Міністерство освіти і науки України

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Конспект лекцій
для студентів спеціальності
121 "Інженерія програмного забезпечення"
денної форми навчання

Луцьк 2016

УДК 004.032.6(045)
А 4.02

Рекомендовано до видання Навчально-методичною радою факультету комп'ютерних наук та інформаційних технологій Луцького НТУ, протокол № від листопада року.

_____ Голова навчально-методичної ради факультету комп'ютерних наук та інформаційних технологій

Розглянуто і схвалено на засіданні кафедри комп'ютерних технологій Луцького НТУ, протокол № від листопада року.

Укладач: _____ В.О.Ліщина, доцент кафедри КТ Луцького НТУ

Рецензент: В. Д. Рудь, доктор технічних наук, професор Луцького НТУ

Відповідальний за випуск: О.О. Герасимчук, завідувач кафедри КТ,
кандидат технічних наук, доцент

Алгоритми та структури даних [Текст] :конспект лекцій для студентів
А4.02 спеціальності 121 "Інженерія програмного забезпечення" / уклад. В.О.Ліщина. –
Луцьк :Луцький НТУ, 2016. – 74с.

Видання містить конспект лекцій з дисципліни " Алгоритми та структури даних ".

Призначені для студентів спеціальності 121 "Інженерія програмного забезпечення".

ЗМІСТ

АНОТАЦІЯ КУРСУ

Теорія алгоритмів як окремий розділ математики, що вивчає загальні властивості алгоритмів, виникла в 30-их роках XX століття. Необхідність точного математичного коректування інтуїтивного поняття алгоритму стала неминучою після усвідомлення неможливості існування алгоритмів розв'язку багатьох масових проблем, у першу чергу пов'язаних з арифметикою та математичною логікою. Для доведення не існування алгоритму треба мати його точне математичне визначення, тому після сформування поняття алгоритму як нової та окремої сутності першочерговою постала проблема знаходження адекватних формальних моделей алгоритму й дослідження їх властивостей. При цьому формальні моделі були запропоновані як для первісного поняття алгоритму, так і для похідного поняття алгоритмічно обчислюваної функції.

В наш час апарат теорії алгоритмів використовується всюди, де зустрічаються алгоритмічні проблеми. Теорія алгоритмів є теоретичним фундаментом програмування.

Метою викладання курсу є отримання студентами ґрунтовної математичної підготовки та знань теоретичних, методичних і алгоритмічних основ інформаційних технологій для їх використання під час розв'язання прикладних і наукових завдань в області інформаційних систем і технологій; забезпечення чіткого уявлення про методи структурного програмування, модульного підходу до побудов алгоритмів, математичних алгоритмів та створенні на їх основі програмних продуктів прикладного значення.

Завданням вивчення курсу є засвоєння теоретичних знань і формування практичних навичок з основ теорії алгоритмів і математичної логіки; ознайомлення з основами структурного програмування; ознайомлення з класичними методами побудови алгоритмів; вивчення математичних основ аналізу алгоритмів та алгоритмічних стратегій; ознайомлення з основами теорії обчислюваності; вивчення сутнісних характеристик алгоритмів сортування, злиття та пошуку; ознайомлення з рекурсивними алгоритмами та фундаментальними алгоритмами на графах і деревах.

У результаті вивчення навчальної дисципліни студенти повинні:

знати:

- теоретичні, методичні і алгоритмічні основи сучасних інформаційних технологій;
- загальні принципи побудови ефективних алгоритмів;
- сучасні методи дослідження та аналізу алгоритмів;
- основні принципи структурованого програмування;
- методи розв'язання класичних задач та недоліки і переваги кожного з них;
- принципи побудови рекурсивних алгоритмів;
- способи та механізми реалізації ефективних алгоритмів у конкретних застосуваннях.

вміти:

- реалізовувати основні алгоритми засобами алгоритмічної мови;
- будувати рекурсивні алгоритми;
- розробляти нові математичні методи, ефективні алгоритми і методи реалізації функцій інформаційних систем і технологій у прикладних областях;
- аналізувати, теоретично та експериментально досліджувати методи, алгоритми, програми апаратно-програмних комплексів і систем;
- створювати та досліджувати математичні та програмні моделі обчислювальних та інформаційних процесів, пов'язаних з функціонуванням об'єктів професійної діяльності;
- аналізувати та вибирати обчислювальні методи розв'язання задач проектування інформаційних систем за критеріями мінімізації обчислювальних витрат, стійкості, складності тощо;
- проектувати елементи математичного забезпечення обчислювальних систем;

ЛЕКЦІЯ 1. ПОВНА ПОБУДОВА АЛГОРИТМУ

Поняття алгоритму. Основні етапи повної побудови алгоритму. Робоча функція алгоритму. Задача про комівояжера. Метод повного перебору побудови алгоритму. Неформальні описи алгоритмів. Формальні описи алгоритму.

Деякі поставлені задачі допускають їх розв'язання за допомогою схематичного, механічного способу дій. Такі схематичні способи розв'язання називають **алгоритмами.** Неформальні описи алгоритмів зустрічаються в багатьох областях застосування. Так, більшості людей добре знайомі з повсякденного життя різні розпорядження, інструкції з використання, вказівки по порядку дії і т.д. При цьому мова йде, як правило, тільки про неточно описані алгоритми.

Якщо розпорядження повинне бути виконане машиною, то необхідний точний, "формальний" опис для відповідного алгоритму. В такому випадку застосовуються, як правило, формальні мови або графічні форми зображення для представлення алгоритмів. У загальному випадку алгоритм описує спосіб дій для розв'язання деякого класу прикладних задач. Наприклад, може бути заданий алгоритм, що здійснює додавання двох десяткових чисел для будь-яких натуральних чисел. Такому алгоритму потрібні два натуральних числа в їхньому десятковому представленні в якості **вхідних даних**, і отримуємо одне число в десятковому представленні як **вихідне дане** (результат). При цьому алгоритм опису є механічним перетворенням представлень інформації. Це є типовим для алгоритмів з переробки інформації.

Дамо неформальні пояснення поняття "алгоритм", а надалі наведемо декілька простих формальних систем для представлення алгоритмів.

1.1. Поняття алгоритму

Алгоритми це способи розв'язання, описані за допомогою розпоряджень по обробці, що задовольняють визначеним умовам.

Алгоритм - це спосіб з точним (тобто вираженим у точно визначеній мові) кінцевим описом застосування ефективних (тобто практично здійсненних) елементарних кроків переробки.

Звичайно, це визначення неточне, тому що воно залежить від розуміння використаних понять, особливо поняття "ефективний". Однак спочатку ця неточність повинна бути зігнорована. Помітимо, що ми, як і взагалі в інформаційних системах, робимо розходження між алгоритмом і його описом.

Алгоритми розв'язують не тільки окремі задачі, але й **класи задач**. Окремі задачі, що підлягають розв'язанню та виділені в міру потреби з розглянутого класу, визначаються за допомогою **параметрів**. **Параметри відіграють роль вхідних даних для алгоритму.** Алгоритми, як правило, для цих вхідних даних видають результати. Ці результати у випадку задач інформаційної обробки можуть бути інформацією (або, точніше, представленням інформації) або ж послідовністю вказівок ("керуючих сигналів"), по яких здійснюються визначені перетворення.

Існує багато різних можливостей для представлення алгоритму. У цьому розділі ми почнемо з неформального опису алгоритмів, а на закінчення буде дана одна проста формальна система, за допомогою якої можуть бути описані алгоритми.

Незалежно від форми опису, для алгоритмів важливо розрізняти наступні питання:

- постановку задачі, що розв'язується за допомогою алгоритму;
- специфічний спосіб, яким розв'язується задача, при цьому для алгоритму розрізняють:

(а) елементарні кроки обробки, що є в розпорядженні;

(б) опис вибору окремих кроків, що виконуються.

Для поставленої задачі завжди існує багато різних способів її розв'язання, тобто різних алгоритмів. Алгоритми можуть дуже відрізнятися, якщо в розпорядженні є різні набори елементарних кроків обробки.

1.2. Неформальні описи алгоритмів

Нижче як приклади дається ряд неформальних описів алгоритмів (інструкцій із застосування). У них вже витримуються типові концепції, що мають місце й у формальних описах алгоритмів.

Приклад 1.1. Арифметичні операції над десятковими числами:

Методи виконання додавання, віднімання, множення і ділення, що розглядаються в шкільних підручниках з математики, є прикладами алгоритмів над натуральними числами в десятковій системі числення.

Приклад 1.2. Алгоритм Евкліда для обчислення найбільшого спільного дільника (НСД): нехай задано два цілих числа a, b, де a > 0 і b > 0; треба знайти найбільший спільний дільник НСД(a, b) чисел a і b.

Алгоритм для обчислення натурального числа НСД(a, b), за Евклідом, говорить наступне:

(1) якщо a = b, то справедливо НСД(a, b) = a;

(2) якщо a < b, то застосовуємо алгоритм НСД до (a, b - a);

(3) якщо b < a, то застосовуємо алгоритм НСД до (a - b, b).

Коректність правила (1) очевидна. Правило (2) базується на наступному законі, що справедливий для всіх додатних натуральних чисел a, b:

$$a < b \Rightarrow НСД(a.\ b) = НСД(a, b - a).$$

Правило (3) базується на законі

$$b < a \Rightarrow НСД(a, b) = НСД(a - b, b).$$

Для цього алгоритму справедливо:

- виконання арифметичної операції "-" і операцій порівняння "<" і "=" вважається ефективним, елементарним кроком обробки;

- якщо в постановці задачі не враховувати обмеження a > 0 і b > 0, то отримаємо алгоритм, що для нерівних від'ємних чисел безперервний (не "закінчується");

- алгоритм є **визначеним**, тобто для кожних вихідних даних послідовність окремих кроків точно визначена.

Тут прослідковується важлива особливість алгоритмів: алгоритми працюють, як правило, над визначеними вхідними значеннями (аргументами, параметрами) і обчислюють визначені результати (вихідні значення). Для детермінованих алгоритмів відношення між вхідними і вихідними значеннями утворить якесь (часткове) **відображення**. Алгоритм називається **коректним**, якщо це відображення відповідає постановці задачі, що повинна розв'язуватися за допомогою цього алгоритму.

Приклад 1.3. Сортування колоди карт.

Є колода карт. Нехай на кожній карті зафіксоване одне натуральне число (для спрощення будемо вважати, що всі числа попарно різні). Потрібно відсортувати, колоду карт так, щоб зафіксовані на картах числа утворювали монотонну (зростаючу або спадаючу) послідовність.

Приведемо чотири різні алгоритми, що базуються на наступних ідеях:

- сортування шляхом попереднього сортування і злиття;

- сортування шляхом вставок;

- сортування шляхом вибору;

- сортування шляхом перестановок.

Сортування шляхом попереднього сортування і злиття

Задана колода *x* сортується за допомогою наступного правила:

(1) якщо *x* порожня або містить одну карту, то *x* відсортована;

(2) якщо *x* містить більш однієї карти, то *x* поділити на дві не порожні колоди; відсортувати кожну з них і далі об'єднати ці колоди в одну відсортовану колоду.

Злиття двох колод карт з одержанням однієї відсортованої колоди знову відповідає постановці задачі, так що воно може бути виконане звичайним чином за допомогою алгоритму.

Сортування шляхом вставок

Задана колода сортується за спаданням за допомогою наступного алгоритму, що у відповідне місце відсортованої колоди *y* вставляє по черзі карти, ті, що обираються з не відсортованої колоди *x* (процес починається з порожньої колоди *y*).

Отже, колода x сортується в *y* за наступним правилом:

(1) якщо *x* порожня, то *y* - шукана відсортована колода;

(2) якщо *x* не порожня, то з *x* береться будь-як карта і вставляється в потрібне місце колоди *y* так, щоб у результаті колода *y* залишилася відсортованою. Цей алгоритм застосовується до зменшення колоди *x* і колоді *y*, що збільшується.

Сортування шляхом вибору

Задана колода *x* сортується за спаданням за наступним алгоритмом, що послідовно вибирає з *x* "найбільшу" карту і додає її в кінець колоди *y* (процес починається з порожньої колоди *y*).

Нехай дані дві колоди *x* та *y*. Нехай колода *y* відсортована за спаданням, і нехай усі карти з *y* "більші" будь-яких карт із *x*. Колода *x* відсортовується в *y* за наступним алгоритмом:

(1) якщо *x* порожня, то *y* - шукана відсортована колода;

(2) якщо *x* не порожня, то з *x* вибирається "найбільша" карта і додається в кінець *y*. Алгоритм застосовується до зменшуваної колоди *x* і колоді, що збільшується, *y*.

Сортування шляхом обміну

Задана колода *x* сортується за наступним алгоритмом:

(1) якщо *x* містить дві сусідні карти, що не відповідають необхідному упорядкуванню, то ці карти міняються місцями, після чого до отриманої колоди застосовується цей же алгоритм;

(2) якщо в *x* не зустрічається ні однієї невпорядкованої пари послідовних карт, то колода *x* відсортована і тим самим є шуканою колодою.

Приклад 1.4. Користування автоматом для видачі квитків.

Автомат має: лічильник, у якому за допомогою кнопок можна встановити вартість потрібного квитка, користуючись наявною на автоматі таблицею, вміст цього лічильника показується в його вікні; прийомну щілину для опускання монет, причому при опусканні кожної монети з лічильника віднімається відповідне їй число; вікно для видачі квитка і кнопку для повернення монет.

Дії по придбанню квитка можна визначити наступним алгоритмом:

(1). установити на лічильнику ціну необхідного квитка;

(2). послідовно опускати монети, поки лічильник не прийме значення нуль або поки не будуть витрачені наявні монети;

(3). якщо не вистачило монет, натиснути кнопку повернення, забрати монети і закінчити дії;

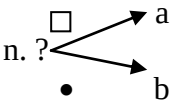
(4). якщо лічильник встановився на нуль і квиток виданий, взяти його і закінчити дії;

(5). якщо лічильник прийняв значення нуль, але квиток не виданий, натиснути кнопку повернення, забрати монети і закінчити дії.

Приклад 1.5. Алгоритми керування учбовим виконавцем “Машина Поста”.

Машина Поста являє собою нескінченну інформаційну стрічку, розділену на позиції (клітини). В кожній клітині може стояти мітка або ні. Вподовж стрічки рухається каретка. За один крок виконується зміщення на одну клітину вліво або вправо. Клітина, в якій встановлена каретка, називають поточною. Задана така система команд:

n. → m	Зрушення каретки на одну позицію вправо та перехід на команду за номером m.
n. ← m	Зрушення каретки на одну позицію вліво та перехід на команду за номером m.
n. • m	В поточній порожній клітинці поставити мітку та перехід на команду за номером m
n. ⊘ m	Витерти мітку в поточній клітинці та перехід

	на команду за номером m
	Перевірка стану клітинки: якщо клітинка порожня, то перехід на команду за номером a, інакше перехід на команду за номером b.
n. !	Зупинка машини.

Початковий стан інформаційної стрічки:

..			•						..

Задано алгоритм:

1. →2
2. •3
3. !

За цим алгоритмом отримуємо результат:

…			•	•					…

1.3. Класифікація алгоритмів

Алгоритм для якої-небудь задачі називається:

- **Терміністичним** (таким, що завершуються), якщо він завжди, для всіх допустимих послідовностей кроків закінчується після кінцевого числа кроків;
- **детерміністичним**, якщо немає ніякої свободи у виборі чергового кроку обробки;
- **Детермінованим**, якщо результат алгоритму визначений однозначно;
- **послідовним**, якщо кроки обробки завжди виконуються один за одним;
- **рівнобіжним**, якщо деякі кроки обробки виконуються одночасно.

Часто для алгоритмів потрібно, щоб вони для будь-яких вхідних даних були детерміністичними або щоб вони завершувалися при будь-яких вхідних даних. У наведеному вище визначенні поняття алгоритму ці обмеження були свідомо упущені, щоб одержати загальне розуміння алгоритму.

У наведених вище прикладах описів алгоритмів увесь час зустрічалися деякі схожі один на одного формулювання. Так, деякі кроки часто виконуються лише при визначеній умові або виконання деяких кроків здійснюється неодноразово. Часто також поставлена задача розв'язується за допомогою розв'язання тієї ж самої задачі, але з трохи зміненими (більш простими) параметрами. У цих випадках говорять про розгалуження, повторення і рекурсію.

Класичні елементи, що зустрічаються в описах алгоритмів, це:

- виконання елементарних кроків;
- розгалуження за умовами;
- повторення і рекурсії.

Схожі концепції мають також місце й у командах машин по обробці інформації. Множина елементарних кроків в алгоритмах відповідає основним машинним операціям. Щоб який-небудь алгоритм можна було описати у формі, здійсненної на одній з машин, для представлення алгоритмів застосовують формальні мови. Опис алгоритму на формально описаній мові називають програмою, а формальна мова - мовою програмування (МП).

1.4. Основні етапи повної побудови алгоритмів

1.4.1.Постановка задачі

Необхідно точно сформулювати задачу. Для цього звичайно необхідно відповісти на наступні запитання:

- Чи зрозуміла термінологія, яка використовується в попередньому формулюванні?

- Що задано?
- Що треба знайти?
- Як визначити розв’язок?
- Яких даних недостатньо і чи всі вони потрібні?
- Які зроблені припущення?

Для прикладу розглянемо класичну **задачу про “Комівояжера**”.

Джек – агент з продажу товарів. На його території N міст. Компанія вертає йому тільки 50% вартості автомобільних поїздок у справах. Джеку відомо вартості поїздок між кожними двома містами на його території. Треба знизити вартість дорожніх видатків.

Постановка задачі: Необхідно отримати список міст, в якому кожне місто згадується тільки один раз, за виключенням базового міста, що стоїть у списку першим та останнім. Сума вартості проїзду – це загальна вартість маршруту. Треба визначити маршрут з найменшою вартістю.

1.4.2. Побудова моделі

Для створення математичної моделі треба визначити:

- Які математичні структури більше всього підходять для задачі?
- Чи існує вже розв’язок аналогічних задач?

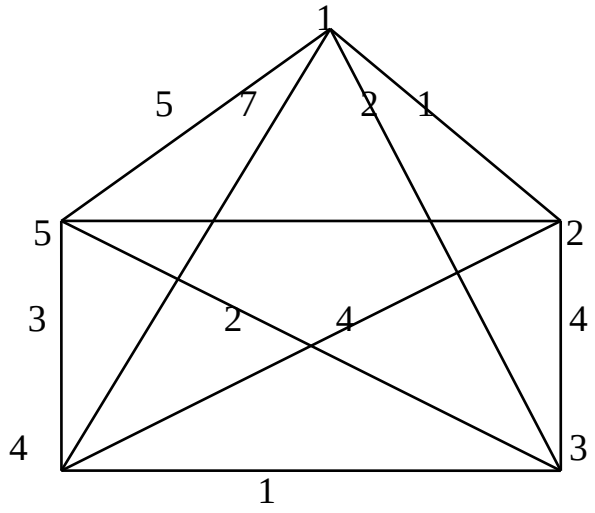
На вибір відповідної структури впливають декілька факторів:

- обмеженість знань відносно невеликою кількістю структур
- зручність подання
- простота обчислень
- корисність різних операцій, які зв’язані з структурою, що розглядається.

Далі треба сформулювати задачу в термінах відповідних математичних об’єктів. В нашому прикладі можна розглянути дві математичні структури: матрицю вартості C та граф:

	1	2	3	4	5
1	-	1	2	7	5
2	1	-	4	4	3
3	2	4	-	1	2
4	7	4	1	-	3
5	5	3	2	3	-

C=



В термінах теорії мереж список міст визначає замкнутий цикл, який починається в базовому місті та повертається сюди після проходження кожного міста по одному разу. Один з турів 1-5-3-4-2-1 має вартість 5+2+1+4+1=13. Чи є він шуканим туром з мінімальною вартістю.

1.4.3. Розробка алгоритму

Два різних алгоритми можуть бути правильними, але дуже відрізнятимуться за ефективністю. Розглянемо наступний алгоритм:

Кожне місто позначимо номером. Кожен тур взаємно-однозначно відповідає перестановці цілих чисел 1, 2, ..., n-1. Можна розв’язати задачу, утворюючи всі перестановки перших n-1 цілих додатних чисел. Для кожної перестановки обчислюємо вартість відповідного туру. Обробляючи таким чином всі перестановки, запам’ятовуємо такий тур, який в поточний момент має найменшу вартість. Опишемо цей алгоритм:

Алгоритм «Повний перебір».
Крок 0. Ініціалізація TOUR←0, MIN←∞
Крок 1. Утворення всіх перестановок (під алгоритм) for i=1 to (n-1)! do до кроку 4
Крок 2. Утворення нової перестановки (під алгоритм).
Крок 3. Побудова нового туру та обчислення вартості S(T(P))
Крок 4. if S(T(P))<MIN then TOUR=T(P), MIN=S(T(P))
кінець.

1.4.4. Правильність алгоритму

Доведення правильності алгоритму є найбільш важким етапом побудови алгоритму. Допустимо, що алгоритм описаний у вигляді послідовності кроків. Необхідно подати деяке доведення правомірності для кожного кроку. Можливо, треба буде довести лему про умови, які діють до та після пройденого кроку. Далі треба довести скінченність алгоритму, при цьому будуть перевірені всі вхідні та вихідні дані. Алгоритм №1 правильний на основі наступного: У зв'язку з тим, що перевіряється кожен тур, то буде перевірено також тур з мінімальною вартістю. А тому його буде запам'ятовано. Алгоритм буде завершено, тому що кількість міст скінчена.

1.4.5. Реалізація алгоритмів

Для реалізації алгоритму його описують за допомогою визначеного програмного засобу. Вибір мови програмування залежить від багатьох чинників. Це і так звана "мода" на мову програмування. Ще декілька років тому алгоритми швидше описувалися за допомогою мов програмування С або Pascal. Тепер алгоритми частіше реалізують за допомогою об′єктно-орієнтованих засобів Visual C, Delphi, Visual Basic. Але треба пам′ятати, що сучасні програмні засоби грунтуються на мовах програмування. Тому при виборі мови програмування основну увагу треба звернути на те, яка потужність мови. Це і можливість використання рекурсії, і динамічна пам′ять і компільованість мови, а звідси і швидкість виконання програми. Рекомендується застосовувати метод програмування зверху-вниз. Під час опису програми виконується синтаксичний та семантичний аналіз програми.

1.4.6. Аналіз алгоритму та його складності

Є потреба мати кількісний критерій для порівняння двох алгоритмів. Нехай А – алгоритм для розв'язання деякого класу задач, n – розмірність окремої задачі з цього класу. Визначимо $f_A(n)$ як робочу функцію, що визначає верхню межу для максимальної кількості основних операцій (додавання, порівняння і т.і.), які потрібно виконати алгоритму А для розв'язку задачі. Кажуть, що алгоритм А поліноміальний, якщо $f_A(n)$ зростає не швидше, ніж поліном від n, в протилежному випадку алгоритм експоненціальний.

Функцію $f_A(n)$ визначають як O[g(n)] та кажуть, що вона порядку g(n) для великих n, якщо

$$\lim \frac{f(n)}{g(n)} = const \neq 0$$

1.4.7. Перевірка програми

Наступним етапом є тестування програми та її аналіз. При виконанні тестування велику увагу треба звернути на область допустимих значень вхідних даних алгоритму. Тестування треба провести на всій області, особливо на "вузьких" ділянках. Для деяких даних алгоритм працює добре, для інших – погано. Ці ділянки треба виявити. Не забувайте, що доведення алгоритму не завжди можливо провести. А при тестуванні програми знаходяться алгоритмічні помилки. Треба тільки поставити перед собою таку задачу.

В кінці побудови алгоритму відбувається **оформлення документації**, що є дуже важливим та трудомістким процесом.

Контрольні запитання

1. Що називають алгоритмом?
2. Які основні властивості алгоритму?
3. Перелічіть етапи розробки алгоритмів?
4. Які питання виникають при постановці завдання?
5. Що являє собою математична модель?
6. Що таке доведення правильності алгоритму?
7. Що називають методом математичної індукції?
8. Як оцінити ефективність алгоритму?
9. Сформулюйте задачу про комівояжера.
10. Сформулюйте алгоритм повного перебору для задачі про комівояжера.
11. Приведіть приклади використання методу повного перебору для побудови алгоритмів.

ЛЕКЦІЯ 2. ФОРМАЛЬНИЙ ОПИС АЛГОРИТМУ ЧЕРЕЗ ЗАМІНУ ТЕКСТІВ

Текстові заміни. Детерміністичні алгоритми текстових замін. Обчислювальні структури. Сигнатури. Основні терми. Схеми. Терми з вільними ідентифікаторами.

2.1. Формальний опис алгоритму через заміну текстів

Для точного опису алгоритму (який допускає машинну обробку, переробку і виконання) потрібна формальна мова (підмножина з V* із заданим набором знаків V) для запису алгоритмів і точне визначення поняття ефективності (виконуваності) елементарних кроків переробки. У найпростішому випадку алгоритми для входу і виходу використовують слова над деяким набором знаків. Оскільки у вигляді слів може бути представлена будь-яка інформація, можна вважати, що алгоритми завжди оперують зі словами.

Однією з найпростіших концепцій елементарних кроків переробки послідовностей знаків є заміна визначених підслів (зразків) в слові яке обробляться іншими словами (або, що те ж саме, підстановка деякого слова замість зразка). Ця концепція веде до алгоритмів у формі **систем текстових замін на послідовностях знаків.**

Нехай V - запас знаків. Пари (v, w) $\in$ V* x V* називається **заміною над V**. Заміна часто записується у виді :

$$v \rightarrow w$$

Кінцеву множину R замін будемо надалі **називати системою текстових замін (СТЗ) над V**. Елементи цієї системи будемо називати також **правилами текстових замін (ПТЗ).** СТЗ використовується для представлення алгоритмів. Окремі кроки цих алгоритмів складаються, таким чином, у застосуванні правил замін.

Як і у випадку законів для булевих термів, правила замін повинні застосовуватися до будь-яких підтермів ("у будь-яких контекстах"). Заміна

$$\mathbf{s \rightarrow t}$$

називається застосуванням правила v $\rightarrow$ w, якщо є слова a, v, w, z з V* такі, що справедливо

$$\mathbf{s = a\ o\ v\ o\ z,\quad t=a\ o\ w\ o\ z.}$$

Завдяки поняттю застосування правил замін з множини R однозначно вибирається відношення на словах з V*. Це відношення також називається **алгебраїчним замиканням R.**

Слово s $\in$ V* називається **термінальним (або терміналом)** у R, якщо не існує слова t $\in$ V* такого, що справедливо наступне: заміна

$$\mathbf{s \rightarrow t}$$

є застосуванням якого-небудь правила з R. Таким чином, до термінального слова *s* не можна більше застосувати ніякого правила заміни.

Приклад 2.1. Заміна

вітрильник $\rightarrow$ вітрила

являє собою застосування правила заміни текстів

ьник$\rightarrow$ а

Якщо система текстових замін R складається тільки з цього правила, то слово “вітрила” є термінальним.

Через повторне застосування ПТЗ, виходячи з початкового заданого слова t_o, виникають обчислення. Якщо t_o, t_1, ..., t_n належать V* і $t_i \rightarrow t_{i+i}$ є застосування правила r_i з R для всіх i, 0 <= i <= n, то послідовність $(ti)_{1 <= i <= n}$ називають **кінцевим обчисленням (послідовністю обчислень)** над R для t_o. Часто обчислення записується в такий спосіб:

$$t_o \rightarrow t_1 \rightarrow t_2 \ ... \rightarrow t_n$$

Слово $\mathbf{t_o}$ називається також **входом** для обчислення. Якщо $\mathbf{t_n}$ є термінал, то обчислення називається кінцевим з результатом t_n. Слово $\mathbf{t_n}$ називається також **виходом** для R при вході $\mathbf{t_o}$. Нескінченне обчислення (послідовність обчислень) $(t_i)_{i\in\ N}$ зі слів $t_i \in$ V*, для яких $t_i \rightarrow ti_{+1}$ є застосування правил замін з R для всіх i $\in$ N, називається **нескінченним** обчисленням.

Приклад 2.2. Розглянемо деякі обчислення за СТЗ.

(1) Для системи текстових замін Q над множиною символів {L, O}, що складається з наступних правил:

$$\mathbf{LL \rightarrow \in \varepsilon,\ O \rightarrow \varepsilon,}$$

через послідовність

$$\mathbf{LOLL \rightarrow LO \rightarrow L}$$

задається обчислення, що завершується, для вхідного слова <LOLL> з результатом <L>.

(2) Для СТЗ над {L, O}, що складається з правил

$$O \rightarrow OO,\ O \rightarrow L,$$

для входу <O> послідовність обчислень

$$O \rightarrow OO \rightarrow OL \rightarrow LL$$

є обчисленням, що завершується, з виходом <LL>, а послідовність

$$O \rightarrow OO \rightarrow OOO \rightarrow OOOO \rightarrow \ldots$$

є обчисленням, що не завершується.

Система текстових замін R у силу наступного розпорядження визначає алгоритм, що використовує слова над V в якості входу і виходу. Для вхідного слова t $\in$V* алгоритм працює в такий спосіб : якщо одне з правил множини R можна застосувати до слова t (тобто існує слово s $\in$ V*, для якого має місце: t -> s є застосування правила з R), то треба використати правило до t і потім використати знову цей же алгоритм до слова s; у протилежному випадку припинити виконання алгоритму.

Слово t служить входом для алгоритму; якщо (після кінцевого числа кроків) виникає термінальне слово, тобто слово, до якого неможна застосувати ніяке правило, то це слово є виходом (результатом обчислень). Якщо така ситуація ніколи не виникає, то алгоритм не завершується. Алгоритми, визначені в такий спосіб за допомогою СТЗ, завжди є послідовними. При цьому вибір застосовуваного правила є не детерміністичним.

Часто для АТЗ як входи використовують тільки слова зовсім визначеної форми (**нормальна форма**). Визначені знаки не входять у ці слова (а також і у вихідні слова) - ці знаки використовуються винятково як допоміжні знаки в словах, що виникають у процесі обчислень.

Приклад 2.3. Приклади використання алгоритмів текстових замін.

(1) Додавання двох натуральних чисел, представлених у вигляді кількості штрихів. Натуральне число представляється у вигляді кількості штрихів з обмежувальними дужками, тобто число n $\in$ N презентується словом <|||…|>, причому в середину дужок входить n штрихів. У цьому випадку алгоритм складається з одного єдиного правила заміни ($\boldsymbol{\varepsilon}$ означає порожнє слово):

$$> + < \rightarrow \boldsymbol{\varepsilon}$$

Для входу <| ...|>+ <|... | > алгоритм дає суму штрихів.

<|||>+<|>

↓

<||||>

(2) Множення двох натуральних чисел (у такому ж поданні)

Застосовуються допоміжні знаки d, e, m. Алгоритм складається з наступних правил замін:

|>*< → >*<d

d| → | md

dm → md

d > → >

< > * < → < e

e| → e

em → | e

e> → >

Для вхідного слова <|...|> * <|...|> з nl штрихами в першому операнді і п2 штрихами в другому операнді алгоритм дає вихідне слово <|...|> з nl * п2 штрихами.

Послідовності, утворені з допоміжних знаків, представляють цілком визначені ситуації в обчисленнях. Слова, які виникають, можуть трактуватися знову, як ускладнені подання чисел. Для входу <||> * <|||> виникає показаний на мал. 1.1 граф можливих обчислень. Всі обчислення закінчуються одержанням слова <||||||>. Цей алгоритм підстановки не детерміністичний, але детермінований, тобто дає, незважаючи на різні обчислення, завжди той самий результат.

(3) Додавання двійкових чисел шляхом текстових підстановок:

Нехай заданий алфавіт

V = {L, O, <, >, (,), [,], +, b, c}. Передбачається наступна форма вхідного слова: $<a_i...a_n> + <b_i...b_m>$, де $b_i, a_i \in \{L, 0\}$.

СТЗ утвориться за наступними правилами:

> + < → >< b

bL→Lb (видалити +

bO→Ob позначити кінець правого

b>→c> операнда символом c)

Oc→(O)c (виділення чергового розряду

Lc→(L)c правого доданка)

L(L)→(L)L

O(L)→(L)O (транспортування виділеного розряду

L(O) →(O)L у початок доданка)

O(O)→(O)O

|L|L→ L|L|

|O|L→L| O| (транспортування чергового розряду

|L|O→O|L| результату на своє місце)

|O|O→O|O|

|L|(L) →(L) |L|

|O| (L)→(L)|O|

|L|(O)→(O) |L| (транспортування результату / операнда)

|O| (O)→(O) |O|

L><(L)→>L<|O|

O><(L)→>< |L|

L><(O)→><|L| (додавання без переносу)

O><(O)→><|O|

L>L<(L)→>L<|L|

O>L<(L)→>L<|O|

L>L<(O)→>L<|O| (додавання з переносом)

O>L<(O)→><|L|

|L|c→c (формування чергового
|O|c→c розряду результату)

<>L<(L)→<>L<|O|
<>L<(O)→<> <|L| (закінчення обробки
<><(L)→<>< |L| лівого операнда)
<><(O)→<>< |O|
L>L<c→>L<c

O>L<c→><c (закінчення обробки
L><c→><c правого операнда)

O><c→><c0
<><c→< (загальне закінчення обробки)
<>L<c→<L

Як видно з останнього прикладу, навіть для відносно простих постановок задач завдання СТЗ часто виявляється досить важкою справою, до того ж і коректність такої системи відразу зовсім не очевидна, тобто зовсім не просто, наприклад, у випадку наведеного вище алгоритму, переконатися в тім, що алгоритм дійсно здійснює додавання двійкових чисел і тим самим розв'язує поставлену задачу.

Алгоритми у виді текстових замін у загальному випадку є не детерміністичними і не детермінованими. Для вхідного слова t існують, як правило, багато різних обчислень з різними результатами. При цьому для однієї і тієї ж задачі можуть існувати такі обчислення, які завершуються і які не завершуються.

Приклад 2.4. Не детерміністична СТЗ з обчисленнями, що не завершуються.

У системі, що містить правила

aa →b, a → aa

кожне слово t, що містить символ **a**, не є термінальним. Якщо друге правило застосовується тільки тоді, коли перше правило неможна застосувати, то обчислення завжди закінчується. При більш вільному застосуванні правил кожний символ **a** у вихідному слові можна перевести в будь-яке число символів **b** у результаті, або навіть обчислення ніколи не закінчаться, наприклад, якщо застосовувати тільки друге правило.

2.2. Детерміністичні алгоритми текстових замін

Часто віддають перевагу детерміністичним АТЗ, тобто алгоритмам, які для кожного вхідного слова однозначно задають обчислення і тим самим у випадку їхнього завершення породжують цілком визначений результат. Це може бути забезпечено, наприклад, встановленням пріоритетів застосування правил. Такі пріоритети можуть бути задані просто порядком описів правил.

Прикладом детерміністичних алгоритмів є так звані **алгоритми Маркова.** Інформація, що обробляється за алгоритмом Маркова, є словом в деякій фіксованій абетці А. Алгоритм (програма) подається послідовністю пар слів в абетці А. Пари, що складають алгоритм, є підстановками $\alpha \rightarrow \beta$

де α, β – слова в абетці А, причому β може бути порожнім (λ). Програма має вигляд

$\boldsymbol{\alpha_1 \rightarrow \beta_1}$
$\boldsymbol{\alpha_2 \rightarrow \beta_2}$
.....

$\alpha_i \rightarrow \beta_i!$

.....

$\alpha_n \rightarrow \beta n.$

Деякі підстановки помічаються знаками оклику та називаються заключними.

В алгоритмах Маркова правила замін лінійно упорядковані (цей порядок визначається послідовністю опису правил). Тоді застосування правил встановлюється в такий спосіб.

Марківська стратегія застосування: якщо можна застосувати кілька правил, то фактично застосовується те з цих правил, яке в описі алгоритму зустрічається першим. Якщо правило можна застосувати в декількох місцях слова, то вибирається саме ліве з цих місць.

Таким чином, марківські алгоритми завжди є детермінованими і детерміністичними. Зокрема , справедливе наступне:

- кожне обчислення за марківською стратегією є також загальним обчисленням у СТЗ;
- для кожного вхідного слова існує **точно одне** кінцеве або ж нескінченне марківське обчислення; алгоритми Маркова є детерміністичними (а звідси результат, якщо він існує, однозначно визначений).

У вхідному слові знаходять фрагмент, який співпадає з лівою частиною першої підстановки. Якщо він знайдений, то його у вхідному слові замінюють на праву частину, в протилежному випадку розглядається друга підстановка з алгоритму і т.д. Обчислення закінчується, коли жодна з лівих частин підстановок не є фрагментом слова, що обробляється або коли виконана заключна підстановка. Таким чином процес може бути нескінченим.

Алгоритми Маркова складають теоретичну основу систему програмування, що використовує мову РЕФАЛ.

Приклад 2.5. Нехай задана абетка A = {1, +}та наступний алгоритм:

1. **1+ → +1**
2. **++ → +**
3. **+ → λ!**

При виконанні алгоритму слово виду

11...1+11...11

перетворюється в слово

11...1

в якому кількість символів “1” таке , як у вхідному слові.

Приклад 2.6. Нехай абетка A = {1, *, y, z} та наступний алгоритм:

1. ***11 → v*1**
2. ***1 → v**
3. **1v → v1z**
4. **zv →vz**
5. **z1 →1z**
6. **v1 → v**
7. **vz → z**
8. **z → 1**
9. **1 → 1!**

Розглянемо протокол обчислення на вхідному слові **11*111**.

11*111	***11→ v*1**
11v*11	***11→ v*1**
11vv*1	***1 → v**
11vvv	**1v→ v1z**
1v1zvv	**1v→ v1z**
v1z1zvv	**zv → vz**
v1z1vzv	**1v → v1z**
v1zv1zzv	**zv → vz**
v1vz1zzv	**1v → v1z**

vv1zz1zzv zv → vz
vv1zz1zvz zv →vz
vv1zz1vzz 1v →v1z
vv1zzv1zzz zv → vz
vv1zvz1zzz zv → vz
vv1vzz1zzz 1v → v1z
vvv1zzz1zzz z1→ 1z
vvv1zz1zzzz z1 → 1z
vvv1z1zzzzz z1→ 1z
vvv11zzzzzz v1→ v
vvv1zzzzzz v1 →v
vvvzzzzzz vz → z
vvzzzzzz vz→ z
vzzzzzz vz → z
zzzzzz z → 1
1zzzzz z → 1
11zzzz z → 1
111zzz z → 1
1111zz z → 1
11111z z →1
111111 1 →1!

Якщо вважати, що у вхідному слові закодована задача добутку 2*3 в унарній системі числення, то у вихідному слові будемо мати 6.

Приклад 2.7. Нехай задана система текстових замін R на множині символів

{v, ¬ , true, false, (,)}

для спрощення логічних виразів, що побудовані тільки із символів даної множини, до нормальної форми. Ця система складається з наступних правил:

(1) ¬ ¬ → **ε**
(2) ¬true → false
(3) ¬false → true
(4) (true) → true
(5) (false) → false
(6) false ∪ → **ε**
(7) ∪ false → **ε**
(8) true ∪ true→true

Алгоритм, визначений даною системою, працює за марківською стратегією коректно для замкнутих булевих термів (тобто замкнуті булеві терми переводяться в семантично еквівалентні однозначні нормальні форми, що складаються з true і false). Це справедливо навіть для замкнутих термів з не розставленими дужками. Однак у цьому випадку можливі такі застосування, що не є еквівалентними перетвореннями. Для терма

¬ true∪true

за марківською стратегією отримаємо обчислення

¬ true ∪true → (правило (2))
false∪true → (правило (6))
true

У загальній не детерміністичній стратегії додатково одержують коректне обчислення

¬ true ∪ true → (правило (8))
¬ true → (правило (2))
false

Завдяки марківській стратегії однозначно визначається вибір правила, яке можна застосувати, що для багатьох задач спрощує формулювання алгоритму. У певних випадках може також виявитися корисним введення часткового упорядкування правил (частковий порядок над правилами замін).

2.3. Відображення, яке відтворюється алгоритмами текстових замін

Шляхом зіставлення вихідного слова кожному вхідному слову при кінцевому обчисленні детерміністичні алгоритми обчислюють **часткові функції**. **Функції є частковими, тому що іноді при деяких вихідних даних алгоритми не завершуються і тому результат обчислень не визначений.** Явного використання часткових функцій можна уникнути шляхом введення особливого символу $\perp$ ("дно"), що символізує відсутній "результат", що не завершується ("розбіжного" обчислення).

Кожен детермінований алгоритм R у формі СТЗ на послідовностях символів V* визначає відображення:

$$f_R: V^* \to V^* \cup \{\perp\}$$

внаслідок наступних правил. Нехай справедливо:

(1) f(t) = r, якщо слово r є результат обчислень за R для вхідного слова t;

(2) f(t) = $\perp$, якщо обчислення за R для вхідного слова t не закінчується.

Тоді ми говоримо: алгоритм R обчислює функцію f .

Символ $\perp$, таким чином, позначає псевдо результат обчислення, що не завершується. За допомогою введення цього символу обходять явну роботу з частковими відображеннями.

Якщо слова t $\in$ V* розуміти як подання визначені інформації з множини A, то існує функція інтерпретації така, що (A, V*, I) утворить інформаційну систему, і якщо функція f_R , відтворена алгоритмом R, погоджена з інтерпретацією, то R відтворює також функцію між інформаціями, таким чином R відтворює відображення інтерпретацій.

Приклад 2.8. Розглянемо алгоритм множення чисел, що представлені кількістю штрихів. Нехай інтерпретація числа штрихів визначається відображенням

$$I:\{<, |, >\}^* \to N$$

з I(<|...|>) = n для слова <|...|> з n штрихами. Тоді алгоритм множення так представлених чисел із входом

<|...|>	*	<|...|>	→	<|...|>
n		m		n*m
I↓		I↓		I↓
Mult(n,		m)=		n*m

зображає відображення пари чисел на їхній добуток, тобто відображення множення. Утім, доведення того, що цей алгоритм справді представляє множення числа штрихів, технічно досить важко. До того ж необхідно також ввести функцію інтерпретації для слів, утворених за допомогою допоміжних символів, що можуть зустрічатися в процесі обчислень.

Детерміністичні АТЗ (алгоритми текстових замін) породжують часткові відображення на словах і тим самим, оскільки слова служать для представлення інформації і відображення погоджене з інтерпретацією, часткові відображення між відповідними інформаціями. **Недетерміновані алгоритми визначають відношення**. Із системою замін R на V ми зв'язуємо відношення (граф, що обчислюється через алгоритм)

$G_R \subseteq V^* \times (V^* \cup \{\perp\})$,

обумовлене в такий спосіб:

G_R = { (t, r) $\in$ V* x V*: r - вихідне слово обчислення за R із вхідним словом t}

$\cup$ { (t , $\perp$) $\in$ V* x {$\perp$}:існує нескінченне обчислення за R для вхідного слова t}.

Звернемо увагу, що у відношенні G_R для кожного входу t існує вихід $r \in V^* \cup \{\perp\}$, тобто існує щонайменше одне $r \in V^* \cup \{\perp\}$ з $(t, r) \in G_R$.

Приклад 2.9.. Розглянемо не детерміністичний алгоритм із неоднозначним результатом:

Нехай вхід має форму

∠<|||...|>

а система R задана правилами:

∠< → <∠

∠| → |+|

∠| → ||

∠> → |>

Так заданий алгоритм R по будь-якому натуральному числу n (представленому у вигляді числа штрихів) породжує натуральне число m, більше, ніж n, або не завершується. На рис.2. 1 приведена схема дерева обчислень для входу ∠ <|||>.

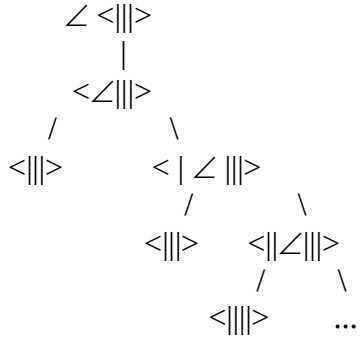


Рис. 2.1. Дерево обчислень

Отримумо наступне відношення (при інтерпретації послідовності штрихів, як натурального числа):

$$G_R = \{(n,m): n \in N \wedge ((m \in N \wedge n < m) \vee m = \perp)\}$$

Детерміністичні алгоритми обчислюють функції. Виникає питання про те, чи можуть усі математичні функції обчислюватися за допомогою алгоритмів. Один з фундаментальних результатів сучасної математичної логіки належать Курту Геделю і відповідає на запитання про обчислювальні функції, пред'явлених через алгоритми. Спрощуючи формулювання, результат Геделя говорить, що не усяка функція допускає її обчислення за допомогою алгоритму. Ті **функції**, для обчислення яких може бути заданий алгоритм, називають **обчислювальними.**

Існує багато різних формалізацій поняття "алгоритм". Деякими прикладами цього є:

- текстові і термові заміни (редукція);
- рекурсивні функції;
- машини Т'юрінга (а також реєстрові машини).

Однак усі ці формалізації ведуть до того ж самого поняття "обчислювальні функції". Точне обговорення цих взаємозв'язків було розпочато в інформатиці за назвою "теорія обчислення" . Теоретично, поняття алгоритму, заснованого на заміні текстів, цілком достатньо. Кожна обчислювальна функція може бути обчислена за допомогою АТЗ. Утім, зовсім не так просто при великих і громіздких СТЗ зрозуміти відображення, що задаються за їхньою допомогою, і відповідно впевнено встановити, що бажане відображення дійсно обчислюється. Розуміння АТЗ особливо стає важливим в силу наступних обставин:

- при складних постановках задач часто потрібно дуже багато правил;
- важко встановити відображення, представлене за допомогою АТЗ;

- дуже важливим стає композиція і структурування алгоритмів у вигляді текстових замін.

Тому шукають інші структурні, більш зручні для використання можливості представлення алгоритмів, при яких було б легше доводити, що алгоритм дійсно обчислює бажану функцію.

2.4. Обчислювальні структури

Алгоритми працюють над елементами даних, що можуть бути об'єднані в так званий носій (множину даних). Для формулювання алгоритмів, поряд з елементами даних що використовуються, досить важливі наявні в розпорядженні ефективні функції від цих елементів. Носії, що фігурують в алгоритмах, і операції можуть трактуватися разом як обчислювальні структури. **Обчислювальна структура охоплює сімейство носіїв (дані) і сімейство відображень між цими носіями.** Обчислювальні структури виявляються у всіляких проявах. Наприклад , кишеньковий калькулятор, так само як і могутня ЕОМ, можуть математично сприйматися й описуватися як обчислювальні структури.

2.4.1. Сімейства функцій і множин як обчислювальні структури

Поняття обчислювальної структури близьке до поняття математичної структури або алгебри. Обчислювальна структура складається із сімейства множин, названих носіями, і множини відображень між носіями. Нехай S і F - множини позначень.

Обчислювальна структура A складається із сімейства $\{ S^A : S \in S\}$ носіїв S^A і сімейства $\{fa^A : f \in F\}$ відображень f^A між цими носіями.

Позначимо:

$$A=(\{s^A : s \in S\}, \{f^A : f \in F\}).$$

Елементи $s \in S$ є позначеннями для носіїв і називаються **типами.** Елементи $f \in F$ є позначення для відображень і називаються **символами функцій або знаками операцій**. Для кожного $f \in F$ існує одне $n \in N$ таке, що має місце: f^A є n-місцева функція, і існують типи $S_1, S_2;, .., S_{n+i} \in S$ такі, що

$f^A : S_1^A \times S_2^A \times ... \times S_n^A \rightarrow S^A_{n+1}$

Може також бути і n = 0 ("нульмісні" відображення). Це такі відображення, що з порожнім списком аргументів одержують один елемент з області значень, що визначений у типі S_1 .

Для усунення часткових відображень знову використовується спеціальний елемент $\perp$("дно") для представлення невизначеного значення функції. Нехай M - множина , що не містить $\perp$. Множина $M^\perp$ визначається як

$M^\perp$ = def M $\{\perp\}$.

Елемент $\perp$ представляє "невизначений" результат функції, наприклад у випадку алгоритму, що не завершується. Відображення

$f : M_1^\perp \times ... \times M_n^\perp \rightarrow M_{n+1}^\perp$

називається **суворим**, коли справедливо: якщо одним з аргументів функції є $\perp$, то результат функції теж є $\perp$. Це відповідає простому припущенню, що результат застосування функції до списку аргументів визначений тільки в тому випадку, коли визначені всі аргументи. Поширення часткових відображень на усі відображення шляхом додавання $\perp$ до носіїв приводить до суворих відображень.

Для усунення часткових відображень надалі (якщо не обговорено щось інше) буде передбачатися, що кожний носій містить спеціальний елемент $\perp$ і що всі розглянуті відображення є суворими.

Приклад 2.10. Приклади обчислювальних структур:

(1) Обчислювальна структура BOOL булевих значень

Множина S типів обчислювальної структури **BOOL** задана так :

S = {**Bool**}.

Множина F символів функцій структури **BOOL** задана так :

F = {true, false, $\neg$, $\vee$,$\wedge$}.

Множина носіїв $B^{\perp}$ відповідає типу **Bool**, таким чином має місце

Bool $^{\mathbf{BOOL}}$ = $\mathbf{B}^{\perp}$ = {L, O, $\perp$ }.

Для кращого розуміння символи двомісних функцій f часто записуються в **інфіксній** формі. Замість f(x, y) ми пишемо вираження x f y. Вживання інфіксного запису вказується через коментарі при завданні функціональної залежності. Аналогічно будемо записувати і символ f одномісної функції . Символи з F позначають наступні функції:

True ^{Bool} :$\rightarrow B^{\perp}$,

false ^{Bool} : $\rightarrow B^{\perp}$,

$\neg^{Bool}$: $B^{\perp} \rightarrow B^{\perp}$ (префікс без застосування дужок),

$\wedge^{Bool}$: $B^{\perp}$ X $B^{\perp}$ $\rightarrow$ $B^{\perp}$ (інфікс),

$\vee^{Bool}$: $B^{\perp}$ X $B^{\perp}$ $\rightarrow B^{\perp}$ (інфікс),

Причому для a, b $\in$ B має місце

True ^{Bool} = L ,

false ^{Bool} = O ,

$\neg^{Bool}$ b = not(b),

a $\vee^{Bool}$ b = or(a, b),

a $\wedge^{Bool}$ b = and(a, b).

Тут функції not, and, or - є суворими і тому їхні значення для випадку, коли один з аргументів є $\perp$, також встановлені.

(2) Обчислювальна структура NAT натуральних чисел

Множина S типів обчислювальної структури **NAT** задане через **S={bool, nat}.** Множина F символів функцій обчислювальної структури **NAT** задане через

F = {true, false, $\neg$, $\wedge$, $\vee$, zero, succ, pred, add, mult, sub, div, $\leq$, $\stackrel{?}{=}$ }.

Типам обчислювальної структури NAT запропоновані носії

$\mathbf{Bool}^{\mathbf{NAT}} = \mathbf{B}^{\perp}$, $\mathbf{nat}^{\mathbf{NAT}} = N^{\perp}$.

Символам з F обчислювальної структури NAT запропоновані наступні функції (для $\neg$,$\wedge$,$\vee$ прийняті ті ж визначення, що й в обчислювальній структурі BOOL):

zero ^{NAT} : $\rightarrow N^{\perp}$,

succ ^{NAT} : $N^{\perp} \rightarrow N^{\perp}$,

pred ^{NAT} : $N^{\perp} \rightarrow N^{\perp}$,

add ^{NAT} : $N^{\perp}$ x $N^{\perp} \rightarrow N^{\perp}$,

mult ^{NAT} : $N^{\perp}$ x $N^{\perp} \rightarrow N^{\perp}$,

sub ^{NAT} : $N^{\perp}$ x $N^{\perp} \rightarrow N^{\perp}$,

div ^{NAT} : $N^{\perp}$ x $N^{\perp} \rightarrow N^{\perp}$,

$\leq$ ^{NAT} : $N^{\perp}$ x $N^{\perp} \rightarrow B^{\perp}$ (інфікс) ,

$\overset{?}{=}$: N x N → B (інфікс) .

Для символів функцій zero, succ, pred, add, mult, sub, div ми застосовуємо часто інфіксний спосіб запису 0, +1, -1, +, *, -, /. Функції специфікуються в такий спосіб:

$zero^{NAT} = 0$.

Нехай x, y ∈ N. Тоді має місце

$succ^{NAT}(x) = x + 1$,

$pred^{NAT}(x) = x - 1$ (якщо $x => 1$) ,

$pred^{NAT}(x) = \perp$,

$add^{NAT}(x , y) = x + y$,

$mult^{NAT}(x , y) = x * y$,

$div^{NAT}(x , y) = x / y$,

$div^{NAT}(x , 0) = \perp$,

Тут x/y позначає ділення натуральних чисел *x* на *y* з ігноруванням залишку.

$(O \leq^{NAT} x) = L$,

$(x+1 \leq^{NAT} 0) = O$,

$(x+1 \leq^{NAT} y+l) = (x \leq^{NAT} y)$,

$(x \overset{?}{=}^{NAT} У) = and\ (x \leq^{NAT} y, y \leq^{NAT} x)$.

Завдяки умові строгості також встановлено, які значення мають відображення у випадку, коли один з аргументів є ⊥ ; у таких випадках значенням відображення завжди є ⊥. Особливо зверніть увагу, що значення терму $(x \overset{?}{=} y)$ для аргументів x, y, не рівних ⊥ , узгоджується зі значенням (x = y), а для x =⊥ або y = ⊥ значенням терму завжди є ⊥ .

(3) Обчислювальна структура кишеньковий калькулятор CALC

Кишеньковий калькулятор також може бути описаний як обчислювальна структура. Множина S типів цієї структури задається так:

S = {key, state) (key - клавіші, state - стан)

Типам обчислювальної структури **CALC** запропоновані наступні носії:

$key^{CALC} = T$ з множиною T = {0, ..., 9, +, *, =},

$state^{CALC} =$ **Z** , причому множина Z станів і область позначень W визначається в такий спосіб:

Z = {(s, p, d): s ∈ W ∧ p ∈ {+,*,=} ∧ d ∈ W} ∪ {помилка},

$W = \{0, ... , 10^{10} - 1\}$.

Множина Z складається, таким чином, із трійок (s, p, d) і особливого елемента "помилка". Компонента d - це "дисплей", тобто видиму індикацію; p позначає символ операції, яку тільки що запам'ятали, а s - значення, яке зберігають всередині калькулятора. Множина W визначає область значень в арифметиці калькулятора. Множина символів функцій обчислювальної структури **CALC** задається так:

F = {ein, tip, 0,1, 2, ... , 9, +, *, =}.

Символам з F структури **CALC** запропоновані наступні функції:

0, 1, ..., 9,+, *, = відповідають одномісним символам функцій, що видають як результат відповідні клавіші. Символам функцій ein і tip запропоновані функції

$ein^{CALC} : \rightarrow Z$, $tip^{CALC} : T \times Z \rightarrow Z$.

Ці функції описуються в такий спосіб:

$ein^{CALC} = (0, =, 0)$.

Для x ∈ {0, ..., 9} має місце

$$tip^{CALC}(x, (s, p, d)) = \begin{cases} помилка, & \text{я якщо } *10 + x \in W) \\ (s, p, d, *10 + x) & \text{інакше} \end{cases}$$

$$\text{tip}^{CALC}(=, (s, p, d)) = \begin{cases} (s, =, s * d) \text{ при } s * d \in W \text{ і } p = * \\ (s, =, s + d) \text{ при } s + d \in W \text{ і } p = + \\ (d, =, 0) \text{ при } p = "=" \\ \text{помилка інакше} \end{cases}$$

tip ^{CALC} (+, (s, p, d)) = (d, +, 0) ,
tip ^{CALC} (*, (s, p, d)) = (d, *, 0).

Послідовності натискань клавіш після включення обчислювача, наприклад, послідовності 9 1+ 1 2=* 2 =

відповідає терм

tip(=, tip(2, tip(*, tip(=, tip(2, tip (1, tip(+, tip(l, tip(9, ein))...),

що веде до стану (103, =, 206).

2.4.2. Сигнатури

Щоб встановити множину символів функції і типів, що зустрічаються в обчислювальній структурі, а також встановити, яким чином символи функції змістовно можуть бути зв'язані між собою використовуються сигнатури.

Сигнатура C - це пари (S, F) множин S і F, причому

- S позначає множин типів, або імен для носіїв;
- F позначає множин символів (імен) функцій;

Для кожного символу функції f ∈ F нехай задана її функціональність **fct f ∈ S $^{+}$.**

Надалі для кращого розуміння при завданні функціональності для f ми будемо писати **fct f= (S_1, ... , S_n) S_{n+l},** щоб виразити, що f^A в обчислювальній структурі A з відповідною сигнатурою S використовується для позначення відображення

$$f^A : S_1^A \text{ x ... x } S_n^A \rightarrow S^A_{n+1}$$

Приклад 2.11. Сигнатура обчислювальної структури **NAT** булевих значень і натуральних чисел з вищенаведеного прикладу дає приклад сигнатури.

S_{NAT} = **(bool, nat),**

F_{NAT} = {true, false, , , , zero, succ, pred, add, mult, sub, div, ≥ , ? , =},

fct true = bool,
fct false = bool,
fct ¬ = (bool) bool (префікс без дужок),
fct ∨ = (bool, bool) bool (інфікс),
fct ∧ = (bool, bool) bool (інфікс),
fct succ = (nat) nat, ...

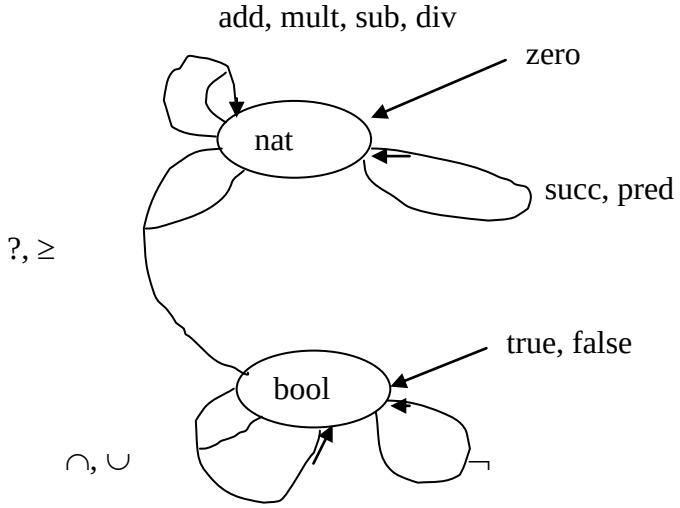


Рис.2.2. Діаграма сигнатури

Сигнатури допускають наочне графічне подання у вигляді **діаграми сигнатури**. Така діаграма для кожного типу містить вузол. Для кожного n-місцевого символу операції - ребро з n вхідними вузлами і одним вихідним вузлом. Для обчислювальної структури NAT ми одержуємо діаграму сигнатури, показану на рис. 2.2.

Задання однієї тільки сигнатури, звичайно, недостатньо для того, щоб однозначно охарактеризувати обчислювальну структуру. Існує багато різних обчислювальних структур з однієї і тією же сигнатурою.

Приклад 2.12. Обчислювальна структура INT цілих чисел, з точністю до іменування типів, має ту ж сигнатуру, що й обчислювальна структура натуральних чисел:

S = {boot, int}, F = { succ, pred, zero, ...}, з **int** ^{INT} **= Z .**

При цьому розрізняються зв'язані з елементами сигнатури носії і відображення:

pred ^{INT} (z) = z - 1.

Отже, з однієї і тією же сигнатурою можна зв'язати різні обчислювальні структури.

Приклад 2.13. Для сигнатури

S ={bs, bit}, F={null, eins, empty, ra}

з функціональностями fct null = bit, fct eins = bit, fct empty = bs, fct ra = (bs, bit) bs існують у якості можливих обчислювальні структури, наприклад BN, BS, що визначаються наступним чином:

bit $^{\mathbf{BN}}$ = {0,1}, bit $^{\mathbf{BS}}$ = B,
bs $^{\mathbf{BN}}$ = N, bs $^{\mathbf{BS}}$ = B*,
null $^{\mathbf{BN}}$ = 0, null $^{\mathbf{BS}}$ = O,
eins $^{\mathbf{BN}}$ = 1, eins $^{\mathbf{BS}}$ = L,
empti $^{\mathbf{BN}}$ = 0, null $^{\mathbf{BS}}$ = ?,
ra $^{\mathbf{BN}}$ (n, x) = 2*n+x , ra $^{\mathbf{BS}}$ = n o <x>,

Обчислювальні структури вказують на структурні властивості інформаційних систем. Зокрема, має місце:

$$(\{s^A \in S\}, S, T)$$

утворить знову інформаційну систему з

$$T: S \rightarrow \{ s^A: s \in S\}, \text{ причому } T[s] = s^A$$

Також $(\{f^A \; f \in F\}, F, I)$ з $I: F \rightarrow \{ f^A: f \in F\}$, причому $I[f] = f^A$ є інформаційною системою.

Для заданої обчислювальної структури з визначеною сигнатурою завжди існує спеціальна структура, алгебра термів, що складається з множини термів, що можуть бути сформульовані над сигнатурою.

Контрольні запитання

1. Що представляє собою формалізоване подання алгоритму?
2. Що називають системою текстових замін?
3. Приведіть приклади СТЗ?
4. Що таке терм?
5. Яка заміна називається термінальною?
6. Що таке детерміністичний алгоритм?
7. Який алгоритм є алгоритмом Маркова?
8. Приведіть приклад алгоритмів, в яких використовується марківська стратегія.
9. Що таке часткова функція?
10. Що таке відображення, яке відтворюється алгоритмами текстових замін?
11. В якому випадку відтворюються часткові відображення, а коли повні?
12. Що називають обчислювальними функціями?
13. Що називають обчислювальною структурою?

ЛЕКЦІЯ 3. АЛГОРИТМИ ЯК СИСТЕМИ ПІДСТАНОВКИ ТЕРМІВ

Правила підстановки термів. Система підстановки термів. Алгоритми підстановки термів. Коректність систем підготовки термів.

3.1. Основні терми

При заданій сигнатурі існує множина основних термів, що можуть бути утворені за допомогою символів функцій сигнатури. Нехай Σ = (S, F) є сигнатура. **Множина основних термів типу s з s ∈ S визначається таким способом:**

(1) кожен нульмісний символ функції f ∈ F з fct f = s утворює основний терм типу s;

(2) кожна послідовність символів $f(t_1, ..., t_n)$ з $f \in F$ і $fct\ f= (S_1, ..., S_n)\ s$ є основний терм типу s, якщо для всіх i, $1 <= i <= n$, t_i є основний терм типу s_i.

Множину всіх основних термів сигнатури Σ позначимо через W_Σ, а множину основних термів типу s - через W_Σ^s. Якщо не існує нульмісних символів функцій, то множина W_Σ порожня.

Прикладами основних термів типу **nat** обчислювальної структури NAT натуральних чисел є:

succ (zero), add (succ (succ (zero)), pred (succ (zero))).

Поряд із класичним математичним способом запису p(a, b) застосування функції для утворення основних термів, коли символ функції p передує взятому у дужки списку (a1, a2) аргументів (дужковий префіксний запис), для спеціальних символів функцій p часто використовуються й інші способи запису (операторні способи), наприклад:

- a p b (*інфіксна форма*);
- p a b (*префіксна форма без застосування дужок*),
- a b p (*постфіксна форма*).

Прикладами інфіксної форми запису є використання символів операцій +, *, ∧, ∨, ∈ і ін. При одномісних символах функцій використовується як префіксна (для ¬), так і постфіксна форма без застосування дужок (наприклад ! для функції факторіал).

Якщо встановлюється "потужність" (кількість аргументів) символу функції, то при чисто префіксній або чисто постфіксній формі символ функції однозначно зіставляється з її аргументами. Для інфіксної форми це не має місця. Наприклад, якщо немає якої-небудь домовленості про старшинство операції, то неясно, чи означає запис x + y * z терм (x + y) * z або терм x + (y * z). Щоб досягти однозначності також і для інфіксної форми запису, можна поряд з використанням дужок ввести визначене старшинство (пріоритети) на множині операторів. Якщо не існує пріоритетів, то розставляються дужки, завжди зліва направо. Це значить що x + y + z позначає (x + y) + z.

Іноді використовують змішану форму запису, щоб можна було записати вираз в більш доступному для розгляду вигляді. Для цього існує ряд способів для спрощення і скорочення запису. Та обставина, що символ функції треба застосовувати в операторній формі запису, може бути виражена у вказівці функціональності. Наприклад, можна за допомогою спеціального символу (як точка) задати позиції аргументів.

Приклад. Для терму x < y ∧ y < z допускається скорочений запис x < y < z. Цей спосіб запису відповідає тримісному символу функції з функціональністю

fct .<.<.= (nat, nat, nat) bool;

точки позначають позиції аргументів.

Якщо є обчислювальна структура A с сигнатурою Σ, то основні терми в A допускають інтерпретацію. **Перехід від основного терму t (представлення) типу s до відповідного елемента a з множини A називають інтерпретацією t в A.** Інтерпретація I^A тому означає відображення

$I^A: Wc \rightarrow \{a \in s^A : s \in S\}$.

Для кожного основного терма t запис I^A [t] позначає інтерпретацію t в A. Пишуть також t^A замість I^A[t]. Інтерпретація виходить заміною в основному термі символів функцій на відповідні функції:

I^A [f(tl,..,tn)] =f^A(I^A [tl],..., I^A [tn]}).

Зокрема , ({a $\in s^A$: s $\in$ S}, Wc, I^A) утворить знову інформаційну систему.

Приклад 3.1. З типом **nat** ми пов'язуємо відповідно до домовленості носій N . Для заданих термів ми одержуємо наступні інтерпретації:

I^{NAT} [succ (zero)] = 1, I^{NAT} [pred (zero)] = $\perp$,

I^{NAT} [pred (add (succ (succ (zero)), zero))] = $\perp$.

У класичній математиці часто задана інтерпретація опускається і замість t^A просто записується t; різницею між основним термом і його інтерпретацією там свідомо зневажають.

Для кожної обчислювальної структури A сигнатури Σ основні терми типу s $\in$ S можуть використовуватися як представлення елементів з множини s^A які зв'язані з типом s в A. Якщо для кожного елемента a, що не дорівнюють $\perp$, носіїв з A є представлення терма, тобто для кожного s і кожного a $\in s^A$ ($\neq\perp$) існує основний терм типу s з t^A=a, то A називається такою, що утворює терм; це аналогічно вимозі, що у відповідній інформаційній системі відображення інтерпретацій є сюрективним.

Інтерпретація ("значення") основного терма допускає відповідно обчислення терма. Один із простих способів організації такого обчислення являють собою схеми.

3.2. Обчислення основних термів: схеми

Основні терми мають характерну внутрішню структуру. Основний терм утворюється із символів функцій і послідовності (іноді порожній) основних термів ("підтермів"), що є термами-аргументами.

Схема для основного терма - це графічне подання обчислень при інтерпретації цього терма; схема складається з прямокутників, у які заноситься інтерпретація основних термів, і з підсхем для обчислень підтермів. Обчислення інтерпретації основного терма допускає зручне його проведення за схемою. Оскільки інтерпретація основного терма виконується через значення інтерпретації його підтермів, ця інтерпретація підтермів упорядковується за допомогою схеми, структура якої аналогічна структурі самого основного терма.

Приклад 3.2. :

(1) Основному терму ((1 + 2) * 3) - 4 з інтерпретацією в N відповідає схема, показана на рис. 3.1

(2) Основному терму (true $\wedge$ false) $\vee$ (false $\vee$ (($\neg$ false $\wedge$ true)) з інтерпретацією в **BOOL** відповідає схема, приведена на рис. 3.2

(3) Обчислення за допомогою кишенькового калькулятора: Основному терму

tip (=, tip (3, tip (+, tip (2, ein))))

відповідає схема на рис. 3.3

Основні терми можуть дуже легко застосовуватися для представлення елементів з множини носіїв обчислювальної структури. Щоб можна було визначити відображення між цими елементами, використовують терми з вільними ідентифікаторами.

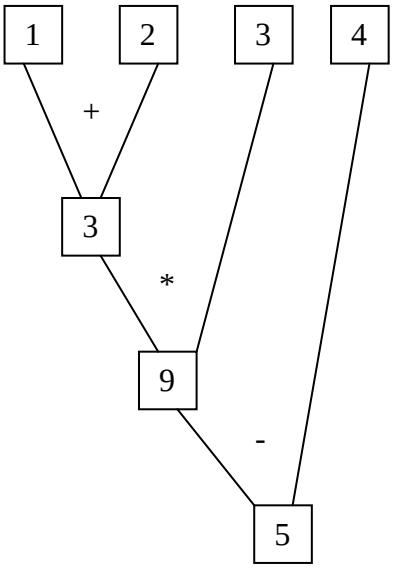


Рис. 3.1.. Обчислювальна схема

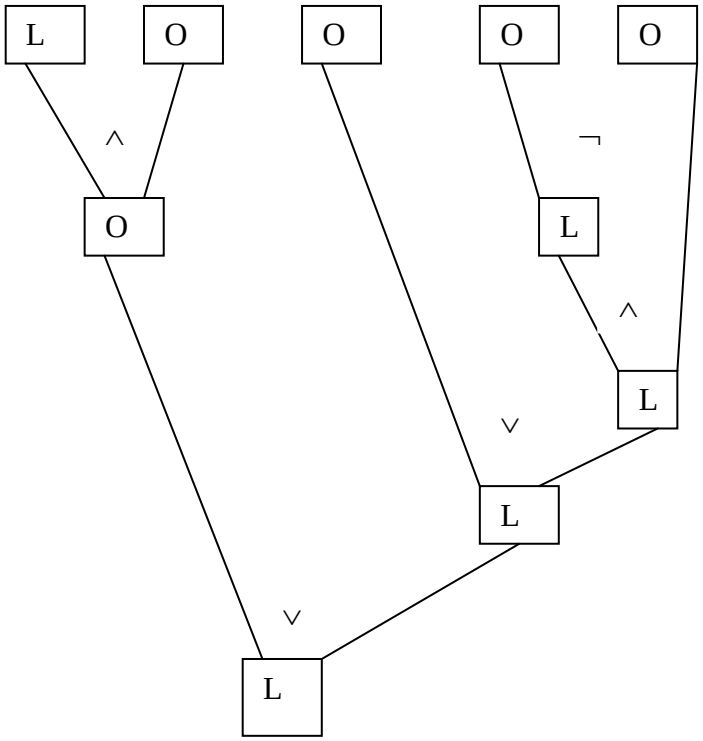


Рис. 3.2. Обчислювальна схема

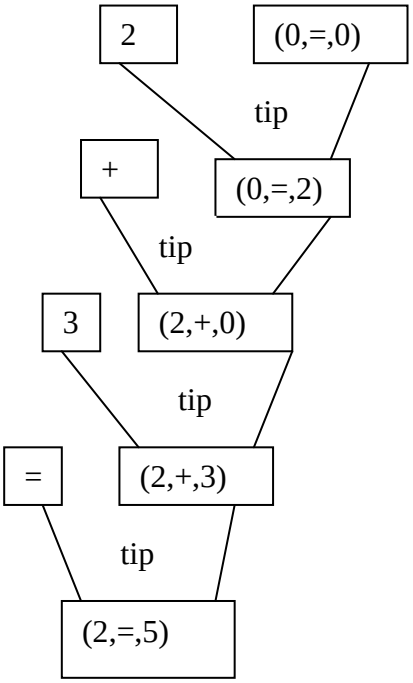


Рис. 3.3.. Обчислювальна схема

3.3.Терми з вільними ідентифікаторами

Ідентифікатор ("змінна", "невідоме") - це власник місця ("ім'я") для терма (або елемента), що пізніше може бути підставлений на це місце. Ідентифікатори можуть розумітися як імена термів або елементів, що будуть конкретизовані тільки пізніше.

Нехай Σ = (S, F) - сигнатура, і X = {Xs: s ∈ S} - сімейство множин ідентифікаторів. Нехай множини Xs ідентифікаторів попарно не перетинаються і відмінні від символів

функцій у F. W_Σ (X) позначає алгебру термів, розповсюджену на X, тобто $W_{\Sigma i}$ з ΣI=(S, F U {x ∈ Xs: s ∈ S}) і fct x = s для x ∈ Xs, де Xs позначає множину ідентифікаторів (власників місць, позначень) типу s.

Приклади 3.3:

(1) Рівняння з "невідомими" у математиці - це рівняння між термами з ідентифікаторами, наприклад

$$ax^2 + bx + c = 0.$$

(2) Часто терми використовують вільні ідентифікатори, щоб визначати функції. Функція f може бути визначена так:

$$f: N \rightarrow N \text{ з } f(x) = 2x + 1$$

Терми з вільними ідентифікаторами називаються також **поліномами**. Замість ідентифікаторів у терми можуть підставлятися інші терми. Відповідне відображення називається **підстановкою в терми з вільними ідентифікаторами.**

Нехай t - терм з ідентифікаторами, x - ідентифікатор типу s і r -терм типу s; через t[r/x] позначається терм, що виходить, коли ідентифікатор x замінюється на r. Цей процес називається **підстановкою**. Підстановки описуються формально, аналогічно побудові термів, за допомогою наступних рівнянь:

x [t/x] = t,

y [t/x] = y, якщо x і y - різні ідентифікатори, f(t1,...,tn) [l/x] = f(t_1 [t/x], ..., t_n [t/x] де f ∈ F з fct f= (S_1, ..., S_n) S_{n+1} і терми t, мають типи S_i.

Через t [t_1/x_1, ..., t_n/x_n] позначається терм, що виникає з терму t при одночасній підстановці t_i замість попарно різних ідентифікаторів X_i.

Нехай t - терм із вільними ідентифікаторами. Терм r назвемо **екземпляром** t, якщо r виходить з t шляхом заміни у ньому визначених вільних ідентифікаторів.

Приклад . Нехай задано терм t з вільними ідентифікаторами x, y, z:

$t =_{def}$ mult (add (succ (x), y), z).

Ми одержуємо екземпляр t через

t [zero/x, succ (zero)/y, succ (succ (zero))/z].

Виконання підстановки дає

mult (add (succ (zero), succ (zero)), succ (succ (zero))).

Аналогічно булевим термам з вільними ідентифікаторами також і загальні терми з вільними ідентифікаторами можуть інтерпретуватися через виконання підстановок.

3.4.Інтерпретація термів з вільними ідентифікаторами

Нехай A - обчислювальна структура із сигнатурою Σ = (S, F), а X - сімейство множин ідентифікаторів. Відображення

B : {x ∈ Xs: s ∈ S} → {a ∈ s^A : s ∈ S},

що кожному ідентифікатору x в X типу s ставить у відповідність елемент a ∈ s^A структури даних s^A типу s, називається **відображенням** X в A. Для кожного відображення B визначається інтерпретація I_B^A терму t з вільними ідентифікаторами з X з допомогою наступних рівностей:

$I_B^A[x] = B(x)$,

I_B^A [F(t_1,...,t_n)] = f^A ($I_B^A[t_1]$, ... , $I_B^A[t_n]$),

для n = 0 виходить $I_B^A[f] = f^A$.

Для зміни відображення в окремій точці використовується спеціальний запис, що схожий на запис, який використовується при підстановках: нехай B - функція (зокрема, відображення):

B : X →M,

тоді для m ∈ M запис B[m/x] означає ту функцію B', B': X ' M, для якої має місце

$$B'(z) = \begin{cases} m, \text{при } z = x, \\ B(z), \text{при } z \neq x. \end{cases}$$

Запис B[m/x] означає відображення, яке для всіх аргументів, крім x, збігається з B, і тільки для x відображення дає значення m.

Те, що для підстановок і змінних у точці по пунктах підстановки використовується той самий запис, підтверджуться наступним твердженням: нехай b, r - терми, x - ідентифікатор, а B -відображення; тоді справедливо

$I_B [t[r/x]] = I_B' [t]$, де B' = B [I_B [r]/x].

Інтерпретація терму t, у якій x заміняють на r при відображенні B, рівносильна інтерпретації терму t з відображенням B', що відрізняється від B тільки в відображенні x і там має значення інтерпретації x при відображенні B.

3.5.Терми з вільними ідентифікаторами як схеми

Терми з вільними ідентифікаторами можуть відігравати роль схем, у яких не всі значення визначені.

Приклад 3.4.. Площа S кільця s із внутрішнім радіусом r і зовнішнім радіусом R виходить по формулі

$$S = \pi(R^2 - r^2) .$$

Терму в правій частині формули відповідає схема, приведена на рис. 3.4

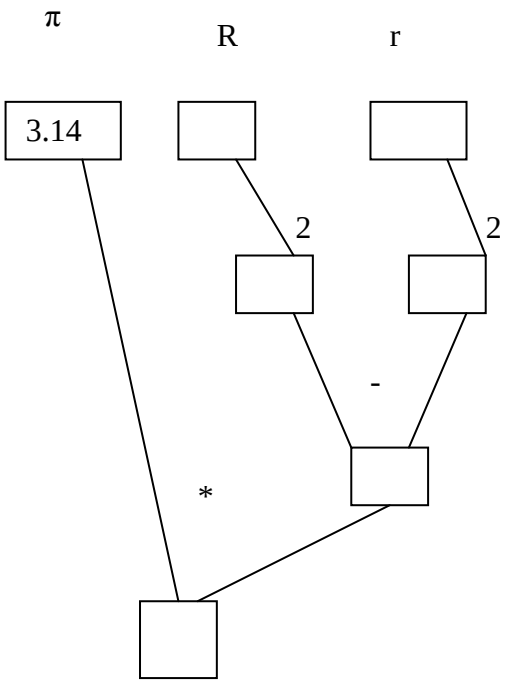


Рис. 3.4. Обчислювальна схема

Терм із вільними ідентифікаторами визначає обчислювальну схему. Схеми можна знайти у трохи іншій формі у багатьох місцях у управлінні. Наприклад, є схеми для нарахування зарплатні.

Якщо в термі визначені ідентифікатори зустрічаються багаторазово, то замість схем типу дерева одержують діаграми (тобто нециклічні орієнтовані графи).

Приклад 3.5. Терм (x - y) * (x + y) має схему, яку показано на рис. 3.5.

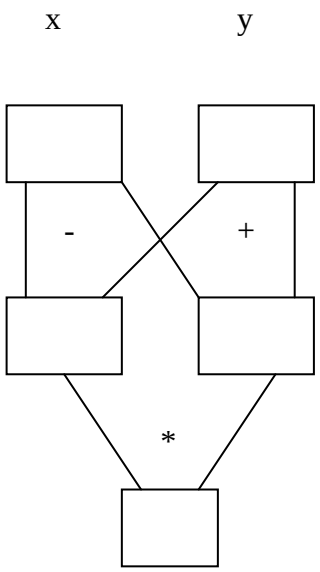


Рис. 3.5. Обчислювальна схема

Терми можуть використовуватися в системах заміни термів для представлення алгоритмів.

3.6. Алгоритми як системи підстановки термів

Найбільш наочний метод опису алгоритмів як системи текстових замін пропонують системи підстановки термів. Терми можуть будуватися над заданою сигнатурою за визначеними, чіткими правилами. До сигнатури і терму над нею можуть бути задані інтерпретації над обчислювальною структурою, тобто над алгеброю. Виникає інформаційна система, при якій терми виступають як представлення. Через інтерпретацію задається семантична еквівалентність на термах. Правила перетворення термів можуть тоді бути визначені так, що терми завжди будуть переводиться в семантично еквівалентні терми.

Множини правил для алгоритмів можуть бути пред'явлені у формі системи підстановки термів. Спеціальна структура термів, що виходить з їхньої побудови, може використовуватися для завдання правил підстановок і їх застосування.

3.6.1.Правила підстановки термів

Для заданої сигнатури Σ і заданого сімейства X множини ідентифікаторів пари (t,r) термів t , r однакового типу з вільними ідентифікаторами з X називається підстановкою термів (правилом підстановки термів - ППТ) або також схеми підстановки термів. Правило записується у вигляді $t \rightarrow r$.

У більшості випадків для ППТ потрібно, щоб всі ідентифікатори, що зустрічаються в r, також входили в t. Якщо в t і r замінюють визначені ідентифікатори X_1, ..., X_n на терми t_1, ..., t_n відповідних типів, то одержують екземпляр (частковий, конкретний випадок) правила. Відповідно

$$t [t_1/X_l, ... , t_n/X_n] \rightarrow r [t_1/X_1, ..., t_n/X_n]$$

називається екземпляром правила $t \rightarrow r$. Якщо t і r - основні терми, то екземпляр називається **повним.**

Приклад 3.6. Для правила

pred (succ (x)) -> x прикладами екземплярів є:

pred (succ (zero)) $\rightarrow$ zero, pred (succ (succ (y))) $\rightarrow$ succ (y).

З правила утворюються кроки підстановки термів завдяки тому, що правило застосовується до будь-якому підтерму наявного терму. Нехай $t \rightarrow r$ є екземпляр правила. Нехай заданий терм c, в якому зустрічається вільний ідентифікатор x. Тоді

$$C [t/X] \rightarrow C [r/X]$$

називається **безумовним застосуванням правила** до терму c[t/x].

Приклад 3.7. До терму

succ (succ (pred (succ (zero))))

може бути застосоване правило попереднього приклада. Це дає:

succ (succ (pred (succ (zero)))) -> succ (succ (zero)).

Аналогом алгоритмів у виді підстановки текстів є алгоритми у вигляді систем підстановки термів (алгоритми підстановки термів - АПТ).

3.6.2. Система підстановки термів

Множина (у загальному випадку кінцева) R правил підстановки термів на сигнатурі **Σ** називається **системою підстановки термів** (СПТ) над **Σ**. Якщо для послідовності термів t_i (0 <= i <= n) справедливо для i = 0,...,n- 1: $t_i \rightarrow t_{i+1}$ є застосування правила із системи підстановки термів R, то послідовність термів є **обчисленням** у R для t_o. Терм називається **термінальним** для системи R, якщо не існує терма такого, що

$$t \rightarrow r$$

є застосування правила з R.

Якщо в обчисленні, заданому за допомогою термів t_i , 0 <= i<= n, терм t_n є термінальним, то обчислення називається **термінованим** (таким, що завершується), а t_n - **результатом або виходом обчислення для входу t_o.**

Нескінченна послідовність $(t_i)_{i \in N}$ термів, що задовольняють приведену вище умову, називається **не термінованим** обчисленням, що завершується, R для t_o. Система R називається **в загальному випадку термінованою, якщо не існує обчислень, що не завершуються.**

Термінальні основні терми визначають нормальну форму. Вони часто також називаються **термами в нормальній формі відносно R.** Над системою R ми можемо терму t поставити у відповідність термінальний терм r як нормальну форму, якщо r є результат обчислень із входом t. Як правило, система нормальних форм, індукованих через СПТ, не є ні однозначною, ні повною. Терми, для яких існують тільки нескінченні обчислення, не мають нормальних форм. Для визначених термів можуть існувати обчислення з різними результатами.

Приклад 3.8. Символи функцій , ¬ , ∧, ∨ будуть використовуватися в префіксній і інфіксній формах запису з застосуванням дужок. Правила підстановок термів говорять.

(¬true) -> false

(¬false) -> true

(false ∨ x) -> x

(x ∨ false) -> x

(true ∨ true) -> true

(x y) →(¬ ((¬ x) (¬ y)))

Ця СПТ редукує кожний основний терм типу **bool** до терму true або false.

Повторне застосування ППТ із заданої множини правил ми можемо трактувати як алгоритм, що працює з термами як входу і виходу.

3.6.3. Алгоритми підстановки термів

Нехай задана система підстановок R; R визначає алгоритм:

1. Якщо R містить правило підстановки з застосуванням t -> м, то далі алгоритм продовжується з використанням r замість t.
2. Якщо R не містить правила підстановки з застосуванням t -> r, то алгоритм закінчується з r в якості результату.

Алгоритм підстановки термів (АПТ) тим самим виконує обчислення в R для кожного основного терму t. Часто обчислення складається в розв'язанні задачі перетворення заданого основного терму у визначену заздалегідь задану нормальну форму.

Якщо АПТ починає працювати з заданим основним термом t, то t називають також **входом для алгоритму**; якщо алгоритм завершується основним термом r, то r називається також **виходом або результатом**

Приклад .Нехай задана сигнатура **з** типом nat і типами функції

fct zero = nat

fct succ = (nat) nat

fct pred = (nat) nat

і нормальною формою succ (...(succ (zero))...) для терму типу nat. Зрозуміло, що, наприклад, терм pred(zero), при звичайній інтерпретації терма над натуральними числами має значення $\perp$, не є представленням у нормальній формі і йому не відповідає ніякий семантично еквівалентний терм в нормальній формі.

Приклад 3.9. Прості АПТ для редукції основних термів до заданої нормальної форми.

1. Для виключення символу функції pred в основному термі, значення якого відмінно від $\perp$, застосовуємо наступне правило:

pred (succ (x)) -> x.

Основні терми, як pred(zero) або succ(zero), є термінальними, але тільки succ(zero) представлений у нормальній формі.

2. Якщо розширити сигнатуру за рахунок символу функції add зfct add = (nat, nat) nat, то правила підстановки термів

add (succ (x), y) → succ (add (x, y)),

add (zero, y) → y

визначають алгоритм для переводу основних термів із символом функції add у семантично еквівалентний терм у нормальній формі, якщо інтерпретації усіх часткових термів, що фігурують, і тим самим саме значення основного терма відмінне від $\perp$.

3. Якщо сигнатуру натуральних чисел розширити символом функції mult з

fct mult = (nat, nat) nat ,

те правила підстановки термів

mult (succ (x), y) → add (y, mult (x, y))

mult (x, succ (y)) → add (x, mult (x, y))

mult (zero, zero) → zero

дають алгоритм для обчислення множення. Порівняння з АТЗ для множення чисел у вигляді кількості штрихів показує, наскільки простіше і наочніше АПТ.

Алгоритми підстановки термів виконуються над основними термами сигнатури. Вони здійснюють перетворення основних термів за допомогою обчислень. При цьому не розглядається, наскільки ці перетворення відповідають визначеним чисельним властивостям символів функцій. Усе-таки завдяки концепції обчислювальних структур стає можливою особливо ясна, проста концепція коректності СПТ.

3.6.4. Коректність систем підстановки термів

Нехай R - СПТ над сигнатурою **Σ**, і A - обчислювальна структура сигнатури **Σ.** Правило підстановки термів t→r над сигнатурою **Σ** називається **частково-коректним** стосовно **A**, якщо для будь-якої відображення B в A справедливо

$$I_B^A [t] = I_B^A [r],$$

причому I_B^A означає інтерпретацію термів в обчислювальній структурі A. Правило t→r тільки тоді коректно стосовно A, коли t = r дійсно є рівністю в A. СПТ R називається **частково-коректної стосовно обчислювальної структури A, якщо кожне правило є частково-коректним стосовно A.** Частково-коректні алгоритми підстановки термів переводять заданий терм завжди в семантично еквівалентний терм.

Приклад 3.10. Усі правила, наведені в попередніх розділах, є частково-коректними стосовно класичної інтерпретації. Для доведення часткової коректності правила

add (succ (x), y) → succ (add (x, y))

треба показати, що для всіх конкретизацій B справедливо

(B(x) + 1) + B(y) = (B(x) + B(y)) + 1.

Справедливість цієї рівності отримується безпосередньо з комутативності й асоціативності додавання. Часткова коректність інших правил може бути легко показана аналогічно.

Якщо алгоритм підстановки термів частково-коректний, то інформація, задана в обчисленнях, завжди зберігається. СПТ R називається **цілком коректною** по відношенню до A, якщо для основного терма t з $t^A = \perp$ не існує ніяких нескінченних обчислень, і для термінальних відносно R різних основних термів t_1, t_2 з $t_1^A \neq \perp$, $t_2^A \neq \perp$ завжди має місце

$$t_1^A \neq t_2^A$$

Це означає, що основний терм t з $t^A \neq \perp$ через R за кінцеве число кроків переводиться в однозначну нормальну форму. Нормальна форма задається через множину термінальних для R основних термів. У цілком коректних СПТ для основних термів t з $t^A \neq \perp$ приймається лише, що вони за допомогою правил з R переводяться в семантично еквівалентні терми. Обчислення для таких термів можуть як завершуватися (з результатом r з $r^A = \perp$), так і не завершуватися.

Часткова коректність є властивість окремих правил і тим самим може показуватися для окремих правил незалежно. Повна коректність є властивість усієї СПТ.

Приклад 3.11.. СПТ для основного терма t з інтерпретацією $\neq \perp$ за допомогою множини термінальних основних термів однозначно визначає нормальну форму. Ці однозначні нормальні форми утворять основні терми виду

$succ^n$ (zero) з $n \in N$, причому

$succ^0$ (zero) = zero,

$succ^{n+1}$ (zero) = succ ($succ^n$ (zero)).

Якщо основний терм t не в нормальній формі, то він містить символ функції з множини {pred, add, mult}. Тоді в t існує також підтерм виду

pred (t_0), add(t_l,t_2) або mult (t_l, t_2),

причому t_0, відповідно t_l і t_2 - основні терми в нормальній формі. Тоді (за винятком t_0 = zero) завжди застосовується одне з правил. Отже, терм t або не в заданій нормальній формі, або справедливо $t^{NAT} = \perp$..

Тепер залишається показати, що для СПТ не існує обчислень, що не завершуються, якщо для заданого терму t справедливо $t^{NAT} \neq \perp$. Для цього визначимо відображення, що кожному основному терму зіставляє обчислення (вагу):

$g: \{t \in \mathbf{W_\Sigma: t^{NAT} \neq \perp}\} \rightarrow N$

g("zero") = O,

g("succ(t)") = g(t),

g("pred(t)") = g(t) + 1,

g("add(tl, t2)") = I^{NAT} [tl]+ g(tl) + g(t2) +1,

g("mult(tl, t2)") = (I^{NAT} [t1]+ 2 * g(tl) + 2) * (I^{NAT} [t2] + 2 * g(t2) + 2).

Тепер для кожного застосування правила $t \rightarrow r$ розглянутої СПТ можна сказати , що справедливо

$$g(t) > g(r)$$

Звідси виходить , що до кожному основному терму можливо якнайбільше g(t) застосувань правил.

Описані в наведеному вище прикладі методи для доведення того, що СПТ завершуються (терміністичні), демонструють загальний принцип доведення для терміністичності СПТ. Нехай R є СПТ над сигнатурою Σ і

$$g : W_\Sigma \rightarrow N$$

є відображення, що кожному основному терму ставить у відповідність вага. Якщо для кожного повного правила застосування $t \rightarrow r$ справедлива умова $g(t)>g(r)$, то завершується кожне обчислення в R.

Контрольні запитання

1. Що називають основними термами?
2. Що таке інфіксна, префіксна та постфіксна форма запису функції?
3. Що називають інтерпретацією терму в множину А?
4. Що таке схема для основного терму? Приведіть приклади.
5. Що таке терми з вільними ідентифікаторами? Особливості відображення. Схеми.
6. Правила підстановки термів з вільними ідентифікаторами.
7. Що таке екземпляр правила ПТ?
8. Що таке система підстановки термів?
9. Що таке терм в нормальній формі?
10. Що називають частково коректною СПТ?
11. Яка СПТ є цілком коректною?

ЛЕКЦІЯ 4. АЛГОРИТМИ ТА ОБЧИСЛЮВАЛЬНІ ФУНКЦІЇ

Основні означення. Оператор суперпозиції. Оператор примітивної рекурсії. Оператор мінімізації.

4.1. Основні означення

Період до початку XX століття можна вважати етапом накопичування інформації про алгоритми. Інтуїтивне розуміння алгоритму було достатнім, щоб вважати ту чи іншу процедуру розв'язання задач алгоритмом. Проте одночасно нагромаджувалися проблеми, пошук алгоритму розв'язання яких ні до чого не приводив. Та для того щоб довести, що алгоритм існує, достатньо його показати, а ось для доведення його відсутності необхідне вже строге визначення шуканого об'єкта. Цим питанням і потрібно було зайнятися.

Оскільки в математиці поняття алгоритму тісно пов'язане з поняттям функції, то історично першою формалізацією алгоритму став клас обчислювальних функцій (К. Гьодель, А.Черч, 1935–1936 рр.). Основна ідея в тому, що довільний алгоритм можна звести до обчислення значення деякої числової функції, тобто з кожним алгоритмом можна зв'язати функцію, яку він обчислює. При цьому виникає ряд запитань: чи для будь-якої функції існує обчислювальний її алгоритм; для яких функцій алгоритми існують і як описати такі алгоритмічні функції? Пошук відповіді на ці питання й привів до створення теорії рекурсивних функцій.

При цьому необхідно зазначити, що в теорії обчислювальних функцій визначають множину натуральних чисел $N = \{0, 1, 2, 3,\ldots\}$ і розглядають тільки числові функції, розуміючи під ними функції k змінних (k – місні функції), аргументи і значення яких належать N. Тобто об'єкти з областю визначення $D_f \subseteq N^k$ (k – ціле додатне) та з областю значень $R_f \subseteq N$ будемо називати κ - місними частковими функціями. Термін "часткова" свідчить про те, що функція визначена на підмножині N^k (звичайно, може статися, що $D_f = N^k$, в такому разі функція стає всюди визначеною). Виходячи з вищезазначеного, дамо означення обчислювальної функції.

Означення 4.1.1. Числову функцію $f: N^k \to N$ називають обчислювальною, якщо існує алгоритм, за допомогою якого можна обчислити значення функції для будь-якого набору значень аргументів із області визначення функції.

Надалі використовуватиме ідею К. Гьоделя та С. Кліні (1936 р.), за якою всі обчислювальні функції можна одержати із множини базисних функцій та алгебраїчних операцій. Самі операції прийнято називати операторами.

Розглянемо клас числових функцій, що використовуються **як базис для побудови обчислювальних функцій:**

1. $O(x)=0$ – нуль-функція (можна задати і n - місну нуль - функцію $O^n(x_1, x_2, \ldots, x_n) = 0$).
2. $S(x)=x+1$ – функція слідування або наступності (але не додавання одиниці).
3. $I^n_m\ (x_1, x_2, x_3,.., x_m, \ldots x_n) = x_m$ – функція проекції або введення фіктивних змінних або вибору аргументу.

Як оператори, застосування яких до базисних функцій приводить до утворення нових функцій, оберемо такі три оператори:

1) **оператор суперпозиції;**
2) **оператор примітивної рекурсії;**
3) **оператор мінімізації або найменшого кореня.**

У цьому розділі застосуємо алгебраїчний підхід до визначення класу обчислюваних функцій. Кожну обчислювальну функцію будемо одержувати з деяких найпростіших обчислювальних базисних функцій за допомогою операцій, обчислювальність яких також не викликає сумніву. Загальна схема знаходження, яка дала назву цьому підходу, – рекурсія, спосіб задання функції шляхом визначення кожного її значення в термінах раніше визначених її значень та інших уже визначених функцій.

Таким чином, нетривіальні обчислювальні функції можна одержати за допомогою композиції (суперпозиції) вже відомих обчислювальних функцій. Цей спосіб є явно алгоритмічним.

4.2. Оператор суперпозиції

Операція суперпозиції полягає у підстановці одних арифметичних функцій замість аргументів інших функцій.

Нехай задана m-місна функція $F(x_1, x_2, x_3,.., x_m)$ та
m-на кількість n-місних функцій $f_1(x_1, x_2, x_3,.., x_n)$, $f_2(x_1, x_2, x_3,.., x_n)$, $f_3(x_1, x_2, x_3,.., x_n)$,...,$f_m(x_1, x_2, x_3,.., x_n)$. Тоді говорять, що n-місна функція $\varphi(x_1, x_2, x_3,.., x_n)$ утворилася внаслідок підстановки у функцію F замість її аргументів m функцій $f_1, f_2, f_3,..., f_m$. Така підстановка називається суперпозицією S^n_m. Тоді

$S^n_m(F, f_1, f_2, ..., f_m) = F(f_1(x_1, x_2, .., x_n), f_2(x_1, x_2, .., x_n),...,f_m(x_1, x_2, .., x_n)) = \varphi(x_1, x_2, .., x_n)$.

Приклад 4.2.1.. Здійснити суперпозицію нуль-функції та функції слідування, тобто знайти $S^2(S(x); O(x))$.

Розв'язання. Для визначення результату операції суперпозиції потрібно функцію $O(x) = 0$ підставити в $S(x) = x+1$ замість значення аргументу. Отримаємо $S^2(S(x);O(x)) = S(O(x)) = O(x) + 1 = 0+1 = 1$.

4.3. Оператор примітивної рекурсії

Оператор примітивної рекурсії (R^n) дозволяє будувати (n+1)-місну арифметичну функцію $f(x_1, x_2, x_3,.., x_n, y)$ з двох заданих функцій, одна з яких є n-місна $\varphi(x_1,x_2,..,x_n)$, а інша – n+2-місна функція $\psi(x_1,x_2,..,x_n,y,z)$ за такою схемою: $f(x_1, x_2, x_3,.., x_n,0) = \varphi(x_1,x_2,..,x_n)$;

$f(x_1, x_2, x_3,.., x_n,y+1) = \psi(x_1,x_2,..,x_n,y, f(x_1, x_2, x_3,.., x_n, y))$. (4.3.1)

Таким чином, $f(x_1, x_2, x_3,.., x_n, y) = R^n(\varphi,\psi)$.

Для розуміння операції примітивної рекурсії необхідно зазначити, що будь-яку функцію від меншої кількості аргументів можна розглядати як функцію від більшої кількості аргументів. Зокрема, функції - константи ($n = 0$) є одномісними і відповідно: $f(x)=a$; $f(x, y+1) = \psi(x,y, f(y))$, де a – константа.

Схеми примітивної рекурсії визначають функцію f тільки через інші функції φ та ψ, а й через значення f у попередніх точках – значення f у точці (y+1) залежить від значення f у точці (y).

Приклад 4.3.1. Знайти значення функції $f(3,2)$, якщо вона задана співвідношеннями

$f(x, 0) = 0$; $f(x, y+1) = f(x, y) + x$.

Розв'язання. У даному разі за схемою (4.3.1) маємо : $f(x, 0) = \varphi(x) = 0$, $\psi(x,y,z) = y + z$. Виходячи з того, що

$f(x, 0) = \varphi(x) = 0$ при будь-якому x, тоді й $f(2, 0) = 0$. Обчислюючи послідовно, одержимо:

$f(2, 1) = f(2 ,0) + 2 = 0+ 2 = 2$;

$f(2, 2) = f(2, 1) + 2 = 2+ 2 = 4$;

$f(3, 2) = f(2, 2) + 2 = 4+ 2 = 6$, що є остаточною відповіддю. Нескладно довести, що в даному прикладі $f(x,y) = x \cdot y$.

4.4. Оператор мінімізації

Розглянемо обчислювальну (n+1) -місну функцію $f(x_1, x_2, x_3,.., x_n, x_{n+1})$. Зафіксуємо значення $x_1, x_2, x_3,.., x_n$ її перших n аргументів та розглянемо рівняння $f(x_1, x_2, x_3,.., x_n, y) = 0$, тобто знайдемо значення, y при якому функція дорівнює нулю. Більш складною буде задача відшукати найменше з усіх значень y, при якому $f(x_1, x_2, x_3,.., x_n, y) = 0$. Оскільки результат знаходження залежить від $x_1, x_2, ...,x_n$, то й найменше значення y є також їх функцією. Введемо позначення

$\varphi(x_1, x_2, ...,x_n) = \mu_y [f(x_1, x_2, ...,x_n, y) = 0]$, (4.4.1)

де y таке, що $f(x_1, x_2, ...,x_n, y) = 0$, а μ_y – оператор мінімізації.

Для знаходження функції $\varphi(x_1, x_2, ...,x_n)$ можна запропонувати таку процедуру.

1. Обчислюємо $f(x_1,...x_n,0)$; якщо її значення буде нуль, то $\varphi(x_1,...x_n)=0$. Якщо $f(x_1, ...x_n,0) \neq 0$, то переходимо до наступного кроку.

2. Обчислюємо $f(x_1,...x_n, 1)$; якщо її значення буде нуль, то $\varphi(x_1,...x_n)= 1$. Якщо $f(x_1, ...x_n,0) \neq 0$, то переходимо до наступного кроку.

І так далі, поки не знайдемо перше значення у, при якому $f(x_1, ...x_n, y) = 0$.

Якщо визначиться, що для всіх у $f(x_1, ...x_n, y) \neq 0$, то функція $\varphi(x_1,...x_n)$ вважається невизначеною.

Приклад 4.4.1. Розглянемо функцію $\varphi(x,y) = x - y$, яку можна отримати за допомогою μ-оператора $d(x,y) =$

$= \mu_z[y+z = x] = \mu_z[S((I^3_2 (x ,y, z), I^3_3 (x, y, z),)] = I^3_1 (x, y, z)$ та обчислимо $f (7, 2)$.

Розв'язання. Задамо y значення 2 та встановимо змінній z послідовно значення 0, 1, 2.., кожного разу обчислюючи суму $y + z$. Як тільки при якомусь першому в заданому порядку z сума дорівнюватиме 7, то відповідне значення візьмемо за значення $d(7,2)$. Проведемо обчислення:

$z = 0$ $\quad 2 + 0 = 2 < > 7$
$z = 1$ $\quad 2 + 1 = 3 < > 7$
$z = 2$ $\quad 2 + 2 = 4 < > 7$
$z = 3$ $\quad 2 + 3 = 5 < > 7$
$z = 4$ $\quad 2 + 4 = 6 < > 7$
$z = 5$ $\quad 2 + 5 = 7 = 7$

Таким чином, $d (7, 2) = 5$.

4.5. Гіпотеза Черча та примітивно рекурсивні функції

Означення 4.5.1. Функцію називають примітивно рекурсивною, якщо вона утворена з найпростіших функцій за допомогою скінченного числа застосувань операторів суперпозиції S^n_m та примітивної рекурсії R^n.

Означення 4.5.2. Функцію називають частково рекурсивною, якщо вона утворена з найпростіших функцій за допомогою скінченного числа застосувань операторів суперпозиції S^n_m, примітивної рекурсії R^n та мінімізації μ_y.

З цих означень та зауваження щодо зберігання операторами властивості обчислювальності функцій випливає, що кожна стандартно задана частково рекурсивна функція є обчислювальною за певною процедурою, яка відповідає інтуїтивному уявленню алгоритму, а з іншого боку – які б досі не будувалися класи точно визначених алгоритмів, завжди з'ясовувалося, що числові функції, які обчислювалися за алгоритмами цих класів, були частково рекурсивними. Тому загальноприйнятою є така наукова гіпотеза (теза Черча).

Гіпотеза (теза) Черча. *Клас алгоритмічно обчислювальних часткових числових функцій збігається з класом усіх частково рекурсивних функцій.*

До формулювання цієї тези входить інтуїтивне поняття обчислювальності, тому його не можна ні спростувати, ні довести. Це факт, на користь якого свідчить багаторічна математична практика.

Історично це була перша гіпотеза щодо зв'язків між класами інтуїтивних і точних алгоритмів.

Зрозуміло, що будь-яка примітивно рекурсивна функція є частково рекурсивною, оскільки за визначенням оператори для побудови частково рекурсивних функцій містять у собі оператори для побудови примітивно рекурсивних функцій. Також зрозуміло, що примітивно рекурсивна функція визначена скрізь і тому є загально рекурсивною функцією (у примітивно рекурсивної функції немає приводу «*зависати*», тому що при її побудові використовуються оператори, що визначають скрізь визначені функції).

Досить складно довести існування і навести приклад загально рекурсивної функції, що не є примітивно рекурсивною.

Функція Аккермана – приклад такої обчислювальної функції, яка не є примітивно рекурсивною. Вона набуває два невід'ємних цілих числа як параметри і повертає натуральне

число, позначається *A(m;n).* Ця функція зростає дуже швидко, наприклад, число *A* (4;4) настільки велике, що кількість цифр у порядку цього числа у багато разів перевершує кількість атомів в спостережуваної частини Всесвіту. Функція Аккермана визначається рекурсивно для невід'ємних цілих чисел m та n таким чином:

$$A(m,n)=\begin{cases} n+1, & m=0; \\ A(m-1,1), & m>0, n=0; \\ A(m-1,A(m,n-1)), & m>0, n>0 \end{cases} \qquad (4.5.1)$$

За означенням примітивно рекурсивної функції, наведеним вище, неважко знайти процедуру, що породжує всі примітивно рекурсивні функції.

Надамо цьому означенню більш формального індуктивного вигляду.

1. Функції $O(x) = 0$, $S(x) = x + 1$, $I^n_m(x_1, x_2, \ldots, x_m, \ldots, x_n) = x$ для всіх натуральних n, m є примітивно рекурсивними.

2. Якщо $g_1(x_1,\ldots,x_n)$, …, $g_m(x_1,\ldots,x_n)$, $h(x_1,\ldots,x_m)$ – примітивно рекурсивні функції, то $S^n_m(h,g_1,\ldots,g_m)$ – примітивно рекурсивні функції для будь-яких натуральних n,m.

3. Якщо $g(x_1,\ldots,x_n)$, та $h(x_1,\ldots,x_n,y,z)$ – примітивно рекурсивні функції, то $R^n(g,h)$ – примітивно рекурсивна функція.

4. Інших примітивно рекурсивних функцій немає.

Приклад 4.5.1. Навести приклади арифметичних функцій, що є примітивно рекурсивними.

Розв'язання. **Функція додавання** двох натуральних чисел $Sum(a,b) = a + b$. Ця функція може бути розглянута як примітивно рекурсивна функція двох змінних, одержувана внаслідок застосування оператора примітивної рекурсії до функцій I^1_1 та F, другу з яких одержимо підстановкою функції проекції I^3_3 у функцію слідування S. Покажемо це математично:

$Sum(x,0) = I^1_1(x)$,

$Sum(x,y+1) = x + y + 1 = Sum(x,y) + 1 = S(Sum(x,y))$,

$Sum(x, y) = R^1(I^1_1(x), F(x, y, z))$, де $F(x, y, z) = S(I^3_3(x, y, z)) = z + 1$.

Функція множення *двох натуральних чисел*

$Mul(a, b) = a \bullet b$.

Ця функція може бути розглянута як примітивно рекурсивна функція двох змінних, одержувана внаслідок застосування оператора примітивної рекурсії до функцій O та G, другу з яких одержимо підстановкою функції проекції I^1_3 та I^3_3 у функцію додавання Sum. Покажемо це математично.

$Mul(x, 0) = O(x)$,

$Mul(x,y+1) = x \times y + x = G(x, y, Mul(x, y))$,

$G(x, y, z) = Sum(I^1_3(x, y, z), I^3_3(x, y, z))$,

$Mul(x, y) = R^1(O(x), G(x, y, z))$.

Контрольні запитання

1. Сформулюйте основну ідею про спосіб побудови обчислювальних функцій.
2. Які функції належать до базисних функцій?
3. Назвіть основні оператори, що застосовуються для породження класу обчислювальних функцій?
4. Яка функція називається частково рекурсивною?
5. Яка гіпотеза є історично першою щодо зв'язків між класами інтуїтивних та точних алгоритмів?
6. Подайте за допомогою діаграм Ейлера-Венна співвідношення між класами обчислювальних та частково-рекурсивних функцій.

ЛЕКЦІЯ 5. АЛГОРИТМІЧНІ МОДЕЛІ

*Основні визначення. Машини Тюрінга. Нормальні алгоритми Маркова***.**

5.1. Основні визначення

Точне визначення класу рекурсивних функцій разом із тезою Черча надає один із можливих варіантів уточнення поняття алгоритму. Проте це уточнення в цілому не пряме, оскільки поняття обчислювальної функції є вторинним по відношенню до поняття алгоритму. Виникає питання, чи не можна уточнити безпосередньо саме поняття алгоритму, а вже потім за його допомогою визначити точно клас обчислювальних функцій?

Основна ідея полягала в тому, що алгоритмічні процеси – це процеси, які може здійснювати відповідним чином влаштована "машина". У зв'язку з розвитком сучасної обчислювальної техніки така алгоритмічна модель становить особливий інтерес, тому що в ній поняття алгоритму базується на командно-адресному принципі. Для наукового аналізу обчислювальних процесів, які можуть бути реалізовані машиною, бажано було знайти просту за своєю логічною структурою схему алгоритмічної машини, яка може бути предметом точної математичної теорії.

Відповідно до цієї думки вчені створили в точних математичних термінах "машинні моделі" (математичні абстракції), на яких вдалось імітувати дію алгоритмів. Це було зроблено Постом і Тюрінгом у 1936–1937 рр. незалежно один від одного і майже одночасно із працями Черча. Пізніше, у 50-х роках двадцятого століття, до їх приєднався А. А. Марков. Алгоритмічні моделі цих вчених є претендентами на право бути основними формалізаціями поняття "алгоритм". Це означає, що вони повинні бути універсальними, тобто в їх рамках можна описати будь-який алгоритм. Це дійсно так, тому що, по-перше, можна довести зведення однієї моделі до іншої, тобто будь-який алгоритм, описаний засобами однієї моделі, може бути описаний засобами іншої; по-друге, завдяки взаємному зведенню моделей у теорії алгоритмів удалося виробити інваріантну по відношенню до моделей систему понять, що дозволяє вести мову про властивості алгоритмів незалежно від того, яка формалізація обрана. Ця система понять базується на понятті обчислювальної функції, тобто для обчислення якої існує алгоритм.

Було доведено, що клас функцій, обчислюваних на цих машинах, збігається із класом рекурсивних функцій. Таким чином, це є ще одним підтвердженням тези Черча.

5.2. Машини Тюрінга

Перший важливий і достатньо широкий клас алгоритмів був уведений А. Тюрінгом та Е. Постом у 1936–1937 рр. Алгоритми цього класу здійснюються особливими машинами, що називаються зараз машинами Тюрінга-Поста, або просто машинами Тюрінга. Машини Тюрінга копіюють в істотних рисах роботу людини і часто розглядаються як математичні моделі для вивчення функціонування людського мозку.

Машина Тюрінга – це математична модель, яка породжує обчислювальні процеси. Ідея машини Тюрінга базується на загальному аналізі процесів обчислення значень функцій обчислювачем. При цьому спостерігається **основна гіпотеза теорії алгоритмів (теза Тюрінга) : будь-який алгоритм може бути реалізований у машині Тюрінга.** Теза має сенс у тому, що кожна функція, для якої складений алгоритм знаходження її значень, представлена машиною Тюрінга, тобто є обчислювальною.

Зафіксуємо два скінченних алфавіти –

$A = \{a_0, a_1, \ldots, a_n\}$, де $n \geq 0$, і $Q = \{q_0, q_1, \ldots, q_m\}$, де $m \geq 1$. При цьому A будемо називати зовнішнім алфавітом, а Q – внутрішнім алфавітом, або алфавітом станів. Додатково поставимо вимогу, щоб $A \cap Q = \varnothing$ і символи $\rightarrow$, L, R не належать $A \cup Q$. Один символ з A називають порожнім, зазвичай його позначають Λ. Усі інші букви з A називають не порожніми.

Візьмемо об'єднаний алфавіт $B = A \cup Q \{\rightarrow , L, R\}$. Виділимо серед слів алфавіту B команди, які представляють слова одного з трьох видів:

1) $q_i a_j \rightarrow q_k a_l$; 2) $q_i a_j \rightarrow q_k L$; 3) $q_i a_j \rightarrow q_k R$;

де $1 \le i \le m$, $0 \le j \le n$, $0 \le k \le m$, $0 \le l \le n$.

Ці команди читаються відповідно так:

1) перебуваючи у стані q_i і спостерігаючи символ a_j, перейти у стан q_k і написати символ a_l;

2) перебуваючи в стані q_i і спостерігаючи символ a_j, перейти у стан q_k і перемістится вліво на один символ ;

3) перебуваючи в стані q_i і спостерігаючи символ a_j, перейти у стан q_k і перемістится вправо на один символ .

Скінченна послідовність команд становить програму P із зовнішнім алфавітом A і внутрішнім алфавітом Q. У кожній команді програми виділимо її ліву частину – це підслово до символа "→" і праву частину – підслово після символа "→".

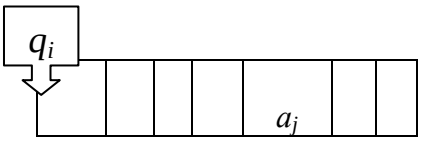
Означення 5.2.1. Машиною Тюрінга (МТ) називають упорядковану шістку $\{ A, Q, a_0, q_0, q_1, P \}$, яка задовольняє такі умови:

1) множини A і Q скінченні, не перетинаються і не містять символів $\rightarrow , L, R;$

2) $a_0 \in A$, $q_0 \in Q$, $q_1 \in Q$. При цьому a_0 називається символом порожньої комірки, q_1 – початковий стан машини, q_0 – стан, у якому машина зупиняється;

3) P – програма із зовнішнім алфавітом A і внутрішнім алфавітом Q, причому *а)* програма не містить двох різних команд з однаковими лівими частинами; *б)* будь-яка з команд не починається символом q_0.

Отже, МТ – це алгоритмічна модель, а не фізична машина. Проте для кращого розуміння роботи цієї машини її можна уявити у вигляді автоматичного пристрою (див. рисунок).



Пристрій спостерігає стрічку, поділену на комірки, яку можна уявити необмеженою в обох напрямках. У кожний конкретний момент часу пристрій знаходиться в деякому стані q_i, спостерігаючи символ a_j, записаний в одній із комірок. Залежно від того, в якому стані перебуває пристрій та який символ він спостерігає, пристрій здійснює дію, яка може полягати в тому, що пристрій на місце a_j запише a_l, а сам перейде в новий стан q_k *(виконується команда першого виду),* або переміститься вліво на одну комірку і змінить свій стан на q_k *(виконується команда другого виду),* або переміститься вправо на одну комірку і змінить стан на q_k *(виконується команда третього виду).*

Охарактеризуємо тепер роботу машини Тюрінга більш детально.

Означення 5.2.2. Машинним словом, або конфігурацією, називається будь-яке слово в алфавіті $A \cup \{ q_i \}$, для якого $q_i \in Q$, причому символ q_i входить у це слово один раз і не на останньому місці.

Приклад 5.2.1 Нехай задані алфавіти $A = \{a_0, 1\}$, $Q = \{q_0, q_1, q_2\}$.

Тоді слова $a_0 1 a_0 a_0 q_1 a_0$, $a_0 1 q_2 1$, $q_0 111 a_0$ є конфігураціями, а слова $a_0 111 q_1$, $q_1 1 a_0 q_2 1 a_0$, $a_0 q_1 q_1 1$ – до конфігурацій не належать.

У процесі роботи МТ здійснюється перехід від одного машинного слова до іншого. При цьому відбувається певне перетворення слів алфавіту A. Будемо говорити, що МТ перетворює машинне слово α в машинне слово γ (записують МТ(α)= γ), якщо існує скінченна послідовність слів $\gamma_0, \gamma_1, \ldots, \gamma_n$, для яких виконуються умови :

1) $\gamma_0 = \alpha$, $\gamma_n = \gamma$;

2) слово γ_{i+1} одержане з γ_i дією МТ в один крок за визначеним правилом.

МТ може перетворити задане слово тільки в одне слово. Якщо МТ не перетворює слово α ні в яке слово, то говорять, що ця машина не застосовна до слова α, або значення $MT(\alpha)$ не визначене.

Приклад 5.2.2. Знайти результат застосування МТ із зовнішнім алфавітом $\{a_0, 1,2\}$, внутрішнім алфавітом $\{q_0,q_1,q_2,q_3,q_4,q_5\}$ і програмою $q_1 1 \rightarrow q_2 R$, $q_2 2 \rightarrow q_2 1$, $q_2 1 \rightarrow q_3 R$, $q_3 2 \rightarrow q_3 R$, $q_3 1 \rightarrow q_4 2$, $q_4 2 \rightarrow q_4 R$, $q_4 a_0 \rightarrow q_5 L$, $q_5 2 \rightarrow q_5 L$, $q_5 1 \rightarrow q_5 L$, $q_5 a_0 \rightarrow q_0 R$ до слова 1221.

Розв'язання. Виходячи з початкової конфігурації $q_1 1221$, знайдемо кінцеву конфігурацію, якщо вона існує. Маємо перетворення МТ за заданою програмою:

$q_1 1221 \to 1 q_2 221 \to 1 q_2 121 \to 11 q_3 21 \to 112 q_3 1 \to 112 q_4 2 \to$

$\to 1122 q_4 a_0 \to 112 q_5 2 \to 11 q_5 22 \to 1 q_5 122 \to q_5 1122 \to$

$\to q_5 a_0 1122 \to q_0 1122.$

Отже, слово 1221 МТ перетворює в слово 1122.

Значно складнішою, ніж задача застосування конкретних машин до конкретних слів, є задача створення МТ із наперед заданими властивостями.

Важливим класом МТ є машини, які обчислюють значення тих чи інших числових функцій для будь-якого набору значень аргументів із області визначення функції.

Нагадаємо, що числовою називають функцію $f(x_1,x_2,\ldots,x_n)$, значеннями якої та значеннями її аргументів є невід'ємні цілі числа. Розглянемо часткові числові функції, визначені загалом не для всіх значень аргументів.

Означення 5.2.3. Функція називається обчислювальною за Тюрінгом, якщо існує машина Тюрінга, яка обчислює значення цієї функції для будь-якого набору значень аргументів із області визначення функції і не застосовна до наборів значень аргументів, що не входять до області визначення цієї функції.

Для обчислення числових функцій на МТ часто використовують спеціальне кодування чисел. Наприклад, невід'ємне ціле число m можна позначити словом (задати набором) з $(m+1)$ одиниць, або скорочено 1^{m+1}. Тобто 0 задають як 1, 1 – як 11, 2 – як 111 тощо. Символом порожньої комірки буде слово 0. Тоді для побудови МТ буде достатньо зовнішнього алфавіту $A = \{0, 1\}$.

5.3. Нормальні алгоритми Маркова

Розглянемо ще один підхід до уточнення поняття алгоритму, запропонований російським математиком А. А. Марковим на початку 1950-х років. Досвід вивчення і застосування математики показує, що всі відомі алгоритми можна розбити на найпростіші кроки — елементарні операції. Як елементарну операцію, на базі якої побудовано нормальні алгоритми, А. А. Марков запропонував застосувати підстановку одного слова замість іншого.

Нормальні алгоритми Маркова (НАМ) перетворюють рядки, задані у будь-якому скінченному алфавіті , у рядки у тому самому алфавіті.

Перейдемо до точних означень. Алфавітом будемо називати непорожню скінченну множину E деяких символів. Символи "→ " та "• " не повинні належати алфавіту E. Елементи алфавіту називатимемо також буквами. *Слово* в алфавіті E — це скінченна, або порожня, послідовність його букв. Порожнє слово позначимо Λ . Множину всіх слів в алфавіті E позначимо через E^*. Нехай $P, Q \in E^*$, тоді вирази $P \to Q$, $P \to \bullet Q$ називаються відповідно формулами простої та заключної підстановки. При цьому перша з них означає, що замість P потрібно вставити слово Q та перейти до наступної підстановки, а в другій формулі після підстановки процес закінчується.

Нехай $P \to (\bullet) Q$ означає будь-яку з формул підстановки (просту чи заключну).

Нормальний алгоритм в алфавіті E вважається заданим, якщо задана скінченна схема (таблиця) формул підстановок слів алфавіту $\boldsymbol{E}$:

$$S = \begin{cases} P_1 \to (\bullet) Q_1 \\ P_2 \to (\bullet) Q_2 \\ \ldots\ldots\ldots\ldots \\ P_n \to (\bullet) Q_n \end{cases} \qquad (5.3.1)$$

Означення 5.3.1. Нормальним алгоритмом Маркова (НАМ) в алфавіті Е називають пару$\{\boldsymbol{E,S}\}$, що складається з алфавіту $\boldsymbol{E}$ та схеми $\boldsymbol{S}$ в цьому алфавіті.

Роботу нормального алгоритму Маркова можна описати у такий спосіб. Нехай задане слово $\alpha \in \boldsymbol{E}^*$. Знаходимо першу в схемі $\boldsymbol{S}$ таку формулу підстановки $\alpha_i \to (\bullet) \beta_i$, що α_i є

підсловом α. Підставляємо в слово α слово β_i замість першого входження α_i в α. Нехай γ_i – результат цієї підстановки. Якщо формула підстановки виявилася заключною, тобто $\alpha_i \to \bullet\beta_i$, то робота алгоритму закінчується і $A(\alpha) = \gamma_i$. Якщо формула підстановки виявилася простою, тобто $\alpha_i \to \beta_i$, то до слова γ_i застосовуємо той самий пошук, який застосовувався до слова α і так далі. Якщо врешті-решт одержимо таке слово γ_i, що жодне із слів $\alpha_1, \alpha_2, \ldots \alpha_n$ не входить у γ_i як підслово, то робота алгоритму закінчується і $A(\alpha) = \gamma_i$. Якщо описаний процес не закінчується ніколи, то говоримо, що алгоритм А не застосовний до слова α.

Приклад 5.3.1. Розглянемо алгоритм, заданий у алфавіті $E=\{a,b\}$ схемою підстановок :

$$S = \begin{cases} ab \to bb, \\ aaa \to \bullet a, \\ ba \to aa. \end{cases} \qquad (5.3.1)$$

Застосуємо його дію до слова $X = abba$.

Першою підстановкою зі слова "$\underline{ab}ba$" одержимо слово $X_1 = bbba$. Далі перша і друга підстановки до слова X_1 не діють, за третьою підстановкою зі слова "$bb\underline{ba}$" одержимо слово $X_2 = b\underline{ba}a$, до якого застосовна лише третя підстановка, що перетворює його у слово $X_3 = baaa$. До слова X_3 застосовні друга і третя підстановки, причому спочатку повинна виконуватися друга підстановка, але оскільки вона є заключною, то після її дії процес перетворення слів закінчується. Отже, слово «$b\underline{aaa}$» переходить у слово $X_4 = =ba$. Таким чином, $A(abba) = ba$.

Означення нормального алгоритму Маркова, на перший погляд, не свідчить про якусь універсальність цього поняття. Проте виявляється, що клас нормальних алгоритмів має досить широкі можливості, зокрема під час обчислення значень функцій.

Означення 5.3.2. *Функція називається нормально обчислювальною, якщо існує такий нормальний алгоритм, який обчислює значення цієї функції для будь-якого набору значень аргументів із області визначення функції і не застосовний до наборів значень аргументів, що не входять до області визначення цієї функції.*

Постає питання про клас функцій, які можна обчислювати за допомогою нормальних алгоритмів. З цього приводу А. А. Марковим була сформульована гіпотеза, що одержала назву принципу нормалізації Маркова.

Принцип нормалізації Маркова. Клас нормально обчислювальних функцій збігається із класом обчислювальних функцій.

Контрольні запитання

1. Що називається машиною Тюрінга?
2. Що називається нормальним алгоритмом Маркова?
3. Яка функція називається обчислювальною за Тюрінгом?
4. Сформулюйте гіпотезу Тюрінга.
5. Яка функція називається нормально обчислювальною?

ЛЕКЦІЯ 6. СКЛАДНІСТЬ АЛГОРИТМІВ

*Асимптотичні оцінки складності. Класи складності***.**

Створення та реалізація алгоритму відповідно до свого призначення визначає його складність. Проте не існує інтегрованого показника складності алгоритму, хоча існує спеціальний навіть розділ – метрична теорія алгоритмів, що займається саме проблемами складності. Інтуїтивно можна виділити такі основні складові складності алгоритму:

1. Логічна складність – кількість людино-місяців, витрачених на створення алгоритму.
2. Статична складність – довжина опису алгоритмів (кількість операторів).
3. Тимчасова складність – час виконання алгоритму.
4. Ємнісна складність – кількість умовних одиниць пам'яті, необхідних для роботи алгоритму.

Головною метою теорії складності є забезпечення механізму класифікації алгоритмів за складністю. Складність алгоритму дозволяє визначитися з вибором ефективного алгоритму серед існуючих, що побудовані для розв'язання конкретної проблеми. А саме вибір серед уже існуючих алгоритмів дозволяє не розглядати логічну та статичну складність, а оцінювати ті ресурси, що знадобляться під час реалізації обраних алгоритмів.

Означення. Складність алгоритму – *це кількісна характеристика, що відображує використані алгоритмом ресурси під час свого виконання.*

Основними ресурсами, що оцінюються, є час виконання і простір пам'яті.

Інтуїтивно це поняття досить зрозуміле. В алгоритму є вхід – опис завдання, яке потрібно вирішити. На його розв'язання алгоритм витрачає певний час (тобто кількість операцій). Складність – це функція від довжини входу, значення якої дорівнює максимальному (за будь-якими входами даної довжини) кількості операцій, необхідних алгоритму для отримання відповіді.

Приклад 6.1. Нехай дана послідовність з нулів та одиниць і нам потрібно з'ясувати, чи є там хоч одна одиниця. Яку складність матиме алгоритм розв'язання цієї задачі?

Розв'язання. Нехай n – кількість символів у послідовності. Алгоритм буде послідовно перевіряти, чи немає одиниці в поточному місці заданої послідовності, а потім рухатися далі, поки вхід не скінчиться. Оскільки одиниця дійсно може бути тільки одна, для отримання точної відповіді на це питання в гіршому випадку доведеться перевірити всі n символів входу. В результаті отримуємо складність порядку cn, де c – кількість кроків, що потрібна для перевірки поточного символа і переходу до наступного. Оскільки такого роду константи сильно залежать від конкретної реалізації, математичного сенсу вони не мають, і їх зазвичай приховують за символом O (О – велике) : в даному випадку фахівець із теорії складності визначив би, що алгоритм має складність $O(n)$ (про символіку складності див. 14.1); іншими словами, він лінійний.

Існує і продовжує розширюватися клас варіантів поняття складності: складність знизу, зверху й у середньому, алгебраїчна складність, мультиплікативна складність, бітова складність, фундаментальні асимптотичні оцінки складності, оцінка алгоритмів за їх належністю до класів складності самих проблем, що вони розв'язують, і т. д.

6.1. Асимптотичні оцінки складності

Одним зі спрощених видів аналізу складності алгоритмів, що використовують при комп'ютерній їх реалізації, є асимптотичний аналіз алгоритмів. Він використовується з метою порівняння витрат часу та інших ресурсів різноманітними алгоритмами, призначеними для розв'язання одного і того самого завдання. Досліджуючи зростання часу роботи алгоритму при вхідних даних, досить великих розмірів, ми тим самим вивчаємо асимптотичну ефективність алгоритмів. Це означає, що нас цікавить тільки те, як час роботи алгоритму зростає зі збільшенням розміру вхідних даних, коли цей розмір збільшується до нескінченності. Зазвичай алгоритм, більш ефективний в асимптотичному сенсі, буде більш

продуктивним для всіх вхідних даних, за винятком дуже маленьких. Нижче перелічені основні оцінки складності.

Позначення, що вводяться для опису асимптотичної поведінки часу роботи алгоритму, використовують функції, область визначення яких – множина невід'ємних цілих чисел N = {0,1,2, ...}. Подібні позначення зручні для опису часу роботи *T(n)* в найгіршому випадку як функції, визначеної тільки для цілих чисел, що становлять розмір вхідних даних.

Означення. *Функція складності алгоритму f(n) має оцінку* Θ(тета) *й записується як f(n) = Θ(g(n)), якщо існує невід'ємна функція g(n) та додатні* n_0, c_1, c_2 *такі, що*

$$c_1 g(n) \le f(n) \le c_2 g(n), \qquad (6.1)$$

при $n > n_0$.

У такому разі говорять, що функція *g(n)* є асимптотично точною оцінкою функції *f(n),* оскільки за визначенням функція *f(n)* не відрізняється від функції *g(n)* з точністю до сталого множника. Важливо розуміти, що Θ *(g(n))* є не однією функцією, а множиною функцій для опису зростання *f(n)* з точністю до сталого множника. Іншими словами, функція *f(n)* належить множині Θ *(g(n)),* якщо існують додатні c_1 та c_2 , що дозволяють обмежити цю функцію у рамки між функціями $c_1 g(n)$ та $c_2 g(n)$ для достаньо великих значень n.

Наприклад, для методу сортування послідовності чисел алгоритмом heapsort оцінка трудомісткості становить $f(n) = \Theta(n \log_2 n)$, тобто в цьому разі $g(n) = n\log_2 n$.

З означення $f(n) = \Theta(g(n))$ випливає, що
$g(n) = \Theta(f(n))$.

Інтуїтивно зрозуміло, що при асимптотично точній оцінці асимптотично невід'ємних функцій, доданками нижчих порядків у них можна знехтувати, оскільки при великих n вони стають неістотними. Навіть невеликої частки доданка найвищого порядку достатньо для того, щоб перевершити доданки нижчих порядків. Таким чином, для виконання нерівностей (6.1) достатньо як c_1 вибрати значення, яке дещо менше коефіцієнта при самому старшому доданку, а як c_2 – значення, яке дещо більше цього коефіцієнта.

Тому коефіцієнт при старшому доданку можна не враховувати, тому що він лише змінює зазначені константи.

Приклад 6.2. Розглянемо квадратичну функцію
$f(n) = an^2 + bn + c$, де *a, b, c* – константи, причому $a > 0$. Відкидаючи доданки нижчих порядків за перший та ігноруючи коефіцієнт a, одержимо $f(n) = \Theta(n^2)$.

Узагалі кажучи, для будь-якого полінома *p(n)* маємо $p(n) = \Theta(n^d)$, де *d* – степінь полінома при $a_d > 0$. Оскільки будь-яка константа – це поліном нульового степеня, то сталу функцію можна записати як $\Theta(n^0)$ або $\Theta(1)$.

Означення. *Функція складності алгоритму f(n) має оцінку O («O – велике») й записується як f(n) = O(g(n)), якщо існує невід'ємна функція g(n) та додатні* n_0, *с такі, що*

$$0 \le f(n) \le cg(n), \qquad (6.2)$$

при $n > n_0$.

Означення. *Функція складності алгоритму f(n) має оцінку Ω(«омега-велике») й записується як f(n) = Ω(g(n)), якщо існує невід'ємна функція g(n) та додатні* n_0, *с такі, що*

$$0 \le cg(n) \le f(n) \qquad (6.3)$$

при $n > n_0$.

O- позначення застосовуються, коли необхідно вказати верхню межу функції з точністю до сталого множника. В позначеннях теорії множин $\Theta(g(n)) \subset O(g(n))$. Оскільки O-позначення описують верхню межу, то в ході їх використання для обмеження часу роботи алгоритму в найгіршому випадку ми отримуємо верхню межу цієї величини для будь-яких вхідних даних. Таким чином, асимптотична оцінка $O(n^2)$ для часу роботи алгоритму у найгіршому випадку застосовна для часу виконання завдання з будь-якими вхідними даними, чого не можна сказати про Θ-позначення. Наприклад, оцінка в $\Theta(n^2)$ для часу сортування вставками в найгіршому випадку не придатна для довільних вхідних даних. Наприклад, якщо вхідні елементи, що подаються на сортування, вже відсортовані в потрібному порядку , час роботи алгоритму сортування вставкою оцінюється як $\Theta(n)$.

Оскільки Ω-позначення використовуються для визначення нижньої межі часу роботи алгоритму в найкращому випадку, то вони визначають нижню межу часу роботи алгоритму

при довільних вхідних даних. Наприклад, час роботи алгоритму сортування вставками знаходиться в межах між $\Omega(n)$ та $O(n^2)$, тобто між лінійною та квадратичною функціями від n.

На рис. 6.1 подані введені вище позначення.

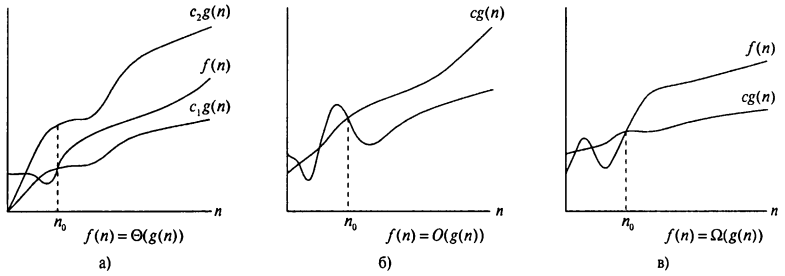


Рис. 6.1 – Графічні приклади асимптотичних позначень

Теорема 6.1. Для будь-яких двох функцій *f(n)* і *g(n)* співвідношення $f(n) = \Theta(g(n))$ виконується тоді й тільки тоді, коли $f(n) = O(g(n))$ і $f(n) = \Omega(g(n))$.

На практиці ця теорема застосовується для визначення асимптотично точної оцінки алгоритму за допомогою асимптотично верхньої та нижньої меж.

Асимптотичний аналіз алгоритмів має не лише практичне, а й теоретичне значення. Так, наприклад, доведено, що всі алгоритми сортування ґрунтуються на попарному порівнянні елементів, відсортують n елементів за час, не менший $\Omega(n\log_2 n)$.

6.2. Класи складності

На початку 1960-х років, у зв'язку з початком широкого використання обчислювальної техніки для розв’язання практичних завдань, виникло питання про межі практичної застосовності цього алгоритму розв'язання задачі в сенсі обмежень на її розмірність. Які завдання можуть бути вирішені на ЕОМ за реальний час? Відповідь на це питання була дана у працях Кобхема (Alan Cobham, 1964) та Ед-мондса (Jack Edmonds, 1965), де були введені класи складності задач.

У теорії алгоритмів *класами складності називаються множини обчислювальних задач, приблизно однакових за складністю обчислення.* Говорячи більш вузько, класи складності – це множини предикатів (функцій, які отримують на вхід слово і повертають відповідь 0 або 1), що використовують для обчислення ресурси приблизно однакової кількості.

У рамках класичної теорії здійснюється класифікація завдань за класами складності (P-складні, NP-складні, експоненціально складні та ін.).

Кожен клас складності (у вузькому сенсі) визначається як множина предикатів, що мають деякі властивості. Типове визначення класу складності має такий вигляд.

Означення. *Класом складності X називається множина предикатів P (x), що обчислюються на машинах Тюрінга і використовуються для обчислення O (f (n)) ресурсу, де n – довжина слова x.*

Як ресурси зазвичай беруть час обчислення (кількість робочих тактів машини Тюрінга) або робочу зону (кількість використаних осередків на рядку під час роботи). Мови, які розпізнаються предикатами з деякого класу (тобто множини слів, на яких предикат повертає 1), також називаються мовами, що належать до того самого класу. Класи прийнято позначати прописними літерами. Доповнення до класу C (тобто клас мов, доповнення яких належать C) позначається co-C.

Для кожного класу існує категорія задач, які є «найскладнішими». Це означає, що будь-яка задача з класу зводиться до такої задачі, і до того ж сама задача лежить у класі. Такі задачі називають повними задачами для даного класу. Найбільш відомими є NP-повні задачі. Повні задачі – хороший інструмент для доведення рівності класів. Достатньо для однієї такої задачі подати алгоритм, що розв’язує її і належить більш маленькому класу, і рівність буде доведено.

Розглянемо найбільш відомі класи складності задач.

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

Формальне визначення. Алгоритм ототожнюється з детермінованою машиною Тюрінга, що обчислює відповідь при даному на вхідний рядок слові із вхідного алфавіту ∑. Час роботи алгоритму T_M (x) при фіксованому вхідному слові x визначається кількістю робочих тактів машини Тюрінга від початку до зупинки машини.

Складністю функції f: ∑* → ∑*, що обчислюється деякою машиною Тюрінга, називається функція C: N → N, що залежить від довжини вхідного слова і дорівнює максимуму часу роботи машини за всіма вхідними словами фіксованої довжини :

$$C_M(n) = \max_{x:|x|=n} T_M(x) \qquad (6.4)$$

Означення. *Якщо для функції f існує машина Тюрінга M така, що C_M (n) <n^c для деякого числа c і досить великих n, то кажуть, що вона належить класу P, або поліноміальна за часом.*

Тобто, задача називається поліноміальною (належить до класу P), якщо існують константа k та алгоритм, що розв'язує задачу з $F_a(n) = O(n^k)$, де n – довжина входу алгоритму в бітах n = |D| .

Згідно з тезою Черча - Тюрінга будь-який мислимий алгоритм можна реалізувати на машині Тюрінга. Для будь-якої мови програмування можна визначити клас P подібним чином (замінивши у визначенні машину Тюрінга на реалізацію мови програмування). Якщо компілятор мови, на якому реалізований алгоритм, уповільнює виконання алгоритму поліноміально (тобто час виконання алгоритму на машині Тюрінга менший деякий за многочлен від часу виконання його мовою програмування), то визначення класів P для цієї мови і для машини Тюрінга збігаються.

Більш вузьке визначення. Іноді під класом P мають на увазі більш вузький клас функцій, а саме клас предикатів (функцій f: ∑ * → {0, 1 }). У такому разі мовою L, що розпізнає даний предикат, називається множина слів, на яких предикат дорівнює 1. Мовами класу P називаються мови, для яких існують предикати, що їх розпізнають, класу P. Очевидно, що якщо мови L_1 і L_2 належать класові P, то і їх об'єднання, перетин та доповнення також належать класові P.

Задачі класу P – це, інтуїтивно, задачі, розв'язувані за реальний час. Відзначимо такі переваги алгоритмів із цього класу:

- для більшості задач із класу P константа k менше 6;
- клас P інваріантний за моделлю обчислень (для широкого класу моделей);
- клас P має властивість природної замкненості (сума або добуток поліномів є поліном).

Таким чином, задачі класу P є уточненням визначення «практично розв'язної» задачі. Прикладами завдань із класу P є цілочислове додавання, множення, ділення, одержання залишку від ділення, множення матриць, з'ясування зв'язності графів і деякі інші.

2. **Клас NP** (*поліноміальної перевірки*)

Клас NP складається із задач, розв'язання яких можна перевірити протягом поліноміального часу. Мається на увазі, що якщо ми одержимо якимось чином розв'язок деякої задачі цього класу, то протягом часу, який буде поліноміально залежати від розміру вхідних даних задачі , можна перевірити коректність такого розв'язку.

Означення. *$\forall D \in D_A$, |D| = n поставимо у відповідність сертифікат $S \in S_A$, такий, що | S | = O (n^l), та алгоритм $A_S = A_S(D, S)$, такий, що видає «1», якщо розв'язок правильний, і «0» , якщо розв'язок неправильний. Тоді задача належить до класу NP, якщо F (A_S) = O (n^m).*

Еквівалентне визначення можна отримати, використовуючи поняття недетермінованої машини Тюрінга (тобто такої машини Тюрінга, у програми якої можуть існувати різні рядки з однаковою лівою частиною).

Означення. *Мова L належить до класу NP (не детермінованих поліноміальних), якщо вона розпізнається не детермінованою машиною Тюрінга з поліноміальною часовою складністю T(n).*

Отже, основна відмінність класів P і NP та, що до класу P належить задачі, які можуть бути розв'язані за час, що поліноміально залежить від обсягу початкових даних, за допомогою детермінованої обчислювальної машини (наприклад, машини Тюрінга), а до класу NP-задачі, які можуть бути розв'язані за поліноміально виражений час за допомогою не детермінованої обчислювальної машини, тобто машини, наступний стан якої не завжди однозначно визначається попередніми. Роботу такої машини можна подати як процес, що розгалужується на кожній неоднозначності: задача вважається розв'язаною, якщо хоча б одна гілка процесу прийшла до відповіді. Оскільки кожна детермінована машина Тюрінга може розглядатись як не детермінована, але без вибору варіантів кроків, то клас P міститься в класі NP ($P \subseteq NP$). Оскільки клас P міститься в класі NP, приналежність того або іншого завдання до класу NP часто відображає наше поточне уявлення про способи розв'язання даної задачі й носить неостаточний характер.

До задач класу складності NP належать, наприклад, задачі: розв'язність логічного виразу, трикольорове розфарбування графу, побудова кліки з k-вершин на неорієнтованому графі, задача покриття множин та інші.

У загальному випадку немає підстав вважати, що для тієї або іншої NP-задачі не може бути знайдений P-розв'язок. Питання про можливу еквівалентність класів P і NP (тобто про можливість знаходження P-розв'язку для будь-якої NP-задачі) вважається багатьма одним із основних питань сучасної теорії складності алгоритмів. Сама постановка питання про еквівалентність класів P і NP можлива завдяки введенню поняття NP-повних задач.

3. **Клас NPC** (*N -повні задачі*)

Неформально задача належить класу NPC, якщо вона належить класу NP і є такою самою «складною», як і будь-яка задача із класу NP. NP-повні задачі складають підмножину NP-задач і відрізняються тією властивістю, що всі NP-задачі можуть бути тим або іншим способом зведені до них. З цього виходить, що якщо для NP-повної задачі буде знайдений P-розв'язок, то P-розв'язок буде знайдений для всіх задач класу NP.

Поняття NP-повноти було введене на початку 1970-х років і ґрунтується на понятті зведення однієї задачі до іншої.

Зведення може бути подане у такий спосіб: якщо ми маємо задачу 1 та алгоритм, що розв'язує цю задачу, тобто видає правильну відповідь для всіх конкретних проблем, що становлять задачу, а для задачі 2 алгоритм розв'язання невідомий, але якщо ми можемо переформулювати (звести) задачу 2 у термінах задачі 1, то ми розв'язуємо задачу 2.

Таким чином, якщо задача 1 задана множиною конкретних проблем D_{A1}, а задача 2 – множиною D_{A2}, й існує функція f_s(алгоритм), що зводить конкретну постановку задачі 2 (d_2) до конкретної постановки задачі 1(d_1):

fs ($d_{(2)} \in D_{A2}$) = $d_{(1)} \in D_{A1}$, то задача 2 зводиться до задачі 1.

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

Означення. *Задача належить до класу NPC, якщо виконуються такі дві умови: по-перше, задача повинна належати до класу NP (L є NP), і, по-друге, до неї поліноміально повинні зводитися всі задачі із класу NP (Lx = < p, для кожного Lx є NP).*

Виконання цього означення показане на рис. 6.2.

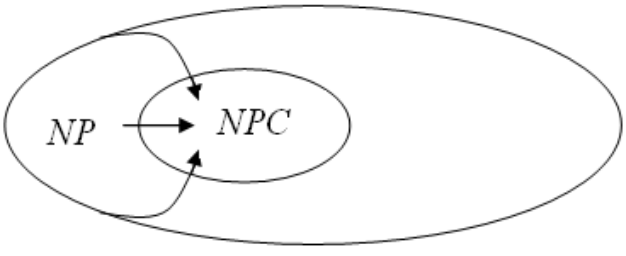


Рис. 6.2 – Зведення і клас NPC

Теорема 6.1. *Якщо існує задача, що належить до класу NPC, для якої існує поліноміальний алгоритм розв'язання ($F = O(n^k)$), то клас P збігається із класом NP, тобто P = NP.*

Схема доведення буде складатися зі зведення будь-якої задачі з NP до даної задачі із класу NPC з поліноміальною трудомісткістю й розв'язання цієї задачі за поліноміальний час (за умовою теореми).

У цей час доведено існування сотень NPC задач, але для жодної з них поки не вдалося знайти поліноміального алгоритму розв'язання. У цей час дослідники припускають таке співвідношення класів, що наведене на рис. 6.3, де $P \neq NP$ і задачі із класу NPC не можуть бути розв'язані (сьогодні) з поліноміальною трудомісткістю.

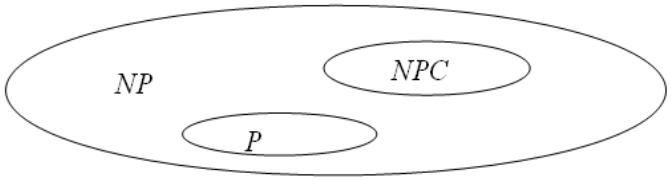


Рис. 6.3 – Співвідношення класів P, NP, NPC

Оскільки метод зведення базується на тому, що для деякої задачі відома її NP – повнота, то для доведення NP – повноти різних задач нам знадобиться «перша» NP – повна задача. Як таку використовуємо задачу на здійсненність, у якій задана булева комбінаційна схема, що складається з логічних елементів (або задано КНФ – кон'юнктивну нормальну форму). В задачі запитується, чи існує набір вхідних булевих величин, для яких на виході буде видане значення 1.

Прикладами NP-повних задач є, крім задачі про кон'юнктивну форму, задача комівояжера, розфарбування графа, задача про гамільтонів цикл у графі та інші.

4. І**нші класи складності**

Дослідження складності алгоритмів дозволили по-новому поглянути на розв'язання багатьох класичних математичних задач і знайти для ряду таких задач (множення многочленів і матриць, розв'язання лінійних систем рівнянь та ін.), розв'язання, що вимагають менше ресурсів, ніж традиційні.

Прості класи складності визначаються такими факторами:

- *типом обчислювальної задачі:* найбільш часто використовуються задачі прийняття рішень;
- *моделлю обчислень:* найбільш поширеною моделлю обчислень є детермінована машина Тюрінга, але багато класів складності визначають на основі недетермінованої машини Тюрінга, логічних схем, квантової машини Тюрінга, монотонних схем і т. д.;
- *ресурсами, які мають межі:* вказується, як "поліноміальний час", "логарифмічний простір", "стала глибина" тощо.

Усі класи складності знаходяться в ієрархічному відношенні: одні містять у собі інші. Однак про більшість включень невідомо, чи є вони строгими. Одна з найбільш відомих відкритих проблем у цій сфері – рівність класів P і NP. На даний момент найбільш поширеною є гіпотеза про невиродженість ієрархії (тобто всі класи різні).

Наведемо приклад побудови однієї з можливих ієрархічних структур класів складності.

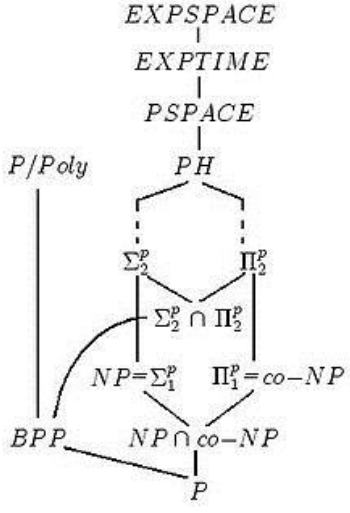


Рис. 6.4 – Ієрархія класів складності

На цій ієрархії показані, наприклад, такі класи, що побудовані на ймовірнісних машинах Тюрінга (BPP), префіксом *co* позначені доповнення до класів, PSPACE – клас, допустимий детермінованою машиною Тюрінга з поліноміальним обмеженим простором і т. д.

Контрольні запитання

1. Сформулюйте основні асимптотичні оцінки, що застосовуються в теорії алгоритмів.
2. Розмістіть у порядку зростання асимптотичні оцінки $O(n)$, $O(n^2)$, $O(1)$, $O(\log_2 n)$, $O(n\log_2 n)$, $O(\log_2\log_2 n)$, $O(n^3)$, $O(2^n)$ для $n > 100$.
3. Зазначте основні класи складності алгоритмів та побудуйте діаграму Ейлера про їх взаємозв’язок.
4. Сформулюйте основну проблему теорії складності.
5. Сформулюйте означення терміна «клас складності».

ЛЕКЦІЯ 7. NP - ПОВНІ, СКЛАДНІ ТА АЛГОРИТМІЧНО НЕРОЗВ'ЯЗНІ ПРОБЛЕМИ

Розв'язні та нерозв'язні проблеми, NP-повнота, складність, зведення. Приклади NP-повних задач. Алгоритмічно нерозв'язні проблеми.

7.1. Розв'язні та нерозв'язні проблеми, NP-повнота, складність, зведення

Теорія алгоритмів розподіляє всі проблеми, що потребують розв'язання, на дві широкі категорії : алгоритмічно розв'язні та нерозв'язні. Останні проблеми будуть розглянуті дещо пізніше. Зупинимося на перших.

Бурхливий розвиток комп'ютерної індустрії підсвідомо наводить на думку, що вже не так важливо, які алгоритми застосовувати для розв'язної проблеми, – сучасний комп'ютер може все. Однак, наприклад, алгоритм «грубої сили» для розв'язування задачки на триста-п'ятсот змінних може зажадати такої кількості операцій, що набагато більше, ніж у Всесвіті елементарних частинок. Наприклад, якщо розв'язання задачі на найсучасніших комп'ютерах займе 10^{10} років, то очевидно, такий алгоритм для нас не має сенсу.

У попередньому розділі ми ввели поняття «складність алгоритму». Зрозумілим є бажання аналогічно визначити і складність завдання − наприклад, як складність найефективнішого алгоритму, що розв'язує цю задачу (для даних розміру n). На жаль, це бажання нездійсненне. Доведено, що є завдання, для яких не існує найшвидшого алгоритму, тому що будь-який алгоритм для такого завдання можна «прискорити», побудувавши більш швидкий алгоритм, що розв'язує цю задачу. Це твердження називають теоремою Блюма про прискорення. Якщо відволіктися від технічних деталей, то спрощений варіант теореми Блюма можна сформулювати таким чином.

Теорема 7.1 (теорема Блюма про прискорення). *Існує така алгоритмічно розв'язна проблема Z, що будь-який алгоритм A, що вирішує цю проблему, можна прискорити таким чином: існує інший алгоритм A_1, що також розв'язує Z, і такий, що $T_{A1}(n) \le logT_A(n)$ для майже всіх n.*

Зауважимо, що теорема Блюма не стверджує, що прискорення можливе для будь-якої задачі. Більше того, для деяких задач існує оптимальний (найшвидший) алгоритм. Однак твердження теореми Блюма про існування «незручних» задач не дозволяє визначити універсальне (застосовне до всіх задач) поняття «оптимального алгоритму».

Тому в теорії складності алгоритмічно розв'язних проблем використаний інший підхід – через класи складності.

Означення 7.1. *Нехай f (n) – деяка функція, що відображає N в N. Клас складності C (f (n)) − це множина всіх задач, для яких існує хоча б один алгоритм, складність якого не перевищує O (f (n)).*

Це визначення в певному сенсі умовне − позначення *C(f (n))* ніколи не застосовують. Чому? Тому що для завдання реального класу задач необхідно ще уточнити:

• що ми розуміємо під «алгоритмом»;

• яка складність (тимчасова, ємнісна чи ще якась) нас цікавить.

При різних відповідях на ці питання одержимо різні класи завдань, і для кожного класу використовується спеціальне позначення.

Термінологія класів складності із 70-х років минулого століття значно змінилася й поповнилася новими термінами, розширила рамки вже існуючих. Незмінними аспектами при побудові глобальної картини ієрархії складності залишаються такі.

• У теорії складності під «алгоритмом» зазвичай розуміють той чи інший різновид машини Тюрінга. Винайдення нових різновидів приводить до появи й нових класів складності. Наприклад, окрім уже відомих класів P, NP, NPC, з'явилися класи : EXPTIME (іноді має назву EXP) – задачі, які розв'язуються детермінованою машиною Тюрінга за O ($2^{p(n)}$) часу, де p(n) є поліномом функції n; EXPSPACE – задачі, для розв'язання яких детермінованій машині Тюрінга потрібно O($2^{p(n)}$) ємнісного простору, де p(n) є поліномом функції n; BPP – проблеми, розв'язні за допомогою ймовірнісної машини Тюрінга в

поліноміальний час із похибкою ймовірністю не більше 1/3 та інші. Є й екзотичні класи, наприклад, P^{NP} – клас задач, що розв'язуються за поліноміальний час детермінованою машиною Тюрінга з оракулом для NP-завдання.

- Для формальних визначень класів складності зазвичай розглядають не довільні алгоритми, а алгоритми для так званих задач розв'язання (decision problem), у яких потрібно визначити, належить чи ні певний елемент деякій множині. Враховуючи необхідність кодування даних, що подаються на вхід машині Тюрінга, ці задачі абсолютно еквівалентні задачам розпізнавання мов, коли на деякому алфавіті $\sum$ розглядається підмножина слів $L \subset \sum^*$ і для довільного слова $l \in \sum^*$ потрібно визначити, чи належить воно мові L.
- У сучасній теорії складності обчислень поняття поліноміального алгоритму є адекватним математичним уточненням інтуїтивного поняття «ефективний алгоритм», а клас P становить «клас ефективно розв'язуваних задач». Тобто прості задачі (розв'язувані) – це задачі, які розв'язуються за поліноміальний час. Важкорозв'язні задачі – це задачі, які не розв'язуються за поліноміальний час, або алгоритм розв'язання за поліноміальний час не знайдений.
- Побудовано ієрархічне співвідношення класів складності : $P \subseteq NP \subseteq BPP \subseteq PH \subseteq EXPSPACE \subseteq EXPTIME$, тобто одні містять у собі інші (більшість включень не доведено). Однак, зважаючи на потужності сучасних засобів обчислювальної техніки, задачі інших класів складності розглядаються поки що лише в теорії, а однією з найбільш відомих відкритих проблем залишається питання про рівність класів P та NP.

У теорії складності обчислень одним із основних інструментаріїв, що застосовуються, є відомий науковий метод зведення – перетворення однієї задачі в іншу. У загальному випадку, якщо у нас є алгоритм, що перетворить екземпляри задачі P_1в екземпляри задачі P_2, які мають ту самому відповідь (так / ні), то говорять, що P_1 зводиться до P_2. Таким чином, зведення – це відношення між двома задачами. За допомогою такого зв'язку може бути доведена приналежність до того або іншого класу складності.

Зведення алгоритмічне – одне з основних понять теорії алгоритмів. Виникло у зв'язку з тим, що нерозв'язність (і розв'язність) багатьох алгоритмічних проблем установлюється здебільшого не безпосередньо, а шляхом зведення до досліджуваної проблеми такої алгоритмічної проблеми, нерозв'язність якої вже доведена.

Поняття алгоритмічного зведення було уточнено
А. Тюрінгом: *якщо, деяка машина Тюрінга переробляє послідовність закодованих значень функції g у послідовність закодованих значень функції f, то функція f зводиться до функції g.* Якщо f зводиться до g за Тюрінгом, $f \leq_T g$, а g зводиться до f, $g \leq_T f$, то кажуть, що f і g мають один і той степінь нерозв'язності, або $f \equiv_T g$.

Останнім часом частіше застосовуються зведення за Карпом або Куком.

Означення 7.2. *Будь-яка мова програмування L_1 називається зведеною за Карпом до мови L_2, якщо існує функція $F: \Sigma^* \to \Sigma^*$, що обчислюється за поліноміальний час, де F (x) належить L_2 в тому разі, якщо x належить L_1.*

Розглянемо дві мови L_1 і L_2 над алфавітами Σ і Г. Зведення L_1 до L_2 за Карпом – це функція $f: \Sigma^* \to \Gamma^*$, обчислювана за поліноміальний час, така, що $\forall x \in L_1 \leftrightarrow f(x) \in L_2$. Таким чином, неформально мова L_1 «не складніша» за мову L_2. Якщо така функція існує, кажуть, що L_1 зводиться за Карпом до L_2, і пишуть $L_1 \leq_K L_2$. Відмітимо, що зведення за Карпом є частковим випадком зведення за Куком.

Означення 7.3. *Зведення задачі R_1до задачі R_2 за Куком – це поліноміальний за часом алгоритм (іншими словами, машина Тюрінга з поліноміальним часом роботи), що розв'язує задачу R_1 за умови, що функція, що знаходить розв'язок задачі R_2, йому дана як оракул, тобто звернення до неї займає один крок.*

Якщо такий алгоритм існує, кажуть, що R_1 зведена за Куком до R_2 і пишуть $R_1 \leq_C R_2$.

Поява деяких нових класів складності пов'язана із залученням до процесу їх вивчення так званої «пророчої машини» – машини Тюрінга з оракулом. Пророк, у цьому розрізі, розглядається як сутність, здатна відповідати на набір питань і зазвичай представлена як

деяка підмножина натуральних чисел. Тоді, інтуїтивно, пророча машина може виконувати всі звичайні дії машини Тюрінга і також може запитати у пророка: «Чи х належить А?».

Машина Тюрінга взаємодіє з оракулом шляхом запису на свій рядок вхідних даних для оракула і потім запускає оракул на виконання. За один крок оракул обчислює функцію, стирає вхідні дані й пише вихідні дані у рядок.

Клас складності задач, розв'язуваних алгоритмом з класу А з оракулом для задачі класу В, позначають A^B. Наприклад, клас задач, розв'язуваних за поліноміальний час детермінованою машиною Тюрінга з оракулом для
NP-завдання, позначають P^{NP}.

Повернемося до одного з основних питань – яке ж практичне значення має вивчення теорії складності й класифікація завдань з точки зору NP-повноти (NPC)? Відповідь очевидна – часто набагато розумніше та ефективніше знайти доказ того, що розглянута задача належить до класу NPC, й відповідно до цього зайнятися пошуком досить точних наближених алгоритмів, ніж безрезультатно витрачати час на відшукання поліноміальних алгоритмів її розв'язання. Зрозуміло, що саме NPC-задачі відіграють тут центральну роль – справа в тому, що поліноміальний час є, хоча й першим, але досить гарним наближенням поняття «практичної можливості розв'язання задачі».

Нагадаємо, що задачі класу NPC відповідають двом умовам: по-перше, вони обов'язково належать класу NP; по-друге, будь-яка довільна задача з класу NP зводиться до цих задач. Якщо опустити першу умову, то одержимо клас задач *NPH(non-deterministic polynomial-time hard)* – клас задач, які "принаймні так складні, як найбільш важкі задачі в NP". Клас NPH містить у собі NPC (див. рис. 7.1), але виходить із меж класу NP(за гіпотези $P \neq NP$). Серед задач NPC виділяють також задачі *NPCS – NP- повні в сильному сенсі*, як ті, що належать класу NP, є NP-повними та мають максимальне значення величин, що зустрічаються в цій задачі, обмежені зверху поліномом від довжини входу.

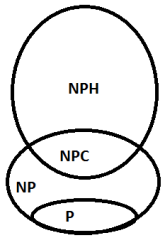


Рис. 7.1 – Співвідношення між класами P, NP, NPC, NPH

7.2. Приклади NP-повних задач

Значення класу NP-повних проблем полягає ще й у тому, що йому належать дуже багато відомих і важливих у прикладному відношенні задач.

Проблемі доведення повноти цих задач присвячені дослідження багатьох видатних вчених з теорії алгоритмів. Будь-які доведення NP-повноти складні, але процедуру доведення можна спростити, застосувавши поняття зведення.

Для цього достатньо показати, що будь-яка з відомих NP-повних задач може бути зведена до цієї задачі.

Наприклад, Річард Карп у своїй праці «Зводимість між комбінаторними задачами» опублікував цілий список, що складається з формулювання та доведення NP-повноти 21 задачі. Перелічимо деякі з них.

1. Задача розбиття

Задана множина додатних цілих чисел x_1, x_2, ..., x_n. Чи можна розбити її на дві підмножини так, щоб суми чисел в обох підмножинах були однаковими?

2. Задача ізоморфізму підграфа

Нехай є два графи G_1 і G_2. Множину вершин першого графа позначимо V_1, а другого – V_2. Нехай | V1 |> | V2 |. Потрібно відповісти на запитання: чи знайдеться в графі G_1 підграф H, ізоморфний графу G_2?

3. Задача виконуваності булевих формул (SAT)

Задача виконуваності булевих формул (SAT) важлива для теорії обчислювальної складності. Об'єктом задачі SAT є булева формула, що складається тільки з імен змінних, дужок та операцій $\wedge$(І), $\vee$(АБО) і $\neg$(НІ). Задача полягає в такому: чи можна призначити всім змінним, що зустрічаються у формулі, значення «ІСТИННІСТЬ» і «ХИБНІСТЬ» так, щоб формула набула значення «ІСТИННІСТЬ»?

4. Задача про кліку

Маємо граф G з m вершинами і ціле додатне число n.

Граф називається клікою, якщо кожна вершина в ньому зв'язана ребром з кожною. Кількість вершин у кліці назвемо її потужністю. Чи правда, що в даному графі G є кліка потужності не менш а ніж n?

5. Задача про гамільтонів цикл

Маємо граф G з n вершинами. Чи існує у графі простий цикл, що проходить через усі вершини графа? Простим називається цикл, у якому вершини не повторюються. Таким чином, гамільтонів цикл – це послідовність вершин і ребер (дуг) графа, що містить усі вершини графа G по одному разу, але щодо ребер обмежень не встановлено, то може бути, що містить не всі дуги.

6. Задача комівояжера

Нехай є граф G з n вершинами. Кожному ребру графа приписано додатне ціле число d_i, що задає довжину ребра. Крім того, задане деяке додатне ціле число L. Потрібно відповісти на запитання: чи знайдеться у графі G маршрут, що проходить через усі вершини графа G, такий, що його довжина не перевищує L?

7. Вершинне покриття

Задано граф G із m вершинами і ціле додатне число n. Вершинним покриттям називається підмножина $U \subseteq V$ множини V вершин графа така, що будь-яке ребро графа G інцидентне хоча б однієї з вершин множини U (друга вершина ребра може або належати, або не належати множині U). Чи існує вершинне покриття, що має в собі не більше ніж n вершин?

7.3. Алгоритмічно нерозв’язні проблеми

За час свого існування людство придумало безліч алгоритмів для розв’язання різноманітних практичних і наукових проблем. Поставимо запитання: а чи існують такі проблеми, для яких неможливо придумати алгоритми їх розв’язання?

Твердження про існування алгоритмічно нерозв'язних проблем є дуже переконливим, адже ми констатуємо, що не знаємо відповідного алгоритму не лише зараз, а й не можемо принципово ніколи його знайти. Виявляється, що існують такі класи задач, для розв’язання яких немає і не може бути єдиного універсального прийому. Проблеми розв’язання такого роду задач називають алгоритмічно нерозв'язними проблемами. Однак алгоритмічна нерозв'язність проблеми розв'язання задач того чи іншого класу зовсім не означає неможливість розв’язання будь-якої конкретної задачі із цього класу. Мова йде про неможливість розв’язання всіх завдань даного класу одним і тим самим прийомом. *Алгоритмічна нерозв'язність* – найважливіша властивість деяких класів коректно поставлених завдань, що допускають застосування алгоритмів, яка полягає в тому, що задача кожного із цих класів у принципі не має загального, універсального алгоритму розв’язання, що об'єднує цей клас.

Алгоритмічна нерозв'язність не є проблемою теорії алгоритмів або невдачею, вона є науковим фактом. Популярність досліджень алгоритмічної нерозв'язності не поступається будь-яким іншим дослідженням у теорії алгоритмів і потребує іноді значних зусиль. Наприклад, доведення алгоритмічної нерозв'язності 10-ї проблеми Гільберта, одержане Ю. В. Матіясевичем, відносять до видатних наукових досягнень XX століття.

Знання можливої алгоритмічної нерозв’язності має в галузі комп’ютерних технологій таке саме значення, як для фізика знання про неможливість створення вічного двигуна. Припустимо, що до Білла Гейтса приходить винахідник з пропозицією надати йому грант (грошові інвестиції) для написання програми, яка приймає на вхід будь-яку програму і видає відповідь, потрапить ця програма в нескінченний цикл чи ні за будь-яких вхідних даних.

Б. Гейтс відмовить йому оскільки знає, що ця задача є алгоритмічно нерозв'язною.

Означення 7.4. *Алгоритмічною проблемою називається проблема побудови алгоритму, що має такі чи іншими властивості.*

Ми отримаємо розв'язання алгоритмічної проблеми, якщо або знайдемо шуканий алгоритм (тобто подано його опис), або доведемо, що такого алгоритму не існує.

Однією з найбільш загальних теорем теорії алгоритмів, що пояснює природу багатьох проблем у теорії алгоритмів, є теорема Райса.

Уведемо деякі визначення.

Означення 7.5. *Характеристичною функцією множини А будемо називати функцію*

$$\chi_A(x) = \begin{cases} 1, \text{ якщо } x \in A, \\ 0, \text{ якщо } x \notin A. \end{cases} \qquad (7.1)$$

Множина А називається рекурсивною, якщо її характеристична функція рекурсивна.

Означення 7.6. *Нехай Q – деяка властивість одномісних частково-рекурсивних функцій. Властивість Q називається нетривіальною, якщо існують функції, що мають цю властивість, так і функції, що не мають її.*

Усі властивості, що розглядаються в теорії алгоритмів, зазвичай є нетривіальними. Наприклад, властивості функцій: усюди визначеність, взаємна однозначність, тотожна рівна нулю та інші.

Ураховуючи те, що частково рекурсивні функції можна задати програмою їх обчислення, виникає питання: чи можна за програмою визначити, чи має відповідна функція певну нетривіальну властивість чи ні?

Відповідно до тез Черча і Тюрінга задача є алгоритмічно розв'язуваною тоді і тільки тоді, коли існує деяка машина Тюрінга М, що розв'язує цю задачу. За поставленим завданням необхідно визначити, чи характерна функції $f_м(x)$, що реалізується машиною М, властивість Q. Функцію $f_м(x)$ будемо розуміти як функцію, що відповідає програмі з номером Геделя *x*.

Теорема Райса. *Якою б не була нетривіальною властивість Q одномісних частково-рекурсивних функцій, задача розпізнання цієї властивості алгоритмічно нерозв'язна, тобто не існує машини Тюрінга М, що розв'язує цю задачу.*

Наслідки з теореми Райса. *Неможливо за допомогою універсального алгоритму перевірити всюди визначеність функції $f_м(x)$; не існує алгоритму перевірки того, що програма обчислить нульову функцію; питання про те, обчислюють чи ні дві програми одну й ту саму функцію, масово нерозв'язна; не існує алгоритму, що визначає за текстом програми, чи буде ця програма обчислювати деяку конкретну обчислювальну функцію.*

Теорема Райса є однією з найбільш загальних теорем теорії алгоритмів, що пояснює природу багатьох проблем нерозв'язності у практиці програмування.

Першою фундаментальною теоретичною працею, пов'язаною з доведенням алгоритмічної нерозв'язності, була праця Курта Геделя – його відома теорема про неповноту символічних логік. Це була строго сформульована математична проблема, для якої не існує розв'язного її алгоритму. Зусиллями різних дослідників список алгоритмічно нерозв'язних проблем був значно розширений. Сьогодні прийнято під час доведення алгоритмічної нерозв'язності деякої проблеми зводити її до класичної задачі – «задачі зупину».

Теорема 7.3. *Не існує алгоритму (машини Тюрінга), що дозволяє за описом довільного алгоритму і його вихідних даних (і алгоритм, і дані задані символами на рядку машини Тюрінга) визначити, зупиняється цей алгоритм на цих даних чи працює нескінченно.*

Таким чином, фундаментально алгоритмічна нерозв'язність пов'язана з нескінченністю виконуваних алгоритмом дій, тобто неможливістю передбачити, що для будь-яких вихідних даних розв'язок буде отриманий за кінцеву кількість кроків. Проте можна спробувати сформулювати причини, що призводять 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.

Приклад 7.1. *Розподіл дев'яток у записі числа π*

Визначимо функцію f (n) = i, де n – кількість дев'яток поспіль у десятковому записі числа π, a_i – номер найлівішої дев'ятки з n дев'яток поспіль:

π = 3,141592 ... f (1) = 5.

Завдання 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) у десятковому записі числа. Обчислення f (n) пов'язане з обчисленням наступних цифр у розкладанні, доти, 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 – ми навіть не знаємо в принципі (за природою числа π), чи існує розв'язок для всіх n.

Приклад 7.2. *Десята проблема Гільберта*

Вона полягає у знаходженні універсального методу цілочислового розв'язання довільного алгебраїчного діофантового рівняння. Доведення алгоритмічної нерозв'язності цієї задачі зайняло близько двадцяти років і було завершене Ю. В. Матіясевичем у 1970 році.

Формально мова йде про цілочислове розв'язання рівнянь вигляду

$P(x_1, x_2, \ldots, x_n) = 0$, де P – многочлен із цілими коефіцієнтами й цілими показниками степенів. Доведено, що такого алгоритму не існує, тобто відсутній загальний метод визначення цілих коренів цього рівняння.

Контрольні запитання

1. У чому суть теореми Блюма про прискорення?
2. Алгоритмічні зведення за Карпом, Куком, Тюрінгом. Яка між ними різниця і що є однаковим?
3. Як довести, що завдання є NP-повним?
4. Що означає, що одна задача зводиться до іншої за поліноміальний час?
5. Як довести, що завдання A зводиться (за поліноміальний час) до задачі B?
6. Дайте означення алгоритмічної нерозв'язності.
7. Що називається характеристичною функцією множини A?
8. Сформулюйте теорему Райса та її наслідки.

ЛЕКЦІЯ 8. АЛГОРИТМИ СОРТУВАННЯ

Сортування включенням із зменшуваними відстанями – алгоритм Шелла. Сортування обміном на великих відстанях – алгоритм Quick Sort. Сортування вибором при допомозі дерева – алгоритм Tree Sort. Сортування вибором при допомозі дерева – алгоритм Heap Sort

8.1 Сортування включенням із зменшуваними відстанями – алгоритм Шелла

Шелл вдосконалив пряме включення. Він запропонував проводити послідовне впорядкування підмасивів з елементів, які знаходяться на великих відстанях. При цьому на кожному наступному етапі відстані між елементами в групах мають зменшуватися.

Для ілюстрації алгоритму розглянемо його покрокове описання. Не обмежуючи загальності, в якості прикладу спочатку зупинимося на масиві з кількістю елементів, що є степенем двійки, тобто $N = 2^m$:

1) на першому етапі окремо групуються і сортуються елементи, розміщені на відстані **N/2** . Це є впорядкування **N/2** підмасиві по **2** елементи, яке називатимемо **N/2-сортування .**

2) на другому етапі виконується впорядку вання **N/4** підмасивів по **4** елементи на відстані **N/4 - N/4-сортування** і т.д.

На останньому етапі виконується **одинарне сортування** (впорядкування на відстані **1**). Наприклад :

44 55 12 42 94 18 06 67
етап I-сортування
44 18 06 42 94 55 12 67
етап II-сортування
06 18 12 42 44 55 94 67
етап III-сортування
06 12 18 42 44 55 67 94

Виникає питання, чи використання багатьох процесів сортування із залученням всіх елементів не збільшить кількість операцій, тобто складність алгоритму? Але на кожному проході або впорядковується відносно мало елементів (початкові етапи), або елементи вже досить добре впорядковані і вимагається відносно мало перестановок (кінцеві етапи). Кожне **i**-те сортування об’єднує дві групи, вже впорядковані **2i**-тим сортуванням. Очевидно, що відстань між елементами груп можна зменшувати по-різному, головне, щоб остання була одиничною. Останній прохід в найгіршому випадку і виконує основну роботу. Варто зауважити, що такий довільний підхід при зменшенні відстаней не погіршує результату і у випадку кількості елементів **N**, що не є степенем двійки.

Нехай виконується **t** етапів. Відстані між елементами в окремих групах на кожному етапі позначимо : $h_1, h_2, ..., h_t$, де $h_t=1$, $h_{i+1}<h_i$, $i=1, 2, ..., t-1$. Таким чином розглядаються **h**-ті сортування. Кожне **h**-те сортування можна реалізувати будь-яким із прямих методів. Зокрема, вибір включення оправданий кращою в порівнянні з іншими алгоритмами ефективністю по перестановках ключів. Однак, чи варто для спрощення умови припинення пошуку місця включення чергового елемента користуватися методом бар’єрів? Оскільки кожне **h**-те сортування потребує свого власного бар’єра, то прийдеться розширити масив не на одну компоненту a_0 , а на h_1 компонент. На першому етапі - це практично половина всіх елементів. У випадку "довгих" масивів прийдеться порушити правило сортування "на своєму місці". Тому не варто заради економії однієї логічної операції на кожному етапі впорядкування жертвувати такими об’ємами пам’яті.

Кількість етапів сортування **t** як і відстані на кожному з них можна вибирати довільно. Зокрема, це може бути кількість цілочисельних поділів числа **N** на **2**, тобто **t=[log(N)]**. В якості прикладу пропонується процедура сортування методом Шелла для масиву із **16** елементів :

```
Procedure Shell_Sort;
Const t=4;
```

```
Var
m, i, j, k : integer;
h : array [1..t] of integer;
x : basetype;
Begin
h[1]:=8; h[2]:=4; h[3]:=2; h[4]:=1;
for m:=1 to t do
begin
k:=h[m];
for i:=k+1 to N do
begin
x:=a[i]; j:=i-k;
while (x<a[j]) and (j>0) do
begin
a[j+k]:=a[j]; j:=j-k
end;
a[j+k]:=x
end
end
End;
```

Аналіз алгоритму Шелла. Поки що не має чітко обгрунтованих виборів відстаней, які давали б найкращу ефективність. Самим цікавим є те, що ці відстані не повинні бути множниками один одного, а тим більше степенями деяких чисел. Це дозволяє уникнути явища, коли на певному етапі взаємодіють дві групи, які до цього ніде ще не перетиналися. Взагалі, бажано, щоб взаємодія окремих ланцюгів відбувалася якомога частіше. Вірною є наступна теорема :

Теорема. Якщо **k**-відсортовану послідовність **i**-відсортувати, то вона при цьому залишиться **k**-відсортованою.

Кнут рекомендує використовувати такі послідовності відстаней, записані в зворотньому порядку :

1, 4, 13, 40, 121, ... , де $\mathbf{h_{i-1}=3h_i+1}$, $\mathbf{h_t=1}$, $\mathbf{t=[\log_3 N]-1}$, або

1, 3, 7, 15, 31, ... , де $\mathbf{h_{i-1}=2h_i+1}$, $\mathbf{h_t=1}$, $\mathbf{t=[\log_2 N]-1}$.

З обчислювальної практики відомо, що загалом метод Шелла має ефективність порядку $O(N^{1,2})$.

8.2 Сортування обміном на великих відстанях – алгоритм Quick Sort

Основна причина повільної роботи алгоритму прямого обміну полягає в тому, що всі порівняння і перестановки елементів в послідовності $\mathbf{a_1, a_2, ..., a_N}$ відбуваються для пар із сусідніх елементів. При такому способі потрібно відносно більше часу, щоб поставити деякий ключ, який знаходиться не на своєму місці, в потрібну позицію у сортованій послідовності. Природньо попробувати пришвидшити цей процес, порівнюючи пари елементів, що знаходяться далеко один від одного в масиві. К. Хоор розробив алгоритм Quick Sort із середнім часом роботи порядку **O(N*lnN).**

Припустимо, що перший елемент масиву, що сортується, є хорошим наближенням елемента, який вкінці опиниться на своєму місці у відсортованій послідовності. Приймемо його значення в якості ведучого елемента, відносно якого ключі будуть мінятися місцями. Для зручності реалізації алгоритму використаємо два вказівники **I, J**, перший з яких вестиме відлік вздовж розглядуваної частини масиву зліва, а другий - справа. Початково їх значення будуть відповідно **I=1, J=N**. Таким чином ведучим елементом буде значення **a[I]**. Перестановки ключів проводяться за такими принципами :

1) порівнюються елементи **a[I]** та **a[J]**; якщо **a[I]≤a[J]**, то здійснюється крок вліво **J:=J-1** і порівняння повторюється; зменшення **J** продовжується доти, поки не виконається умова **a[I]> a[J]**;

2) якщо при порівнянні елементів досягнута умова **a[I]> a[J]**, то проводиться обмін місцями ключів **a[I]** та **a[J]** і здійснюється крок вправо **I:=I+1**; таким чином ведучий елемент перейшов в позицію **J**; порівняння ключів із збільшенням **I** продовжується доти, поки знову не виконається умова **a[I]> a[J]**;

3) у випадку виконання умови **a[I]> a[J]** проводиться обмін місцями ключів **a[I]** та **a[J]** і здійснюється крок вліво **J:=J-1**; при цьому ведучий елемент знову повертається в позицію **I**.

Цей процес із почерговим зменшенням **J** та збільшенням **I** повторюється з обох кінців послідовності до "середини"до тих пір, поки не досягнеться **I=J**.

Тепер мають місце два факти. По-перше, ключ, що був першим у вихідній послідовності, в результаті такого впорядкування опиняється на "своєму" місці. По-друге, всі елементи зліва будуть меншими за нього, а всі ключі справа - більшими.

Ту ж саму процедуру можна використати для впорядкування лівої і правої підпослідовностей і т. д. до повного сортування всього масиву. Таким чином розглянутий алгоритм має чітко виражений рекурсивний характер. Виходячи з цього, значення індексів крайніх елементів меншої з двох невідсортованих підпослідовностей варто помістити в стекову структуру даних, і приступити до впорядкування більшої підпослідовності.

Оскільки короткі послідовності скоріше сортуються при допомозі прямих методів, то алгоритм Quick Sort матиме вхідний параметр - деяке число **S**, що визначає нижню межу його використання. Провівши нескладний математичний аналіз нерівності, яка пов’язує ефективності алгоритму Quick Sort та прямих алгоритмів сортування

$$\frac{N^2 - N}{2} \geq N * \ln(N),$$

можна встановити значення числа **S**, яке буде нижньою межею використання швидкого сортування. Остання нерівність дає результат $S \geq 7$.

Однак, якщо крім основних операцій порівняння ключів ще враховувати порівняння індексів та перестановки елементів, то це значення можна збільшити в 2-3 рази.

В якості прикладу наводиться програмна реалізація цього алгоритму у вигляд процедури **Quick_Sort**. В ній використовується два масиви **left** і **right**, де зберігатимуться індексні номери відповідно лівої і правої границь підпослідовностей, які ще будуть впорядковуватися на наступних етапах.

```
Procedure Quick_Sort;
Const S=20;
Var
k, L, R, i, j : integer;
x : basetype;
left, right : array [1..N] of integer;
Begin
k:=1; left[k]:=1; right[k]:=N;
while k>0 do
begin
L:=left[k]; R:=right[k]; k:=k-1;
while R-L>S do
begin
i:=L; j:=R; x:=a[i];
while j>i do
begin
while x<a[j] do j:=j-1;
if j>i then begin
a[i]:=a[j]; a[j]:=x; i:=i+1
end;
while a[i]<x do i:=i+1;
if j>i then begin
a[j]:=a[i]; a[i]:=x; j:=j-1
```

```
end
end;
k:=k+1;
if R-i<=i-L then
begin
left[k]:=i+1; right[k]:=R; R:=i-1
end
else
begin
left[k]:=L; right[k]:=i-1; L:=i+1
end
end;
for i:=L+1 to R do
begin
x:=a[i]; j:=i-1;
while (x<a[j]) and (j>=L) do
begin
a[j+1]:=a[j]; j:=j-1
end;
a[j+1]:=x
end
end
End;
```

Аналіз алгоритму Quick Sort. Щоб оцінити ефективність алгоритму, позначимо через **Q(N)** середню кількість кроків, необхідних для впорядкування **N** елементів. Припустимо також, **S=1**, тобто не використовується сортування прямими методами на коротких послідовностях.

При першому проходженні Quick Sort порівнює всі елементи з ведучим і виконується не більше ніж за **C*N** кроків, де **C** - деяка константа. Потім потрібно відсортувати дві підпослідовності довжинами **I-1** та **N-I**. Тому середня кількість кроків, потрібних для впорядкування **N** елементів, залежить від середньої кількості кроків, потрібних для впорядкування **I-1** та **N-I** елементів відповідно. Оскільки всі можливі значення $I=\overline{1,N}$ є рівноймовірними, то спрведлива наступна оцінка :

$$Q(N)\le C*N+\frac{1}{N}\sum_{i=1}^{N}\left(Q(I-1)+Q(N-I)\right).$$

Врахувавши, що

$$\sum_{i=1}^{N}\left(Q(I-1)+Q(N-I)\right)=Q(0)+Q(N-1)+Q(1)+Q(N-2)+L\ +Q(N-2)+Q(1)+Q(N-1)+Q(0),$$

отримаємо

$$Q(N)\le C*N+\frac{2}{N}\sum_{i=0}^{N-1}Q(i) \quad .(1)$$

Покажемо за індукцією по **N**, що $Q(N)\le K*N*\ln(N)$ для $N\ge 2$, де **K=2C+2B**, **B=Q(0)=Q(1)**. Останнє співвідношення означає, що Quick Sort вимагає постійної однакової кількості кроків для впорядкування **0** або **1** елемента.

1) **N=2** : $Q(2)=2C+Q(0)+Q(1)=2C+2B=K\le K*2*\ln(2)$;

2) припустимо, що $Q(I)\le K*I*\ln(I)$ для **I=2, 3, ..., N-1** ;

3) перевіримо справедливість для **I=N**. Співвідношення (1) з врахуванням попереднього припущення можна переписати у вигляді

$$Q(N) \le C*N + \frac{2}{N}\left(Q(0)+Q(1)\right) + \frac{2}{N}\sum_{i=2}^{N-1} K*I*\ln(I)$$ або

$$Q(N) \le C*N + \frac{4B}{N} + \frac{2}{N}\sum_{i=2}^{N-1} K*I*\ln(I)$$.(2)

Оскільки функція **I*ln(I)** є опуклою вниз, то для цілих значень аргументу справедлива оцінка

$$\sum_{i=2}^{N-1} I*\ln(I) \le \int_{2}^{N} x\ln(x)dx \le \frac{N^2\ln(N)}{2} - \frac{N^2}{4}$$

Врахувавши останню нерівність, із співвідношення (2) одержимо

$$Q(N) \le C*N + \frac{4B}{N} + \frac{2K}{N}\left(\frac{N^2\ln(N)}{2} - \frac{N^2}{4}\right) = C*N + \frac{4B}{N} + K*N*\ln(N) - \frac{K*N}{2}$$.

Оскільки $N \ge 2$, то

$$C*N + \frac{4B}{N} \le C*N + B*N = \frac{(2C+2B)}{2}N = \frac{K*N}{2}$$.

Остаточно отримаємо

$$Q(N) \le K*N*\ln(N)$$.(3)

Таким чином ефективність алгоритму Quick Sort є величина порядку **O(N*ln(N)).**

Всі наведені викладки справедливі для аналізу по операціях порівняння. Кількість же перестановок залежить від початкового розміщення елементів у послідовності. Характерно, що для цього методу у випадку зворотньо впорядкованого масиву об'єм переміщень ключів не буде максимальним. Адже на кожному етапі ведучий елемент буде найбільшим і опиниться на своєму місці після першого ж порівняння і перестановки, тобто **M=N-1.** Максимальна кількість переприсвоєнь ключів співпадатиме з кількістю порівнянь, мінімальна - рівна нулю.

Алгоритм Quick Sort, як і розглянуті прямі методи, описує процес стійкого сортування.

8.3 Сортування вибором при допомозі дерева – алгоритм Tree Sort

Алгоритм сортування деревом TreeSort власне кажучи є поліпшенням алгоритму сортування вибором. Процедура вибору найменшого елемента удосконалена як процедура побудови так званого сортуючого дерева. Сортуюче дерево - це структура даних, у якій представлений процес пошуку найменшого елемента методом попарного порівняння елементів, що стоять поруч. Алгоритм сортує масив у два етапи.

I етап : побудова сортуючого дерева;

II етап : просівання елементів по сортуючому дереву.

Розглянемо приклад: Нехай масив **A** складається з 8 елементів. Другий рядок складається з мінімумів елементів першого рядка, які стоять поруч. Кожний наступний рядок складений з мінімумів елементів, що стоять поруч, попереднього рядка.

Ця структура даних називається сортуючим деревом. У корені сортуючого дерева розташований найменший елемент. Крім того, у дереві побудовані шляхи елементів масиву від листів до відповідного величині елемента вузла -розгалуження.

Коли дерево побудоване, починається етап просівання елементів масиву по дереву. Мінімальний елемент пересилається у вихідний масив B і усі входження цього елемента в дереві заміняються на спеціальний символ **M**. Потім здійснюється просівання елемента уздовж шляху, відзначеного символом M, починаючи з листка, сусіднього з M і до кореня.

Крок просівання - це вибір найменшого з двох елементів, що зустрілися на шляху до кореня дерева і його пересилання у вузол, відзначений **M**.

a6 = min(M, a6)
a6 = min(a6, a8)
a3 = min(a3, a6)
b2 := a3

Просівання елементів відбувається доти, поки весь вихідний масив не буде заповнений символами **M**, тобто n раз:

For i := 1 to n do begin
Відзначити шлях від кореня до листка символом M;
Просіяти елемент уздовж відзначеного шляху;
B[I] := корінь дерева
end;

Обґрунтування правильності алгоритму очевидно, оскільки кожне чергове просівання викидає в масив **У** найменший з елементів масиву **A**, що залишилися.

Сортуюче дерево можна реалізувати, використовуючи або двовимірний масив, або одномірний масиві **ST[1..N],** де **N = 2n-1**.

Аналіз алгоритму Tree Sort.

Оцінимо складність алгоритму в термінах M(n), C(n). Насамперед відзначимо, що алгоритм Tree Sort працює однаково на усіх входах, так що його складність у гіршому випадку збігається зі складністю в середньому.

Припустимо, що **n** - ступінь **2 (n = 2l)**. Тоді сортуюче дерево має **l + 1** рівень (глибину l). Побудова рівня **I** вимагає **n / 2I** порівнянь і пересилань. Таким чином, **I**-ий етап має складність:

C1(n) = n/2+n/4+ ... + 2+1 = n-1, M1(n) = C1(n) = n - 1

Для того, щоб оцінити складність II-го етапу **C2(n)** і **M2(n)** помітимо, що кожен шлях просівання має довжину **l**, тому кількість порівнянь і пересилань при просіванні одного елемента пропорційно l. Таким чином,

M2(n) = O(l n),
C2(n) = O(l n).

Оскільки **l = log2n, M2(n)=O(n log2 n)), C2(n)=O(n log2 n), Але З(n) = C1(n) + C2(n), M(n) = M1(n) + M2(n)**. Тому що **C1(n) < C2(n), M1(n) < M2(n)**,остаточно одержуємо оцінки складності алгоритму Tree Sort за часом:

(n) = O(n log2 n), C(n) = O(n log2 n),

У загальному випадку, коли **n** не є ступенем 2, сортуюче дерево будується трохи інакше. "Зайвий" елемент (елемент, для якого немає пари) переноситься на наступний рівень. Легко бачити, що при цьому глибина сортуючого дерева дорівнює **[log2 n] + 1**. Удосконалення алгоритму II етапу очевидно. Оцінки при цьому змінюють лише мультиплікативні множники. Алгоритм **Tree Sort** має істотний недолік: для нього потрібно додаткова пам'ять розміру 2n - 1.

8.4 Сортування вибором при допомозі дерева – алгоритм Heap Sort

Прямий вибір - повторюваний пошук найменшого елемента серед **N** елементів, **N-1** елементів, **N-2** і т.д. Кількість порівнянь при цьому **$(N^2-N)/2$**. Для підвищення ефективності необхідно залишати після кожного етапу побільше інформації окрім ідентифікації найменшого ключа.

Після **N/2** порівнянь можна знайти в кожній парі елементів найменший, після **N/4** порівнянь - менший із пари вже вибраних на попередньому кроці і т.д. Виконавши загалом **N/2+N/4+...+2+1=N-1** порівнянь, можна побудувати дерево вибору та ідентифікувати його корінь як шуканий найменший елемент. Наприклад

крок I \/ \/ \/ \/
44 12 06
крок II \/ \/

12 06
крок III \ /
06

На наступному етапі сортування проводиться рух вздовж віток, які відмічені мінімальними елементом, і вилучення його з дерева шляхом заміни на пустий елемент.

44[]
\ / \ / \ / \ /
44 12 18 []
\ / \ /
12 []
\ /
[]

Далі здійснюється заповнення "дірок" у дереві. На першому рівні залишається "дірка" від вилученого елемента, а на наступних знову вибирається менший із двох сусідніх попереднього рівня. "Дірка" при порівнянні вважається як завгодно великим значенням.

44[]
\ / \ / \ / \ /
44 12 18 67
\ / \ /
12 18
\ /
12

Елемент, що опинився в корені, - знову найменший. Після **N** таких кроків дерево стане пустим, в ньому будуть лише одні "дірки" (сортування закінчене). На кожному з **N** етапів виконується **log(N)** порівнянь. Тому на весь процес впорядкування потрібно порядку **N*log(N)** операцій плюс **N-1** операцій для побудови дерева. Це значно краще ніж $\mathbf{N^2}$ для прямих методів і навіть краще ніж $\mathbf{N^{1,2}}$ для алгоритму Шелла. Однак при цьому виникає проблема збереження додаткової інформації. Тому кожен окремий етап в алгоритмі ускладнюється.

Корисно було б, зокрема, позбутися від "дирок", якими вкінці буде заповнене все дерево, і які породжують багато непотрібних порівнянь. Крім того, виникає потреба такої організації даних за принципом дерева, яка б вимагала **N** одиниць пам'яті, а не **2N-1**. Цього вдалося добитися в алгоритмі Heap Sort, який розробив Д. Уілльямс. Він використав спеціальну деревовидну структуру - піраміду.

Піраміда - це означене, тобто задане елементами кореневе бінарне дерево, яке визначається як послідовність ключів $\mathbf{a_L, a_{L+1}, ..., a_R}$, для якої справедливі нерівності

$$a_i \le a_{2i} \text{ та } a_i \le a_{2i+1} \text{ для } i = \overline{L, R/2}. \quad (1)$$

Таким чином бінарне дерево сортувань виду

a_1
/ \
a_2=42a_3=06
/ \ / \
a_4=55a_5=94a_6=18a_7=12

являє собою піраміду, а елемент $\mathbf{a_1}$ буде найменшим в розглядуваній множині : $\mathbf{a_1 = min(a_1, a_2, ..., a_N)}$.

Припустимо, що є деяка піраміда із заданими елементами $\mathbf{a_{L+1}, ..., a_R}$ для певних значень **L** та **R,** і потрібно ввести новий елемент **x**, утворюючи таким чином розширену піраміду $\mathbf{a_L, a_{L+1}, ..., a_R}$. В якості вихідної візьмемо піраміду $\mathbf{a_1, a_2, ..., a_7}$ із попереднього прикладу і добавимо до неї зліва елемент $\mathbf{a_1 = 44}$. Нова піраміда отримується так : спочатку **x** ставиться зверху деревовидної структури, а потім він поступово опускається вниз кожен раз в напрямку меншого з двох прилеглих до нього елементів, а сам цей менший елемент переміщується вгору. Процес просіювання продовжується доти, поки в жодній з прилеглих вершин не буде елемента меншого за нововведеного. В розглядуваному прикладі ключ **44**

спочатку поміняється місцями з ключем **06**, а потім з **12**, і в результаті отримується таке дерево

```
      06
      / \
      42
      / \ / \
       94 18
```

Характерно, що такий метод просіювання залишає незмінними умови (1), які визначають піраміду.

Р. Флойд запропонував певний "лаконічний" алгоритм побудови піраміди "на тому ж місці". Вважається, що деяка частина елементів масиву $\mathbf{a_m, a_2, ..., a_N}$ (**m=Ndiv2**) вже утворює піраміду - нижній шар відповідного бінарного дерева, для них ніякої впорядкованості не вимагається. Тепер піраміда розширюється вліво; кожен раз добавляється і просіюваннями ставитться у відповідну позицію новий елемент. Ці дії реалізуються проседурою Sift :

```
Procedure Sift(L, R : integer);
Var
i, j : integer; x : basetype;
Begin
i:=L; j:=2*L; x:=a[L];
if (j<R) and (a[j+1]<a[j]) then j:=j+1;
while (j<=R) and (a[j]<x) do
begin
a[i]:=a[j]; a[j]:=x; i:=j; j:=2*j;
if (j<R) and (a[j+1]<a[j]) then j:=j+1
end
End;
```

Таким чином, процес формування піраміди із **N** елементів $\mathbf{a_1, ..., a_N}$ "на тому ж місці" є повторюваним виконанням процедури Sift при зміні параметра **L=Ndiv2, ..., 1** :

```
L:=N div 2 +1;
while L>1 do
begin
L:=L-1;
Sift(L, N)
end;
```

Для ілюстації алгоритму розглянемо попередній варіант масиву :

```
44  |
44 |
44 |
44 | 42
06
```

Тут жирним шрифтом виділені добавлювані до піраміди елементи; підкреслені - елементи, з якими проводився обмін.

Для того, щоб отримати не тільки часткове, а і повне впорядкування серед елементів послідовності, потрібно виконати **N** зсувних етапів. Після кожного проходу на вершину дерева виштовхуватиметься черговий найменший ключ. Знову виникає питання : де зберігати "спливаючі" верхні елементи і чи можна проводити перестановки "на тому ж місці"? Це легко реалізувати, якщо кожен раз брати останню компоненту піраміди - це буде просіюваний ключ **x**, ховати верхній елемент з попереднього етапу в звільнене позицію, а **x** зсувати на відповідне місце. Зрозуміло, що після кожного етапу розглядувана піраміда буде скорочуватися на один елемент справа.

Процес сортування описується за допомогою процедури Sift таким чином:

```
R:=N;
```

```
while R>1 do
begin
x:=a[1]; a[1]:=a[R]; a[R]:=x;
R:=R-1;
Sift(1, R)
end;
```

Як видно з прикладу, отриманий порядок ключів фактично є зворотнім. Це легко виправити, помінявши напрямок відношення порівняння в процедурі Sift на протилежний. Остаточно процедура сортування масиву методом Heap Sort матиме вигляд :

```
Procedure Heap_Sort;
Var
L, R : integer; x : basetype;
Procedure Sift(L, R : integer);
Var
i, j : integer; x : basetype;
Begin
i:=L; j:=2*L; x:=a[L];
if (j<R) and (a[j]<a[j+1]) then j:=j+1;
while (j<=R) and (x<a[j]) do
begin
a[i]:=a[j]; a[j]:=x; i:=j; j:=2*j;
if (j<R) and (a[j]<a[j+1]) then j:=j+1
end
End;
Begin
L:=N div 2 +1; R:=N;
while L>1 do
begin L:=L-1; Sift(L, N) end;
while R>1 do
begin
x:=a[1]; a[1]:=a[R]; a[R]:=x;
R:=R-1;
Sift(1, R)
end
End;
```

Аналіз алгоритму Heap Sort. Як вже раніше відмічалося, складність алгоритму по операціях порівняння є величиною порядку **O(N*log(N)+N)**. Кількість переміщень елементів суттєво залежить від стартового розміщення ключів в послідовності.

Контрольні запитання

1. Опишіть метод сортування включенням із зменшуваними відстанями – алгоритм Шелла.
2. Сформулюйте основні етапи алгоритму Шелла.
3. В чому полягає сортування обміном на великих відстанях?
4. У чому суть методу сортування вибором при допомозі дерева – алгоритм Tree Sort?

ЛЕКЦІЯ 9. ОСНОВНІ ЕВРИСТИЧНІ АЛГОРИТМИ

Метод розгалужень і меж. Евристичні алгоритми. Застосування принципу оптимальності.

9.1. Метод розгалужень і меж

Обхід усіх вузлів дерева пошуку варіантів може виявитися надто довгим. Наприклад, якщо в дереві всі вузли є допустимими, кожний проміжний вузол має m синів, а глибина дерева n, то всього в дереві $1+m+m^2+ \dots +m^n=(m^{n+1}-1)/(m-1)$ вузлів. Уже за $m=10$ та $n=10$ це більш, ніж 10^{10}. Якщо припустити, що комп'ютер здатний обробити 10^5 вузлів за секунду, то обхід такого дерева триватиме 10^5 секунд, або приблизно добу.

Існує багато практичних задач, де вимагається відшукати чи побудувати не всі можливі варіанти, а лише один із них, найкращий у деякому розумінні, визначеному в задачі. Отже, тут з'являється таке поняття, як ***цінність*** варіантів. Загальним принципом розв'язання таких задач є скорочення обходу дерева варіантів. У ньому відкидаються деякі гілки, про які можна стверджувати, що вони не містять варіантів більш цінних, ніж уже знайдені. Розглянемо приклад.

Задача про три процесори. Нехай є три процесори, здатні виконувати завдання з однаковою швидкістю. Є набір завдань, про кожне з яких відомий час його виконання. Порядок виконання завдань неважливий. Якщо процесор почав виконувати завдання, то виконує його до кінця протягом зазначеного часу. Переключення процесора на виконання нового завдання відбувається миттєво. Треба так розподілити завдання між процесорами, шоб момент закінчення останнього завдання був мінімальним. Назвемо цю величину ***вартістю розподілу***. Отже, займемося обчисленням мінімальної вартості серед можливих розподілів. Сам розподіл, що забезпечує таку вартість, для початку нас не цікавитиме.

Приклад. Нехай є 6 завдань, час виконання яких відповідно 7, 8, 9, 10, 11, 12. Якщо в зазначеному порядку розподілити перші три завдання між процесорами, а потім давати їх у тому ж порядку процесорам, що звільняються, то перший процесор закінчить роботу в момент 7+10=17, другий – у момент 8+11=19, а третій – 9+12=21. Маємо вартість 21. Проте їх можна розподілити інакше – 7+12, 8+11, 9+10, одержавши вартість 19.←

Перше, що ми зробимо в розв'язанні задачі – упорядкуємо завдання за незростанням часу їх виконання. Отже, нехай $P_1, \dots , P_n$ – завдання, часи виконання $T_1, \dots , T_n$ яких задовольняють нерівності $T_1 \ge \dots \ge T_n$. Розподіл можна подати послідовністю пар вигляду $(i; k)$, де i – номер завдання, k – номер процесора, на якому воно виконується. Наприклад, за часів 12, 11, 10, 9, 8, 7 найкращий розподіл подається як

$$<(1; 1), (2; 2), (3; 3), (4; 3), (5; 2), (6; 1)>.$$

Подібно до розміщень ферзів, можна говорити про повний розподіл – довжини n, та неповний – меншої довжини. Так само утворимо дерево пошуку розподілів. Його коренем є порожній розподіл, синами кореня – три розподіли $<(1; 1)>$, $<(1; 2)>$, $<(1; 3)>$ тощо, тобто синами кожного розподілу вигляду

$$v=<(1; k_1), \dots , (i; k_i)>$$

за $i<n$ є три розподіли

$$v_1=<(1; k_1), \dots , (i; k_i), (i+1; 1)>,$$
$$v_2=<(1; k_1), \dots , (i; k_i), (i+1; 2)>,$$
$$v_3=<(1; k_1), \dots , (i; k_i), (i+1; 3)>.$$

Повні розподіли є листками вигляду $<(1; k_1), \dots , (n; k_n)>$.

Тепер займемося упорядкуванням обходу дерева таким чином, щоб варіанти з меншою вартістю оброблялися якомога раніше, а варіанти з більшою вартістю – якомога пізніше. За розподілом $v=<(1; k_1), \dots , (i; k_i)>$, де $i \le n$, неважко обчислити трійку часів роботи процесорів (S_1, S_2, S_3) з його виконання. Очевидно, його вартістю є найбільше з S_1, S_2, S_3. Такий розподіл за $i<n$ та час T_{i+1} дають три варіанти трійок, відповідних його розподілам-синам v_1, v_2, v_3:

(S_1+T_{i+1}, S_2, S_3), (S_1, S_2+T_{i+1}, S_3), (S_1, S_2, S_3+T_{i+1}).

За $i+1=n$ неважко вибрати найменшу з цих трьох вартостей. Проте за $i+1<n$ нас будуть цікавити не стільки вартості цих неповних розподілів, скільки нижні оцінки вартості тих

повних розподілів, які з них можна одержати. Цією оцінкою є *вартість, менше якої не може бути вартість повних розподілів*.

Розглянемо найпростіший спосіб такого оцінювання. Очевидно, що за неповного розподілу *v* перших *i* завдань із трійкою часів (S_1, S_2, S_3) всі розподіли, що є його нащадками, мають вартість не меншу, ніж

$$E(v)=\max\{S_1, S_2, S_3, \min\{S_1, S_2, S_3\}+T_{i+1}\}.$$

Отже, оцінка $E(v)$ є ***нижньою межею*** для вартості нащадків розподілу *v*.

Організуємо обхід дерева розподілів таким чином, що:

1. для кожного з вузлів обчислюється зазначена оцінка вартості,
2. вузли розглядаються у порядку зростання їх оцінок,
3. вузли з оцінкою, більшою від вартості вже одержаного повного розподілу, взагалі не розглядаються.

Ці міркування складають суть ***методу розгалужень і меж***. Упорядкування вузлів робить обхід цілеспрямованим, а відкидання явно неперспективних піддерев скорочує його.

Уточнимо організацію даних для обробки вузлів у зазначеному порядку. Оскільки нас цікавлять не самі розподіли, а лише їх вартість, у вузлах дерева будемо зберігати тільки трійку часів та номер завдання, розподіленого останнім. Маючи список часів $T[1], \dots, T[n]$ обробки завдань, неважко за цими даними обчислити оцінку вартості для неповних розподілів та саму вартість для повних. Для наочності цю величину також зберігатимемо у вузлі. Отже, вузол дерева подається трійкою часів $S[1]$, $S[2]$, $S[3]$, номером завдання *i* та оцінкою вартості *E*, яка за $i<n$ обчислюється як

$$\max\{ S[1], S[2], S[3], \min\{ S[1], S[2], S[3]\}+T[i+1]\}.$$

Очевидно, що за $i=n-1$ ця величина є вартістю повного розподілу, який подається "кращим із синів" цього вузла дерева.

Проміжні вузли записуються не в магазин, а в чергу, елементи якої упорядковано за зростанням оцінок вартості. Таким чином, для подання черги зручно скористатися лінійним списком (п.16.3.3). Вузли, відповідні повним розподілам, в чергу не записуються, оскільки оцінка вартості є власне їх вартістю.

Очевидно, що спочатку з трьох розподілів <(1;1)>, <(1;2)>, <(1;3)> в чергу достатньо записати лише один, для визначеності <(1; 1)>. Очевидно також, що коли обробляється вузол із однаковими часами $S[1]$, $S[2]$, $S[3]$, то з трьох його синів до черги достатньо додати лише одного. Якщо ж два з трьох часів у вузлі рівні, то до черги не додається один із двох синів, що відрізняються лише порядком часів.

Опишемо обробку вузлів дерева таким алгоритмом.

Занести до черги розподіл (T[1], 0, 0; 1; T[1]);
Cmin:=∞ ;
while (*черга не порожня*) **and** (*її перший елемент має оцінку* E<Cmin)
do
begin
Вилучити з черги її перший елемент Node=(S[1], S[2], S[3]; i; E);
if i=n-1 **then** {синами вузла є листки}
Обчислити вартість синів вузла Node *та за необхідності запам'ятати нову поточну мінімальну вартість* Cmin
else
Обчислити оцінку вартості синів вузла Node *та додати до черги лише тих із них, чия оцінка не більше* Cmin
end

Уточнення цього алгоритму залишаємо вправою.

Розглянемо приклад обчислення мінімальної вартості розподілу за наведеним алгоритмом. Нехай задано час виконання п'яти завдань 9, 8, 7, 5, 4. Очевидно, що найкращий розподіл (9, 8+4, 7+5) має вартість 12. Значення *Cmin* та зміст черги, що виникають за наведеним алгоритмом, подамо таблицею:

C	**Черга**

min	
∞	<9,0,0; 1; 9>
∞	<9,8,0; 2; 9> <17,0,0; 2; 17>
∞	<9,8,7; 3; 12> <9,15,0; 3; 15> <16,8,0; 3; 16> <17,0,0; 2; 17>
∞	<9,8,12; 4; 12> <9,13,7; 4; 13> <9,8,11; 4; 13> <9,15,0; 3; 15> <16,8,0; 3; 16> <17,0,0; 2; 17>
12	<9,13,7; 4; 13> <9,8,11; 4; 13> <9,15,0; 3; 15> <16,8,0; 3; 16> <17,0,0; 2; 17>

Як бачимо, перший елемент черги має оцінку вартості, гіршу за *Cmin*, тому подальше дослідження дерева варіантів не відбувається. За виконання алгоритму до черги додається 9 проміжних вузлів, а вилучається 4. Між тим, неважко підрахувати, що з урахуванням симетричних варіантів дерево містить 19 проміжних вузлів. Фактично, ми одержали потрібний розподіл взагалі без перебирання варіантів.

У загальному випадку *метод розгалужень і меж не позбавляє перебирання*. У цьому неважко переконатися, імітувавши наведений алгоритм на прикладі часів виконання завдань (12, 8, 7, 5, 4, 2).

Задача про розподіл завдань представляє чималу групу задач, які розв'язуються методом розгалужень і меж. Подивимося на цю задачу більш узагальнено. Розподіл (повний чи частковий) $v(i)=<(1; k_1), \dots, (i; k_i)>$ подамо як послідовність $<a_1, a_2, \dots, a_i>$, де a_j позначає пару $(j; k_j)$. Очевидно, що $v(i)$ одержується з $v(i\text{-}1)$ додаванням компонента a_i. Вартість розподілу при цьому не зменшується, тобто

$$C(v(i\text{-}1)) \le C(v(i)). \quad (19.1)$$

Існує чимало задач, в яких розв'язок-послідовність $<a_1, a_2, \dots, a_n>$ будується шляхом "нарощування" часткових розв'язків $<a_1, a_2, \dots, a_{i-1}>$ новими компонентами a_i. Умова (19.1) дозволяє відкидати ті часткові розв'язки та всіх їх нащадків, якщо їх вартість не може бути меншою вартості *Cmin* уже побудованого повного розв'язку. Таким чином, *Cmin* виступає *верхньою межею* для вартості розв'язків, які є сенс будувати. Але, як правило, обчислити вартість повного розв'язку можна лише після його побудови. Для запобігання побудови всіх повних розв'язків треба мати можливість *оцінювати знизу* їх вартість за вартістю побудованих часткових. Чим точнішою буде така оцінка, тим ефективнішим буде скорочення перебору.

Отже, алгоритм розв'язання багатьох задач за методом розгалужень і меж має таку загальну структуру:

Для кожного можливого a_1 *занести до черги частковий розв'язок* $<a_1>$;
Обчислити нижню оцінку E *вартості його нащадків, що є повними розв'язками*;
Cmin:=∞ *;*
while (*черга не порожня*) **and** (*її перший елемент має оцінку* E<Cmin)
do
begin
Вилучити з черги її перший елемент Node;
if *синами вузла* Node *є листки* **then**
Обчислити вартість синів Node *та за необхідності запам'ятати нову поточну мінімальну вартість* Cmin

else

Обчислити оцінку вартості синів вузла Node *та*
додати до черги лише тих із них, чия оцінка не більше Cmin

end.

9.2. Евристичні алгоритми

Повернемося до задачі про розподіл завдань по трьох процесорах і спробуємо розв'язати її у зовсім інший спосіб.

Нехай ми маємо неповний розподіл (S_1, S_2, S_3) усіх завдань, крім останнього. У цьому випадку найкраще розподілити останнє завдання, додавши його час до найменшого з S_1, S_2, S_3, тобто передати його до найменш завантаженого процесора.

Тепер правилом "*передати чергове завдання до найменш завантаженого процесора*" будемо керуватися при розподілі *кожного з завдань*. Застосування цього правила виражається алгоритмом, за яким завдання розподіляються без будь-якого перебирання варіантів:

розподілити перші три завдання по одному на процесор;
for i:=4 **to** n **do**
begin
обчислити k – *номер найменшого з* S[1], S[2], S[3];
додати T[i] *до* S[k]
end

За цим алгоритмом завдання (12, 8, 7, 5, 4) розподіляються як (12, 8+4, 7+5). Очевидно, що краще не може бути.

Проте розподіл завдань за цим алгоритмом не завжди є найкращим. Наприклад, завдання (12, 8, 7, 5, 4, 2) розподіляються за ним як (12+2, 8+4, 7+5) з вартістю 14, хоча є кращий розподіл (12, 8+5, 7+4+2) з вартістю 13.

Правило "*передати чергове завдання до найменш завантаженого процесора*", яким ми керувалися при розподілі завдань, є прикладом ***евристики***. Взагалі, значенням цього слова є "мистецтво відшукання істини", а в інформатиці ***евристика*** – це правило, метод або прийом, призначений для підвищення ефективності пошуку розв'язку задачі [Сл].

Алгоритм, побудований на основі застосування евристики, називається ***евристичним***. Як правило, евристичні алгоритми дозволяють швидко побудувати розв'язок задачі, але не завжди гарантують, що він дійсно буде найкращим.

Приклад 1. Розглянемо ще одну задачу та дві евристики для неї. Нехай, як і раніше, задано упорядкований за незростанням список часів виконання завдань T_1, T_2, … , T_n, але кількість процесорів не фіксовано. Замість цього задано час T_0, і треба визначити найменшу кількість процесорів, яка забезпечує виконання всіх завдань у межах T_0. Зрозуміло, що $T_{0\geq} T_1$.

Спочатку переформулюємо цю задачу в інших термінах. Час виконання завдання можна розглядати як об'єм предмету, а час T_0 – як об'єм ящиків, по яких розподіляються предмети (форма ящиків та предметів неважлива). Отже, треба обчислити найменшу кількість ящиків, необхідних для розподілу всіх предметів. Тепер сформулюємо дві евристики.

Е1. "*Перший прийнятний*". Перший предмет кладемо в перший ящик. Другий також, якщо він там уміщається. Якщо не уміщається, то кладемо його в другий ящик. Взагалі, *черговий предмет кладемо в ящик із найменшим номером, в якому він уміщається*.

Е2. "*Найкращий прийнятний*". Черговий предмет кладеться в той ящик, у якому залишається *найменший ще допустимий незайнятий об'єм*. Якщо таких ящиків кілька, то з них вибираємо ящик із найменшим номером.

Запис відповідних евристичних алгоритмів залишаємо вправою.

9.3. Застосування принципу оптимальності

Знайомство з принципом оптимальності почнемо з розв'язання задачі.

Приклад 2. Нехай паперовий прямокутник складено з клітин – m по вертикалі та n по горизонталі. У кожній клітині записано натуральне число. Уявімо, що з прямокутника зробили вертикальний циліндр, з'єднавши першу та останню вертикалі. Ми можемо рухатися по клітинах циліндра та підраховувати суму чисел у них. Рух починається з будь-якої

клітини першого кільця. Далі, якщо ми перебуваємо в якійсь клітині, то можемо перейти на наступне кільце в одну з тих трьох клітин, що мають спільні точки з поточною. Рух закінчується на останньому, m-му кільці клітин. Треба обчислити найбільшу суму, яку можна набрати на одному з можливих шляхів довжини m.

Якщо m=1, то достатньо вибрати клітину з найбільшим числом. Нехай m>1. Занумеруємо клітини кожного кільця числами від 0 до n-1. Позначимо через C_{ki} число, записане в клітині з номером i у кільці k, а через S_{ki} – найбільшу суму, яку можна набрати на шляху, що веде в цю клітину. Очевидно, що $S_{1i} = C_{1i}$. Для початку обчислимо для кожної клітини другого кільця найбільшу суму S_{2i} на шляху довжини 2. За умовою задачі очевидно, що

$S_{2i}=C_{2i}+\max\{S_{1,\,i-1}, S_{1i}, S_{1,\,i+1}\}$ за i=1, … , n-2,

$S_{20}=C_{20}+\max\{S_{1,\,n-1}, S_{10}, S_{11}\}$, $S_{2,n-1}=C_{2,\,n-1}+\max\{S_{1,\,n-2}, S_{1,\,n-1}, S_{10}\}$.

За цими сумами можна аналогічно підрахувати суми для клітин третього кільця. Так само при переході до четвертого кільця достатньо знати лише найбільші суми для клітин третього кільця тощо. Дістявши суми для клітин останнього кільця, вибираємо найбільшу з них, і задачу розв'язано.

Уточнення алгоритму залишаємо вправою. Скажемо лише, що суми S_{ki}, k=2, … , m, i=0, … , n-1, обчислюються за єдиною формулою

$S_{ki}=C_{ki}+\max\{S_{k-1,\,(i-1+n)\bmod n}, S_{k-1,\,i}, S_{k-1,\,(i+1)\bmod n}\}$.

Оцінимо складність наведеного алгоритму. Очевидно, що при переході на наступне кільце обчислюються n сум за сталу кількість дій кожна. Таких переходів відбувається m-1, тому загальна кількість дій оцінюється як $O(nm)$.←

У наведених обчисленнях сум ми керувалися правилом: *при переході на наступне кільце неважливо, якими були шляхи до клітин попереднього кільця. Аби вони давали найбільші суми, можливі для їх кінцевих клітин*. Ішими словами, *вибір шляхів від клітин попереднього кільця в клітини наступного не залежить від того, як саме ми вибирали клітини раніше*.

Наведене правило є окремим конкретним випадком ***принципу оптимальності***, одного з головних у теорії ***динамічного програмування***. Її автор, Р.Беллман, сформулював цей принцип так:

"Оптимальна поведінка має таку властивість, що, якими б не були початковий стан і рішення в початковий момент, наступні рішення повинні складати оптимальну поведінку відносно стану, який одержується в результаті першого рішення."

Обсяг книжки не дозволяє викладати тут теорію динамічного програмування. Вона велика й серйозна. Наведемо натомість ще один приклад застосування принципу оптимальності.

Приклад 3. Розглянемо обчислення добутку n матриць

$A = A_1 \times A_2 \times \ldots \times A_n$,

де кожна A_i – матриця з s_{i-1} рядками та s_i стовпцями. Як відомо, операція множення матриць є асоціативною, і результат не залежить від порядку її застосування. Але від нього залежить кількість множень їх елементів.

За традиційним алгоритмом множення матриць розмірами $a\times b$ та $b\times c$ відбувається abc множень їх елементів. Наприклад, множення матриць $A_1\times A_2\times A_3$ розмірами 100× 1, 1× 100, 100× 1 відповідно у порядку $(A_1\times A_2)\times A_3$ вимагає 100· 1· 100+100· 100· 1=20000 множень, тоді як у порядку $A_1\times (A_2\times A_3)$ – лише 1· 100· 1+100· 1· 1=200, тобто в 100 разів менше.

Отже, за послідовністю розмірів матриць s_0, s_1, s_2, … , s_n треба обчислити найменшу кількість множень їх елементів, необхідних для обчислення добутку матриць $A = A_1 \times A_2 \times \ldots \times A_n$.

Очевидно, що при обчисленні добутку останнім виконується одне з множень, тобто $A=(A_1\times \ldots \times A_i)\times (A_{i+1}\times \ldots \times A_n)$, де $1\le i\le n$-1. Якщо добутки $A_1\times \ldots \times A_i$ та $A_{i+1}\times \ldots \times A_n$ обчислено, то виконання останнього множення вимагає $s_0\cdot s_i\cdot s_n$ множень. Позначимо m_{ik} мінімальну кількість множень, необхідних для обчислення $A_i\times A_{i+1}\times \ldots \times A_k$ за $i<k$, m_{ii}=0. Тоді шукана кількість

$m_{1n} =$ $\{m_{1i}+m_{i+1,n}+s_0.\, s_i.\, s_n\}$

Отже, задача відшукання m_{1n}, тобто задача розміру n, зводиться до $2(n\text{-}2)$ задач меншого розміру. Але головним тут є той факт, що числа m_{1i} та $m_{i+1,n}$

не залежать одне від одного, тобто найменша кількість множень при обчисленні добутку $A_{1\times} \ldots\times A_i$ не залежить від того, як обчислюється добуток $A_{i+1\times} \ldots\times A_n$, і навпаки. *Завдяки саме цій незалежності можна застосувати принцип оптимальності*.

Спочатку обчислимо всі $m_{i,i+1}$ за i=1, … , n-1. Очевидно, $m_{i,i+1}=s_{i-1}.\, s_i.\, s_{i+1}$. Запам'ятавши їх, обчислимо $m_{i,i+2}$ і також запам'ятаємо. Потім обчислимо всі $m_{i,i+3}$ тощо, збільшуючи різницю d між другим та першим індексами, поки не дійдемо до m_{1n}. При цьому ми обчислюємо m_{ij} за формулою

$m_{ij} =$ $\{m_{ik}+m_{k+1,j}+s_{i-1}.\, s_k.\, s_j\}$

Наведений алгоритм уточнюється таким чином:

```
for i:=1 to n-1 do m[i, i+1]:=s[i-1]*s[i]*s[i+1];
for d:=1 to n-1 do
for i:=1 to n-d do
begin
j:=i+d;
```

У m[i, j] *запам'ятати мінімальне зі значень*

m[i,k]+m[k+1,j]+s[i-1]*s[k]*s[j] *по всіх* k=i+1, …, j-1

```
end
```

{m[1, n] – *шукане значення*}

Безпосередньо за алгоритмом неважко переконатися, що оцінкою його складності є $O(n^3)$.←

Підведемо підсумок. В обох прикладах ми будували послідовності – шляхи на циліндрі або послідовності дужок. Характерним для них є те, що, кажучи неформально, коли зафіксовано якийсь компонент у їх середині, то оптимальний вибір компонентів у початку не впливає на оптимальний вибір у кінці, і навпаки. Саме ця незалежність позбавляє необхідності перебирати всі можливі послідовності і забезпечує складність наведених алгоритмів порядку $O(mn)$ та $O(n^3)$ відповідно.

У задачі про три станки такої незалежності рішень на початку їх послідовності та в її кінці немає. Саме це змушує перебирати всі можливі послідовності та зумовлює *незастосовність принципу оптимальності*. Для цієї задачі немає алгоритмів, які б дозволяли будувати розв'язок із незалежних частин подібно до задачі про добуток матриць.

Існує величезний клас задач, розв'язки яких є послідовностями заданого вигляду, причому їх початок і кінець взаємозалежні. Для таких задач побудовано алгоритми складності не менше $O(2^n)$, де n – це величина, що характеризує розмір вхідних даних задачі. Але для них досі не побудовано алгоритмів, складність яких можна було б оцінити поліноміальною функцією від n. Поки що не доведено, що *таких алгоритмів узагалі не можна побудувати*, але саме до такої думки схиляються майже всі, хто мав справу з цими задачами.

Серед задач, розв'язок яких будується перебиранням варіантів, виділяються так звані *NP-складні та NP-повні задачі*. Обсяг і характер цієї книжки не дозволяють розпочинати знайомство з ними, тому зацікавлений читач може подивитися в книги [АХУ, РНД, ГД].

Контрольні запитання

1. Опишіть у чому полягає суть методу розгалужень і меж.
2. Сформулюйте основні принципи евристичних алгоритмів.
3. Застосування принципу оптимальності до розв’язування задач. Описати на прикладі множення матриць

ЛІТЕРАТУРА

1. Акимов О. Е. Дискретная математика. – Москва : Лаборатория Базовых Знаний. 2001. – 352 с.

2. Андерсон Дж. А. Дискретная математика и комбинаторика: пер. с англ. – М. : Вильямс, 2003. – 960 с.

3. Бардачов Ю. М. Дискретна математика /
Ю. М Бардачов, Н. А. Соколова, В. Є. Ходаков – К. : Вища школа, 2002. – 287 с.

4. Бондаренко М. Ф. Комп'ютерна дискретна математика / М. Ф. Бондаренко, Н. В. Білоус, А. Г. Руткас – Харків : Компанія СМІТ, 2004. – 480 с.

5. Босс В. Лекции по математике. Т. 6. От Диофанта до Тьюринга. – М. : КомКнига, 2006. – 208 с.

6. Лекции по математической логике и теории алгоритмов. Часть 3. Вычислимые функции / Н. К. Верещагин, А. Шень 3-е изд. – М. : МЦНМО, 2008. – 192 с.

7. Гусев С. С. Логика / С. С.Гусев, Э. Ф. Караваев,
Г. В. Карпов – Москва : Проспект, 2011. – 675 с.

8. Гринченков Д. В. Математическая логика и теория алгоритмов для программистов. – Москва : Кно Рус,
2010. – 208 с.

9. Євладенко В. М. Математична логіка та теорія алгоритмів. – Кіровоград : КОД, 2009. – 116 с.

10. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М. : Лаборатория Базових Знаний, 2001. –
288 с.

11. Игошин В. И. Математическая логика и теория алгоритмов : учеб. пособие для студ. высш. учеб. заведе-ний. - М. : Издательский центр «Академия», 2008. — 448 с.

12. Клини С. К. Математическая логика. – Москва. : Мир, 1972. – 480 с.

13. Алгоритмы: построение и анализ, 2-е издание / Т. Кормен, Ч. Лейзерсон, Р. Риверст, К. Штайн – М. : Вильямс, 2005. – 1296 с.

14. Колмогоров А. Н. Введение в математическую логику / А. Н. Колмогоров, А. Г. Драгалин – Москва. : Издательство Московского университета, 1982. – 120 с.

15. Кривий С. Л. Вступ до некласичної математичної логіки. – Київ : ВПЦ "Київський університет", 2010. –
205 с.

16. Лавров И. А. Задачи по теории множеств, математической логике и теории алгоритмов / И. А. Лавров, Л. Л. Максимова – М. : ФИЗМАТЛИТ, 2004. – 256 с.

17. Лиман Ф .М. Математична логіка і теорія алго-ритмів. – Суми : Вид-во „Слобожанщина", 1998. – 152 с.

18. Лихтарников Л. М. Математическая логика. –СПб; Москва; Краснодар : Лань, 2008. – 288 с.

19. Любченко К. М. Елементи математичної логіки з комп'ютерною підтримкою. Черкаси :ЧНУ,2004. – 87 с.

20. Мендельсон Э. Введение в математическую логику. – Москва : Наука, 1971. – 320 с.

21. Набебин А. А. Математическая логика и теория алгоритмов. – Москва : Научный мир, 2008. – 343 с.

22. Нікітченко М. С. Математична логіка та теорія алгоритмів. – Київ : ВПЦ, "Київський університет", 2008. – 528 с.

23. Трохимчук Р. М. Збірник задач і вправ з математичної логіки. – Київ : Персонал, 2008. – 114 с.

24. Теория алгоритмов: основные открытия и приложения/ В. А. Успенський, А. Л. Семенов – М. : Наука, 1987. – 245 с.

25. Фейс Р. Модальна логіка. – Москва : Наука,1974. – 520 с.

26. Математическая логика и автоматическое даказательство теорем/ Ч. Чень, Р. Ли – Москва : Наука, 1983. – 256 с.

27. Шапорев С. Д. Математическая логика. – СПб : БХВ – Петербург, 2005. – 405 с.

28. Шкільняк С. С. Математична логіка. Основи теорії алгоритмів. – Київ : Персонал, 2009. – 280 с.

29. Шкільняк С. С. Математична логіка: приклади і задачі. – Київ : ВПЦ "Київський університет", 2002. – 56 с.

А4.02 Алгоритми та структури даних [Текст] :конспект лекцій для студентів спеціальності 121 "Інженерія програмного забезпечення" / уклад. В.О.Ліщина. – Луцьк :Луцький НТУ, 2016. – 74 с.

Комп'ютерний набір В.О.Ліщина

Редактор В.О.Ліщина

Підп. до друку 2016р.

Формат 60х84/16. Папір офс. Гарнітура Таймс.

Ум. друк. арк. ___. Обл.-вид. арк. 2,5.

Тираж ___ прим. Зам. 1.

Редакційно-видавничий відділ

Луцького національного технічного університету

43018 м. Луцьк, вул. Львівська, 75

Друк – РВВ Луцького НТУ