призвели до появи різноманітних методик переходу до тимчасових оцінками.

1) Пооперaційний аналіз

Ідея пооперaційного аналізу полягає в отриманні пооперaційної функції трудомісткості для кожної з використовуваних алгоритмом елементарних операцій з урахуванням типів даних.

Наступним кроком є експериментальне визначення середнього часу виконання цієї елементарної операції на конкретній обчислювальній машині. Очікуваний час виконання розраховується як сума добутків пооперaційної трудомісткості на середні часи операцій:

$$T_A(N) = \sum F_{A\ опер}(N) * \bar{t}_{опер},$$

де $\bar{t}_{опер}$ – час виконання елементарної операції.

2) Метод Гіббсона

Метод передбачає проведення сукупного аналізу за трудомісткістю і перехід до часових оцінок з урахуванням належності розв'язуваної задачі до одного з наступних типів:

- задачі науково-технічного характеру, в яких переважно використовуються операції з операндами дійсного типу;
- задачі дискретної математики з переважанням операцій з операндами цілого типу;
- задачі баз даних з переважанням операцій з операндами рядкового типу.

Далі на основі аналізу великої кількості реальних програм для вирішення відповідних типів задач визначається частотна використання операцій (рис 5.1). Створюються відповідні тестові програми, і визначається середній час на операцію в даному типі задач – $\bar{t}_{тип\ задачі}$.

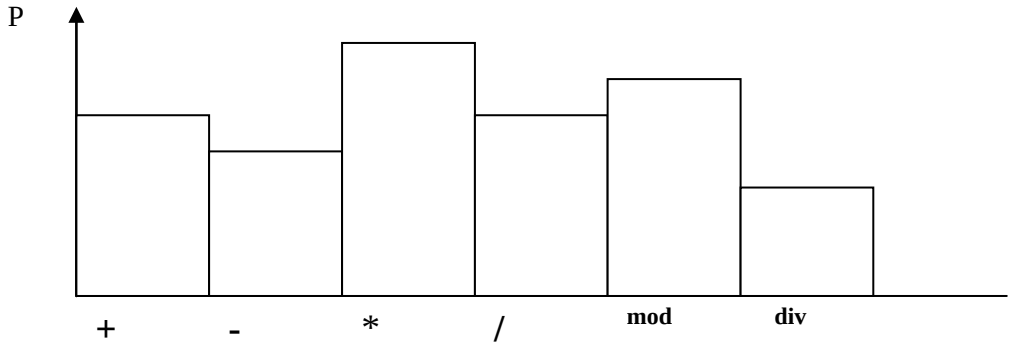


Рис 5.1. Можливий вид частоти використання операцій

На основі отриманої інформації оцінюється загальний час роботи алгоритму у виді:

$$T_A(N) = F_A(N) * \bar{t}_{тип\ задачи}$$

3) Метод прямого визначення середнього часу

У цьому методі так само проводиться сукупний аналіз за трудомісткістю – визначається $F_A\ (N)$. Після цього на основі прямого експерименту для різних значень N_e визначається середній час роботи даної програми T_e і за допомогою відомої функції трудомісткості розраховується середній час на

узагальнену елементарну операцію, що породжується даним алгоритмом, компілятором і комп'ютером – $\bar{t_a}$.

Ці дані можна (у припущенні про стійкість середнього часу по N) інтерполювати або екстраполювати на інші значення розмірності задачі таким чином:

$$\bar{t_a} = T_э(N_э) / F_A(N_A),\ T(N) = \bar{t_a} * F_A(N).$$

5.4 Приклад пооперaційного часового аналізу

У ряді випадків саме пооперaційний аналіз дозволяє виявити особливі аспекти раціонального застосування того чи іншого алгоритму розв'язання задачі.

Як приклад розглянемо задачу множення двох комплексних чисел:

$$(a+bi)*(c+di)=(ac - bd) + i(ad + bc)=e + if$$

1. ***Алгоритм А1*** *(пряме обчислення e, f – чотири множення)*

```
MultComplex1 (a, b, c, d; e, f)
    e←a*c – b*d          f* =4 операцій
    f←a*d + b*c          f± =2 операцій
    Return (e, f)        f← =2 операцій
                         fA1=8 операцій
End.
```

2. ***Алгоритм А2*** *(обчислення e, f за три множення)*

```
MultComplex2 (a, b, c, d; e, f)
    z1←c*(a + b)         f* =3 операцій
    z2←b*(d + c)         f± =5 операцій
    z3←a*(d – c)         f← =5 операцій
    e←z1 – z2            fA2=13 операцій
    f←z1 + z3

    Return (e, f)
End.
```

Пооперaційний аналіз цих двох алгоритмів не є важким (його результати наведені праворуч від запису відповідних алгоритмів).

За сукупною кількістю елементарних операцій алгоритм А2 поступається алгоритму А1, проте в реальних комп'ютерах операція множення вимагає більшого часу, ніж операція додавання.

Чи можна шляхом пооперaційного аналізу відповісти на питання: за яких умов алгоритм А2 переважніше алгоритму А1?

Введемо параметри q і r, що встановлюють співвідношення між часом виконання операції множення, додавання і присвоювання для операндів дійсного типу.

Тоді можна привести часові оцінки двох алгоритмів за часом виконання операції складання/віднімання – t +:

$$t_* = q*t_+,\ q>1;$$
$$t_{\leftarrow}=r*t_+,\ r<1.$$

Тоді приведені до t_+ часові оцінки мають вид:

$$T_{A1} = 4*q*t_+ +2*t_+ +2*r*t_+ =t_+ *(4*q+2+2*r);$$
$$T_{A2} = 3*q*t_+ +5*t_+ +5*r*t_+ =t_+ *(3*q+5+5*r).$$

Рівняння часів буде досягнуто при умові:

$$4*q+2+2*r = 3*q+5+5*r,$$

звідси:

$$q = 3 + 3*r$$

і відповідно при $q > 3 + 3*r$ алгоритм *A2* буде працювати більш ефективно.

Таким чином, якщо середовище реалізації алгоритмів А1 і А2 – мова програмування, обслуговуючий його компілятор і комп'ютер, на якому реалізується завдання, є такими, що час виконання операції множення двох дійсних чисел більш ніж втричі перевищує час складання двох дійсних чисел, (в припущенні, що r <<1, а це реальне співвідношення), то для реалізації більш кращий алгоритм А2.

Звичайно, виграш у часі дуже малий, якщо ми перемножуємо тільки два комплексних числа, проте, якщо цей алгоритм є частиною складної

обчислювальної задачі з комплексними числами, що вимагає суттєво значимого за часом кількості множень, то виграш у часі може бути відчутний.

Оцінка такого виграшу на одне множення комплексних чисел випливає з щойно проведеного аналізу:

$$\Delta T = (q - 3 - 3 * r) * t +$$

5.5 Питання для самоконтролю

1) Елементарні операції в псевдомові високого рівня.
2) Аналіз трудомісткості основних алгоритмічних конструкцій.
3) Побудова функції трудомісткості для підсумовування матриці.
4) Побудова функції трудомісткості для задачі пошуку максимуму.
5) Проблеми при переході від трудомісткості до часових оцінками.
6) Методики переходу від функції трудомісткості до часових оцінок.
7) Можливості пооперaційного аналізу алгоритмів на прикладі задачі множення комплексних чисел.

6. ТЕОРІЇ СКЛАДНОСТІ ОБЧИСЛЕНЬ І КЛАСИ СКЛАДНОСТІ ЗАДАЧ

6.1 Теоретична межа трудомісткості завдання

Розглядаючи деяку задачу, що має алгоритми розв'язку, і аналізуючи один з алгоритмів її вирішення, можна отримати оцінку трудомісткості цього алгоритму в найгіршому випадку – $A(D_A) = O(g(D_A))$. Такі ж оцінки можна отримати і для інших відомих алгоритмів вирішення даної задачі. При дослідженні задачі аналізу алгоритмів виникає питання – а чи існує нижня межа для $g(D_A)$ і якщо «так», то чи існує алгоритм, що її вирішує з трудомісткістю для найгіршого випадку.

Тобто, яка оцінка складності самого «швидкого» алгоритму розв'язку даної задачі в найгіршому випадку? Очевидно, що це оцінка самої задачі, а не будь-якого алгоритму її рішення. Таким чином, визначення поняття теоретичної нижньої межі трудомісткості завдання в найгіршому випадку має вид:

$$F_{min} = min\ \{\ \Theta(F_a^{\wedge}(D))\ \}$$

Якщо можливо на основі теоретичних міркувань довести існування і отримати функцію оцінки, то можна стверджувати, що будь-який алгоритм, який вирішує дану задачу працює не швидше, ніж з оцінкою Fmin в найгіршому випадку:

$$F_a^{\wedge}(D) = \Omega(F_{min})$$

Наведемо ряд прикладів:

1) Задача пошуку максимуму в масиві $A = (a_1, ..., an)$ – для цього завдання, очевидно, повинні бути переглянуті всі елементи і $Fmin = Q(n)$.

2) Задача множення матриць – для цього завдання можна зробити припущення, що необхідно виконати деякі арифметичні операції з усіма вхідними даними, теоретичне обґрунтування якої-небудь іншої оцінки на сьогодні не відомо [5], що приводить нас до оцінки $Fmin = Q(n^2)$.

Відзначимо, що кращий алгоритм множення матриць має оцінку $Q(n^2, 34)$ [6].

Розбіжність між теоретичною межею і оцінкою кращого відомого алгоритму дозволяє припустити, що або існує, але ще не знайдений більш швидкий алгоритм множення матриць, або оцінка $Q(n^2, 34)$ повинна бути доведена, як теоретична межа.

6.2 Класи складності задач

На початку 1960-х років, у зв'язку з початком широкого використання обчислювальної техніки для вирішення практичних задач, виникло питання про межі практичної застосовності даного алгоритму розв'язання задачі в сенсі обмежень на її розмірність. Які задачі можуть бути вирішені на ПК за реальний час?

Відповідь на це питання було дано в роботах Кобм (Alan Cobham, 1964), і Едмондса (Jack Edmonds, 1965), де були введені класи складності задач.

1) **Клас P** (задачі з поліноміальною складністю)

Задача називається поліноміальною, тобто відноситься до класу P, якщо існує константа k і алгоритм, що вирішує задачу з

$$F_a(n) = O(n^k),$$

де n – довжина входу алгоритму в бітах n = |D| [5].

Задачі класу P – це задачі, які вирішуються за реальний час.

Відзначимо переваги алгоритмів з цього класу:

- для більшості задач з класу P константа k менше 6;
- клас P інваріантний до моделі обчислень (для широкого класу моделей);
- клас P має властивість природної замкнутості (сума або добуток поліномів є поліном).

Таким чином, задачі класу P є уточнення визначення «практично вирішуваної» задачі.

2) **Клас NP** (задачі, що перевіряються поліноміально)

Нехай деякий алгоритм дає рішення деякої задачі. Чи відповідає отримана відповідь поставленій задачі, і на скільки швидко можна перевірити правильність рішення?

Розглянемо, наприклад, задачу про суму:

Задано N чисел – A = (a_1, ... an) і число V.

Треба знайти вектор (масив) X = (x_1, ..., xn), xi є {0,1}, такий, що $\sum a_i * x_i = V$.

Тобто треба визначити, чи може бути представлено число V у вигляді суми будь-яких елементів масиву A.

Якщо деякий алгоритм визначає масив X, то перевірка правильності цього результату може бути виконана з поліноміальною складністю: перевірка $\sum a_i * x_i = V$ вимагає не більше Q (N) операцій.

Формально: $\forall D \in D_A$, $|D|=n$ поставимо у відповідність сертифікат $S \in S_A$, такий, що | S | = O (n^l) і алгоритм As = As (D, S), такий, що він видає « 1 », якщо рішення правильно, і «0», якщо рішення невірно.

Тоді задача належить класу NP, якщо F (As) = O (n^m) [6].

Змістовно задача відноситься до класу NP, якщо її рішення за допомогою деякого алгоритму може бути швидко (поліноміально) перевірено.

6.3 Проблема P = NP

Після введення в теорію алгоритмів понять класів складності Едмондсом (Edmonds, 1965) була сформульована основна проблема теорії складності –

Чи вірно твердження P = NP?

Словесне формулювання проблеми має вигляд: чи можна всі задачі, вирішення яких перевіряється з поліноміальною складністю, вирішити за поліноміальний час? [5].

Очевидно, що будь-яка задача, що належить класу P, належить і класу NP, оскільки вона може бути поліноміально перевірена – задача перевірки рішення може складатися просто в повторному рішенні задачі.

На сьогодні відсутні теоретичні докази як збігу цих класів (P = NP), так і їх неспівпадання.

Припущення полягає в тому, що клас P є власною підмножиною класу NP, тобто NP \ P не порожньо – рис 6.1.

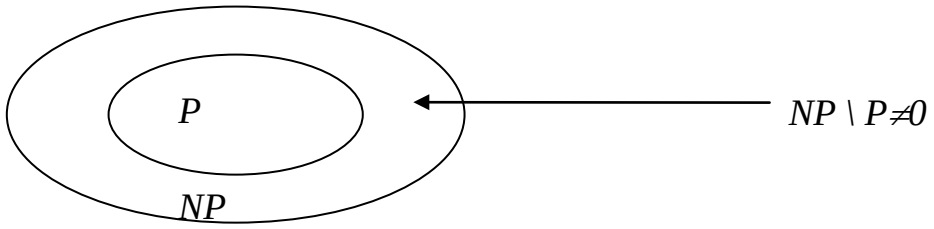


Рис 6.1. Співвідношення класів P і NP

6.4 Клас NPC (NP – повні задачі)

Поняття NP – повноти було введено незалежно Куком (Stephen Cook, 1971) і Левіним (журнал «Проблеми передачі інформації», 1973, т. 9, вип. 3) і ґрунтується на понятті зводимості однієї задачі до іншої [5].

Зводимість може бути представлена наступним чином: якщо ми маємо задачу 1 і алгоритм, що вирішує цю задачу і видає правильну відповідь для всіх вхідних даних, що складають задачу, а для задачі 2 алгоритм рішення невідомий, то, якщо можна переформулювати (звести) задачу 2 в термінах задачі 1, то можна вирішити задачу 2.

Таким чином, якщо задача 1 задана множиною конкретних проблем D_{A1}, а задача 2 – множиною, і існує функція f_s (алгоритм), що зводить конкретну постановку задачі 2 (D_{A2}) до конкретної постановки задачі 1 (D_{A1}):

$$f_s(d_{(2)} \in D_{A2}) = d_{(1)} \in D_{A1},$$

то задача 2 зводиться до задачі 1.

Якщо при цьому F_A (f_s) = O (n^k), тобто алгоритм зведення належить до класу P, то говорять, що задача 1 поліноміально зводиться до задачі 2 [5].

Прийнято говорити, що задача задається деякою мовою. Тобто якщо задача 1 задана мовою L1, а задача 2 – мовою L2, то поліноміальна зводимість визначається наступним чином:

$$L2 \leq_p L1.$$

Визначення класу NPC (NP-complete) або класу NP-повних задач вимагає виконання наступних двох умов: по-перше, задача має належати класу

$$NP\ (L \in NP),$$

і, по-друге, до неї поліноміально повинні зводитися всі задачі з класу

$$NP\ (L_x \leq_p L, \text{ для кожного } L_x \in NP),$$

що схематично представлено на рис 6.2.

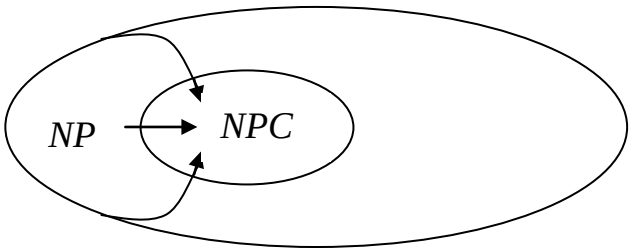


Рис 6.2. Зводимість NP класу і NPC

Для класу NPC доведена наступна теорема:

Якщо існує задача, що належить класу NPC, для якої існує поліноміальний алгоритм розв'язку ($F = O(n^k)$), то клас P збігається з класом NP, тобто P = NP [5].

Схема доказу полягає у зведенні будь-якої задачі з NP до даної задачі з класу NPC з поліноміальною трудомісткістю і вирішенні цієї задачі за поліноміальний час (за умовою теореми).

В даний час доведено існування сотень NP-повних задач [5,6], але ні для однієї з них **поки не вдалося** знайти поліноміального алгоритму рішення.

Крім того дослідники припускають наступне співвідношення класів, показане на рис 6.3.

$$P \neq NP,$$

тобто $NP \setminus P \neq 0$, і задачі з класу NPC не можуть бути вирішені (сьогодні) з поліноміальною трудомісткістю.

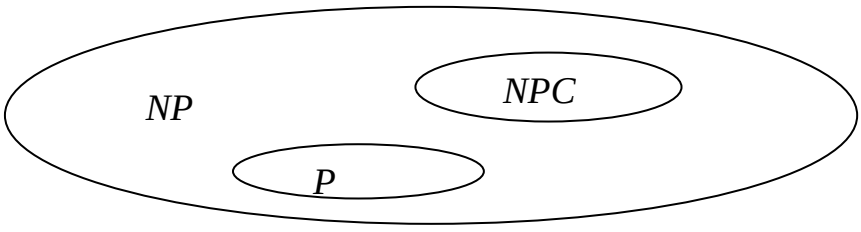


Рис 6.3. Співвідношення класів P, NP, NPC

6.5 Приклади NP – повних задач

6.5.1 Задача про виконуваність схеми

Розглянемо схему з функціональних елементів «і», «або», «ні» з n бітовими входами і одним виходом, що складається не більше, ніж з $O(n^k)$ елементів – рис 6.4.

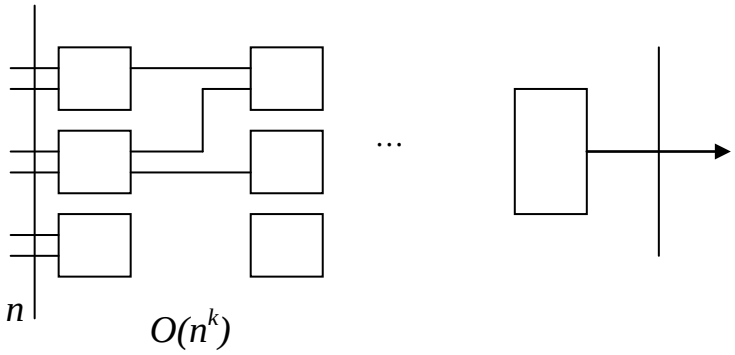


Рис 6.4. Абстрактна функціональна схема

Будемо вважати під виконуючим набором значень з множини {0,1} на вході схеми, такий набір входів – значення x1, ..., xn, при якому на виході схеми буде значення «1».

Формулювання задачі – чи існує для даної схеми набір значень входу x1, ..., xn.

Задача належить класу NP – перевірка знайденого набору x1, ..., xn не складніше кількості функціональних елементів, і отже не більше ніж $O(n^k)$.

Це була одна з перших задач, для якої було доведено її NP повнота, тобто будь-яка задача з класу NP поліноміально зводиться до задачі про здійснимість схеми [5].

Вирішення цієї задачі може бути отримано перебором всіх 2n можливих значень входу з подальшою перевіркою на відповідність умові ви-виконуючого набору. У найгіршому випадку доведеться перевірити всі можливі значення входу, що призводить до оцінки $F^{\wedge}(n) = O(n^k * 2^n)$.

Для цієї, як і для всіх інших NP-повних задач, не відомий поліноміальний алгоритм вирішення.

6.5.2 Задача про суму

Вже розглянута задача про суму також є NP-повною. Відмітимо, якщо кількість доданків фіксована, то складність задачі є поліноміальної, так як:

- для 2-х доданків $\Rightarrow C_N^2=(N*(N-1))/(1*2) = O(N^2);$
- для 3-х доданків $\Rightarrow C_N^3=(N*(N-1)*(N-2))/(1*2*3) = O(N^3).$

В загальному випадку доведеться перебирати 2N різних варіантів, оскільки за біноміальною теоремою

$$(a+b)^N = \sum c_N^k * a^{N-k} * b^k,$$

а при $a=b=1,$ маємо:

$$(1+1)^N = \sum C_N^k = 2^N,$$

отже,

$$F_A (N, V) = O(N * 2^N).$$

6.5.3 Задача про клік

Нехай дано граф G = G (V, E), де V – безліч з n вершин, а E – безліч ребер. Будемо розуміти під кліком максимальний за кількістю вершин повний підграф на графі в G.

Задача полягає у визначенні кліку в заданому графі G.

Оскільки в повному графі на m вершинах мається m*(m-1)/2 ребер, то перевірка, чи є даний граф повним, має складність O (m^2).

Якщо ми розглядаємо підграф з **m** вершинами в графі G з вершинами (m <n), то всього існує Cnm різних підграфів. Якщо в задачі про клік кількість вершин кліки фіксоване, то алгоритм розв'язку має поліноміальну складність:

$$F(m, n) = O(m^2 * C_n^m) = O(m^2 * n^m).$$

Однак у загальному випадку доведеться перевіряти всі підграфи з кількістю вершин m = (2, n) на їх повноту і визначити максимальне значення m для якого в даному графі G існує повний підграф, що призводить до оцінки в найгіршому випадку:

$$F^\wedge(n) = \sum_k O(k^2 * C_n^k) \Rightarrow O (n^2 * 2^n)$$

6.6 Питання для самоконтролю

1) Теоретична межа трудомісткості завдання.
2) Основні задачі теорії складності обчислень, поняття реально вирішуваних завдань.
3) Поняття с класів складності задач, клас P.
4) Клас складності NP, поняття сертифіката.
5) Проблема P = NP, та її сучасний стан.
6) Зводимість мов і визначення класу NPC.
7) Приклади NP – повних задач.
8) Задача про клік та її особливості.

7. ПРИКЛАД ПОВНОГО АНАЛІЗУ АЛГОРИТМУ ВИРІШЕННЯ ЗАДАЧІ ПРО СУМУ

7.1 Формулювання задачі і асимптотична оцінка

Словесно задача про суму формулюється як задача знаходження таких чисел з даної сукупності, які в сумі дають задане число. Класично завдання формулюється в термінах цілих чисел [5].

У термінах структур даних мови високого рівня завдання формулюється, як задача визначення таких елементів вихідного масиву S з N чисел, які в сумі дають число V (відзначимо, що задача належить до класу NPC).

Детальне формулювання:

Задано: *Масив S[i], i={1, N} і число V.*
Необхідно: *Визначити такі* S_j, *що* $\sum S_j = V$

Тривіальне рішення визначається рівністю V=Sum, де Sum=$\sum S_i$, умови існування рішення мають вид:

$$\text{Min } \{S[i], i=1,N\} \le V \le \text{Sum}.$$

Отримаємо асимптотичну оцінку складності рішення даної задачі для алгоритму, що використовує прямий перебір всіх можливих варіантів. Оскільки вихідний масив містить N чисел, то перевірці на рівність V підлягають такі варіанти рішень:

- V містить 1 доданок $\Rightarrow$ CN1 = N варіантів;
- V містить 2 доданків $\Rightarrow$ CN2 = (N * (N-1)) / (1 * 2) варіантів;
- V містить 3 доданків $\Rightarrow$ CN3 = (N * (N-1) * (N-2)) / (1 * 2 * 3) варіантів;
- І т.д. до перевірки одного варіанту з N доданками.

Оскільки сума біноміальних коефіцієнтів для ступеня N дорівнює

$$(1+1)^N = \sum C_N^k = 2^N$$

і для кожного варіанту необхідно виконати підсумовування (з оцінкою O(N)) для перевірки на V, то оцінка складності алгоритму в гіршому випадку має вигляд:

$$F^{\wedge}_A(N, V) = O(N*2^N) \qquad (7.1)$$

7.2 Алгоритм точного рішення задачі про суму (метод перебору)

Визначимо допоміжний масив, що зберігає поточне поєднання вихідних чисел в масиві S, що підлягають перевірці на V – масив *X[j]*, елемент масиву дорівнює «0», якщо число S [j] не входить в V і дорівнює «1», якщо число S [j] входить до V.

Рішення отримано, якщо

V = $\sum S[j]*X[j]$.

Можуть бути запропоновані наступні дві реалізації механізму повного перебору варіантів:

• перебір за всілякими сполученням з k елементів по N. Тобто спочатку алгоритм намагається представити V як один з елементів масиву S, потім перебираються всі можливі пари, потім всі можливі трійки тощо;

• перебір за допомогою бінарного лічильнику, реалізованому в масиві X.

Друга ідея алгоритмічно більш проста і зводиться до вирішення задачі про збільшення двійкового лічильника в масиві X на «1»:

• при 00 ... 0111 збільшення на «1» призводить до скиду всіх правих «1» і установці в «1» наступного самого правого «0»;

• при 00 ... 1000, коли останній елемент лічильника дорівнює «0» збільшення на «1» призводить до переустановлення останнього елемента в масиві *X* з «0» в «1».

Розглядаючи масив X як інформацію про елементи масиву S, що підлягають підсумовуванню в даний момент, треба робити підсумовування і перевірку на рівність V, до тих пір, поки рішення не буде знайдено або ж безрезультатно будуть переглянуті всі можливі варіанти.

Таким чином, алгоритм точного рішення задачі про суму методом прямого перебору має у формальній системі мови високого рівня наступний вигляд:

```
        TASKSUM(S,N,V; X,FL)
FL ← false
i ← 1
repeat
  X[i] ← 0
```

```
        i ← i+1
Until i > N
X[N] ← 1
Repeat
  Sum ← 0
  i ← 1
  Repeat
Sum ← Sum + S[i] * X[i]
    i ← i+1
  Until i > N
if Sum = V
        FL ← true
  Return (X,FL)
j ← N
While X[j] = 1
                X[j] = 0
                j ← j-1
  X[j] ← 1
Until X[0] = 1
Return(X,FL)
```

7.3 Аналіз алгоритму точного рішення задачі про суму

Розглянемо найкращий і найгірший випадок для даного алгоритму:

а) У кращому випадку, коли останній елемент масиву збігається зі значенням V : V = S [N], необхідно виконати тільки одне підсумовування, що призводить до оцінки:

$F_a\check{}(N)=Q(N);$

б) У гіршому випадку, якщо рішення взагалі немає, то доведеться перевірити всі варіанти, і

$F_a\hat{}(N) = Q (N*2^N).$

Отримаємо детальну оцінку для гіршого випадку, використовуючи прийняту методику підрахунку елементарних операцій у вигляді:

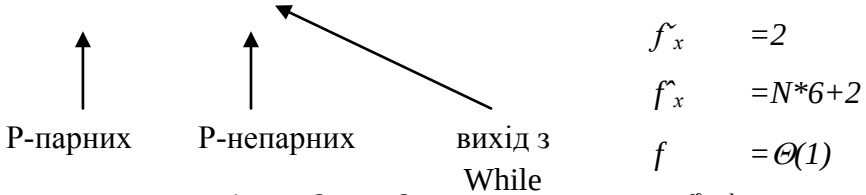
$$F_a\hat{}(N) = 2+N*(3+2)+2+(2^N-1)*\{2+N*(3+5)+1+1+f_{cnt}+2+2\} \qquad (7.2)$$

Для отримання значення fcnt – кількості операцій, необхідних для збільшення лічильника на «1» розглянемо по кроках проходи циклу While, в якому виконується ця операція:

X	кількість проходів в While	операцій
001	1	6+2
010	0	2
011	2	2*6+2
100	0	2
101	1	6+2
110	0	2
111	3	3*6+2

Таким чином:

$f_{cnt} = (1/2)*(2)+(1/2)*(2)+(1/2)*((1/2)*1*6+(1/4)*2*6+(1/8)*3*6+\ldots) =$



$=2 + 1/2 * 6 * (1/2^1+2/2^2+3/2^3+\ldots) = 2+ 3 * (\sum_{k=1}^{\infty} k/2^k);$

Оскільки $\sum k*x^k = x/(1-x)^2$, [5]

то $\sum k*(1/2)^k = (1/2)/(1-(1/2))^2 = 2,$

і, відповідно:

$f_x = 8$ (! і не залежить від довжини лічильника)

Підстановка f_x в (7.2) дає:

$F\hat{}_A(N) = 4+5*N+(2^N-1)*(8*N+16),$

і остаточно:

$$F\hat{}_A(N) = 8*N*2^N + 16*2^N - 3*N - 12,$$

що узгоджується з асимптотичною оцінкою – формула (1).

7.4 Питання для самоконтролю

1) Формулювання завдання про суму.
2) Асимптотична оцінка складності алгоритму для прямого перебору.
3) Алгоритм розв'язання задачі про суму.
4) Підалгоритм збільшення на одиницю довічного лічильника.
5) Оцінки трудомісткості для кращого і гіршого випадку.
6) Функція трудомісткості алгоритму для вирішення завдання про суму в гіршому випадку.

8. РЕКУРСИВНІ ФУНКЦІЇ І АЛГОРИТМИ

8.1 Рекурсивні функції

Однією з ідей процедурного програмування, яка оформилася на початку шістдесятих років XX століття, стало активне застосування в практиці програмування деякого методу, заснованого на організації серій взаємних звернень програм (функцій) один до одного. Питання про ефективність використання даного методу при розробці алгоритмічних моделей актуальні і в даний час, незважаючи на існування різних парадигм програмування, створення нових і вдосконалення існуючих мов програмування. Мова йде про рекурсивному методі в програмуванні, який розглядається альтернативним по відношенню до ітераційного.

По суті один і той же метод, стосовно до різних областей носить різні назви – це індукція, рекурсія і рекурентні співвідношення – відмінності стосуються особливостей використання.

Під індукцією розуміється метод доведення тверджень, який будується на базі індукції при n = 0,1, потім твердження покладається правильним при n = n_b і проводиться доказ для n +1.

Рекурсія – це визначення об'єкта через звернення до самого себе.

Рекурсивний алгоритм – це алгоритм, в описі якого прямо або побічно міститься звернення до самого себе. У техніці процедурного програмування дане поняття поширюється на функцію, яка реалізує рішення окремого блоку завдання за допомогою виклику зі свого тіла інших функцій, в тому числі і себе самої. Якщо при цьому на черговому етапі роботи функція організовує звернення до самої себе, то така функція є рекурсивною.

Термін **рекурентні** співвідношення пов'язаний з американським науковим стилем і визначає математичне завдання функції за допомогою рекурсії.

Пряме звернення функції до самої себе припускає, що в тілі функції міститься виклик цієї ж функції, але з іншим набором фактичних параметрів. Такий спосіб організації роботи називається прямою рекурсією. Наприклад, щоб знайти суму перших n натуральних чисел, треба суму перших (n-1) чисел скласти з числом n, тобто має місце залежність: Sn = Sn-1 + n. Обчислення

відбувається за допомогою аналогічних міркувань. Такий ланцюжок взаємних звернень в кінцевому підсумку зведеться до обчислення суми одного першого елемента, яка дорівнює самому елементу.

При непрямому зверненні функція містить виклики інших функцій зі свого тіла. При цьому одна або декілька з викладенихя функцій на певному етапі звертаються до вихідної функції зі зміненим набором вхідних параметрів. Така організація звернень називається **непрямою** рекурсією. Наприклад, пошук максимального елемента в масиві розміру n можна здійснювати як пошук максимуму з двох чисел: одне з них – це останній елемент масиву, а інше є максимальним елементом в масиві розміру (n-1). Для знаходження максимального елемента масиву розміру (n-1) застосовуються аналогічні міркування. У підсумку рішення зводиться до пошуку максимального з перших двох елементів масиву.

Рекурсивний метод в програмуванні передбачає розробку рішення задачі, грунтуючись на властивостях рекурсивності окремих об'єктів або закономірностей. При цьому вхідна задача зводиться до вирішення аналогічних підзадач, які є більш простими і відрізняються іншим набором параметрів.

Розробці рекурсивних алгоритмів передує рекурсивна тріада – етапи моделювання завдання, на яких визначається набір параметрів і співвідношень між ними. Рекурсивну тріаду складають **параметризація, виділення бази і декомпозиція.**

На етапі **параметризації** з постановки задачі виділяються параметри, які описують вхідні дані. При цьому деякі подальші розробки рішення можуть вимагати введення додаткових параметрів, які не обумовлені в умови, але використовуються при складанні залежностей. Необхідність у додаткових параметрах часто виникає також при вирішенні завдань оптимізації рекурсивних алгоритмів, в ході яких скорочується їхня тимчасова складність.

Виділення **бази рекурсії** припускає знаходження в розв'язуваній задачі тривіальних випадків, результат для яких очевидний і не потребує проведення розрахунків. Вірно знайдена база рекурсії забезпечує завершеність рекурсивних звернень, які в кінцевому підсумку зводяться до базового випадку. Перевизначення бази або її динамічне розширення в ході рішення задачі часто дозволяють оптимізувати рекурсивний алгоритм за рахунок досягнення базового випадку за більш короткий шлях звернень.

Декомпозиція являє собою зведення загального випадку до більш простих підзадач, які відрізняються від вихідної задачі набором вхідних даних. Декомпозиційні залежності описують не тільки зв'язок між задачам і підзадачами, а й характер зміни значень параметрів на черговому кроці. Від обраних відносин залежить трудомісткість алгоритму, так як для однієї і тієї ж задачі можуть бути складені різні залежності. Перегляд відносин декомпозиції доцільно проводити комплексно, тобто паралельно з коригуванням параметрів і аналізом базових випадків.

Існує декілька категорій завдань, що допускають рекурсивні визначення. Одна з категорій – математичні формули, визначення яких рекурсивне за своєю суттю.

Основним завданням дослідження рекурсивних функцій є отримання F(n) в явній або як ще кажуть «замкнутій» формі, тобто у вигляді аналітично заданої функції.

Класичний приклад функції, визначення якої може задаватися в рекурсивної формі, F (n) = n!

F(n)=n*(n-1)*…*1, 0!=1

Рекурсивне визначення цієї функції має вид:

$$F(n) = \begin{cases} 1, n = 0 \\ n * F(n-1), n > 0 \end{cases}$$

де F(n-1)=(n-1)!

Другий приклад – це опис синтаксису конструкцій мов програмування за допомогою Бэкуса Наура форми (БНФ).

Приклад.
Визначення ідентифікатора в мові Паскаль: зараз С++
<літера>::=a|b|…|z,
<цифра>::=0|1|…|9;
<ідентифікатор>::=<літера>|<ідентифікатор><літера>|<ідентифікатор><цифра>.

8.2. Рекурсивні процедури і функції

Категорії задач, що дозволяють рекурсивні визначення:

1. Задачі, математичні моделі яких записуються у вигляді рекурсивно визначених функцій.
2. Структура даних задачі визначаться рекурсивним чином. Наприклад: графи, дерева, списки.
3. Методи рішення задачі допускають рекурсивне визначення (ігри, головоломки тощо)

Рекурсивні алгоритми реалізуються у вигляді підпрограм, які визначаються в програмі, як процедури або функції.

Підпрограма називається рекурсивною, якщо в її визначенні присутній прямо або побічно виклик самої обумовленої підпрограми.

Явна рекурсія характеризується існуванням у визначеній підпрограмі оператора звернення до неї самої.

Неявна (непряма) рекурсія характеризується тим, що одна підпрограма звертається до іншої, яка через ланцюжок викликів інших підпрограм рекурсивно звертається до першої.

Реалізація рекурсивних підпрограм

Стек – структура даних, яка містить впорядкований набір елементів. Якщо в стек заноситься новий елемент, він додається в кінець упорядкованого набору (на вершину стека). При видаленні елемента він теж вибирається з кінця набору (з вершини стека).

В основі реалізації рекурсивної підпрограми лежить структура даних, що називається стеком, в якому зберігаються всі не глобальні дані, що беруть участь у всіх викликах підпрограми, при яких вона ще не завершила свою роботу.

Стек складається з фрагментів, які становлять блоки послідовних осередків.

Кожен виклик підпрограми використовує фрагмент стека, довжина якого залежить від підпрограми, що викликається.

У загальному випадку при виклику процедурою А процедури В відбувається наступне:

1. В вершину стека поміщається фрагмент потрібного розміру. До нього входять такі, дані:

(а) показники фактичних параметрів виклику процедури В;

(б) порожні комірки для локальних змінних, визначених у процедурі В;

(в) адреса повернення (АВ), тобто адреса команди в процедурі А, яку слід виконати після того, як процедура В закінчить свою роботу.

Якщо В – функція, то у фрагмент стеку для В поміщається покажчик осередку у фрагменті стека для А, в якій належить помістити значення цієї функції (адреса значення).

2. Управління передається першому оператору процедури В.

3. При завершенні роботи процедури В управління передається процедурі А за допомогою наступної послідовності кроків:

(а) адреса повернення витягується з вершини стеку;

(б) якщо В – функція, то її значення запам'ятовується в комірці, на яку вказує покажчик адреси значення;

(в) фрагмент стека процедури В витягується з стека, в вершину ставиться фрагмент процедури А;

(г) виконання процедури А поновлюється з команди, зазначеної в адресі повернення.

При виклику підпрограмою самої себе, тобто в рекурсивному випадку, виконується та ж сама послідовність дій.

Рекурсивне визначення складається з двох незалежних частин: ***базової*** **і** ***рекурсивної***.

Базова частина не є рекурсивним ствердженням, вона задає визначення для деякої фіксованої групи об'єктів і визначає умову закінчення рекурсії. У першому прикладі в базовій частині стверджується, що об'єкт 0! = 1.

Рекурсія і ітерація

Не завжди рекурсивне рішення задачі є кращим, більш прості рішення можуть бути знайдені за допомогою ітерації. Клас функцій, що мають визначення виду

$$F_n(x) = \begin{cases} G(x), \text{если } n = 0, \\ H(F_{n-1}(x)), \text{если } n > 0. \end{cases}$$

може завжди бути виражений ітеративно так, що застосування рекурсії необов'язково.

Приклад. Необхідно скласти визначення функції для обчислення значений деяких поліномів, визначення яких має наступний вигляд:

$$S_n(x) = \begin{cases} 0, \textit{если } n = 0, \\ 2x, \textit{если } n = 1, \\ \frac{2n}{n-1} S_{n-1}(x) + \frac{n-1}{2n} S_{n-2}(x), \textit{если } n>1 \end{cases}$$

Розглянемо ряд прикладів:

б) Приклади рекурсивних функцій

1. $\begin{cases} f(0)=0 \\ f(n)= f(n-1)+1 \end{cases}$

Зрозуміло, що $f(n)=n$.

2. $\begin{cases} f(0)=1 \\ f(n)= n*f(n-1) \end{cases}$

Послідовна підстановка дає – f(n) = 1*2*3* ... *n = n!

Для повноти відомостей наведемо формулу Стірлінга для наближеного обчислення факторіала для великих n:

$n! \approx (2\pi n)^{1/2} *(n^n)/(e^n)$

3. $\begin{cases} f(0)=1 \\ f(1)=1 \\ f(n)= f(n-1)+ f(n-2), \quad n\geq 2 \end{cases}$

Ця рекурсивна функція визначає числа Фібоначчі: 1 1 2 3 5 8 13, які досить часто виникають при аналізі різних завдань, у тому числі і при аналізі алгоритмів. Відзначимо, що асимптотично f (n) » [1,618 n] [8].

4. $\begin{cases} f(0)=1 \\ f(n)= f(n-1)+ f(n-2)+\ldots+1 = \Sigma f(i)+1 \end{cases}$

Для отримання функції в явному вигляді розглянемо її послідовні значення: $f(0)=1$, $f(1)=2$, $f(2)=4$, $f(3)=8$, що дозволяє припустити, що $f(n)=2^n$, точний доказ виконується по індукції.

5. $\begin{cases} f(0)=1 \\ f(n)= 2*f(n-1) \end{cases}$

Ми маємо справу з прикладом того, що одна і та ж функція може мати різні рекурсивні визначення – *f(n) = 2n*, як і в прикладі 4, що перевіряється елементарною підстановкою.

6. $\begin{cases} f(0)=1 \\ f(1)=2 \\ f(n)= f(n-1)*f(n-2) \end{cases}$

У цьому випадку ми можемо отримати рішення в замкнутій формі, зіставивши значенням функції відповідають ступені двійки:

$f(2) = 2 = 2^1$

$f(3) = 4 = 2^2$

$f(4) = 8 = 2^3$

$f(5) = 32 = 2^5$

$f(6) = 256 = 2^8$

Позначаючи через F_n – *n -ое* число Фібоначчі, маємо: $f(n)=2^{Fn}$.

8.3 Аналіз трудомісткості рекурсивних алгоритмів методом підрахунку вершин дерева рекурсії

Рекурсивні алгоритми відносяться до класу алгоритмів з високою ресурсоємністю, так як при великій кількості самовивозу рекурсивних функцій відбувається швидке заповнення стекової області. Крім того, організація зберігання та закриття чергового шару рекурсивного стека є додатковими операціями, які вимагають тимчасових витрат. На трудомісткість рекурсивних алгоритмів впливає і кількість переданих функцією параметрів.

Розглянемо один з методів аналізу трудомісткості рекурсивного алгоритму, який будується на основі підрахунку вершин рекурсивного дерева. Для оцінки трудомісткості рекурсивних алгоритмів будується **повне дерево**

рекурсії. Воно являє собою граф, вершинами якого є набори фактичних параметрів при всіх викликах функції, починаючи з першого звернення до неї, а ребрами – пари таких наборів, відповідних взаємним викликам. При цьому вершини дерева рекурсії відповідають фактичним викликам рекурсивних функцій. Слід зауважити, що одні й ті ж набори параметрів можуть відповідати різним вершинам дерева. Корінь повного дерева рекурсивних викликів – це вершина повного дерева рекурсії, відповідна початковим зверненням до функції.

Важливою характеристикою рекурсивного алгоритму є **глибина рекурсивних викликів** – найбільше одночасне кількість рекурсивних звернень функції, що визначає максимальну кількість шарів рекурсивного стека, в якому здійснюється зберігання відкладених обчислень. Кількість елементів повних рекурсивних звернень завжди не менше глибини рекурсивних викликів. При розробці рекурсивних програм необхідно враховувати, що глибина рекурсивних викликів не повинна перевершувати максимального розміру стека використовуваного обчислювального середовища.

При **цьому обсяг рекурсії** – це одна з характеристик складності рекурсивних обчислень для конкретного набору параметрів, що представляє собою кількість вершин повного рекурсивного дерева без одиниці.

8.4 Рекурсивна реалізація алгоритмів

Більшість сучасних мов високого рівня підтримують механізм рекурсивного виклику, коли функція, як елемент структури мови програмування, що повертає обчислене значення по своєму імені, може викликати сама себе з іншим аргументом. Ця можливість дозволяє безпосередньо реалізовувати рекурсивне обчислення певних функцій.

Рекурсивний алгоритм може бути реалізований ітераційною послідовністю дій.

Розглянемо приклад рекурсивної функції, що обчислює факторіал: *F(n).*

If n=0 or n=1 (перевірка можливості прямого обчислення)

Then $F \leftarrow 1$

Else $F \leftarrow n*F(n-1);$ (рекурсивний виклик функції)

Return (F);

End;

Аналіз трудомісткості рекурсивних реалізацій алгоритмів, очевидно, пов'язаний як з кількістю операцій, виконуваних при одному виклику функції, так і з кількістю таких викликів. Графічне представлення породжуваної даними алгоритму ланцюжка рекурсивних викликів називається деревом рекурсивних викликів. Більш детальний розгляд призводить до необхідності врахування витрат як на організацію виклику функції та передачі параметрів, так і на повернення обчислених значень і передачу управління в точку виклику.

Можна помітити, що деяка гілка дерева рекурсивних викликів обривається при досягненні такого значення переданого параметра, при якому функція може бути обчислена безпосередньо. Таким чином, рекурсія еквівалентна конструкції циклу, в якому кожен прохід є виконанням рекурсивної функції з заданим параметром.

Розглянемо приклад для функції обчислення факторіалу (рис 8.1):

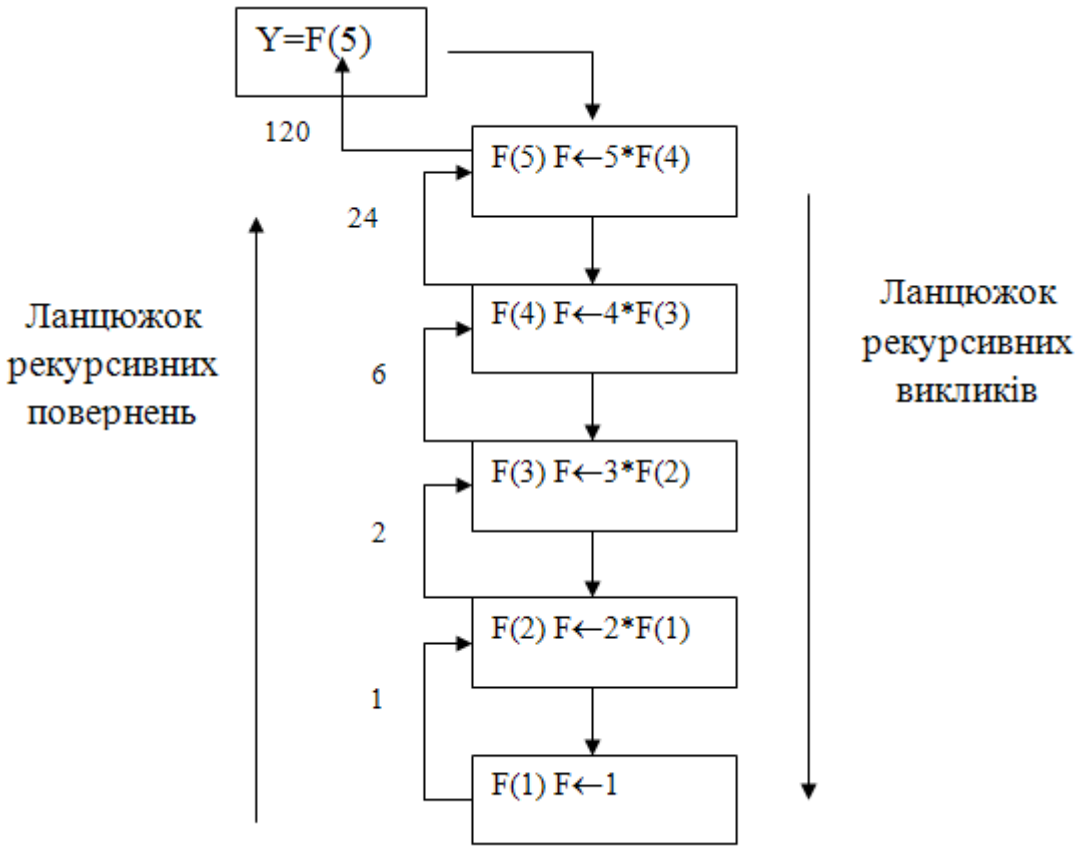


Рис 8.1. Дерево рекурсії при обчисленні факторіалу – F(5)

Дерево рекурсивних викликів може мати і більш складну структуру, якщо на кожному виклику породжується кілька звернень – фрагмент дерева рекурсій для чисел Фібоначчі представлений на рис 8.2.

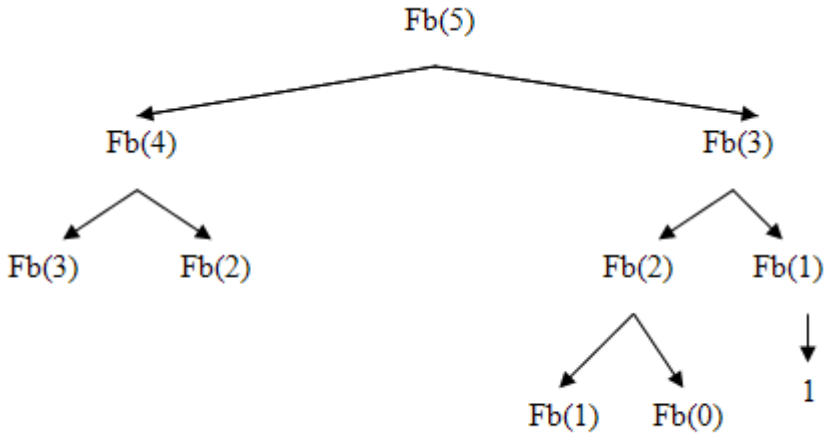


Рис 8.2. Фрагмент дерева рекурсії при обчисленні чисел Фібоначчі – F(5)

Механізм виклику функції або процедури в мові високого рівня суттєво залежить від архітектури комп'ютера і операційної системи. У рамках IBM PC сумісних комп'ютерів цей механізм реалізований через програмний стек. Які передаються в процедуру або функцію фактичні параметри, так і які повертаються з них значення, поміщаються в програмний стек спеціальними командами процесора. Додатково зберігаються значення необхідних регістрів і адреса повернення в процедуру.

Схематично цей механізм ілюстрований на рис 8.3.

Для підрахунку трудомісткості виклику будемо вважати операції поміщення слова в стек і вибірку його з стека елементарними операціями у формальній системі. Тоді при виклику процедури або функції в стек поміщається адреса повернення, стан необхідних регістрів процесора, адреси повернених значень і передані параметри. Після цього виконується перехід за адресою на викликувану процедуру, яка витягує передані фактичні параметри, виконує обчислення, поміщає їх за вказаними в стеці адресами, і при завершенні роботи відновлює регістри, виштовхує з стека адресу повернення і здійснює перехід за цією адресою.

Позначимо через:

m – кількість фактичних параметрів, що передаються,
k – кількість значень, що повертаються процедурою,
r – кількість регістрів в стеку, що оберігаються,

маємо:

$$f_{виклик} = m+k+r+1+m+k+r+1 = 2*(m+k+r+1)$$

елементарних операцій на один виклик і повернення.

Аналіз трудомісткості рекурсивних алгоритмів в частині трудомісткості самого рекурсивного виклику можна виконувати різними способами в залежності від того, як формується підсумкова сума елементарних операцій – розглядом окремо ланцюжка рекурсивних викликів і повернень, або сукупно по вершинах дерева рекурсивних викликів.

8.5 Аналіз трудомісткості алгоритму обчислення факторіала

Для розглянутого вище рекурсивного алгоритму обчислення факторіала кількість вершин рекурсивного дерева одно, очевидно, n, при цьому передається і повертається по одному значенню (m = 1, k = 1), в припущенні про збереження чотирьох регістрів – r = 4, а на останньому рекурсивному виклику значення функції обчислюється безпосередньо – в результаті:

$$fA(n) = n * 2 * (1 + 1 + 4 + 1) + (n-1) * (1 + 3) + 1 * 2 = 18 * n – 2$$

Відзначимо, що n – параметр алгоритму, а не кількість слів на вході.

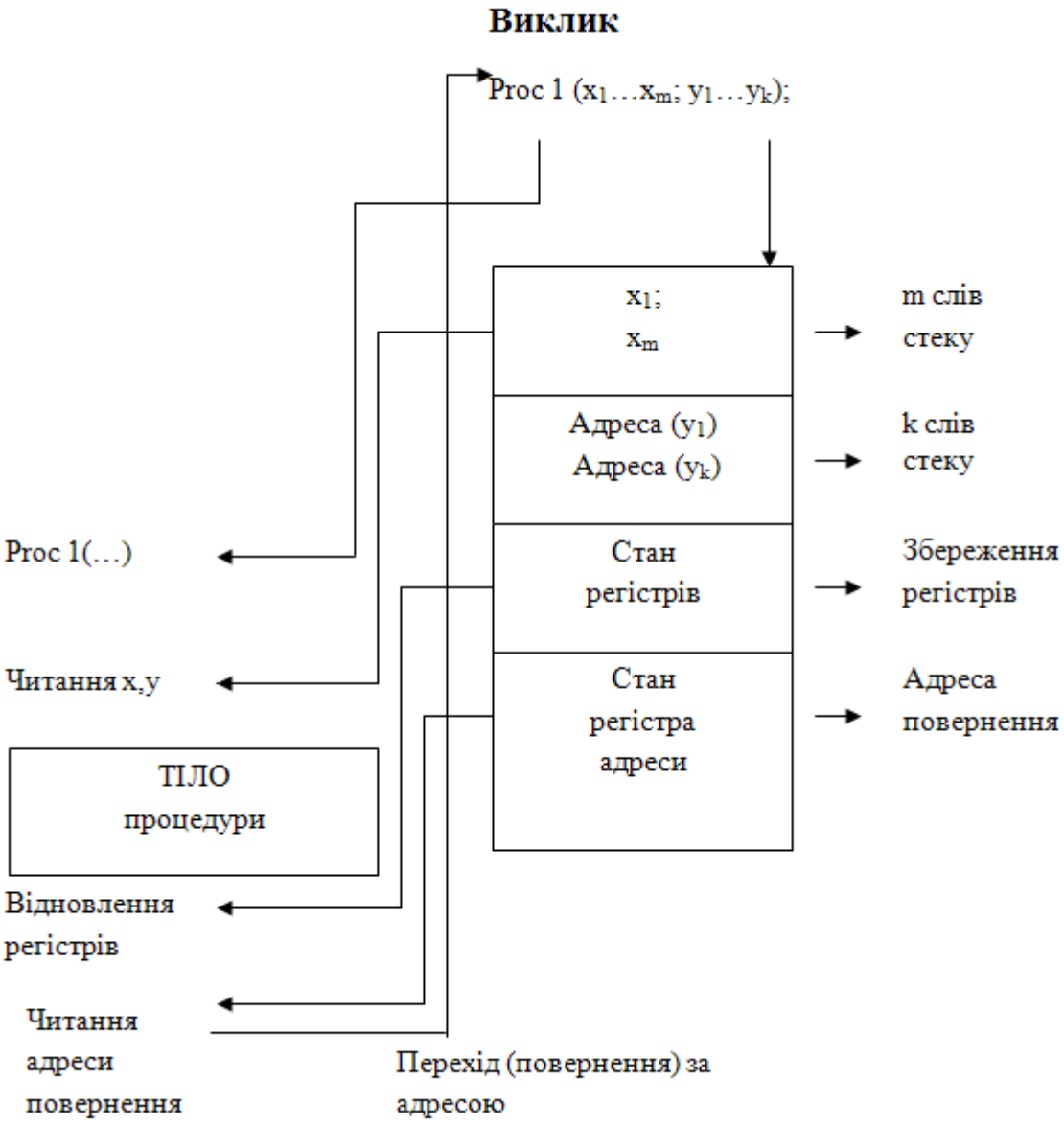


Рис. 8.3. Механізм виклику процедури з використанням програмного стека

8.6 Питання для самоконтролю

1) Поняття індукції і рекурсії.
2) Приклади рекурсивного завдання функцій.
3) Рекурсивна реалізація алгоритмів.
4) Трудомісткість механізму виклику функції в мові високого рівня.
5) Рекурсивне дерево, рекурсивні виклики і повернення.
6) Аналіз трудомісткості рекурсивного алгоритму обчислення факторіала.

9. РЕКУРСИВНІ АЛГОРИТМИ І МЕТОДИ ЇХ АНАЛІЗУ

9.1 Логарифмічні тотожності

При аналізі рекурсивних алгоритмів досить часто використовуються логарифмічні тотожності, далі передбачається, що, a> 0, b> 0, c> 0, основа логарифма не дорівнює одиниці:

$b^{\log_b a} = a;\ e^{\ln x} = x;\ \log_b a^c = c * \log_b a;\log_b a = 1/\log_a b$

$\log_b a = \log_c a / \log_c b \Rightarrow$ записі Q (ln (x))

Основа логарифма не суттєва, якщо вона більше одиниці, оскільки константа ховається в позначенні Q.

$$a^{\log_b c} = c^{\log_b a}$$

Можна показати, що для будь-якого $\xi > 0$ $ln(n) = o(n^{\xi})$, при $n \to \infty$.

9.2 Методи рішення рекурсивних співвідношень

В математиці розроблено ряд методів, за допомогою яких можна отримати явний вигляд функції, які задані у рекурсивному виді [1, 5] – метод індукції, формальні степеневі ряди, метод ітерацій і т.д.

Розглянемо деякі з них:

а) Метод індукції

Метод полягає в тому, що б спочатку вгадати рішення, а потім довести його правильність за допомогою індукції.

Приклад:

$$\begin{cases} f(0)=1 \\ f(n+1)=2*f(n) \end{cases}$$

Припущення: $f(n)=2^n$

Базис: якщо $f(n)=2^n$, *тоді* $f(0)=1$, що виконано за визначенням.
Індукція: Нехай $f(n)=2^n$, тоді для $n+1 \Rightarrow f(n+1)= 2 * 2^n = 2^{n+1}$

Зауважимо, що базис суттєво впливає на рішення. Так, наприклад:

- якщо $f(0)=0$, то $f(n)=0$;
- якщо$f(0)=1/7$, то $f(n)=(1/7)*2^n$;
- *якщо* $f(0)=1/64$, то $f(n)=(2)^{n-6}$

б) Метод ітерації (підстановки)

Суть методу полягає в послідовній підстановці – ітерації рекурсивного визначення, з наступним виявленням загальних закономірностей:

Нехай $f(n)=3*f(n/4)+n$, тоді:

$$f(n)=n+3*f(n/4)=n+3*[\ n/4+3*f(n/16)\ ]=n+3*\ [n/4+3\{\ n/16+3*f(n/64)\ \}\ ],$$

і розкриваючи дужки, отримуємо:

$$f(n)=n+3*n/4+9*n/16+27*n/64+\ldots+3^i*n/4^i$$

Зупинка рекурсії відбудеться при $(n/4^i) \le 1 \Rightarrow i \ge \log_4 n$, в цьому випадку останній доданок не більше, ніж $3^{\log_4 n}*\Theta(1) = n^{\log_4 3}*\Theta(1)$.

$$f(n) = n*\Sigma(3/4)^k + n^{\log_4 3}*\Theta(1),$$

тобто

$$\Sigma(3/4)^k = 4*n,$$

то остаточно:

$$f(n) = 4 * n + n^{\log_4 3}*\Theta(1) = \Theta(n).$$

9.3 Рекурсивні алгоритми

Основний метод побудови рекурсивних алгоритмів – це метод декомпозиції. Ідея методу полягає в розкладі задачі на частини меншої розмірності, отримані рішення для кожної частини і об'єднання рішень.

В загальному вигляді, якщо відбувається розділення задачі на **b** підзадач, яке призводить до необхідності вирішення **a** підзадач розмірністю n/b, то загальний вигляд функції трудомісткості має вигляд [5]:

$$f_A(n) = a * f_A(n/b) + d(n) + U(n) \quad (9.1),$$

де:

$d(n)$ – трудомісткість алгоритму розподілу задачі на підзадачі,

$U(n)$ – трудомісткість алгоритму об'єднання отриманих рішень.

Розглянемо, наприклад, відомий алгоритм сортування злиттям, що належить Дж. Фон Нейману [5]:

На кожному рекурсивному виклику переданий масив ділиться навпіл, що дає оцінку для d(n) = Q(1). Далі рекурсивно викликається сортування отриманих масивів половинній довжини (до тих пір, поки довжина масиву не стане рівною одиниці), і зливаються повернені відсортовані масиви за Q (n).

Тоді очікувана трудомісткість на сортування складе:

$$f_A(n) = 2 * f_A(n/2) + \Theta(1) + \Theta(n)$$

Виникає задача про отримання оцінки складності функції трудомісткості, заданої у вигляді (9.1), для довільних значень a і b.

9.4 Основна теорема про рекурентних співвідношеннях

Наступна теорема належить Дж. Бентлі, Д. Хакену та Дж. Саксу (1980 г.), достатньо повний доказ цієї теореми наведено в [5].

Теорема.

Нехай $a \geq 1$, $b > 1$ – константи, $g(n)$ – функція.

Нехай далі:

$f(n) = a*f(n/b) + g(n)$,

де $n/b = \lfloor (n/b) \rfloor$ або $\lceil (n/b) \rceil$

Тоді:

1) Якщо $g(n) = O(n^{\log_b a - \xi})$, $\xi > 0$, *то* $f(n) = \Theta(n^{\log_b a})$.

Приклад: $f(n) = 8f(n/2) + n^2$, тоді $f(n) = \Theta(n^3)$

2) Якщо $g(n) = \Theta(n^{\log_6 a})$, *то* $f(n) = \Theta(n^{\log_b a} * \log n)$.

Приклад: $f_A(n) = 2 * f_A(n/2) + \Theta(n)$, тоді $f(n) = \Theta(n * \log n)$

3) Якщо $g(n) = \Omega(n^{\log_b a + e})$, $e > 0$, то $f(n) = \Theta(g(n))$.

Приклад: $f(n) = 2 * f(n/2) + n^2$, маємо: $n^{\log_b a} = n^1$,

і відповідно: $f(n) = \Theta(n^2)$

Дана теорема є потужним засобом аналізу асимптотичної складності рекурсивних алгоритмів, на жаль, вона не дає можливості отримати повний вид функції трудомісткості.

9.5 Питання для самоконтролю

1) Аналіз рекурсивних співвідношень методом ітерацій.
2) Аналіз рекурсивних співвідношень методом підстановки
3) Загальний вигляд функції трудомісткості для методу декомпозиції.
4) Основна теорема про рекурентних співвідношеннях.
5) Приклади розв'язання рекурентних співвідношень на основі теореми Дж. Бентлі, Д. Хакена та Дж. Сакса.

10. ПРЯМИЙ АНАЛІЗ РЕКУРСИВНОГО ДЕРЕВА ВИКЛИКІВ

10.1 Алгоритм сортування злиттям

Розглянемо підхід до отримання функції трудомісткості рекурсивного алгоритму, який заснований на безпосередньому підрахунку вершин дерева рекурсивних викликів, на прикладі алгоритму сортування злиттям.

Рекурсивна процедура **Merge Sort** – MS отримує на вхід масив A і два індекси *p* і *q*, що вказують на ту частину масиву, яка буде сортуватися при даному виклику.

Допоміжні масиви *Bp* і *Bq* використовуються для злиття відсортованих частин масиву.

MS(A, p ,q, Bp, Bq)
If p≠q (перевірка останову рекурсії при *p=q*)
then
r←(p+q) div 2
*MS(A, p, r, Bp,Bq) (*рекурсивний виклик для першої частини)
MS(A, r+1, q, Bp,Bq) (рекурсивний виклик для другої частини)
Merge(A, p, r, q, Bp, Bq) (злиття відсортованих частин)
Return (A)

10.2 Злиття відсортованих частин (Merge)

Розглянемо процедуру злиття відсортованих частин масиву A, що використовує додаткові масиви Bp і Bq, в кінець яких з метою зупинки руху індексу поміщається максимальне значення. Алгоритм рекурсивного сортування працює так, що частини масиву A, які об'єднуються, знаходяться поруч один з одним, тому алгоритм злиття спочатку копіює відсортовані частини в проміжні масиви, а потім формує об'єднаний масив безпосередньо в масиві A.

Merge (A,p,r,q,Bp,Bq)	Кількість операцій в даному рядку
Max ← A[r]	*2*
If Max <A[q] Then	*2*
Max ← A[q]	*½*2*
kp ← r – p + 1	*3*
p1 ← p – 1	*2*

For i ← 1 to kp (копіювання першої частини) *1+ kp*3*
Bp [i] ← A[p1 + i] *kp*(4)*
Bp[kp+1] ← Max *3*

kq ← q – r *2*
For i ← 1 to kq (копіювання другої частини) *1+ kq*3*
Bq [i] ← A[r + i] *kq*(4)*
Bq [kq+ 1] ← Max *3*

(зауважимо, що $m=kp + kq = q – p + 1$ – довжина об'єднаного масиву)

pp ← p *1*
pq ← r+1 *2*
For i ← p to q (злиття частин) *1+m*3*
If Bp [pp] < Bq [pq] *m*3*
Then
A[i] ← Bp[pp] *½*m*3*
pp ← pp +1 *½*m*2*
Else
A [i] ← Bq [pq] *½*m*3*
pq ← pq +1 *½*m*2*
Return(A)
End

На підставі зазначеної кількості операцій можна отримати трудомісткість процедури злиття відсортованих масивів в середньому:

$F_{merge}(m)=$
$=2+2+1+3+2+1+kp*7+3+2+1+kq*7+3+1+2+1+m*(3+3+3+2)=$
$= 11*m + 7*(kp+kq) + 23 = 18*m+23.$ *(10.1)*

10.3 Підрахунок вершин в дереві рекурсивних викликів

Алгоритм, що має на вході масив з n елементів, ділить його навпіл при першому виклику, тому розглянемо випадок, коли $n = 2k$, $k = \log_2 n$.

В цьому випадку ми маємо повне дерево рекурсивних викликів глибиною k, що містить n листів, фрагмент дерева показаний на рис 10.1.

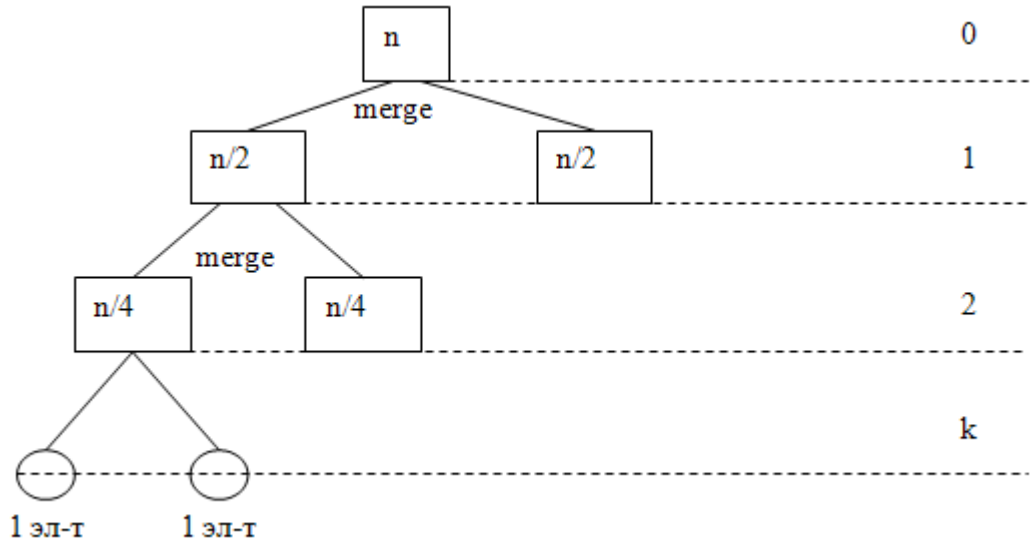


Рис 10.1. Фрагмент рекурсивного дерева при сортуванні злиттям

Позначимо кількість вершин дерева через V:

$V = n + n/2+n/4+n/8+...+1 = n*(1+1/2+1/4+1/8+...+1)=2n – 1=2^{k+1} – 1$

З них всі внутрішні вершини породжують рекурсію, кількість таких вершин – Vr = n-1. Решта n вершин – це вершини, в яких розглядається тільки один елемент масиву, що призводить до зупинки рекурсії.

10.4 Аналіз трудомісткості алгоритму сортування злиттям

Таким чином, для n листів дерева виконується виклик процедури MS с обчисленням r + 1 раз, перевіркою умови p = q і поверненням в названу процедуру для злиття. Ці дії в сумі з урахуванням трудомісткості виклику дають:

$F1(n) = n*2*(5+4+1) + n*2(If\ p=q\ i\ r+1) = 22*n.$

Для n-1 рекурсивних вершин виконується:

- перевірка довжини масиву, що передається,
- обчислюється середина масиву,
- рекурсивний виклик процедур MS,

- повернення.

Оскільки трудомісткість виклику враховується при вході до процедури, то отримуємо:

$$Fr(n) = (n-1)*2*(5+4+1) + (n-1)*(1+3+1) = 24*n - 24.$$

Процедура злиття відсортованих масивів буде викликана n-1 раз. При цьому трудомісткість складається з трудомісткості виклику і власної трудомісткості процедури Merge.

Трудомісткість виклику становитиме (для 6 параметрів і 4-х регістрів):

$$Fm_{виклик}(n) = (n-1)*2*(6+4+1) = 22*n - 22.$$

Оскільки трудомісткість процедури злиття для масиву довжиною m становить 18 * m + 23 (10.1), і процедура викликається n-1 раз з довжинами масиву рівними n, n/2, n/4, ..., причому 2 рази з довжиною n/2, 4 рази з довжиною n/4, то сукупно маємо:

$$Fm_{злиття}(n) = (n-1)*23 + 18*n + 2*18*(n/2) + 4*18*(n/4) + \ldots + =$$
$$= \{\text{враховуючи, що таким чином обробляються } k\text{-}1 \text{ рівнів}\}$$
$$= 18*n*(\log_2 n - 1) + 23*(n-1) = 18*n*\log_2 n + 5*n - 23.$$

Враховуючи всі компоненти функції трудомісткості, одержуємо кінцеву оцінку:

$$Fa(n) = F1(n) + Fr(n) + Fm_{виклик}(n) + Fm_{злиття}(n) =$$
$$= 22*n + 24*n - 24 + 22*n - 22 + 18*n*\log_2 n + 5*n - 23 =$$
$$= 18*n*\log_2 n + 73*n - 69 \qquad (10.2)$$

Якщо кількість чисел на вході алгоритму не дорівнює ступеню двійки, то необхідно проводити більш глибокий аналіз, заснований на вивченні поведінки рекурсивного дерева, проте при будь-яких ситуаціях з даними оцінка головного порядку $\Theta(n*\log_2 n)$ не зміниться [5].

10.5 Питання для самоконтролю

1) Рекурсивний алгоритм сортування злиттям.
2) Процедура злиття двох відсортованих масивів.
3) Оцінка трудомісткості процедури злиття.
4) Підрахунок вершин в дереві рекурсивних викликів для алгоритму сортування злиттям.
5) Аналіз алгоритму рекурсивної сортування методом прямого підрахунку вершин рекурсивного дерева.

11. ТЕОРІЯ І АЛГОРИТМИ МОДУЛЯРНОЇ АРИФМЕТИКИ

11.1 Алгоритм зведення числа в цілу ступінь

Задача про швидке зведення числа на всю ступінь, тобто обчислення значення $y = x^n$ для цілого n лежить в основі алгоритмічного забезпечення багатьох криптосистем [9], відзначимо, що в цьому аспекті застосування обчислення виробляються за mod_k. Представляє інтерес детальний аналіз швидкого алгоритму зведення в ступінь методом послідовного зведення в квадрат [6]. В цілях цього аналізу представляється доцільним введення трьох наступних спеціальних функцій:

1. Функція β(n)

Функція визначена для цілого додатного n, і β(n) є кількість бітів у двійковому поданні числа n. Відзначимо, що функція β (n) може бути представлена у вигляді:

$\beta(n) = [\log_2(n)] + 1 = [\log_2(n+1)]$,
де $[x]$ – ціла частина x, $n > 0$.

2. Функція β_1(n)

Функція визначена для цілого додатного n, і β_1(n) є кількість «1» в двійковому поданні числа n. При цьому, функція β_1(n) не є монотонно зростаючою функцією, наприклад, для всіх $n = 2^k$ $\beta_1(n) = 1$. Графік функції для початкових значень n представлений на рис 11.1.

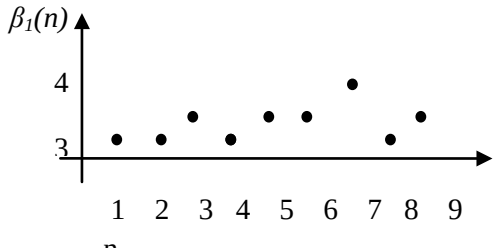


Рис 11.1. Значення функції для $n=1,2,\ldots 9$.

Враховуючи визначення β_1(n) справедлива нерівність:

$$1 \le \beta_1(n) \le \beta(n) = [\log_2(n)] + 1, \text{ т.е. } \beta_1(n) = O(\log_2(n))$$

Відзначимо, що функція $\beta_1(n)$ може бути рекурсивно задана наступним чином [4]:

$$\beta_1(n) = \begin{cases} \beta_1(0)=0;\ \beta_1(1) = 1; \\ \beta_1(2n) = \beta_1(n); \\ \beta_1(2n+1) = \beta_1(n) + 1. \end{cases}$$

3. Функція β_0 (n)

Функція визначена для цілого додатного *n*, і $\beta_0(n)$ є кількість «0» в двійковому поданні числа n. Функція $\beta_0(n)$ не є монотонно зростаючою функцією. Для всіх

$$n = 2^k - 1, \quad \beta_0(n) = 0$$

Для будь-якого n справедливо співвідношення

$$\beta(n) = \beta_0(n) + \beta_1(n).$$

Для подальшого аналізу представляє також інтерес визначення середнього значення функції $\beta_1(n)$ для $n = \{0,1,\ldots,N\}$, де $N=2^k-1$ (тобто двійкове подання числа *N* займає *k* розрядів), позначимо його через $\beta s\ (N)$.

Тоді $\beta_s(N) = \frac{1}{N+1}\sum_{m=0}^{N}\beta_s(m)$.

Оскільки кількість чисел, що мають L одиниць в K розрядах дорівнює кількості сполучень з L по K, то тоді:

$$\sum_{m=0}^{N}\beta_s(m) = \sum_{L=1}^{k} L * C_K^L = \sum_{L=1}^{k} L * \frac{K}{L} C_{K-1}^{L-1} = K * \sum_{L=0}^{k-1} C_K^L = K * 2^{K-1},$$

оскільки $N=2^k-1$, то:

$$\beta_s(N) = \frac{1}{N+1}\sum_{m=0}^{N}\beta_s(m) = \frac{K * 2^{k-1}}{2^K - 1 + 1} = \frac{K}{2} = \frac{\log_2(N+1)}{2} = \frac{\beta(N)}{2} \qquad (11.1).$$

Ідея швидкого алгоритму розв'язання задачі про зведення в ступінь полягає у використанні двійкового розкладання числа *n* і обчислення відповідних ступенів шляхом повторного зведення в квадрат [5].

Нехай, наприклад, $n=11$,

тоді $x^{11} = x^8 * x^2 * x^1$, $x^4 = x^2 * x^2$ і $x^8 = x^4 * x^4$.

Алгоритмічна реалізація ідеї вимагає послідовного виділення бітів, зведення *x* у квадрат і множення *y* на ті ступені x, для яких у двійковому розкладанні *n* присутня одиниця.

```
XstK (x,n;y);
z ← x;
y ← 1;
Repeat
        If (n mod 2) = 1
                then
                        y ← y*z;
        z ← z*z;
        n ← n div 2;
Until n = 0
Return (y)
End
```

Одержимо функцію трудомісткості даного алгоритму, використовуючи введені раніше позначення і прийняту методику рахунків елементарних операцій у формальній системі процедурно-орієнтованої мови високого рівня:

$$Fa(n) = 2 + \beta(n)*(2+2+2+1) + \beta_1(n)*(2) = 7*\beta(n) + 2*\beta1(n)+2 \qquad (11.2)$$

Кількість проходів циклу визначається кількістю бітів у двійковому поданні n – β(n), а кількість повторень операції y ← y*z – кількістю одиниць в цьому поданні – β_1(n), що й відображає формула 11.2.

Визначимо трудомісткість алгоритму для особливих значень n, такими особливими значеннями є випадки, коли $n=2^k$ або $n=2^k - 1$:

– в випадку коли $n=2^k$, то $\beta_1(n)=1$ і $Fa(n)= 7*\beta(n) + 4$;

– в випадку коли $n=2^k - 1$, то $\beta_1(n)= \beta(n)$ і $Fa(n)= 9*\beta(n) + 2$.

Якщо показник ступеня заздалегідь невідомий, то можна отримати середню оцінку, в припущенні, що подання числа n займає не більше k двійкових розрядів, тобто n <2k або log2n <k. Тоді за формулою (11.1)

$\beta_s(N) = \beta(N)/2$, *где* $N=2^k-1$,

звідки:

$$Fa(n) \leq 7*\beta(N) + 2*\beta_s(N)+2 = 8*\beta(N) + 2 = 8*([log_2(n)]+1)+2 = 8*k +2.$$

Таким чином, кількість операцій, виконуваних швидким алгоритмом зведення в ступінь, лінійно залежить від кількості бітів у двійковому поданні показника ступеня.

Введення спеціальних функцій $\beta_1(n)$ і $\beta(n)$ дозволило отримати точне значення функції трудомісткості цього алгоритму.

11.2 Відомості з теорії простих чисел

а) Порівняння

Кажуть, що два числа a і b можна порівняти по модулю c, якщо вони дають при діленні на c рівні залишки.

Операція отримання залишку від ділення a на c записується у вигляді: a modc = d, що еквівалентно поданням: a = k * c + d;

Порівнянність двох чисел по модулю означає, що:

a modc = b modc і записується як $(a \equiv b)\ modc$

Приклади: $(13 \equiv 6)\ mod7,\ (17 \equiv 22)\ mod5$

Якщо, a modc = 0, то, число a ділиться на c без залишку: $(a \equiv 0) modc$

б) Прості числа

Число p називається простим, якщо вона не має інших дільників, крім одиниці і самого себе. Очевидно, що в якості можливих дільників є сенс перевіряти тільки прості числа, менші або рівні квадратному кореню з числа, що перевіряється.

Безліч простих дільників, доказ належить Евкліду:

Нехай p1, ..., pk, – всі прості числа, але тоді число

a = (p1 * p2 * ... * pk + 1)

у залишку від ділення на будь-яке з них дає одиницю

a mod pi = 1 і отже є простим.

в) Функція π (n)

Функція π (n) в теорії простих чисел позначає кількість простих чисел, не переважаючих n.

Наприклад π (12) = 5, оскільки існує 5 простих чисел, не переважаючих 12, а саме: 2,3,5,7,11.

Асимптотична поведінка функції π (n) було отримано наприкінці XIX століття [5] і пов'язане з функцій інтегрального логарифма:

Для великих n – $\pi(n) \approx li(n) \approx n / \ln n$.

Отриманий результат означає, що прості числа не так вже «рідкісні», ймовірність того, що серед взятих випадково ln n чисел, що не перевершують n, одне з них просте, достатньо велика.

Відзначимо, що це використовується при пошуку великих простих чисел в ймовірнісному тесті Міллера-Рабіна [5].

11.3 Питання для самоконтролю

1) Функції підрахунку кількості бітів і кількості одиниць в двійковому поданні числа та їх властивості.
2) Алгоритм швидкого зведення в ступінь.
3) Аналіз трудомісткості алгоритму швидкого зведення в ступінь;
4) Поняття напівгрупи, моноїд і групи, приклади груп.
5) Порівняння та відомості з теорії простих чисел.

12. КРИПТОСИСТЕМА RSA І ТЕОРІЯ АЛГОРИТМІВ

12.1 Мультиплікативна група відрахувань за модулем n

Розглянемо деякі групи, утворені на множині відрахувань за модулем n: Нехай n – ціле додатне число, тоді множина залишків від ділення будь-якого цілого додатного числа на n називається множиною відрахувань за модулем n і позначається як *Zn:*

$Z_n = \{ 0, 1, 2,\ldots, n-1\}$

Якщо як групову операцію розглянути операцію додавання за модулем *n: (+ modn),* то множина *Zn* утворює з цією операцією і нулем, в якості «одиниці», групу $\{Z_n, + modn, 0\}$, яку називають **адитивною групою** відрахувань за модулем n.

Зворотним елементом для $\boldsymbol{a \in Zn}$ буде елемент $a^{-1} = (n - a) modn$

Якщо як групову операцію розглянути операцію множення за модулем n: *(*modn),* то множина ***Z'n*** утворює з цією операцією і одиницею групу $\{Z'_n, *modn, 1\}$, *яку* називають **мультиплікативною групою** відрахувань за модулем n, що позначається звичайно як ***Zn*.***

Зворотний елемент в групі Zn* існує, тільки якщо *НСД (z, n) = 1* Кількість чисел, взаємопростих з n, і, отже, кількість елементів в групі *Zn** може бути отримано за формулою Ейлера [5, 9]:

$|\boldsymbol{Z_n^*}| = \varphi(n) = n*П\,(1-1/p_i),$

де p_i – прості дільники числа *n.*

Наприклад: $\varphi(15) = 15*(1-1/3)(1-1/5) = 15*2/3*4/5=8$

Якщо число *n* – просте число, тобто $n = p$, то $\varphi(p) = p(1-1/p) = (p-1).$

Знаходження зворотного елементу для деякого елемента мультиплікативною групи по множенню зазвичай виконується за допомогою розширеного алгоритму Евкліда [5].

Зауважимо, що теорема про єдинство зворотного елемента в групі *Zn** , а *1 і (n – 1)* є зворотними самі собі, тобто:

$1*1 = 1modn$ і $(n-1)*(n-1) = (n^2 - 2n +1)modn = 1$

Ці числа називаються тривіальними коріннями з одиниці за модулем n.

Розглянемо, наприклад, $Z_7^* = \{ 1, 2, 3, 4, 5, 6 \}$. Зворотним елементом до *2* буде *4*, тобто $2^{-1} = 4 \ mod7$**,** оскільки $2*4 = 8 \ mod7 = 1$.

Зворотним елементом до 3 буде 5, тобто $3^{-1} = 5 \ mod7$,

оскільки $3*5 = 15 \ mod7 = 1$.

12.2 Ступені елементів в Zn* і пошук великих простих чисел

Оскільки групова операція множення за модулем *(*modn)* застосована до будь-якої пари чисел з *Zn*,* то можна визначити ступені елементів:

$(a*a) \ modn = a^2$; $(a^2 * a) \ modn = a^3$.

Для ступенів елементів в групі Zp*,* справедлива мала **теорема Ферма**:

Якщо p – просте число, то для кожного елемента справедливо порівняння:

$\forall a \in Z_p^* : a^{p-1} \equiv 1 \ modp$

Наприклад, для Z_7^* вірно: $5^6 \equiv 4^6 \equiv 3^6 \equiv 2^6 \equiv 1 mod7$

Узагальненням малої теореми Ферма для будь-якого (не обов'язково простого) n є **теорема Ейлера** (Ферма-Ейлера):

$\forall a \in Z_n^* : a^{\varphi(n)} \equiv 1 \ modn$

На теоремі Ферма-Ейлера заснований спеціальний алгоритм пошуку великих простих чисел – ймовірнісний тест Міллера-Рабіна [5]. Нагадаємо, що кількість простих чисел, що не перевершують x – функція $\pi(x)$ має наступну асимптотичну оцінку: $\pi(x) \approx x/\ln x$. Це призводить до оцінки $1/\ln n$ для ймовірності того, що навмання (випадково) взяте число n є простим.

Ідея ймовірнісного тесту Міллера-Рабіна полягає в наступному:

Генерується випадкове число n і вибирається деякий a $\in$ {2, ... n-2}, тоді за теоремою Ферма-Ейлера:

Якщо $(a^{n-1}) \bmod n \neq 1$, то, очевидно, що число n – складене;
Якщо $(a^{n-1}) \bmod n = 1$, то, можливо необхідно перевірити інше a.

Ймовірність помилки тесту експоненціально падає з ростом успішних перевірок з різними значеннями a $\in$ {2, ... n-2}, реально виконується близько декількох десятків перевірок [5].

12.3 Криптосистема RSA

Запропонована в 1977 році РІВЕРСТ, Шамілем і Адлеманом (R. Rivest, A. Shamir, L. Adleman) криптосистема з відкритим ключем – RSA може бути коротко описана наступним чином [9]:

а) Знаходимо два великих простих числа p і q (тест Міллера – Рабіна)

б) Обчислюємо $n = p * q$; $n \approx 2512 – 2768$.

в) За побудовою $\varphi(n) = p*q*(1-1/q)*(1-1/p) = (p-1)*(q-1)$.

г) Обираємо число e, таке, що: $НСД\ (e, \varphi(n)) = 1$.

д) Знаходимо число f зворотне до e за модулем j(n) за допомогою розширеного алгоритму Евкліда

$f = e^{-1} \bmod \varphi(n)$, тобто $(e*f \equiv 1) \bmod \varphi(n)$, або $e*f = k* \varphi(n)+1$.

Шифрування

Розбиваємо повідомлення на блоки M_i

$$\beta(M_i) = \beta(n)-1$$

і обчислюємо

$$C_i = M_i^e \bmod n$$

Дешифрування

За прийнятим повідомленням Сі обчислюємо (всі операції по modn):

$$C_i^f \bmod n = (M^e)^f = M^{e*f} = M^{k*\varphi(n)+1} = M*(M^{\varphi(n)})^k = M_i$$

12.4 Крипостійкість RSA і складність алгоритмів факторизації

Оскільки значення e і n відомі, то задача розтину криптосистеми зводиться до обчислення f, такого, що $(e*f \equiv 1) \bmod \varphi(n)$

На сьогодні теоретично не доведено, що для визначення f необхідно розкласти n на множники, проте, якщо такий алгоритм буде знайдений, то на його основі можна побудувати швидкий алгоритм розкладання числа на прості множники [11].

Тому криптостійкість RSA визначається сьогодні алгоритмічною складністю завдання факторизації – завдання розкладання числа на прості множники. Відзначимо, що за останні 20 років алгоритмічний прогрес в цій області значно перевищує зростання продуктивності процесорів. На сьогодні в області важко розв'язуваних завдань прийнята наступна одиниця виміру тимчасової складності завдання – **1 MY**.

1 MY – це задача, для вирішення якої необхідна робота комп'ютера, що виконує 1 млн. операцій в секунду протягом одного року. У 1977 році Р. Ріверст прогнозував на основі кращого в той час алгоритму розв'язання задачі факторизації методом еліптичних кривих тимчасову складність факторизації складного числа довжиною в 129 десяткових цифр (*129D*) – $n \approx 10^{129}$ в $4* 10^{16}$ років [9]. Однак цей модуль був розкладений на множники за 5000 MY (з використанням мережі Інтернет) в 1994 році алгоритмом, що використовує метод квадратичного решета.

Модуль RSA 140D був факторизований в 1999 році алгоритмом, що використовує метод узагальненого числового сита за 2000 MY. Більш докладні відомості про тимчасову складність завдання факторизації і декілька проектів по факторизації модулів RSA наведені в [9].

Найкращий сьогодні алгоритм факторизації, що використовує метод узагальненого числового сита має наступну тимчасову оцінку:

$T(n) = O (e (\ln n)^{1/3} * (\ln \ln n)^{2/3})$;

Відзначимо, що саме успіхи асимптотичного і експериментального аналізу алгоритмів дозволяють не тільки прогнозувати часову складність розкриття криптосистеми RSA, забезпечуючи тим самим її криптостійкість, але і розраховувати довжину модуля (кількість бітів у двійковому поданні

числа n) необхідну для ефективного шифрування з прогнозованим часом розкриття.

12.5 Питання для самоконтролю

1) Мультиплікативна група відрахувань по модулю N і її властивості.
2) Ступені елементів і теорема Ферма-Ейлера.
3) Ідея імовірнісного тесту Міллера-Рабіна для пошуку великих простих чисел.
4) Криптосистема RSA.
5) Застосування теорії алгоритмів до аналізу крипостійкості RSA.

ЛІТЕРАТУРА

1. Ахо, А. Структуры данных и алгоритмы / Ахо А., Хопкрофт Дж., Ульман Дж. – М.: Издательский дом «Вильямс», 2001. –384 с.
2. Вирт Н. Алгоритмы и структуры данных / Н. Вирт – 2-ое изд., испр. – СПб.: Невский диалект, 2001. – 352 с.
3. Карпов, Ю.Г. Теория автоматов / Ю.Г. Карпов – СПб.: Питер, 2002. – 224 с.
4. Кнут, Д. Искусство программирования. Тома 1, 2, 3. 3-е изд. / Д. Кнут. Уч. пос. – М.: Изд. дом "Вильямс", 2001. – 385 с.
5. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест – М.: МЦНМО, 2001. – 960 с.
6. Макконнел Дж. Анализ алгоритмов. Вводный курс / Макконнел Дж. – М.: Техносфера, 2002. –304 с.
7. Новиков, Ф.А. Дискретная математика для программистов / Ф.А. Новиков – СПб.: Питер, 2001. – 304 с.
8. Романовский, И.В. Дискретный анализ. Учебное пособие для студентов, специализирующихся по прикладной математике / И.В. Романовский – Издание 2-ое, исправленное. – СПб.; Невский диалект, 2000. – 240 с.
9. Чмора, А.Л. Современная прикладная криптография / А.Л. Чмора – М.: Гелиос АРВ, 2001. – 256 с.