

С.М. Прийма

МАТЕМАТИЧНА ЛОГІКА І ТЕОРІЯ АЛГОРИТМІВ

Навчальний посібник

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів
вищих педагогічних навчальних закладів

Мелітополь
2010

УДК 519.683. (075.8)

ББК 32.973-01 я 73

П 15

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів вищих
педагогічних навчальних закладів
(лист №1.4/18-Г-1193 від 26.05.2008 р.)

Рецензенти:

завідувач кафедри прикладної математики і обчислювальної техніки
Таврійського державного агротехнологічного університету
д.т.н., професор **Найдиш А.В.**

завідувач кафедри інформатики і математики
Таврійського національного університету ім. В.І. Вернадського,
к. ф.-м. н., доцент **Кошелева Т. М.**

завідувач кафедри інформатики і кібернетики
Мелітопольського державного педагогічного університету,
д.т.н., професор **Єремєєв В.С.**

Прийма С.М. Математична логіка і теорія алгоритмів:
П 15 Навчальний посібник – Мелітополь: ТОВ „Видавничий
будинок ММД”, 2010. - 196 с.

ISBN 978-966-8563-84-3

У посібнику викладено основні розділи математичної логіки і теорії алгоритмів. Теоретичний матеріал ілюстровано численними прикладами та доповнено практичним курсом, що дозволяє закріпити вивчений матеріал.

Навчальний посібник призначено для студентів вищих педагогічних навчальних закладів.

УДК 519.683. (075.8)

ББК 32.973-01 я 73

ISBN 978-966-8563-84-3

© Прийма С.М., 2010

ЗМІСТ

ВСТУП	5
РОЗДІЛ 1. МАТЕМАТИЧНА ЛОГІКА.....	11
1.1. АЛГЕБРА ЛОГІКИ.....	11
1.1.1. Алгебра висловлень.....	11
1.1.1.1. Поняття висловлення. Види висловлень.	11
1.1.1.2. Логічні операції над висловленнями.....	12
1.1.1.3. Формули алгебри висловлень. Обчислення значень формул.....	18
1.1.1.4. Рівносильні, тотожно істинні (ТІ) і тотожно хибні (ТХ) формули алгебри логіки. Закони логічних операцій. Закон подвійності.	20
1.1.1.5. Рівносильні перетворення формул.....	23
1.1.1.6. Проблема розв'язуваності в алгебрі висловлень і нормальні форми	25
1.1.1.7. Функції алгебри висловлень	37
1.1.2. Алгебра предикатів.....	39
1.1.2.1. Поняття предиката	39
1.1.2.2. Логічні операції над предикатами.....	42
1.1.2.3. Кванторні операції.....	44
1.1.2.4. Поняття формули алгебри предикатів	48
1.1.2.5. Значення формули логіки предикатів	49
1.1.2.6. Рівносильні формули логіки предикатів.....	49
1.1.2.7. Загальнозначущі формули логіки предикатів	51
1.1.2.8. Загальнозначущість і здійсненність формул. Проблема можливості розв'язання.....	52
1.2. ЛОГІЧНІ ЧИСЛЕННЯ	54
1.2.1. Аксиоматичний метод.....	54
1.2.2. Числення висловлень	59
1.2.2.1. Поняття формули числення висловлень.....	59
1.2.2.2. Визначення доказової (виведеної) формули	61
1.2.2.3. Проблеми аксіоматичного числення висловлень.....	62
1.2.3. Числення предикатів	64

РОЗДІЛ 2. ТЕОРІЯ АЛГОРИТМІВ.....	67
2.1. ТЕОРІЯ АЛГОРИТМІВ. ОСНОВНІ ПОЛОЖЕННЯ ТА ОЗНАЧЕННЯ ТЕОРІЇ АЛГОРИТМІВ	67
2.1.1. <i>Поняття про алгоритм. Еволюція поняття алгоритм.....</i>	67
2.1.1.1. Властивості алгоритмів	68
2.1.1.2. Вимоги до алгоритмів	68
2.1.2. <i>Підходи до визначення алгоритму</i>	69
2.1.2.1. Алгоритм як обчислювальна функція.....	69
2.1.2.2. Алгоритмічні моделі на основі детермінованих пристроїв	70
2.1.2.3. Нормальні алгоритми Маркова	70
2.2. АЛГОРИТМІЧНІ МОДЕЛІ	71
2.2.1. <i>Обчислювальні функції</i>	71
2.2.1.1. Поняття про обчислювальну функцію.....	71
2.2.1.2. Примітивно-рекурсивні функції.....	75
2.2.1.3. Частково-рекурсивні функції.....	75
2.2.1.4. <i>Теза Черча.....</i>	75
2.2.2. <i>Алгоритмічні моделі на основі детермінованих пристроїв.....</i>	76
2.2.2.1. Фінітний комбінаторний процес Поста	76
2.2.2.2. Абстрактна обчислювальна машина Тьюрінга... ..	79
2.2.2.3. <i>Машини з довільним доступом</i>	86
2.2.3. <i>Теорія нормальних алгорифмів Маркова</i>	88
2.2.4. <i>Еквівалентність алгоритмічних моделей</i>	92
2.3. АЛГОРИТМІЧНО НЕРОЗВ'ЯЗУВАНІ ПРОБЛЕМИ	95
2.4. СКЛАДНІСТЬ АЛГОРИТМІВ	99
2.4.1. <i>Поняття про складність алгоритмів.....</i>	99
2.4.2. <i>Асимптотична часова складність алгоритмів</i>	122
2.5. МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ	131
2.5.1. <i>Декомпозиція</i>	131
2.5.2. <i>Метод розгалужень і меж</i>	137
2.5.3. <i>Динамічне програмування</i>	143
2.5.4. <i>Евристичні алгоритми.....</i>	152
ПРАКТИЧНИЙ КУРС.....	155
МОДУЛЬНО-ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ.....	169
ЛІТЕРАТУРА	184
ПРЕДМЕТНИЙ ПОКАЗЧИК	188

ВСТУП

Математика є наукою, в якій усі твердження доводяться за допомогою доказових міркувань, тобто шляхом використання законів людського мислення. Ось чому для математики велике значення мають логічні теорії як засоби побудови математичних знань. Вивчення законів людського мислення і різних логічних теорій є предметом логіки.

Як самостійна наука, логіка вперше була сформульована в працях грецького філософа Аристотеля (384-322 р.р. до н.е.). Він систематизував відомі до нього відомості, і ця система стала згодом називатися *формальною* або *Аристотелевою логікою*.

У логічних теоріях описуються процеси міркувань і закони мислення, що дозволяють з істинності одних суджень робити висновок про істинність або хибність інших суджень тільки на підставі форм цих суджень, незалежно від їхнього конкретного змісту.

Формальна логіка проіснувала без серйозних змін більше двадцяти століть. Проте, починаючи з другої половини ХІХ століття для науки взагалі, і математики зокрема, вже було недостатньо такої логіки. У зв'язку зі своєю неточністю формальна логіка не давала можливості чітко описати закони і правила елементарних видів мислення. Виникла потреба в ефективних методах розпізнавання законів і методів мислення. Природно, що математика виявила недостатність Аристотелевої логіки і стимулювала подальший її розвиток.

Застосування математичних методів до логічних проблем призвело до зародження нової науки – *математичної логіки*. Імпульсом до використання методів математики у логіці була потреба описати закони людського мислення точною та суворого математичною мовою, щоб зробити їх справжнім знаряддям наукового дослідження. Характерним для математичної логіки є не тільки вивчення законів і форм мислення, але й використання „мови символів”. Символічна мова абстрагує від конкретного змісту висловлень, розкриває більш тісні закономірності мислення, здійснює необхідні узагальнення, встановлює нові закономірності. Ось чому сучасну математичну логіку часто називає *символічною*.

Вперше в історії ідеї про побудову логіки на математичній основі були висловлені німецьким математиком Г. Лейбницем (1646-1716) наприкінці XVII століття. Він вважав, що основні поняття логіки повинні бути позначені символами, що з'єднуються за особливими правилами. Це дозволить будь-яке міркування замінити обчисленням. “Ми вживаємо знаки не тільки для того, щоб передати наші думки іншим особам, але і для того, щоб полегшити сам процес нашого мислення” (Г. Лейбниц).

Перша реалізація ідеї Г. Лейбниці належить англійському вченому Д. Булю (1815-1864). Він створив алгебру, у якій буквами були позначені висловлення, і це привело до появи *алгебри висловлень*. Уведення символічних позначень у логіку мало для неї таке ж вирішальне значення, як і введення літерних позначень для математики. Це дало можливість формулювати її закони у загальному вигляді і створило передумови для застосування в логіці математичних методів, що призвело до корінної перебудови логічних теорій і до виникнення нової області математики, що одержала назву *математична логіка*. В свою чергу, застосування математики до логіки дозволило представити логічні теорії в новій зручній формі і використовувати обчислювальний апарат для розв'язку задач, малодоступних людському мисленню, що, звичайно, розширило область логічних досліджень.

До кінця XIX сторіччя актуальне значення для математики придбали питання обґрунтування її основних понять і ідей. Ці задачі мали логічну природу і, природно, привели до подальшого розвитку математичної логіки. У цьому відношенні визначними були роботи німецького математика Г. Фреге (1848-1925) і італійського математика Д. Пеано (1858-1932), що застосували математичну логіку для обґрунтування арифметики і теорії множин.

Логіка, що використовувалася для потреб математиків, набула власного, досить досконалого апарату і забезпечила точне та адекватне визначення поняття “*математичний доказ*”. Набувши рис математичної науки, математична логіка стала широко використовуватися для аналізу і побудови математичних теорій, вивчення алгоритмів і створення основ

теоретичної інформатики. Особливості математичного мислення пояснюються особливостями математичних абстракцій і різноманіттям їхніх взаємозв'язків. Вони відбивають у логічній систематизації математики, у доказі математичних теорем.

У зв'язку з вищевказаним сучасну *математичну логіку* визначають як розділ математики, предметом вивчення якого є математичний доказ і питання основ математики.

Математична логіка використовує методи символізації, формалізації, конструктивізму, якими активно користуються математики.

Однієї з основних причин розвитку математичної логіки є широке поширення *аксіоматичного методу* в побудові різних математичних теорій, у першу чергу, геометрії, арифметики, теорії груп тощо.

В аксіоматичній побудові математичної теорії попередньо вибирається деяка *система первісних понять* і *відносини* між ними. Ці поняття і відносини називаються основними. Далі без доказу приймаються основні положення розглянутої теорії – *аксіоми*. Весь подальший зміст теорії виводиться логічно з аксіом. Вперше *аксіоматична побудова* математичної теорії була здійснена Евклідом у побудові геометрії.

Відзначимо, що аксіоматичний підхід до побудови теорії залишалася єдиним до XIX століття. Велику роль у зміні такого підходу зіграли роботи М.І. Лобачевського (1792-1856). Лобачевський вперше висловив переконання в неможливості доказу п'ятого постулату Евкліда і підкріпив це переконання створенням нової геометрії. Пізніше німецький математик Ф. Клейн (1849-1925) довів несуперечність геометрії Лобачевського, чим фактично була доведена і неможливість доказу п'ятого постулату Евкліда. Так виникли і були вирішені в роботах М.І. Лобачевського і Ф. Клейна вперше в історії математики проблеми неможливості доказу і несуперечності в аксіоматичній теорії.

Несуперечність аксіоматичної теорії є однією з основних вимог, що пропонувані до системи аксіом певної теорії. Вона означає, що з даної системи аксіом не можна логічним шляхом вивести два суперечливі один одному твердження.

Доказ несуперечності аксіоматичних теорій можна здійснити різними методами. Одним з них є *метод моделювання* або *інтерпретацій*. Тут в якості основних понять і відношень вибираються елементи деякої множини і відношення між ними, а потім перевіряється, чи будуть виконуватися для обраних понять і відносин аксіоми даної теорії, тобто будується *модель* для даної теорії. Так, аналітична геометрія є арифметичною інтерпретацією геометрії Евкліда. Зрозуміло, що метод моделювання зводить питання про несуперечності однієї теорії до проблеми несуперечності іншої теорії.

Більшість інтерпретацій для математичних теорій (і, зокрема, для арифметики) будується на базі *теорії множин*. Однак наприкінці ХІХ століття в теорії множин були виявлені *протиріччя* (парадокси теорії множин). Яскравим прикладом такого парадоксу є парадокс Б. Расела.

Інші методи обґрунтування математики були розвинуті Д. Гілбертом (1862-1943) і його школою. Вони ґрунтуються на побудові математичних теорій як синтаксичних теорій, у яких всі аксіоми записуються формулами в деякому алфавіті і точно вказуються правила виведення одних формул з інших, тобто в теорію як складова частина входить математична логіка.

Таким чином, математична теорія, несуперечність якої було потрібно довести, стала предметом іншої математичної теорії, яку Гілберт назвав *метаматематикою*, або *теорією доказів*.

У зв'язку з цим виникає задача побудови синтаксичної, тобто *формалізованої аксіоматичної теорії* самої математичної логіки. Вибираючи по-різному системи аксіом і правила виведення одних формул з інших, одержують різні синтаксичні логічні теорії. Кожну з них називають *логічним численням*.

За своєю плідністю, силою і важливістю відкриттів, за своїм значенням для всієї математики математична логіка займає одне з найважливіших місць у сучасній математичній науці.

Сучасна математична логіка, на відміну від традиційної формальної логіки, знайшла широке застосування у різних областях наукових досліджень, зокрема, і у *теорії алгоритмів*.

Поняття «алгоритм» є концептуальною основою різноманітних процесів обробки інформації. З давніх часів у математиці склалося інтуїтивне уявлення про *алгоритм* як формальне розпорядження, що визначає сукупність операцій і порядок їх виконання для розв'язання задач деякого типу. Термін «алгоритм» зобов'язаний своїм походженням великому вченому середньовічного Сходу – Махамад ібн Муса ал-Хорезмі (Магомет, син Мойсея, із Хорезма 783 - 850 рр). У латинських перекладах з арабської арифметичного трактату ал-Хорезмі його ім'я транскрибувалося як *algorismi*.

З алгоритмами, тобто ефективними процедурами, які однозначно приводять до результату, математика мала справу завжди й у своєму розвитку накопичила множина різних алгоритмів. Отримавши відповідну інтерпретацію в конкретних додатках, вони становлять значну і найсуттєвішу частину математичного апарату, який використовується в техніці. Відомі зі шкільної програми методи множення “стовпчиком” та діленням “кутом”, алгоритм Евкліда знаходження найбільшого спільного дільника двох додатних натуральних чисел, метод Гауса розв'язання системи лінійних рівнянь, правило диференціювання складної функції, добування квадратного кореня з раціонального числа із заданою мірою точності, обчислення рангу цілочислової матриці, спосіб побудови трикутника за трьома сторонами – все це алгоритми.

Доки математика мала справу в основному з числами й обчисленнями, а поняття алгоритму ототожнювалося з поняттям методу обчислення, необхідність у вивченні самого поняття алгоритму не виникало. Вважалося, якщо для розв'язання деякого класу задач пропонувався конкретний алгоритм, то вчені доходили згоди вважати його дійсно шуканим алгоритмом. І тільки доведення того, що не існує алгоритму розв'язання певного класу, яке є висловлюванням про всі можливі алгоритми, потребує уточнення цього важливого поняття. Саме уточнення поняття “алгоритм” і виконує така наукова дисципліна як *теорія алгоритмів*. Отже, *теорія алгоритмів* - наукова дисципліна, що вивчає поняття про алгоритм на основі алгоритмічних моделей, їх функціонування та взаємозв'язку.

Математична логіка і теорія алгоритмів використовуються в теорії релейно-контактних схем, в теорії автоматів, у лінгвістиці, в економічних дослідженнях, у фізіології мозку і психології.

Математична логіка і теорія алгоритмів дуже важливі для професійної підготовки математика і програміста. Саме ці навчальні дисципліни дають йому можливість вникнути в сутність поняття доказу, з'ясувати зміст поняття, встановити взаємозв'язок між різного роду теоремами, сприяють розвитку алгоритмічного мислення.

РОЗДІЛ 1. МАТЕМАТИЧНА ЛОГІКА

1.1. АЛГЕБРА ЛОГІКИ

1.1.1. Алгебра висловлень

Література: [7,8-17; 8,29-80; 12,99-130; 12,185-201; 20,8-60]

Ключові поняття: *просте і складне висловлення, логічні операції над висловленнями, таблиця істинності, формула алгебри висловлень, рівносильні, тотожно істинні і тотожно хибні формули алгебри логіки, основні рівносильності, досконалі нормальні форми, алгоритми одержання досконалих нормальних форм, функції алгебри висловлень.*

1.1.1.1. Поняття висловлення. Види висловлень.

Основним (первісним) поняттям математичної логіки є поняття *простого висловлення*.

Під *висловленням*, за звичай, розуміють будь-яку оповідальну пропозицію, що стверджує щось про що-небудь, і при цьому ми можемо сказати, істинна вона чи хибна в даному місці і часу. Логічними значеннями висловлень є *«істина»* і *«хибність»*.

Приведемо приклади висловлень:

1. *Паралелограм має чотири вершини.*
2. *Мелітополь – столиця України.*
3. *Поганий студент – хорошиий солдат.*
4. *Число 2 більше 5.*
5. *Число 6 ділиться на 2 і на 3.*

Висловлення 1), 3) і 5) – істинні, а 2) і 4) – хибні.

Висловлення, що являє собою одне твердження, прийнято називати *простим* або *елементарним*. Висловлення, що виходять з елементарних за допомогою граматичних сполучників «не», «і», «або», «якщо ..., то ...», «тоді і тільки тоді», прийнято називати *складними* або *складеними*. Так, висловлення 4) виходить із простого висловлення, висловлення 5) утворено з елементарних висловлень «Число 6 ділиться на 2», «Число 6 ділиться на 3», з'єднаних сполучником «і».

Аналогічно складні висловлення можуть бути отримані з простих висловлень за допомогою граматичних сполучників «або», «якщо ..., то ...», «тоді і тільки тоді».

Граматичні сполучники «не», «і», «або», «якщо ..., то ...», «тоді і тільки тоді», в математичній логіці прийнято називати *пропозиційними зв'язками* або *функторами*.

В алгебрі висловлень усі висловлення розглядаються тільки з погляду їхнього логічного значення, а від конкретного змісту відволікаються. Вважається, що кожне висловлення або істинне, або хибне, і жодне висловлення не може бути одночасно істинним і хибним.

Слід зазначити, що розрізняють *двозначну* та *багатозначну* алгебру висловлень. Саме двозначна алгебра висловлень оперує лише двома значеннями. Символи „1” та „0” для позначення істинних і хибних висловлень запропонував німецький математик і логік Е.Шредер (1841-1902 р.р.).

Надалі будемо елементарні висловлення позначати буквами латинського алфавіту: $a, b, c, \dots, x, y, z, \dots$; істинне значення – буквою “1” або цифрою 1, а хибне значення – буквою “X” або цифрою 0.

Якщо висловлення a істинне, то будемо писати $a=1$, якщо ж хибне, то $a=0$.

1.1.1.2. Логічні операції над висловленнями.

Заперечення

Запереченням висловлення x називається нове висловлення, що є істинним, якщо висловлення x хибне, і хибним, якщо висловлення x істинне.

Заперечення висловлення x позначається як \bar{x} і читається „не x ” або „не вірно, що x ”. Логіки віддають перевагу виразу „не вірно, що x ”, оскільки саме так підкреслюється заперечення усього висловлення. Заперечивши істинне висловлення, логіки вказують на те, що одержане в результаті заперечення висловлення є хибним.

Логічні значення висловлення \bar{x} можна описати за допомогою таблиці 1.1

Таблиці такого виду прийнято називати *таблицями істинності*.

Таблиця 1.1.

Таблиця істинності заперечення

x	\bar{x}
1	0
0	1

Нехай x висловлення. Враховуючи те, що \bar{x} також є висловленням, то можна утворити заперечення висловлення \bar{x} , тобто висловлення $\bar{\bar{x}}$, що називається *подвійним запереченням* висловлення x . Зрозуміло, що логічні значення висловлень x і $\bar{\bar{x}}$ збігаються.

Наприклад, для висловлення «Мелітополь – столиця України» запереченням буде висловлення «Невірно, що Мелітополь є столицею України» або «Мелітополь не є столицею України», а подвійним запереченням буде висловлення «Невірно, що Мелітополь не є столицею України».

Кон'юнкція (логічне множення).

Кон'юнкцією (від. лат. *conjunctio* – зв'язок, сполучник) двох висловлень x , y називається нове висловлення, що вважається істинним, якщо обоє висловлення x , y істинні, і помилковим, якщо хоча б одне з них хибне (тобто, в інших випадках).

У природній мові аналогом кон'юнкції є вирази „ x і y ”, „ x разом з y ”, „як x , так і y ”, „не тільки x , а й y ”.

Кон'юнкція висловлень x , y позначається символом $x \wedge y$ (або $x \& y$, або $X \cdot Y$), і читається як « x і y ». Висловлення x , y називаються членами кон'юнкції. Усі можливі логічні значення кон'юнкції двох висловлень x і y описуються таблицею істинності (див. *Таблицю 1.2*).

Наприклад, для висловлень «6 ділиться на 2» та «6 ділиться на 3» їх кон'юнкцією буде висловлення «6 ділиться на 2 і 6 ділиться на 3».

З визначення операції кон'юнкції видно, що сполучник «і» в алгебрі логіки вживається в тому ж змісті, що й у повсякденній мові. Але в звичайній мові не прийнято з'єднувати сполучником «і» два висловлення, далекі одне від одного за змістом, а в алгебрі логіки розглядається кон'юнкція двох будь-яких висловлень (наприклад: «У городі бузина і у Києві дядько»).

Таблиця 1.2.

Таблиця істинності кон'юнкції

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

З визначення операцій кон'юнкції і заперечення зрозуміло, що висловлення $x \wedge \bar{x}$ завжди помилкове.

Диз'юнкція (логічне додавання)

Диз'юнкцією двох висловлень x , y називається нове висловлення, що вважається істинним, якщо хоча б одне з висловлень x , y істинне, і хибним, якщо вони обоє хибні.

Диз'юнкція висловлень x , y позначається символом $x \vee y$, читається як « x або y ». Висловлення x , y називаються членами диз'юнкції. Усі можливі логічні значення диз'юнкції двох висловлень x і y описуються таблицею істинності (див. Таблицю 1.3.).

Таблиця 1.3.

Таблиця істинності диз'юнкції

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Наприклад, висловлення “*У трикутнику DFE кут D або кут E гострий*” істинно, тому що обов'язково істинне одне з висловлень: «*У трикутнику DFE кут D гострий*», «*У трикутнику DFE кут E гострий*». У повсякденній мові сполучник «або» вживається в різному змісті: у такому, що виключає, і не виключає. В алгебрі логіки сполучник «або» завжди вживається в такому змісті, що не виключає.

З визначення операцій диз'юнкції і заперечення зрозуміло, що висловлення $x \vee \bar{x}$ завжди істинне.

Різні значення сполучника „або” в логіці фіксуються з'єднувальною (простою), розділовою (суворою) і виключною (антикон'юнкцією) диз'юнкцією.

З'єднувальна (проста) диз'юнкція – складне висловлення, яке буде істинним тоді і тільки тоді, коли буде істинним хоча б одне з висловлень (x або y ; x або y , або обидва) (див. *Таблицю 1.4.*)

Таблиця 1.4.

Таблиця істинності з'єднувальної диз'юнкції

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Розділова(сувора) диз'юнкція – складне висловлення, яке буде істинним тоді і тільки тоді, коли одне з простих висловлень істинне, а інше – обов'язково хибне (x або y , але не обидва; x , якщо не y) (див. *Таблицю 1.5.*)

Таблиця 1.5.

Таблиця істинності розділової диз'юнкції

x	y	$x \underline{\vee} y$
0	0	0
0	1	1
1	0	1
1	1	0

Таблиця істинності виключної диз'юнкції

x	y	$x \mid y$
0	0	1
0	1	1
1	0	1
1	1	0

Виключна (антikon'юнкція) диз'юнкція – складне висловлення, яке буде істинним тоді і тільки тоді, коли у крайньому разі одне з простих висловлень, що його складає, хибне (див. *Таблицю 1.6.*).

Виключна диз'юнкція має також назву операція „штрих Шеффера”.

Імплікація

Імплікацією (від. лат.*implicatio* – сплетення, переплетіння) двох висловлень x , y називається нове висловлення, що вважається помилковим, якщо x істинне, а y – хибне, і істинним у всіх інших випадках.

Імплікація висловлень x, y позначається символом $x \rightarrow y$ (або $x \Rightarrow y$), читається як “якщо x , то y ” або “з x впливає y ”. Висловлення x називають *умовою* або *посилкою*, висловлення y – *наслідком* або *висновком*, висловлення $x \rightarrow y$ – *імплікацією*. Слід зазначити, що прості висловлення, які об'єднуються імплікацією, не можна переставляти місцями.

Логічні значення операції імплікації описуються таблицею істинності (див. *Таблицю 1.7.*).

Таблиця істинності імплікації

X	y	$x \rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

Наприклад, висловлення “якщо число 12 поділяється на 6, то воно поділяється на 3”, мабуть, істинне, тому що тут істинна посилка “Число 12 поділяється на 6” і істинний висновок “Число 12 поділяється на 3”.

Вживання слів “якщо..., то...” в алгебрі висловлень відрізняється від вживання їх у повсякденній мові, де ми, як правило, вважаємо, що, якщо висловлення x хибне, то висловлення “якщо x , то y ” узагалі не має змісту. Крім того, будуючи пропозицію виду “якщо x , то y ” у повсякденній мові, ми завжди маємо на увазі, що пропозиція y впливає з пропозиції x . Вживання слів “якщо..., то...” у математичній логіці не вимагає цього, оскільки в ній зміст висловлень не розглядається. Таким чином, вираз “якщо x , то y ” доцільно читати як „ x імплікує y ”. Це дозволяє уникнути нісенітниць типу „якщо сніг білий, то $2+2=4$ ”.

Імплікація відіграє важливу роль у математичних доказах, тому що багато теорем формулюються в умовній формі “якщо x , то y ”. Якщо при цьому відомо, що x істинне, і доведена істинність імплікації $x \rightarrow y$, то ми вправі зробити висновок про істинність висновку y .

Аналіз імплікації передбачає визначення понять „достатня” і „необхідна” підстава.

Достатньою є підстава, наявність якої обов’язково спричиняє певний наслідок. У разі її відсутності, наслідок може наступити, а може й ні (якщо має місце x , то обов’язково матиме місце y). Наприклад, якщо був дощ, то дахи будинків мокрі.

Необхідною є підстава, відсутність якої зумовлює відсутність конкретного явища (y , тільки якщо x). Наприклад, якщо дахи будинків мокрі, то був дощ.

Еквіваленція (подвійна імплікація).

Еквіваленцією (або *еквівалентністю*) двох висловлень x, y називається нове висловлення, що вважається істинним, коли обое висловлення x, y або одночасно істинні, або одночасно хибні. І хибним y всіх інших випадках.

Еквіваленція висловлень x, y позначається символом $x \leftrightarrow y$ (або \Leftrightarrow , рідше \sim), і читається як “для того, щоб x ,

необхідно і досить, щоб y ”, або “ x тоді і тільки тоді, коли y ”, “якщо x , то y , і навпаки”.

Висловлення x , y називаються членами еквіваленції.

Логічні значення операції еквіваленції описуються таблицею істинності (див. Таблицю 1.8).

Таблиця 1.8.

Таблиця істинності еквіваленції

x	y	$x \leftrightarrow y$
0	0	1
0	1	0
1	0	0
1	1	1

Наприклад, еквіваленція “Трикутник SPQ з вершиною S і основою PQ рівнобедрений тоді і тільки тоді, коли $\angle P = \angle Q$ ” є істинною, тому що висловлення “Трикутник SPQ з вершиною S і основою PQ рівнобедрений” і “У трикутнику SPQ з вершиною S і основою PQ $\angle P = \angle Q$ ” або одночасно істинні, або одночасно хибні.

Еквівалентність відіграє велику роль у математичних доказах. Відомо, що значне число теорем формулюється у формі необхідних і достатніх умов, тобто у формі еквівалентності. У цьому випадку, знаючи про істинність або хибність одного з двох членів еквівалентності і, довівши істинність самої еквівалентності, ми робимо висновок про істинність або хибність іншого члена еквівалентності.

1.1.1.3. Формули алгебри висловлень. Обчислення значень формул.

За допомогою логічних операцій над висловленнями із заданої сукупності висловлень можна будувати різні складні висловлення. При цьому порядок виконання операцій вказується дужками. Наприклад, із трьох висловлень x, y, z можна побудувати висловлення

$$(x \wedge y) \vee \bar{z} \quad \text{і} \quad x \rightarrow (\bar{y} \vee (x \wedge z)) .$$

Перше з них є диз'юнкцією кон'юнкції x, y і запереченням висловлення z , а друге висловлення є імплікацією, посилюю

якої є висловлення x , а висновком – заперечення диз'юнкції висловлення y і кон'юнкції висловлень x, z .

Будь-яке складне висловлення, що може бути побудоване з елементарних висловлень за допомогою застосування логічних операцій заперечення, кон'юнкції, диз'юнкції, імплікації і еквіваленції та дужок, називається *формулою алгебри висловлень*. Формули алгебри висловлень будемо позначати великими буквами латинського алфавіту $A, B, C, \dots, X, Y, Z, \dots$

Для спрощення запису формул прийнятий ряд домовленостей. Наприклад, дужки можна опускати, дотримуючи наступного порядку дій: кон'юнкція виконується раніше, ніж всі інші операції, диз'юнкція виконується раніше, ніж імплікація й еквівалентність. Якщо над формулою стоїть знак заперечення, то дужки теж опускаються. У зв'язку з цим приведені вище формули $(x \wedge y) \vee \bar{z}$ і $x \rightarrow (y \vee (x \wedge z))$ можуть бути написані так: $x \wedge y \vee \bar{z}$ і $x \rightarrow y \vee x \wedge z$, а також $x \vee \bar{z}$ і $x \rightarrow y \vee xz$.

Логічне значення формули алгебри висловлення цілком визначається логічними значеннями вхідних до неї елементарних висловлень. Наприклад, логічним значенням формули $x \wedge y \vee \bar{z}$ у випадку, якщо $x=1, y=1, z=0$ буде істина, тобто $x \wedge y \vee \bar{z}=1$.

Як і у випадку з логічними операціями всі можливі логічні значення формули, у залежності від значень вхідних у неї елементарних висловлень, можуть бути цілком визначені за допомогою таблиці істинності. Наприклад, для формули $\bar{x} \vee y \rightarrow x \wedge \bar{y}$ таблиця істинності має вигляд:

Таблиця 1.9.

Таблиця істинності формули $\bar{x} \vee y \rightarrow x \wedge \bar{y}$

x	y	\bar{x}	\bar{y}	$\bar{x} \vee y$	$x \wedge \bar{y}$	$\bar{x} \vee y \rightarrow x \wedge \bar{y}$
0	0	1	1	1	0	0
0	1	1	0	1	0	0
1	0	0	1	0	1	1
1	1	0	0	1	0	0

Як бачимо, коли формула містить n елементарних висловлень, то вона приймає 2^n значень, що складаються з нулів і одиниць, або, що те ж саме, таблиця містить 2^n рядків.

1.1.1.4. Рівносильні, тотожно істинні (ТІ) і тотожно хибні (ТХ) формули алгебри логіки. Закони логічних операцій. Закон подвійності.

Визначення 1.

Дві формули алгебри логіки A і B називаються *рівносильними*, якщо вони приймають однакові логічні значення на будь-якому наборі вхідних у формули елементарних висловлень. Рівносильність формул будемо позначати знаком \equiv , а запис $A \equiv B$ означає, що формули A і B *рівносильні*.

Формула A називається *тотожно істинною* (*тавтологією* або *загальнозначущою*), якщо вона приймає значення „істина” при всіх значеннях вхідних у неї змінних.

Формула називається *тотожно хибною*, якщо вона приймає значення „хибність” при всіх значеннях вхідних у неї змінних.

Формула називається *здійсненою*, якщо існують значення вхідних у неї змінних при яких вся формула перетворюється „істина”.

Формула називається *суперечливою* (*нездійсненою*), якщо існують значення вхідних у неї змінних при яких вся формула перетворюється „хибність”.

Між поняттями рівносильності та еквівалентності існує наступний зв'язок: якщо формули A і B рівносильні, то формула $A \leftrightarrow B$ – тавтологія, і навпаки, якщо формула $A \leftrightarrow B$ – тавтологія, то формули A і B – рівносильні. Наприклад, в елементарній математиці вираз $(x+y)^2$ рівносильний виразу $x^2+2xy+y^2$.

Рівносильності алгебри висловлень можна виразити законами логічних операцій.

Закони ідемпотентності

$$1. x \wedge x \equiv x$$

$$2. x \vee x \equiv x$$

Закони нуля та одиниці

$$3. x \wedge 1 \equiv x$$

$$4. x \vee 1 \equiv 1$$

$$5. x \wedge 0 \equiv 0$$

$$6. x \vee 0 \equiv x$$

Закон протиріччя

$$7. x \wedge \bar{x} \equiv 0$$

Закон виключення третього

$$8. x \vee \bar{x} \equiv 1$$

Закон зняття подвійного заперечення

$$9. \bar{\bar{x}} \equiv x$$

Закони поглинання

$$10. x \wedge (x \vee y) \equiv x$$

$$11. x \vee (x \wedge y) \equiv x$$

Закони розкриття імплікації та еквівалентності

$$12. x \leftrightarrow y \equiv (x \rightarrow y) \wedge (y \rightarrow x)$$

$$13. x \rightarrow y \equiv \bar{x} \vee y$$

Закони де Моргана

$$14. \overline{x \wedge y} \equiv \bar{x} \vee \bar{y}$$

$$15. \overline{x \vee y} \equiv \bar{x} \wedge \bar{y}$$

$$16. \overline{\bar{x} \wedge \bar{y}} \equiv x \vee y$$

$$17. \overline{\bar{x} \vee \bar{y}} \equiv x \wedge y$$

З рівносильностей цієї групи випливає, що будь-яку формулу алгебри висловлень можна замінити рівносильною їй формулою, що містить тільки дві логічні операції: кон'юнкцію і заперечення або диз'юнкцію і заперечення (див. *Таблицю 1.10.*). Подальше виключення логічних операцій неможливо. Так, якщо ми будемо використовувати тільки кон'юнкцію, то вже така формула як заперечення x не може бути виражена за допомогою операції кон'юнкції.

Таблиця істинності доведення рівносильності 13

x	y	\bar{x}	$x \rightarrow y$	$\bar{x} \vee y$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

Однак існують операції, за допомогою яких може бути виражена кожна з п'яти логічних операцій, якими ми користуємося. Такою операцією є, наприклад, виключна диз'юнкція (штрих Шеффера) (див. *Таблицю 1.11*).

Закони комутативності

18. $x \wedge y \equiv y \wedge x$

19. $x \vee y \equiv y \vee x$

Закони асоціативності

20. $x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$

21. $x \vee (y \vee z) \equiv (x \vee y) \vee z$

Закони дистрибутивності

22. $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$

23. $x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$

Закон склеювання

24. $x \wedge y \vee \bar{x} \wedge y \equiv y, x \wedge y \vee x \wedge \bar{y} \equiv x;$

25. $(x \vee y) \wedge (\bar{x} \vee y) \equiv y, (x \vee y) \wedge (x \vee \bar{y}) \equiv x.$

Закон Блейка – Порецького

26. $x \vee (\bar{x} \wedge y) \equiv x \vee y.$

Закон згортання логічного виразу

27. $(x \wedge y) \vee (\bar{x} \wedge z) \vee (y \wedge z) \equiv (x \wedge y) \vee (\bar{x} \wedge z).$

Закон контрапозиції

28. $x \rightarrow y \equiv \bar{y} \rightarrow \bar{x}$

Додаткові закони алгебри логіки

29. $x \rightarrow y \equiv x \wedge \bar{y}$

30. $x \leftrightarrow y \equiv (\bar{x} \vee y) \wedge (x \vee \bar{y})$

Таблиця істинності штриха Шеффера

x	y	$x y$
0	0	1
0	1	1
1	0	0
1	1	0

1.1.1.5. Рівносильні перетворення формул.

Використовуючи наведені вище рівносильності, можна частину формули або усю формулу замінити рівносильної їй формулою. Такі перетворення формул називаються рівносильними (аналог тотожним перетворенням в арифметиці, алгебрі і тригонометрії).

Рівносильні перетворення використовуються для доказу рівносильностей, для приведення формул до заданого виду, для спрощення формул.

Формула А вважається простіше рівносильної їй формули В, якщо вона містить менше букв, менше логічних операцій. При цьому звичайно операції *еквівалентності* і *імплікації* замінюються операціями *диз'юнкції* і *кон'юнкції*, а *заперечення* відносять до елементарних висловлень.

При проведенні рівносильних перетворень кожен крок ґрунтується на використанні того або іншого закону. Номер відповідної формули ми будемо вказувати над знаком рівносильності, що передує черговому крокові.

Розглянемо ряд прикладів рівносильних перетворень.

Приклад 1. Довести рівносильність $x \leftrightarrow y \equiv \bar{x} \wedge \bar{y} \vee x \wedge y$.

$$\begin{aligned}
 x \leftrightarrow y &\equiv (x \rightarrow y) \wedge (y \rightarrow x) \equiv (\bar{x} \vee y) \wedge (\bar{y} \vee x) \equiv (\bar{x} \vee y) \bar{y} \vee (\bar{x} \vee y) x \equiv \\
 &\equiv \bar{x} \wedge \bar{y} \vee \bar{x} \wedge x \vee y \wedge \bar{y} \vee y \wedge x \equiv \bar{x} \wedge \bar{y} \vee 0 \vee 0 \vee y \wedge x \equiv \bar{x} \wedge \bar{y} \vee y \wedge x \equiv \\
 &\equiv \bar{x} \wedge \bar{y} \vee x \wedge y
 \end{aligned}$$

Приклад 2. Спростити формулу $\overline{(x \vee y \rightarrow x \vee y)} \wedge y$.

$$\overline{(x \vee y \rightarrow x \vee y)} \wedge y \equiv \overline{(x \vee y \vee x \vee y)} \wedge y \equiv (x \vee y \vee x \vee y) \wedge y \equiv (x \vee y) \wedge y \equiv y$$

Приклад 3. Довести ТТ формули $(x \rightarrow y) \rightarrow ((y \rightarrow z) \rightarrow (x \vee y \rightarrow z))$.

$$(x \rightarrow y) \rightarrow ((y \rightarrow z) \rightarrow (x \vee y \rightarrow z)) \equiv$$

$$\equiv \overline{(x \vee y)} \vee \overline{(y \vee z)} \vee \overline{x \vee y \vee z} \equiv$$

$$\equiv \overline{x} \wedge \overline{y} \vee \overline{y} \wedge \overline{z} \vee \overline{x} \wedge \overline{y} \vee \overline{z} \equiv$$

$$\equiv x \wedge \overline{y} \vee y \wedge \overline{z} \vee \overline{x} \wedge \overline{y} \vee \overline{z} \equiv$$

$$\equiv (x \wedge \overline{y} \vee \overline{x} \wedge \overline{y}) \vee (y \wedge \overline{z} \vee \overline{z}) \equiv$$

$$\equiv \overline{y} \wedge (x \vee \overline{x}) \vee (y \vee z) \wedge (\overline{z} \vee z) \equiv$$

$$\equiv \overline{y} \wedge 1 \vee (y \vee z) \wedge 1 \equiv \overline{y} \vee y \vee z \equiv 1 \vee z \equiv 1$$

Приклад 4. Довести закони склеювання.

$$x \wedge y \vee \overline{x} \wedge y \equiv y \wedge (x \vee \overline{x}) \equiv y \wedge 1 \equiv y$$

$$(x \vee y) \wedge (\overline{x} \vee y) \equiv (x \vee y) \wedge \overline{x} \vee (x \vee y) \wedge y \equiv$$

$$\equiv x \wedge \overline{x} \vee y \wedge \overline{x} \vee x \wedge y \vee y \wedge y \equiv$$

$$\equiv 0 \vee y \wedge \overline{x} \vee x \wedge y \vee y \equiv$$

$$\equiv y \wedge \overline{x} \vee x \wedge y \vee y \equiv y \wedge \overline{x} \vee y \equiv y$$

Приклад 5. Довести закон згортання логічного виразу.

$$(x \wedge y) \vee (\overline{x} \wedge z) \vee (y \wedge z) \equiv (x \wedge y) \wedge (z \vee \overline{z}) \vee (\overline{x} \wedge z) \wedge (y \vee \overline{y}) \vee (y \wedge z) \wedge (x \vee \overline{x}) \equiv$$

$$\equiv (x \wedge y \wedge z) \vee (x \wedge y \wedge \overline{z}) \vee (\overline{x} \wedge y \wedge z) \vee (\overline{x} \wedge \overline{y} \wedge z) \vee (x \wedge y \wedge z) \vee (\overline{x} \wedge y \wedge z) \equiv$$

$$\equiv ((x \wedge y \wedge z) \vee (x \wedge y \wedge \overline{z})) \vee ((\overline{x} \wedge y \wedge z) \vee (\overline{x} \wedge \overline{y} \wedge z)) \equiv (x \wedge y) \vee (\overline{x} \wedge z).$$

1.1.1.6. Проблема розв'язуваності в алгебрі висловлень і нормальні форми

Дуже часто виникає завдання: задати єдиний спосіб (алгоритм), який дає для кожної формули з'ясувати, чи є вона здійсненою. Таке завдання має назву *проблеми розв'язуваності*.

Метод перебору, який полягає у визначенні значення формули для кожного окремого набору змінних, звичайно дає принципове вирішення проблеми, але при великій кількості змінних він практично не здійснений через велике число можливих наборів значень змінних (2^n).

Існує інший спосіб, що ґрунтується на зведенні формул до нормальної форми. Якщо у процесі такого зведення формула не перетворилась на ТІ або ТХ, то це свідчить про її здійсненність.

Проблема розв'язуваності.

В математиці часто виникає необхідність знаходження єдиного методу розв'язку деякого класу типових задач. Такі класи, наприклад, утворюють задачі знаходження найбільшого спільного дільника цілих чисел, задачі знаходження найбільшого спільного дільника багаточленів, задачі знаходження рішень систем рівнянь тощо.

Метод, що описаний за допомогою системи вказівок, які необхідно виконати у певній послідовності для одержання з умови задачі за скінчене число кроків її розв'язку, називають *алгоритмом*. Задача знаходження алгоритму для розв'язку даного класу задач отримала назву *проблеми розв'язуваності* даного класу задач.

Якщо для даного класу задач вдається побудувати алгоритм, то говорять, що проблема розв'язуваності для цього класу задач розв'язувана або ж, що дана проблема алгоритмічно нерозв'язувана. У зв'язку з тим, що для розв'язку деякого класу задач не вдалося знайти алгоритм, в математиці виникли задачі доведення неможливості побудови для даного типу задач алгоритму, що їх би розв'язував. Саме така ситуація вимагала чіткого визначення поняття алгоритму як об'єкта математичної теорії та формування основ такої дисципліни як теорія алгоритмів. На сьогодні в межах вказаної дисципліни сформульовано декілька алгоритмічних моделей, що

використовуються для уточнення інтуїтивно зрозумілого поняття алгоритму.

Прикладом проблеми розв'язуваності може бути задача побудови алгоритму, який дає змогу для кожної формули з'ясувати, чи є вона здійсненою (чи не є вона ТІ або ТХ), тобто чи виражає дана формула деякий логічний закон чи ні. Проблема розв'язуваності для даної задачі розв'язувана.

Найпростішим алгоритмом розв'язку цієї задачі є побудова таблиці істинності для даної формули алгебри висловлень. У випадку, якщо всі значення формули істинні, то формула є ТІ, якщо всі хибні – ТХ, інакше формула є здійсненою.

Наведений алгоритм, звичайно, дає принципове вирішення проблеми розв'язуваності. Проте, за умови великої кількості змінних він практично не здійснений через величезне число можливих наборів значень змінних (2^n).

Існує інший алгоритм, що ґрунтується на зведенні формул до нормальної форми. Якщо у процесі такого зведення формула не перетворилася на ТІ або ТХ, то це свідчить про її здійсненність.

У якості нормальних форм використовуються досконалі диз'юнктивні (ДДНФ) та кон'юнктивні (ДКНФ) нормальні форми.

Елементарні диз'юнкції та елементарні кон'юнкції.

Визначення 1. Логічна сума будь-якої кількості різних незалежних змінних, що входять із запереченням або без нього, називається *елементарною диз'юнкцією*.

Наприклад, $x_1 \vee \overline{x_1} \vee x_2$; $x_1 \vee x_2 \vee \overline{x_1} \vee x_3$;
 $\overline{x_0} \vee x_1 \vee \overline{x_0}$.

Визначення 2. Логічний добуток будь-якої кількості різних незалежних змінних, що входять із запереченням або без нього, називаються *елементарною кон'юнкцією*.

Наприклад, $x_1 \wedge x_1 \wedge x_2$; $x_1 \wedge x_2 \wedge x_1 \wedge x_3$;
 $x_0 \wedge x_1 \wedge x_0$.

Якщо до елементарної диз'юнкції (кон'юнкції) входить кожна змінна (із запереченням або без нього) і при цьому тільки один раз, то вона називається *повною елементарною диз'юнкцією* (кон'юнкцією).

Наприклад, для елементарної диз'юнкції $x_1 \vee x_2 \vee x_1 \vee x_3$ повним елементарними диз'юнкціями будуть:

$$\begin{aligned} & x_1 \vee x_2 \vee x_3 \\ & x_1 \vee x_2 \vee \overline{x_3} \\ & x_1 \vee \overline{x_2} \vee x_3 \\ & x_1 \vee \overline{x_2} \vee \overline{x_3} \\ & \overline{x_1} \vee x_2 \vee x_3 \\ & \overline{x_1} \vee x_2 \vee \overline{x_3} \\ & \overline{x_1} \vee \overline{x_2} \vee x_3 \\ & \overline{x_1} \vee \overline{x_2} \vee \overline{x_3} . \end{aligned}$$

Якщо у наведених формулах знаки операцій диз'юнкції змінити знаком операції кон'юнкції, то ми отримаємо повні елементарні кон'юнкції від трьох незалежних змінних.

Кожна елементарна диз'юнкція приймає значення, що дорівнює нулю, тоді і тільки тоді, коли кожна змінна, що не є запереченням, дорівнює нулю, а кожна змінна із запереченням дорівнює одиниці.

Систему значень змінних, для якої дана повна елементарна диз'юнкція приймає значення, що дорівнює нулю, називається *нулем* (конституентною нуля) даної елементарної диз'юнкції.

Так, наприклад, нулями наведених повних елементарних диз'юнкції є система:

$$\begin{aligned} & (0,0,0) \\ & (0,0,1) \\ & (0,1,0) \\ & (0,1,1) \end{aligned}$$

(1,0,0)

(1,0,1)

(1,1,0)

(1,1,1).

По відношенню до повних елементарних кон'юнкції можна сказати, що кожна з них тільки один раз прийме значення, що дорівнює одиниці, - тоді і тільки тоді, коли змінна без заперечення дорівнюватиме одиниці, а із запереченням – нулю.

Таку систему значень змінних називають *одиноцею* (конституентною одиницею) даної елементарної диз'юнкції.

Із наведених означень випливає, що:

- 1) елементарна диз'юнкція тоді є тотожно-істинною (ТІ), коли в ній присутні разом із певною змінною x_i і її заперечення $\overline{x_i}$;
- 2) елементарна кон'юнкція тоді і тільки тоді є тотожно-хибною (ТХ), коли в ній присутні разом із змінною x_i і її заперечення $\overline{x_i}$.

Нормальні форми

Визначення 3. Формула, що задана кон'юнкцією елементарних диз'юнкції будь-якої кількості різних незалежних змінних, називається *кон'юнктивною нормальною формою* (КНФ).

Наприклад, $(x_1 \vee x_2 \vee x_2) \wedge (x_1 \vee x_1 \vee x_2 \vee x_2)$.

Визначення 4. Формула, що задана диз'юнкцією елементарних кон'юнкції будь-якої кількості різних незалежних змінних, називається *диз'юнктивною нормальною формою* (ДНФ).

Наприклад, $(x_1 \wedge x_2 \wedge x_2) \vee (x_1 \wedge x_1 \wedge x_2 \wedge x_2)$.

Зауваження:

- 1) КНФ тоді і тільки тоді є ТІ, коли кожна елементарна диз'юнкція, що входить до її

складу, разом із змінною x_i містить і її заперечення $\overline{x_i}$;

- 2) ДНФ тоді і тільки тоді є ТХ, коли кожна елементарна кон'юнкція, що входить до її складу, разом із змінною x_i містить і її заперечення $\overline{x_i}$.

Досконалі нормальні форми

Визначення 5. Формула, що задана кон'юнкцією різних повних елементарних диз'юнкції будь-якої кількості різних незалежних змінних (при цьому рівність диз'юнкції розуміється з точністю до порядку змінних) називається *досконалою кон'юнктивною нормальною формою* (ДКНФ).

Наприклад, $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3)$.

Визначення 6. Формула, що задана диз'юнкцією повних елементарних кон'юнкцій будь-якої кількості різних незалежних змінних (при цьому рівність кон'юнкцій розуміється з точністю до порядку змінних) називається *досконалою диз'юнктивною нормальною формою* (ДДНФ).

Наприклад, $(x_1 \wedge x_2 \wedge \overline{x_3}) \vee (x_1 \wedge x_2 \wedge x_3) \vee (\overline{x_1} \wedge x_2 \wedge x_3)$.

- Зауваження: 1) ДКНФ не може бути ТІ формулою
2) ДДНФ не може бути ТХ формулою.

Враховуючи те, що в n кількості змінних можна утворити 2^n повних елементарних диз'юнкцій (кон'юнкцій), то, вибираючи можливі комбінації по 1,2,...по n диз'юнкцій (кон'юнкцій) із всіх 2^n повних елементарних диз'юнкцій (кон'юнкцій) і поєднуючи їх в кожній комбінації знаком операції кон'юнкції (диз'юнкції), отримаємо всі можливі ДКНФ

(ДДНФ) від n змінних. Загальна кількість таких форм дорівнює $2^{2^n} - 1$ (виключаючи пусту кон'юнкцію (диз'юнкцію)).

Системою значень змінних ДКНФ, при яких формула приймає значення, що дорівнює нулю, є нулі повних елементарних диз'юнкцій, які складають цю форму. Наприклад, формула $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3)$ має три нулі $(0,0,1)$, $(0,1,0)$ і $(1,1,1)$.

Системою значень змінних ДДНФ, при яких формула приймає значення, що дорівнює одиниці, є одиниці повних елементарних кон'юнкцій, які складають цю форму. Наприклад, формула $(x_1 \wedge x_2 \wedge x_3) \vee (\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}) \vee (x_1 \wedge x_2 \wedge \overline{x_3})$ має три одиниці $(0,1,1)$, $(1,0,0)$ і $(0,1,0)$.

Повертаючись до проблеми побудови алгоритму, який дає змогу для кожної формули з'ясувати чи є вона здійсненою (чи не є вона ТІ або ТХ), пригадаємо, що формула буде ТІ тоді і тільки тоді, коли в кожній елементарній диз'юнкції кон'юнктивної нормальної форми разом із змінною x_i буде присутнє і її заперечення $\overline{x_i}$.

Наприклад, формула $(X \rightarrow Y) \rightarrow (\overline{Y} \rightarrow \overline{X})$ має еквівалентну їй кон'юнктивну нормальну форму $(X \vee \overline{X} \vee Y) \wedge (\overline{X} \vee \overline{Y} \vee Y)$. Як бачимо КНФ є ТІ, а отже і вихідна формула також буде ТІ.

Визначення приналежності формули до класу ТХ відбувається аналогічним чином, проте в даному випадку необхідно і достатньо, щоб кожна елементарна кон'юнкція диз'юнктивної нормальної форми разом із деякою змінною x_i містила б і її заперечення $\overline{x_i}$. Як було сказано раніше, якщо ДКНФ містить всі повні елементарні диз'юнкції, то вона є ТХ. Звідси робимо висновок, що формула є виконуваною тоді і тільки тоді, коли еквівалентна їй ДКНФ не містить всіх повних елементарних диз'юнкцій.

Наприклад, формула $x_1 \wedge (x_2 \rightarrow x_3) \rightarrow \overline{x_1 \vee x_2}$ має еквівалентну формулу $(x_2 \wedge \overline{x_3} \vee \overline{x_1} \vee x_2)$ та ДКНФ $(\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$.

Як бачимо до ДКНФ не увійшли диз'юнкції

$$x_1 \vee x_2 \vee x_3$$

$$x_1 \vee x_2 \vee \overline{x_3}$$

$$x_1 \vee \overline{x_2} \vee x_3$$

$$x_1 \vee \overline{x_2} \vee \overline{x_3}$$

$$\overline{x_1} \vee \overline{x_2} \vee x_3$$

$$\overline{x_1} \vee \overline{x_2} \vee \overline{x_3},$$

нулями яких є відповідно система

$$0, 0, 0$$

$$0, 0, 1$$

$$0, 1, 0$$

$$0, 1, 1$$

$$1, 1, 0$$

$$1, 1, 1.$$

Отже, ми отримали можливість вирішувати питання про виконуваність будь-якої формули лише на основі її вигляду, не вдаючись в зміст самої формули.

Проблема визначення здійсненності довільної формули зведенням її до ДДНФ чи ДКНФ вимагає формулювання алгоритмів переходу від довільної формули алгебри висловлень до ДДНФ та ДКНФ. Сформулюємо ці алгоритми.

Алгоритм переходу від довільної формули алгебри висловлень до ДДНФ через побудову таблиці істинності

1. Побудувати таблицю істинності для формули.
2. Виписати ті значення змінних X , Y і Z , що дали конституенту одиниці.
3. Одержати ДДНФ формули за допомогою з'єднання операцією диз'юнкції записаних конституент одиниці.

Підтвердимо правомірність алгоритму прикладом побудови ДДНФ функції $x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$.

1. Побудуємо таблицю істинності для формули

Таблиця 1.12.

Таблиця істинності формули $x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$

x	y	z	$x \wedge y$	$(x \wedge (\overline{y \vee z}) \vee y \wedge z)$	$\overline{(x \wedge (\overline{y \vee z}) \vee y \wedge z)}$	$x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$
0	0	0	0	0	1	1
0	0	1	0	0	1	1
0	1	0	0	0	1	1
0	1	1	0	1	0	0
1	0	0	0	1	0	0
1	0	1	0	1	0	0
1	1	0	1	0	1	1
1	1	1	1	1	0	1

Таблиця 1.13.

Конституенти одиниці для формули

$$x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$$

x	y	z
0	0	0
0	0	1
0	1	0
1	1	0
1	1	1

2. Випишемо ті значення змінних X, Y і Z, що дали конституенту одиниці (див. *Таблицю 1.13.*).

3. Одержуємо ДДНФ функції за допомогою з'єднання операцією диз'юнкції записаних конституент одиниці.

$$(\overline{x} \wedge \overline{y} \wedge \overline{z}) \vee (\overline{x} \wedge \overline{y} \wedge z) \vee (\overline{x} \wedge y \wedge \overline{z}) \vee (x \wedge y \wedge \overline{z}) \vee (x \wedge y \wedge z)$$

Алгоритм переходу від довільної формули алгебри висловлень до ДКНФ через побудову таблиці істинності

1. Побудувати таблицю істинності для формули.
2. Виписати ті значення змінних X, Y і Z, що дали конституенту нуля.
3. Одержати ДКНФ формули за допомогою з'єднання операцією кон'юнкції записаних конститuent нуля.

Підтвердимо правомірність алгоритму прикладом побудови

ДКНФ функції $x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$.

Таблиця 1.14.

Конституенти нуля для формули

$$x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$$

x	y	z
0	1	1
1	0	0
1	0	1

1. Скористаємось таблицею істинності для формули з попереднього прикладу.
2. Випишемо ті значення змінних X, Y і Z, що дали конституенту нуля (див. *Таблицю*

1.14).

3. Одержуємо ДКНФ функції за допомогою з'єднання операцією кон'юнкції записаних конститuent нуля:

$$(x \vee \overline{y \vee z}) \wedge (\overline{x \vee y \vee z}) \wedge (\overline{x \vee y \vee z})$$

Приведення до досконалих нормальних форм громіздких формул алгебри висловлень спричинило необхідність розробки алгоритму, який би не використовував побудову таблиці істинності.

Алгоритм переходу від довільної формули алгебри висловлень до ДДНФ

1. Виключити константи, використовуючи закони дій з константами.
2. Використовуючи рівносильності, що виражають одні логічні операції через інші, позбутися у формулі імплікацій та еквіваленцій.

3. Опустити знаки заперечення безпосередньо на змінні, використовуючи закони де Моргана.
4. Використовуючи дистрибутивний закон, розкрити дужки. До одержаних елементарних кон'юнкцій застосувати закони ідемпотентності і протиріччя, спростити їх і звести подібні. Результатом виконання вказаних дій є одержання ДНФ функцій.
5. Побудувати константи одиниці функцій введенням у кожен елементарну кон'юнкцію відсутніх змінних, використовуючи закон виключення третього.
6. За допомогою дистрибутивного закону розкрити дужки і звести подібні, використовуючи закон ідемпотентності. Одержана формула відповідає ДДНФ функції.

Підтвердимо правомірність алгоритму прикладом побудови ДДНФ функції $x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$.

1. Опускаємо заперечення безпосередньо на змінні, використовуючи закони де Моргана:

$$\begin{aligned} & \overline{x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)}^{15} = \\ & = \overline{x \wedge y \vee (x \wedge (\overline{y \vee z})) \wedge (y \wedge z)}^{14} = \\ & = \overline{x \wedge y \vee (\overline{x \vee (\overline{y \vee z}))} \wedge (\overline{y \vee z})}^{15} = \\ & = \overline{x \wedge y \vee (\overline{x \vee (y \wedge \overline{z}))} \wedge (\overline{y \vee z})} \end{aligned}$$

2. Побудуємо ДНФ функцій, використовуючи дистрибутивний закон:

$$\begin{aligned} & x \wedge y \vee (\overline{x \vee (y \wedge \overline{z}))} \wedge (\overline{y \vee z})^{22} = \\ & = x \wedge y \vee ((\overline{x \wedge y}) \vee (y \wedge \overline{z} \wedge \overline{y}) \vee (\overline{x \wedge z}) \vee (y \wedge \overline{z} \wedge \overline{z}))^{7,1} = \\ & = (x \wedge y) \vee (\overline{x \wedge y}) \vee 0 \vee (\overline{x \wedge z}) \vee (y \wedge \overline{z})^6 = \\ & = (x \wedge y) \vee (\overline{x \wedge y}) \vee (\overline{x \wedge z}) \vee (y \wedge \overline{z}) \end{aligned}$$

3. Дана функція залежить від трьох змінних, тому до елементарних кон'юнкцій необхідно ввести відсутні змінні, використовуючи закон виключення третього:

$$\begin{aligned} & (x \wedge y) \vee (\overline{x \wedge y}) \vee (\overline{x \wedge z}) \vee (y \wedge \overline{z}) \stackrel{8,3}{=} \\ & = (x \wedge y \wedge (z \vee \overline{z})) \vee (\overline{x \wedge y} \wedge (z \vee \overline{z})) \vee \\ & \vee (\overline{x} \wedge (y \vee \overline{y}) \wedge \overline{z}) \vee ((x \vee \overline{x}) \wedge y \wedge \overline{z}) \end{aligned}$$

4. Використовуючи дистрибутивний закон, розкриємо дужки і зведемо подібні для одержання ДДНФ функції:

$$\begin{aligned} & (x \wedge y \wedge (z \vee \overline{z})) \vee (\overline{x \wedge y} \wedge (z \vee \overline{z})) \vee \\ & \vee (\overline{x} \wedge (y \vee \overline{y}) \wedge \overline{z}) \vee ((x \vee \overline{x}) \wedge y \wedge \overline{z}) \stackrel{22}{=} \\ & \equiv (x \wedge y \wedge z) \vee (x \wedge y \wedge \overline{z}) \vee (\overline{x} \wedge \overline{y} \wedge z) \vee (\overline{x} \wedge \overline{y} \wedge \overline{z}) \vee \\ & \vee (\overline{x} \wedge y \wedge \overline{z}) \vee (\overline{x} \wedge \overline{y} \wedge \overline{z}) \vee (x \wedge y \wedge \overline{z}) \vee (\overline{x} \wedge y \wedge \overline{z}) \equiv \\ & \equiv (x \wedge y \wedge z) \vee (x \wedge y \wedge \overline{z}) \vee (\overline{x} \wedge \overline{y} \wedge z) \vee (\overline{x} \wedge \overline{y} \wedge \overline{z}) \vee (\overline{x} \wedge y \wedge \overline{z}) \end{aligned}$$

Одержуємо ДДНФ заданої функції.

Аналогічним чином визначається і алгоритм переходу від довільної формули алгебри висловлень до ДКНФ.

Алгоритм переходу від довільної формули алгебри висловлень до ДКНФ

1. Виключити константи, використовуючи закони дій з константами.
2. Використовуючи рівносильності, що виражають одні логічні операції через інші, позбутися у формулі імплікацій та еквіваленцій.
3. Опустити знаки заперечення безпосередньо на змінні, використовуючи закони де Моргана.
4. Використовуючи дистрибутивний закон, звести функцію до виду елементарних диз'юнкцій. Застосувати закони ідемпотентності і виключення третього, спростити їх і звести подібні. Результатом виконання вказаних дій є одержання КНФ функцій.
5. Побудувати конституанти нуля функцій введенням у кожну елементарну диз'юнкцію відсутніх змінних, використовуючи закон протиріччя.

6. За допомогою дистрибутивного закону звести функцію до виду кон'юнкції конститuant нуля і спростити формул, використовуючи закон ідемпотентності. Одержана формула відповідає ДКНФ функції.

Підтвердимо правомірність алгоритму прикладом побудови ДКНФ функції $x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z)$.

1. Опускаємо заперечення безпосередньо на змінні, використовуючи закони де Моргана:

$$\begin{aligned} x \wedge y \vee (x \wedge (\overline{y \vee z}) \vee y \wedge z) &= \\ &= x \wedge y \vee \overline{(x \wedge (\overline{y \vee z})) \wedge (y \wedge z)} = x \wedge y \vee (\overline{x \vee (\overline{y \vee z})}) \wedge (\overline{y \vee z}) = \\ &= x \wedge y \vee (\overline{x \vee (y \wedge z)}) \wedge (\overline{y \vee z}) \end{aligned}$$

2. Побудуємо КНФ функцій, використовуючи дистрибутивний закон, закон ідемпотентності і виключення третього:

$$\begin{aligned} x \wedge y \vee (\overline{x \vee (y \wedge z)}) \wedge (\overline{y \vee z}) &= \\ &= x \wedge y \vee (\overline{x \vee y}) \wedge (\overline{x \vee z}) \wedge (\overline{y \vee z}) = \\ &= (x \vee ((\overline{x \vee y}) \wedge (\overline{x \vee z}) \wedge (\overline{y \vee z}))) \wedge (y \vee ((\overline{x \vee y}) \wedge (\overline{x \vee z}) \wedge (\overline{y \vee z}))) = \\ &= (x \vee \overline{x} \vee y) \wedge (x \vee \overline{x} \vee z) \wedge (x \vee \overline{y \vee z}) \wedge (y \vee \overline{x \vee y}) \wedge (y \vee \overline{x \vee z}) \wedge (y \vee \overline{y \vee z}) = \\ &= 1 \wedge 1 \wedge (x \vee \overline{y \vee z}) \wedge (y \vee \overline{x}) \wedge (y \vee \overline{x \vee z}) \wedge 1 = \\ &= (x \vee \overline{y \vee z}) \wedge (y \vee \overline{x}) \wedge (y \vee \overline{x \vee z}) \end{aligned}$$

3. Дана функція залежить від трьох змінних, тому до елементарної диз'юнкції $(y \vee \overline{x})$ необхідно ввести відсутню змінну z , використовуючи закон протиріччя. 4. 4. Після чого, використовуючи дистрибутивний закон, зведемо функцію до виду кон'юнкції конститuant нуля:

$$\begin{aligned} (x \vee \overline{y \vee z}) \wedge (y \vee \overline{x}) \wedge (y \vee \overline{x \vee z}) &= \\ &= (x \vee \overline{y \vee z}) \wedge (\overline{x \vee y \vee z} \wedge \overline{z}) \wedge (y \vee \overline{x \vee z}) = \\ &= (x \vee \overline{y \vee z}) \wedge (\overline{x \vee y \vee z}) \wedge (\overline{x \vee y \vee z}) \end{aligned}$$

Одержуємо ДКНФ заданої функції.

1.1.1.7. Функції алгебри висловлень

Як уже відзначалося, значення формули алгебри логіки цілком залежить від значень вхідних у цю формулу висловлень. Тому формула алгебри логіки є функцією вхідних у неї елементарних висловлень.

Наприклад, формула $(x \wedge y) \rightarrow \bar{z}$ є функцією трьох змінних $f(x,y,z)$. Особливістю цієї функції є та обставина, що її аргументи приймають одне з двох значень: нуль або одиницю, і при цьому функція також приймає одне з двох значень: нуль або одиницю.

Визначення.

Функція алгебри висловлень – це ствердний граматичний вираз, в якому йдеться про певну властивість або відношення предмета при невизначеності самого предмета.

Функцією n змінних є функція, де кожна змінна і результат функції приймають тільки одне з двох значень – 0 або 1.

Зрозуміло, що тотожньо-істинні і тотожньо-хибні формули алгебри висловлень являють собою постійні функції, а дві рівносильні функції виражають ту саму функцію.

З'ясуємо, яке число функцій n змінних. Очевидно, кожну функцію алгебри висловлень (як і формулу) можна задати за допомогою таблиці істинності, що буде містити 2^n рядків. Отже, кожна функція n змінних приймає 2^n значень, що складаються з нулів і одиниць. Таким чином, функція n змінних цілком визначається набором значень з нулів і одиниць довжини 2^n . Загальне ж число наборів, що складаються з нулів і одиниць, довжини 2^n дорівнює 2^{2^n} . Виходить, число різних функцій алгебри логіки n змінних дорівнює 2^{2^n} .

Зокрема, різних функцій однієї змінної чотири, а різних функцій двох змінних – шістнадцять. Випишемо усі функції алгебри логіки однієї і двох змінних.

Таблиця істинності для будь-яких функцій двох змінних має вигляд (див. *Таблицю 1.15.*).

Таблиця істинності для будь-яких функцій двох змінних

x	y	0	\wedge	\rightarrow	\mathcal{X}	\leftarrow'	y	+	\vee	\downarrow	\leftrightarrow	y'	\leftarrow	x'	\rightarrow		1
		f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

- $f_0(x,y)$ - тотожно-хибна
- $f_1(x,y)$ - тотожно-істинна
- $f_2(x,y)$ - кон'юнкція
- $f_3(x,y)$ - диз'юнкція
- $f_4(x,y)$ - антикон'юнкція
- $f_5(x,y)$ - імплікація
- $f_6(x,y)$ - імплікація
- $f_7(x,y)$ - диз'юнкція
- $f_8(x,y)$ - стрілка Пірса
- $f_9(x,y)$ - кон'юнкція
- $f_{10}(x,y)$ - антикон'юнкція
- $f_{11}(x,y)$ - антиімплікація
- $f_{12}(x,y)$ - заперечення анти імплікації
- $f_{13}(x,y)$ - еквівалентність
- $f_{14}(x,y)$ - заперечення еквівалентності
- $f_{15}(x,y)$ - заперечення першого аргументу

Як бачимо, кожній функції привласнюється порядковий номер у вигляді натурального числа. Цікавим є той факт, що стовпчик значень кожної функції у таблиці істинності відповідає двійковому коду цього числа. Молодшим розрядом вважається самий нижчий рядок (1,1,...,1), а старшим – самий верхній (0,0,...,0). Вказаний порядковий номер функції, як двійковий, так і десятковий, повністю визначає функцію алгебри висловлень від n змінних.

Кожній інтерпретації функції також привласнюється свій номер – значення двійкового коду, який зображує інтерпретація. Інтерпретації, що записана у верхньому рядку таблиці істинності, привласнюється номер 0, потім йде 1 тощо.

В самому нижчому рядку розташована інтерпретація за номером 2^n-1 , де n - кількість змінних, від якої залежить функція.

Для пояснення скористаємося прикладом, в якому необхідно знайти порядковий номер функції $f(x,y)$, що приймає такі значення: $f(0,0)=1, f(0,1)=1, f(1,0)=0, f(1,1)=1$.

Побудуємо таблицю істинності для цієї функції (див. *Таблиця 1.16.*).

Двійковий код, що відповідає значенням цієї функції –

1101. Переведемо двійкове число 1101_2 у десяткову систему числення. Для цього кожному розряду двійкового числа привласнимо ваговий коефіцієнт, що кратний відповідному ступеню числа 2, починаючи з нижчого рядку: $2^0, 2^1, 2^2$ тощо. Помноживши ваговий коефіцієнт на відповідну

Таблиця 1.16.

Таблиця істинності функції

$f(x,y)$,

x	y	$f(x,y)$
0	0	1
0	1	1
1	0	0
1	1	1

двійкову цифру і додавши одержані значення, знайдемо порядковий номер функції:

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13_{10}$$

Таким чином, десятковий номер даної функції – 13, тобто розглянута функція імплікації.

1.1.2. Алгебра предикатів

Література: [7, 17-22; 8, 158-170; 12, 207-227; 20, 60-112]

Ключові поняття: *суб'єкт, предикат, одномісний і багатомісний предикат, логічні операції над предикатами, квантори, кванторні операції над предикатами, формули алгебри предикатів, основні рівносильності алгебри предикатів.*

1.1.2.1. Поняття предиката

В алгебрі логіки висловлення розглядаються як нероздільні елементи і тільки з погляду їхньої істинності або хибності.

Ні структура висловлень, ні, тим більше, їхній зміст не враховуються. У той же час і в науці, і в практиці використовуються висновки, що істотно залежать як від структури, так і від змісту використовуваних у них висловлень. Наприклад, у міркуванні “*Будь-який ромб – паралелограм; ABC – ромб; отже, ABC – паралелограм*” посилки і висновки є елементарними висловленнями логіки висловлень і з погляду цієї логіки розглядаються як цілі, неподільні, без врахування їх внутрішньої структури. Проте, достатньо записати вихідний умовивід відповідною формулою алгебри висловлень $A \wedge B \rightarrow C$, а потім звести її до нормальної форми $A \wedge B \rightarrow C \equiv \overline{A \wedge B \vee C} \equiv \overline{A} \vee \overline{B} \vee \overline{C}$ і ми побачимо, що дана формула не є загальнозначущою (тавтологією), адже на інтерпретації (1,1,0) одержана формула дорівнюватиме нулю. Це означає, що C не є наслідком A і B. Така обмеженість можливостей формалізації пов’язана з тим, що в атомі A не враховується внутрішня змістова особливість узагальнення „кожен”. Отже, алгебра логіки, будучи важливою частиною логіки, виявляється недостатньою в аналізі багатьох міркувань. У зв’язку з цим виникає необхідність у розширенні логіки висловлень, у побудові такої логічної системи, засобами якої можна було б досліджувати структуру тих висловлень, що у рамках логіки висловлень розглядаються як елементарні.

Такою логічною системою є *логіка предикатів*, що містить логіку висловлень як свою складову частину.

Логіка предикатів, як і традиційна формальна логіка, поділяє елементарне висловлення на *суб’єкт* (буквально – підмет, хоча воно може грати і роль доповнення) і *предикат* (буквально – присудок, хоча воно може грати і роль визначення).

Суб’єкт – це те, про що щось затверджується у висловленні;

предикат – це те, що стверджується про суб’єкта.

Наприклад, у висловленні “7 - *просте число*”, “7” – суб’єкт, “*просте число*” – предикат. Це висловлення стверджує, що “7” має властивість “*бути простим числом*”.

Якщо в розглянутому прикладі змінити конкретне число 7 змінної x з множини натуральних чисел, то одержимо висловлювану форма “ x – *просте число*”. За одного значення x (наприклад, $x=13$, $x=17$) ця форма дає істинні висловлення, а при інших значеннях x (наприклад, $x=10$, $x=18$) ця форма дає хибні висловлення.

Стає зрозумілим, що ця висловлювана форма визначає функцію однієї змінної x , визначеної на певній множині і приймає значення з множини $\{1;0\}$.

Тут предикат стає функцією суб'єкта і виражає властивість суб'єкта.

Дамо декілька визначень, що відносяться до предикатів.

Одномісним предикатом $P(x)$ називається довільна функція змінної x , визначена на множині M , що приймає значення з множини $\{1; 0\}$.

Множина M , на якій визначений предикат $P(x)$, називається *областю визначення предиката* $P(x)$.

Множина всіх елементів $x \in M$, при яких предикат приймає значення “істина”, називається множиною (областю) істинності предиката $P(x)$. Так, наприклад, предикат $P(x)$ – “ x – *просте число*” визначений на множині N , а множина істинності для нього є множина усіх простих чисел.

Предикат $F(x)$ – “*діагоналі паралелограма x взаємно перпендикулярними*” визначений на множині всіх паралелограмів, а його множиною істинності є множина усіх ромбів.

З приведених прикладів бачимо, що одномісні предикати виражають властивості предметів (суб'єктів).

Визначення 2.

Предикат $P(x)$, визначений на множині M , називається *тотожньо-істинним*, якщо його множина істинності збігається з областю визначення.

Визначення 3.

Предикат $P(x)$, визначений на множині M , називається *тотожньо-хибним*, якщо його множина істинності є порожньою множиною.

Природним узагальненням поняття одномісного предиката є поняття *n*-місного предиката, за допомогою якого виражаються відношення між предметами.

Прикладом *бінарного відношення*, тобто відношення між двома предметами, є відношення “менше”. Нехай це відношення введене на множині Z цілих чисел. Воно може бути охарактеризовано висловлюваною формулою “ $x < y$ ”, де $x, y \in Z$, таким чином є функцією двох змінних $P(x, y)$, визначеної на множині впорядкованих пар цілих чисел з множиною значень $\{1; 0\}$.

Визначення 4.

Двомісним предикатом $P(x, y)$ називається функція двох змінних x і y , що визначена на множині M і приймає значення з множини $\{1; 0\}$.

У числі прикладів двомісних предикатів можна назвати такі предикати: $Q(x, y)$ – “ $x=y$ ” - предикат рівності; $F(x, y)$ – “ x рівнобіжний y ”, “*пряма x рівнобіжна прямій y* ”, визначений на множині прямих, що лежать на даній площині.

Аналогічно вводиться поняття *тримісного предиката*. Приведемо приклад тримісного предиката (функції трьох змінних): $S(x, y, z)$ – “ $x+y=z$ ”. Підстановка в нього значення $x=3$ перетворює його в двомісний предикат: $S(y, z)$ – “ $3+y=z$ ”, а підстановка $x=3, z=2$ – в одномісний предикат $S(y)$ – “ $3+y=2$ ”. Підстановка ж $S(2, 3, 5)$ перетворює його в істинне висловлення, а $S(1, 7, 4)$ – у помилкове.

Аналогічно визначається поняття *i*-місного предиката (функції n -змінних). Приклад n -місного предиката: $R(x_1, x_2, \dots, x_n): a_1x_1 + \dots + a_nx_n = 0$, який, як бачимо, являє собою алгебраїчне рівняння з n невідомими.

При $n=0$ будемо мати *нульмісний предикат* – це логічна змінна, що приймає значення з множини $\{1; 0\}$.

1.1.2.2. Логічні операції над предикатами

Предикати так само, як висловлення, можуть приймати два значення: “істина” (1) і “хибність” (0), тому до них застосовуються всі операції логіки висловлень, у результаті чого

з елементарних предикатів формуються складені предикати (як і в логіці висловлень, де з елементарних висловлень формувалися складені). Розглянемо застосування операцій логіки висловлень до предикатів на прикладах одномісних предикатів. Ці операції в логіці предикатів зберігають той же зміст, що був їм властивий у логіці висловлень.

Нехай на деякій множині M визначені два предикати $P(x)$ і $Q(x)$.

Визначення 1.

Запереченням предиката $P(x)$ називається новий предикат $\overline{P(x)}$ або $\overline{P(x)}$, що приймає значення “істина” при всіх значеннях $x \in M$, при яких предикат $P(x)$ приймає значення “хибність”, і приймає значення “хибність” при тих значеннях $x \in M$, при яких предикат $P(x)$ приймає значення “істина”.

Визначення 2.

Кон'юнкцією двох предикатів $P(x)$ і $Q(x)$ називається новий (складений) предикат $P(x) \wedge Q(x)$, що приймає значення “істина” при тих і тільки тих значеннях $x \in M$, при яких кожний із предикатів приймає значення “істина”, і приймає значення “хибність” у всіх інших випадках.

Так, наприклад, для предикатів $P(x)$: “ x – парне число” і $Q(x)$: “ x кратне 3” кон'юнкцією $P(x) \wedge Q(x)$ є предикат “ x – парне число і x кратне трьом”.

Визначення 3.

Диз'юнкцією двох предикатів $P(x)$ і $Q(x)$ називається новий предикат $P(x) \vee Q(x)$, що приймає значення “хибність” при тих і тільки тих значеннях $x \in M$, при яких кожний із предикатів приймає значення “хибність”, і приймає значення “істина” у всіх інших випадках.

Визначення 4. Імплікацією предикатів $P(x)$ і $Q(x)$ називається новий предикат $P(x) \rightarrow Q(x)$, що є хибним при тих і тільки тих значеннях $x \in M$, при яких одночасно $P(x)$ приймає значення “істина”, а $Q(x)$ – значення “хибність”, і приймає значення “істина” у всіх інших випадках.

Визначення 5. Еквіваленцією предикатів $P(x)$ і $Q(x)$ називається новий предикат $P(x) \leftrightarrow Q(x)$, що приймає значення “істинна” при всіх тих і тільки тих $x \in M$, при яких $P(x)$ і $Q(x)$ обидва приймають істинні або обидва хибні значення.

1.1.2.3. Кванторні операції

Розглянемо операції, що перетворюють предикати у висловлення.

Нехай предикат $P(x)$ визначений на множині M . Якщо “ a ” – деякий елемент із множини M , то підстановка його замість x у предикат $P(x)$ перетворює цей предикат у висловлення $P(a)$. Таке висловлення називають *одиничним*. Наприклад, $R(x)$: “ x – парне число” – предикат, а $R(6)$ – істинне висловлення, $R(3)$ – хибне висловлення.

Це ж відноситься і до n – місних предикатів: якщо замість усіх предметних змінних x_i , $i = \overline{1, n}$ підставити їх значення, то одержимо висловлення.

Поряд з утворенням із предикатів висловлень у результаті таких підстановок у логіці предикатів розглядаються ще дві операції, що перетворюють одномісний предикат у висловлення. Ці операції називаються *операціями квантифікації* (або просто *квантифікацією*, або *зв'язуванням кванторами*, або *навішенням кванторів*). При цьому розглядаються два типи так званих *кванторів*.

Квантор (від лат. *quantum* - скільки)- це логічний оператор, який описує відношення між суб'єктом та предикатом і несе кількісну характеристику виразу, перед яким він поставлений.

Квантор загальності

Нехай $P(x)$ – предикат, визначений на множині M . Під виразом $\forall xP(x)$ розуміють висловлення, істинне тоді, коли $P(x)$ істинний для кожного елемента x з множини M , і хибний в іншому випадку. Це висловлення вже не залежить від x . Відповідний йому словесний вираз звучить так: “Для будь-якого x $P(x)$ істинне”.

Символ \forall називають *квантором загальності* (від нім. *Alles - всі*). Змінну x у предикаті $P(x)$ називають *вільною* (їй можна давати різні значення з M), у висловленні ж $\forall xP(x)$ x називають *зв'язаною* квантором загальності.

Квантор існування

Нехай $P(x)$ – предикат визначений на множині M . Під виразом $\exists xP(x)$ розуміють висловлення, що є істинним, якщо існує елемент $x \in M$, для якого $P(x)$ істинне, і хибне – в іншому випадку. Це висловлення вже не залежить від x . Відповідний йому словесний вираз звучить так: “Існує x , при якому $P(x)$ істинне”. Символ \exists називають *квантором існування* (від нім. *Existieren - існувати*). У висловленні $\exists xP(x)$ змінна x зв'язана цим квантором.

Кванторні операції застосовуються і до багатомісних предикатів. Нехай, наприклад, на множині M заданий двомісний предикат $P(x,y)$. Застосування кванторної операції до предиката $P(x,y)$ за змінною x ставить у відповідність двомісному предикатові $P(x,y)$ одномісний предикат $\forall xP(x,y)$ (або одномісний предикат $\exists xP(x,y)$), що залежить від змінної y і не залежний від змінної x . До них можна застосувати кванторні операції за змінної y , що приведуть вже до висловлень наступних

виглядів:

$$\forall y \forall x P(x, y), \exists y \forall x P(x, y), \forall y \exists x P(x, y), \exists y \exists x P(x, y).$$

Розглянемо предикат $P(x)$ визначений на множині $M = \{a_1, \dots, a_n\}$, що складається з скінченної кількості елементів. Якщо предикат $P(x)$ є тотожно-істинним, то істинними будуть

Теоретичний курс

висловлення $P(a_1), P(a_2), \dots, P(a_n) \dots$. При цьому істинними будуть висловлення $\forall x P(x)$ і кон'юнкція $P(a_1) \wedge P(a_2) \wedge \dots \wedge P(a_n)$.

Якщо ж хоча б для одного елемента $a_k \in M$ $P(a_k)$ виявиться помилковим, то помилковими будуть висловлення $\forall x P(x)$ і кон'юнкція $\bigwedge_{i=1}^n P(a_i)$. Отже, справедлива рівносильність $\forall x P(x) \equiv P(a_1) \wedge P(a_2) \wedge \dots \wedge P(a_n) = \bigwedge_{i=1}^n P(a_i)$.

Слід зазначити, що в операціях квантифікації розподіляють два види змінних – зв'язані і вільні.

Логічні змінні, які входять в область дії квантора загальності і існування у формулі, і замість яких неможливо підставити імена предметів відповідної області, називаються *зв'язаними* змінними. Наприклад, $\forall x \exists y (x \geq y)$ - „для будь-якого x існує такий y , що x більше або дорівнює y ”. Змінна у формулі вважається *вільною*, якщо вона не входить у область дії кванторів і може бути замінена іменами предметів відповідної області.

Чисельні квантори

У математиці часто зустрічаються вирази вигляду “щонайменше n ” (“хоча б n ”), “не більш ніж n ”, “ n і тільки n ” (“рівно n ”), де n – натуральне число.

Ці вирази, що називаються *чисельними кванторами*, мають чисто логічний зміст; вони можуть бути замінені рівнозначними виразами, що не містять числівників, складається тільки з логічних термінів, знаків \equiv або \sim , що означає тотожність (збіг) об'єктів.

Нехай $n=1$. Пропозиція “Щонайменше один об'єкт має властивість P ” має той же зміст, що і пропозиція “Існує об'єкт, що володіє властивістю P ”, тобто $\exists x (P(x))$. (*)

Пропозиція “не більш ніж один об'єкт має властивість P ” рівнозначна пропозиції “Якщо є об'єкти, що володіють властивістю P , то вони збігаються”, тобто $\forall x \forall y (P(x) \wedge P(y) \rightarrow x \equiv y)$. (**). Пропозицію “один і тільки один об'єкт має властивість P ” рівнозначно кон'юнкції вищевказаних пропозицій (*) і (**).

Заперечення пропозицій із кванторами

Відомо, що часто для заперечення деякої пропозиції досить подати присудкові цієї пропозиції негативну частку “не”. Наприклад, запереченням пропозиції “*Ріка x впадає в Чорне море*” є пропозиція “*Ріка x не впадає в Чорне море*”. Чи можливе використання цього прийому для побудови заперечень пропозицій із кванторами? Розглянемо приклад.

Пропозиції “*Усі птаха літають*” і “*Усі птаха не літають*” не є запереченнями одна одної, тому що вони обоє помилкові. Пропозиції “*Деякі птахи літають*” і “*Деякі птахи не літають*” не є запереченням один одного, тому що вони обоє істинні. Таким чином, пропозиції, що отримані додаванням частки “не” до присудка пропозицій “*Усі x суть P*” і “*Деякі x суть P*” не є запереченнями цих пропозицій. Універсальним способом побудови заперечення даної пропозиції є додавання словосполучення “*невірно, що*” на початку пропозиції. Таким чином, запереченням пропозиції “*Усі птаха літають*” є пропозиція “*Невірно, що усі птаха літають*”; але ця пропозиція має той же зміст, що і пропозиція “*Деякі птахи не літають*”. Запереченням пропозиції “*Деякі птахи літають*” є пропозиція “*Невірно, що деякі птахи літають*”, що має той же зміст, що і пропозиція “*Усі птаха не літають*”.

Домовимося заперечення пропозиції $\forall x(P(x))$ записувати як $\overline{\forall x(P(x))}$, а заперечення пропозиції $\exists x(P(x))$ – як $\overline{\exists x(P(x))}$. Очевидно, що пропозиція $\overline{\forall x(P(x))}$ має той же зміст, а отже, те ж значення істинності, що і пропозиція $\overline{\exists x(P(x))}$, а пропозиція $\overline{\exists x(P(x))}$ – той же зміст, що $\forall x(\overline{P(x)})$. Інакше кажучи, $\overline{\forall x(P(x))}$ рівносильно $\exists x(\overline{P(x)})$; $\overline{\exists x(P(x))}$ рівносильно $\forall x(\overline{P(x)})$.

Квантори загальності та існування називають *двоїстими* відносно один одного. З'ясуємо тепер, як будувати заперечення пропозиції, що починається з декількох кванторів, наприклад, такого: $\forall x \exists y \forall z (P(x, y, z))$.

Послідовно застосовуючи сформульоване вище правило, одержимо: $\overline{\forall x \exists y \forall z (P(x, y, z))}$ рівносильно $\exists x (\overline{\exists y \forall z (P(x, y, z))})$, що рівносильно $\exists x \forall y (\overline{\forall z (P(x, y, z))})$, що рівносильно $\exists x \forall y \exists z (\overline{P(x, y, z)})$.

1.1.2.4. Поняття формули алгебри предикатів

В алгебрі предикатів кожне висловлення, як змінне, так і постійне, є *формулою*.

Якщо F – n -місна предикатна змінна або постійний предикат, а x_1, x_2, \dots, x_n – предметні змінні або предметні сталі (не обов'язково всі різні), то $F(x_1, x_2, \dots, x_n)$ є формулою. Така формула називається *елементарною*, у ній предметні змінні є вільними, не зв'язаними кванторами.

Якщо A і B – формули, причому, такі, що та сама предметна змінна не є в одній з них зв'язаною, а в іншій – вільною, то висловлення $A \vee B$, $A \wedge B$, $A \rightarrow B$ є формулами. У цих формулах ті змінні, котрі у вихідних формулах були вільними, є вільними, а ті, котрі були зв'язаними, є зв'язаними.

Якщо A – формула, то \overline{A} – формула, і характер предметних змінних при переході від формули A к формулі \overline{A} не змінюється.

Якщо $A(x)$ – формула, у яку предметна змінна x входить вільно, то слова $\forall x A(x)$ і $\exists x A(x)$ є формулами, причому, предметна змінна входить до них зв'язано.

Наприклад, якщо $P(x)$ і $Q(x, y)$ – одномісний і двомісний предикати, а q, r – змінні висловлення, то формулами будуть, наприклад, вирази:

$$q, P(x), P(x) \wedge Q(x, y), \forall x P(x) \rightarrow \exists x Q(x, y), \overline{(Q(x, y) \vee q)} \rightarrow r.$$

Не є формулою, наприклад, вираз: $\forall x Q(x, y) \rightarrow P(x)$. Тут порушена певна умова, тому що у вираз $\forall x Q(x, y)$ змінна x входить зв'язано, а у вираз $P(x)$ змінна x входить вільно.

З визначення формули логіки предикатів зрозуміло, що усяка формула алгебри висловлень є формулою логіки предикатів.

1.1.2.5. Значення формули логіки предикатів

Про логічне значення формули логіки предикатів можна говорити лише тоді, коли задана множина M , на якій визначені вхідні в цю формулу предикати. Логічне значення формули логіки предикатів залежить від значень трьох видів змінних:

- значень вхідних у формулу змінних висловлень;
- значень вільних предметних змінних з множини M ;
- значень предикатних змінних.

При конкретних значеннях кожного з трьох видів змінних формула логіки предикатів стає висловленням, що має істинне або хибне значення.

Як приклад розглянемо формулу $\exists y \forall z (P(x, y) \rightarrow P(y, z))$, (1) у якій двомісний предикат $P(x, y)$ визначений на множині $M \times M$, де $M = \{0, 1, 2, \dots, n, \dots\}$, тобто $M \times M = N \times N$.

У формулу (1) входить змінний предикат $P(x, y)$, предметні змінні x, y, z , дві з яких y і z – зв'язані кванторами, а x – вільна.

Візьмемо за конкретне значення предиката $P(x, y)$ фіксований предикат $P^0(x, y)$: “ $x < y$ ”, а вільній змінній x додамо значення $x^0 = 5 \in M$. Тоді при значеннях y , менших $x^0 = 5$, предикат $P^0(x^0, y)$ приймає значення “хибність”, а імплікація $P(x, y) \rightarrow P(y, z)$ при усіх $z \in M$ приймає значення “істина”, тобто висловлення $\exists y \forall z (P^0(x, y) \rightarrow P^0(y, z))$ має значення “істина”.

1.1.2.6. Рівносильні формули логіки предикатів

Визначення 1.

Дві формули логіки предикатів A і B називаються *рівносильними* на області M , якщо вони приймають однакові логічні значення при всіх значеннях вхідних у них змінних, віднесених до області M .

Визначення 2.

Дві формули логіки предикатів A і B називаються *рівносильними*, якщо вони рівносильні на будь-якій області.

Зрозуміло те, що всі рівносильності алгебри висловлень будуть вірні, якщо в них замість змінних висловлень підставити формули логіки предикатів. Але, крім того, мають місце рівносильності самої логіки предикатів. Розглянемо основні з цих рівносильностей.

Нехай $A(x)$ і $B(x)$ – змінні предикати, а C – змінне висловлення (або формула, що не містить x).

Тоді мають місце рівносильності :

$$1. \overline{\forall x A(x)} \equiv \exists x \overline{A(x)}.$$

$$2. \overline{\exists x A(x)} \equiv \forall x \overline{A(x)}.$$

$$3. \overline{\forall x A(x)} \equiv \overline{\exists x \overline{A(x)}}.$$

$$4. \overline{\exists x A(x)} \equiv \overline{\forall x \overline{A(x)}}.$$

$$5. \forall x A(x) \wedge \forall x B(x) \equiv \forall x [A(x) \wedge B(x)]$$

$$C \wedge \forall x B(x) \equiv \forall x [C \wedge B(x)].$$

$$C \vee \forall x B(x) \equiv \forall x [C \vee B(x)]$$

$$C \rightarrow \forall x B(x) \equiv \forall x [C \rightarrow B(x)]$$

$$\forall x [B(x) \rightarrow C] \equiv \exists x B(x) \rightarrow C.$$

$$\exists x [A(x) \vee B(x)] \equiv \exists x A(x) \vee \exists x B(x).$$

$$\exists x [C \vee B(x)] \equiv C \vee \exists x B(x).$$

$$\exists x [C \wedge B(x)] \equiv C \wedge \exists x B(x).$$

$$\exists x A(x) \wedge \exists y B(y) \equiv \exists x \exists y [A(x) \wedge B(y)].$$

$$\exists x [C \rightarrow B(x)] \equiv C \rightarrow \exists x B(x).$$

Рівносильність 1 означає той простий факт, що, якщо не для всіх x істинно $A(x)$, то існує x , при якому буде істиною $\overline{A(x)}$.

Рівносильність 2 означає той простий факт, що, якщо не існує x , при якому істинно $A(x)$, то для всіх x буде істиною $\overline{A(x)}$.

Рівносильності 3 і 4 виходять з рівносильностей 1 і 2, відповідно, якщо від обох їхніх частин узяти заперечення і скористатися законом подвійного заперечення.

1.1.2.7. Загальнозначущі формули логіки предикатів

$$\overline{\forall x}P(x) \equiv \exists x\overline{P(x)}.$$

$$\overline{\exists x}P(x) \equiv \forall x\overline{P(x)}.$$

$$\forall xP(x) \equiv \overline{\exists x\overline{P(x)}}.$$

$$\exists xP(x) \equiv \overline{\forall x\overline{P(x)}}.$$

$$\forall x[P(x) \wedge Q(x)] \equiv \forall xP(x) \wedge \forall xQ(x).$$

$$\exists x[P(x) \vee Q(x)] \equiv \exists xP(x) \vee \exists xQ(x).$$

$$\forall x\forall yP(x, y) \equiv \forall y\forall xP(x, y).$$

$$\exists x\exists yP(x, y) \equiv \exists y\exists xP(x, y).$$

$$\exists x\forall yP(x, y) \Rightarrow \forall y\exists xP(x, y).$$

$$\forall xP(x) \vee \forall xQ(x) \Rightarrow \forall x[P(x) \vee Q(x)].$$

$$\exists x[P(x) \wedge Q(x)] \Rightarrow \exists xP(x) \wedge \exists xQ(x).$$

$$\forall x[P(x) \Rightarrow Q(x)] \Rightarrow [\forall xP(x) \Rightarrow \forall xQ(x)].$$

$$\forall xP(x) \Rightarrow \exists xP(x).$$

$$P(y) \Rightarrow \exists xP(x) \equiv \exists x[P(y) \Rightarrow P(x)].$$

$$C \Rightarrow \exists xP(x) \equiv \exists x[C \Rightarrow P(x)].$$

$$C \wedge \forall xP(x) \equiv \forall x[C \wedge P(x)].$$

$$C \vee \forall xP(x) \equiv \forall x[C \vee P(x)].$$

$$C \Rightarrow \forall xP(x) \equiv \forall x[C \Rightarrow P(x)].$$

$$\exists x[C \vee P(x)] \equiv C \vee \exists xP(x).$$

$$\exists x[C \wedge P(x)] \equiv C \wedge \exists xP(x).$$

$$\exists xP(x) \wedge \exists yQ(y) \equiv \exists x\exists y[P(x) \wedge Q(y)].$$

$$\exists x[P(x) \Rightarrow C] \equiv \forall xP(x) \Rightarrow C.$$

$$\forall xP(x) \vee \forall xQ(x) \equiv \forall xP(x) \vee \forall yQ(y) \equiv$$

$$\equiv \forall x[P(x) \vee \forall yQ(y)] \equiv \forall x\forall y[P(x) \vee Q(y)]$$

$$\exists xP(x) \wedge \exists xQ(x) \equiv \exists xP(x) \wedge \exists yQ(y) \equiv$$

$$\equiv \exists x[P(x) \wedge \exists yQ(y)] \equiv \exists x\exists y[P(x) \wedge Q(y)].$$

1.1.2.8. Загальнозначущість і здійсненність формул. Проблема можливості розв'язання

Визначення 1.

Формула А логіки предикатів називається *здійсненою* в області М, якщо існують значення змінних вхідних у цю формулу і віднесених до області М (інакше – існує модель), при яких формула А приймає істинні значення.

Визначення 2.

Формула А логіки предикатів називається *здійсненою*, якщо існує область, на якій ця формула здійсненна.

Визначення 3.

Формула А логіки предикатів називається *тотожньо-істинною* в області М (здійсненої), якщо вона приймає істинні значення для всіх значень змінних, вхідних у цю формулу і віднесених до цієї області.

Визначення 4.

Формула А логіки предикатів називається *загальнозначущою*, якщо вона тотожньо-істинна на будь-якій області (на будь-якій моделі).

Якщо дві рівносильні формули логіки предикатів з'єднати знаком еквівалентності \leftrightarrow , то отримана формула буде приймати значення і для будь-якого набору змінних у будь-якій області, тобто буде *загальнозначущою*.

Це поняття є узагальненням поняття тотожньої істинності формули логіки висловлень. Усі логічні закони, подані у логіці висловлень наведеними формулами, є загальнозначущими формулами логіки предикатів і виражають, як і інші загальнозначущі формули, закони логіки мовою логіки предикатів.

Загальнозначимість формули логіки предикатів, наприклад, F позначається як $\vdash F$. Усі загальнозначущі формули можуть бути джерелами нових \vdash формул. Наприклад, підставляючи у закон виключеного третього $x \vee \bar{x}$ – замість x предикат $P(x_1, \dots, x_n)$, одержуємо загальнозначущу формулу $P(x_1, \dots, x_n) \vee \bar{P}(x_1, \dots, x_n)$. При $n=1$ маємо загальнозначущу формулу $P(x) \vee \bar{P}(x)$, і, таким чином, $\forall x [P(x) \vee \bar{P}(x)]$ - загальнозначуща формула логіки предикатів.

З тотожно-істинної формули логіки висловлень $x \vee y \equiv y \vee x$ підстановкою замість x предиката $P(x, y)$, а замість y - предиката $Q(x, y)$ одержуємо загальнозначущу формулу $P(x, y) \vee Q(x, y) \equiv Q(x, y) \vee P(x, y)$ і т.д.

Визначення 5.

Формула A логіки предикатів називається *тотожно-хибною в області M* , якщо вона приймає помилкові значення для всіх значень змінних, входних у цю формулу і віднесених до цієї області (іншими словами, на даній моделі).

Визначення 6.

Формула A логіки предикатів називається *тотожно-хибною (нездійсненою)*, якщо вона тотожно-хибна на будь-якій області (на всякій моделі).

Наприклад, формула $\forall x [P(x) \wedge \bar{P}(x)]$ є тотожно-хибною (нездійсненою) формулою логіки предикатів.

З приведених визначень з очевидністю випливає:

- Якщо формула A загальнозначуща, то вона і здійсненна на будь-якій області (моделі).
- Якщо формула A тотожно-істинна в області M , то вона і здійсненна в цій області.
- Якщо формула A тотожно-хибна в області M , то вона не здійсненна в цій області.
- Якщо формула A не здійсненна, то вона тотожно-хибною помилковою на всякій області (на всякій моделі).

Для того, щоб формула A логіки предикатів була загальнозначуща, необхідно і досить, щоб її заперечення було не здійсненне.

Для того, щоб формула A логіки предикатів була здійсненою, необхідно і досить, щоб формула \bar{A} була не загальнозначуща.

Проблема можливості розв'язання в логіці предикатів формулюється так само, як і в алгебрі висловлень: *чи існують алгоритми, що дозволяють для будь-якої формули A логіки предикатів встановити, до якого типу (класу) вона відноситься, тобто чи є вона загальнозначущою, здійсненою або тотожно-хибною (нездійсненою)*. Якби такий алгоритм

існував, то, як і в алгебрі висловлень, він зводився б до критерію тотожньої істинності будь-якої формули логіки предикатів. Відзначимо, що, на відміну від алгебри висловлень, у логіці предикатів не застосовується метод перебору усіх варіантів значень змінних, що входять у формулу, тому що таких варіантів може бути нескінченна множина.

1.2. ЛОГІЧНІ ЧИСЛЕННЯ

1.2.1. Аксиоматичний метод

Література: [7,31-39;8,147-187;12,201-207;12,227-230;20,127-164]

Ключові поняття: *аксіоматичний метод, формальна система, формалізована мова, об'єкт-мова, мета-мова, алфавіт, аксіома, правило виведення, логічне числення, аксіоматична система натуральних чисел.*

Одна з важливих причин появи і розвитку математичної полягає в широкому розповсюдженні аксіоматичного методу для побудови формальних теорій (систем), зокрема, таких математичних теорій як геометрія, арифметика, теорія груп тощо. *Аксиоматичний метод* побудови наукових теорій полягає у тому, що певні початкові твердження, які називають *аксіомами*, приймаються як даність без доведення, а всі інші твердження виводяться із них суто логічним шляхом. Уточнення аксіоматичного методу проведемо на основі поняття формальної системи. При належній конкретизації формальних систем отримуємо аксіоматичні системи для того чи іншого розділу математики.

Під *формальними системами* розуміють системи операцій над об'єктами. Термін «формальний» підкреслює, що об'єкти й операції над ними розглядаються суто формально, без будь-яких змістовних інтерпретацій символів. Передбачається, що між символами не існує ніяких зв'язків і відношень, крім тих, які явно описуються засобами самої формальної системи.

Таким чином, під формальною системою будемо розуміти деяку абстрактну (математичну) систему, що складається із

множини аксіом та множини правил виведення, яку можна реалізувати машинним способом, наприклад за допомогою програми ЕОМ.

У формальних системах прагнуть використати тільки синтаксичні аспекти мови, абстрагуючись від питань семантики та прагматики. Дуже часто формалізацію визначають як процес вивчення змісту за допомогою засобів *формалізованої мови*.

Формалізована мова – будь-яка сукупність спеціалізованих мовних засобів із суворо зафіксованими правилами утворення різноманітних виразів і правилами приписування цим виразам певних значень.

Переваги використання формалізованої мови:

- можливість стисло фіксувати і передавати різні відносини між об'єктами, що вивчаються („квадрат суми двох чисел дорівнює квадрату $\equiv (x+y)^2 = x^2 + 2xy + y^2$ ”);
- можливість оцінити характер відношень між об'єктами, що зафіксовані у певній формі (хімічні формули, наприклад, H_2O);
- можливість однозначно виявити логічну структуру думку, відрізнити логічний синтаксис від лінгвістичного.

В структурі формалізованої мови розрізняють:

- *об'єкт-мову* (система знакових засобів, сукупність різноманітних формул, що фіксує в знаковій формі логічну структуру міркувань, логічні властивості складових елементів міркувань та логічні відносини між елементами міркувань (синтаксична частина));
- *мета-мову* (логічний аналіз об'єкт-мови, що пояснює, носіями яких саме властивостей і відношень є ті чи інші знаки об'єкт-мови, які саме логічні функції фіксують відповідні комбінації і утворення об'єкт-мови (семантична частина)).

При аксіоматичній побудові формальних систем, зокрема, математичної теорії, попередньо вибирається деяка система первісних понять і відношень між ними. Ці поняття і відношення називаються *основними*. Далі, без доведення, тобто як *аксіоми*, береться деяка система властивостей основних понять і відношень. Після того всі нові поняття теорії визначаються через основні або раніше визначені поняття, а всі твердження «логічно» виводяться з аксіом або з вже доведених тверджень. *Правилами виведення* називають чітко сформульовані процедури, які дають змогу утворювати нові поняття з тих, що вже існують.

Система основних понять і аксіом теорії може вибиратися по-різному. Однак теорія буде цікавою і життєвою лише в тому випадку, якщо система її основних понять і аксіом відбиває певні об'єкти і зв'язки між ними, що існують у реальному світі.

Математична логіка широко використовується при обґрунтуванні математики як формальної теорії. У зв'язку з цим виникає завдання побудови формалізованої аксіоматичної теорії самої математичної логіки. Вибираючи різні системи аксіом і правила виведення одних формул з інших, ми будемо отримувати різні синтаксичні логічні теорії. Кожну з цих теорій прийнято називати *логічним численням*. Далі розглянемо класичні числення висловлень і предикатів.

Попередньо ще раз відзначимо, що будь-яка формальна аксіоматична теорія характеризується:

1. *алфавітом*, тобто множиною символів, що використовуються для побудови формул теорії;
2. *системою аксіом*, тобто деякою множиною формул, що називаються аксіомами;
3. *правилами виведення*, що дозволяють з одних формул одержувати інші формули розглянутої формальної теорії.

Формальна аксіоматична система повинна бути:

- *несуперечливою* (з аксіом системи не можна зробити два взаємовиключні висновки);
- *повною* (система аксіом припускає лише одну реалізацію);

- незалежною (жодну з аксіом цієї системи не можна вивести з інших аксіом, тобто довести як теорему, базуючись на решті аксіом системи).

Зауважимо, що формальна система має бути такою, яка алгоритмічно реалізується. Мається на увазі, що для пошуку будь-якого виведення у формальній системі повинен не тільки існувати алгоритм виведення, але цей алгоритм має ще допускати реалізацію на існуючих ЕОМ. Інакше формальна система залишиться абстрактною.

Історично теорія формальних систем, як і *теорія алгоритмів*, виникла в межах основ математики при дослідженні побудови аксіоматичних теорій та методів доведення в них. З їх вивчення і продовжимо знайомство.

Наведемо приклад застосування аксіоматичного методу в арифметиці.

Ми використовуємо термін “*натуральні числа*” у змісті, що відрізняється від звичайного – *нуль* ми теж включаємо в натуральні числа. Таке використання цього терміна прийняте серед західних математиків. Ми пишемо “*натуральні числа*” для того, щоб не писати кожного разу “*цілі додатні числа*”. Слід зазначити, що саме такий підхід є більш прийнятним для формальних систем теорії алгоритмів.

Аксіоми натуральних чисел

Ми розглядаємо множину ω об'єктів, які називаємо *натуральними числами*. Одне з натуральних чисел називається *нулем* і позначається як “0”. Для будь-якого натурального числа n одне з натуральних чисел називається *наступним* за числом n і позначається як n' .

Множина натуральних чисел така, що задовольняє наступним аксіомам:

Аксіома 1.

Для будь-якого натурального числа n : $n' \neq 0$.

Аксіома 2.

Для будь-яких натуральних чисел m і n : якщо $m' = n'$, то $m = n$.

Аксиома 3.

Нехай A є підмножиною множини ω з наступними властивостями:

$$0 \in A;$$

для будь-якого натурального числа n : якщо $n \in A$, то $n' \in A$, тоді $A = \omega$.

Ці аксіоми були введені Джузеппе Пеано в 1889 році.

Початкові задачі

Визначення 1.

$$1 = 0', 2 = 1', 3 = 2', 4 = 3'.$$

$$n' = n + 1$$

Визначення 2 (Сума).

Функція $g(m, n)$ називається *сумою* m і n і позначається як $m + n$.

Так, для будь-яких натуральних чисел m і n :

$$m + 0 = m,$$

$$m + n' = (m + n)'$$

Таким чином,

$$g(m, 0) = m,$$

$$g(m, n') = g(m, n)'$$

Визначення 2 (Добуток).

Функція $f(m, n)$ називається *добутком* m і n і позначається як $m * n$.

Так, для будь-яких натуральних чисел m і n :

$$m * 0 = 0,$$

$$m * (n + 1) = (m * n) + m \quad \text{або} \quad (m * n') = (m * n) + m$$

Таким чином,

$$f(m, 0) = 0,$$

$$f(m, n + 1) = f(m, n) + m \quad \text{або} \quad f(m, n') = f(m, n) + m$$

Отже, дана формальна аксіоматична система характеризує натуральні числа.

1.2.2. Числення висловлень

Література: [7,31-39;8,147-187;12,201-207;20,127-164]

Ключові поняття: *числення висловлень, формули, доказ формул, числення висловлень, проблеми аксіоматичного числення висловлень.*

Даючи опис алгебри висловлень, ми користувалися логічними значеннями висловлень (“істина”, “неправда”). Але поняття істинності і хибності не математичні. Ці поняття в багатьох випадках суб’єктивні і, скоріше, відносяться до філософії.

У зв’язку з цим бажано побудувати математичну логіку, не користуючись поняттями істинності і хибності. Необхідно також при цій побудові не застосовувати самих законів логіки.

1.2.2.1. Поняття формули числення висловлень

Числення висловлень – це аксіоматична логічна система, інтерпретацією якої є алгебра висловлень.

Опис будь-якого числення містить у собі опис символів цього числення (алфавіту), формул, що є скінченими конфігураціями символів, і визначення виведених формул.

Алфавіт числення висловлень складається із символів трьох категорій:

1. Символи першої категорії: $x, y, z, \dots, x_1, x_2, \dots$. Ці символи будемо називати *змінними висловленнями*.

2. Символи другої категорії: \neg, \rightarrow . Вони носять загальну назву *логічних зв’язок*.

3. Третю категорію складає пари символів „(, ,)”, що називаються дужками.

Інших символів числення висловлень не має.

Числення висловлень визначається наступним чином:

1. Задаються символи числення висловлень;

2. Задаються формули:

- всі пропозиційні змінні $x, y, \dots, z, A, B, \dots, Z$ – формули
- якщо A, B, \dots, Z – формули, то всі їх комбінації є також формулами

3. Задаються аксіоми числення.

Аксіома 1(A1). $A \rightarrow (B \rightarrow A)$

Аксіома

2(A2).

$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

Аксіома 3(A3). $(\bar{B} \rightarrow \bar{A}) \rightarrow ((\bar{B} \rightarrow A) \rightarrow B)$

Імплікація та інверсія є достатніми для побудови будь-якої формули. При необхідності вводяться додаткові формули:

$A \wedge B \equiv (A \rightarrow \bar{B})$

$A \vee B \equiv (\bar{A} \rightarrow B)$

$A \leftrightarrow B \equiv ((A \rightarrow B) \wedge (B \rightarrow A))$

4. Задаються правила виведення, наприклад

$$\frac{A, A \rightarrow B}{B}$$
 - правило *Modus Ponens*

Формули числення висловлень являють собою послідовності символів алфавіту числення висловлень. Для позначення формул будемо користуватися великими буквами латинського алфавіту. Ці букви не є символами числення. Вони являють собою тільки умовні позначки формул.

Будь-яка змінна x, y, z, \dots є формулою. Якщо A і B - формули, то слова $(A \rightarrow B), \bar{A}$ - теж формули. Ніякий інший рядок символів не є формулою. Змінні висловлення будемо називати *елементарними формулами*.

Одночасно з поняттям формули вводиться поняття *підформули* або частини формули. Підформулою елементарної формули є вона сама.

Якщо формула має вигляд \bar{A} , то її підформулами є: вона сама, формула A і всі підформули формули A .

Якщо формула має вигляд $(A*B)$ (тут і надалі під символом $*$ будемо розуміти символ \rightarrow), то її підформулами є: вона сама, формули A і B і всі підформули формул A і B .

1.2.2.2. Визначення доказової (виведеної) формули

Наступним етапом у побудові числення висловлень є виділення класу *доказових* (виведених) формул.

Визначення доказових формул має той же характер, що і визначення формули.

Спочатку визначаються вихідні доказові виведені формули (аксіоми), а потім визначаються правила виведення, що дозволяють з наявних доказових формул одержати нові доказові формули.

Утворення доказової формули з вихідних доказових формул шляхом застосування правил виведення називається *виведенням* (*доказом*) даної формули з аксіом.

Скінчена послідовність формул F називається *виведенням*, якщо кожна формула цієї послідовності або є аксіомою, або випливає з попередніх формул послідовності за правилами виведення. Елементи послідовності формул F називаються посилками виведення або гіпотезами.

Формула A для якої існує виведення, називається *теоремою*.

Факт утворення теореми A з послідовності F позначають як $F \vdash A$. Звідси випливає, що аксіоми – це $\vdash A$.

Приклад доказу деякої теореми:

Довести: $\vdash F \rightarrow F$

Доведення: необхідно побудувати виведення формул $F \rightarrow F$ з аксіом.

Таким чином. Виведення матиме наступний вигляд:

1. $(F \rightarrow ((F \rightarrow F) \rightarrow F)) \rightarrow ((F \rightarrow (F \rightarrow F)) \rightarrow (F \rightarrow F))$
2. $F \rightarrow ((F \rightarrow F) \rightarrow F)$
3. $(F \rightarrow (F \rightarrow F)) \rightarrow (F \rightarrow F)$
4. $(F \rightarrow (F \rightarrow F))$
5. $F \rightarrow F$

Пояснення:

- формула 1 є аксіомою A_2 , в якості формул A і C взята формула F , а в якості B – формула $F \rightarrow F$;
- формула 2 є аксіомою A_1 , де B – це формула $F \rightarrow F$;

- формула 3 отримана з формул 1 і 2 за правилом *modus ponens*;
- формула 4 є аксіомою A1;
- формула 5 отримана з формул 3 і 4 за правилом *modus ponens*.

Таким чином, процес одержання доказових формул будемо називати *доказом (висновком) формул*. Це процес послідовного переходу від однієї доказової формули до іншої за допомогою аксіом, правила підстановки і правила висновку на кожному кроці (у визначеному змісті це аналог рівносильним перетворенням в алгебрі логіки), так що висновок навіть простої формули може виявитися, у силу його складності, досить громіздким.

1.2.2.3. Проблеми аксіоматичного числення висловлень

Всяка аксіоматична теорія для її обґрунтування вимагає розгляду чотирьох проблем:

- проблеми можливості розв'язання,
- проблеми несуперечності,
- проблеми повноти,
- проблеми незалежності.

Проблема можливості розв'язання числення висловлень

Проблема можливості розв'язання числення висловлень полягає в доказі існування алгоритму, що дозволив би для будь-якої заданої формули числення висловлень визначити, чи є вона доказовою або ні.

Має місце теорема: проблема можливості розв'язання для числення висловлень розв'язна.

Дійсно, будь-яка формула числення висловлень може розглядатися як формула алгебри висловлень, і, отже, можна розглядати її логічні значення на різних наборах значень вхідних у неї змінних.

Нехай A – будь-яка формула числення висловлень, а x_1, x_2, \dots, x_n – перелік вхідних у неї змінних. Обчислимо $R_{a_1, a_2, \dots, a_n}(A)$ на всіх наборах значень a_1, a_2, \dots, a_n вхідних у неї

змінних. Якщо при цьому $R_{a_1, a_2, \dots, a_n}(A) = 1$, на всіх наборах a_1, a_2, \dots, a_n , то формула A – тотожно-істинна, і, виходить, за теоремою про можливість доведення тотожно-істинної формули вона доказова.

Якщо ж існує набір значень змінних $a_1^0, a_2^0, \dots, a_n^0$ таких, що $R_{a_1^0 \dots a_n^0}(A) = 0$, то формула A – не тотожно-істинна, і, виходить, за теоремою 18 вона не доказова.

Проблема несуперечності числення висловлень

Логічне числення називається несуперечливим, якщо в ньому не доказові ніякі дві формули, з яких одна є запереченням іншої.

Інакше кажучи, аксіоматичне числення називається несуперечливим, якщо в ньому не існує така формула A , що доказово як формулу A , так і формулу \bar{A} .

Проблема несуперечності полягає в з'ясуванні питання: є дане числення несуперечливим чи ні?

Якщо у численні виявляються доказові формули виду A і \bar{A} , то таке числення називається суперечливим. У розглянутому нами численні висловлень неможливо вивести одночасно формули A і \bar{A} , тобто це числення висловлень несуперечливо.

Проблема повноти числення висловлень

Визначення 1.

Аксіоматичне числення висловлень називається повним у вузькому змісті, якщо додавання до списку його аксіом будь-якої не довідної у численні формули як нову аксіому приводить до суперечливого числення.

Визначення 2.

Числення висловлень називається повним у широкому змісті, якщо будь-яка тотожно-істинна формула в ньому доказова.

З цих визначень випливає, що проблема повноти числення висловлень містить два питання:

- чи можна розширити систему аксіом аксіоматичного числення шляхом додавання до неї в якості нової

аксіоми яку-небудь недовідну в цьому численні формулу?

- чи є всяка тотожно-істинна формула алгебри висловлень доказовою у численні висловлень?

Розгляньте нами числення висловлень повно як у вузькому змісті, так і в широкому.

Проблема незалежності аксіом числення висловлень

Для будь-якого аксіоматичного числення виникає питання про незалежність його аксіом. Питання це формулюється так : *чи можна яку-небудь аксіому вивести з інших аксіом, застосовуючи правила висновку даної системи?*

Якщо для деякої аксіоми системи це можливо, то цю аксіому можна виключити зі списку аксіом системи, і логічне числення при цьому не зміниться, тобто клас доказових формул залишиться без змін.

Визначення 3.

Аксіома А називається *незалежною* від всіх інших аксіом числення, якщо вона не може бути виведена з інших аксіом. Система аксіом числення називається *незалежною*, якщо кожна аксіома системи незалежна.

Розглянута нами система аксіом числення висловлень незалежна.

1.2.3. Числення предикатів

Література:[7,31-39;8,147-187;12,227-230;20,127-164]

Ключові поняття: *числення предикатів, аксіоми числення предикатів, правила виведення числення предикатів.*

У алгебрі предикатів, на відміну від алгебри висловлення, не існує ефективного способу для розпізнавання загальнозначущості формул. Тому аксіоматичний метод стає істотним при вивченні формул, які містять квантори. Визначення загальнозначущих формул так само, як і в численні висловлень, здійснюється заданням деякої сукупності формул, що називаються аксіомами, і заданням правил виведення, що

дають змогу з одних загальнозначущих формул одержувати інші.

У численні предикатів будемо користуватися наступним алфавітом:

- символи p, q, r, \dots - змінні висловлення, що приймають два значення: 1- істина, 0 – хибність.
- предметні змінні – x, y, z, \dots , які збігаються зі значеннями з деякої множини M ;
- x^0, y^0, z^0 – предметні константи, тобто значення предметних змінних.
- $P(\cdot), Q(\cdot), F(\cdot), \dots$ - одномісні предикатні змінні;
- $Q(\cdot; \dots, \cdot), R(\cdot; \dots, \cdot)$ – n -місні предикатні змінні.
- $P^0(\cdot), Q^0(\cdot; \dots, \cdot)$ – символи постійних предикатів.
- символи логічних операцій: $\wedge, \vee, \rightarrow, \neg$.
- символи кванторних операцій: $\forall x, \exists x$.
- допоміжні символи: дужки, коми.

Аксіоми числення предикатів

Перша група аксіом

$$P(x) \rightarrow (Q(x) \rightarrow P(x))$$

$$(P(x) \rightarrow (Q(x) \rightarrow F(x))) \rightarrow$$

$$\rightarrow ((P(x) \rightarrow Q(x)) \rightarrow (P(x) \rightarrow F(x)))$$

Друга група аксіом (до імплікації присдана кон'юнкція):

$$P(x) \wedge Q(x) \rightarrow P(x)$$

$$P(x) \wedge Q(x) \rightarrow Q(x)$$

$$(F(x) \rightarrow P(x)) \rightarrow$$

$$\rightarrow ((F(x) \rightarrow Q(x)) \rightarrow (F(x) \rightarrow P(x) \wedge Q(x)))$$

Третя група аксіом (до імплікації присдана диз'юнкція):

$$P(x) \rightarrow P(x) \vee Q(x)$$

$$Q(x) \rightarrow P(x) \vee Q(x)$$

$$(P(x) \rightarrow F(x)) \rightarrow$$

$$\rightarrow ((Q(x) \rightarrow F(x)) \rightarrow (P(x) \vee Q(x) \rightarrow F(x)))$$

Теоретичний курс

Четверта група аксіом (до імплікації приєдналося заперечення):

$$(P(x) \rightarrow Q(x)) \rightarrow (\overline{Q(x)} \rightarrow \overline{P(x)})$$

$$\overline{\overline{P(x)}} \rightarrow P(x)$$

$$\overline{P(x)} \rightarrow \overline{\overline{P(x)}}$$

П'ята група аксіом (використання кванторів):

$$\forall x P(x) \rightarrow P(x)$$

$$P(x) \rightarrow \exists x P(x)$$

Правила виведення предикатів

Правило виведення *modus ponens*

$$\frac{P(x), P(x) \rightarrow Q(x)}{Q(x)}$$

$$Q(x)$$

Правило зв'язування квантором загальності

$$\frac{Q(x) \rightarrow P(x)}{Q(x) \rightarrow (\forall x P(x))}$$

$$Q(x) \rightarrow (\forall x P(x))$$

Правило зв'язування квантором існування

$$\frac{P(x) \rightarrow Q(x)}{(\exists x P(x)) \rightarrow Q(x)}$$

$$(\exists x P(x)) \rightarrow Q(x)$$

РОЗДІЛ 2. ТЕОРІЯ АЛГОРИТМІВ

2.1. ТЕОРІЯ АЛГОРИТМІВ. ОСНОВНІ ПОЛОЖЕННЯ ТА ОЗНАЧЕННЯ ТЕОРІЇ АЛГОРИТМІВ

2.1.1. Поняття про алгоритм. Еволюція поняття алгоритм.

Література: [8,108-113;8,116-118;12,360-362;18,7-12]

Ключові поняття: *алгоритм, властивості алгоритму, вимоги до алгоритмів, підходи до визначення алгоритму, обчислювальна функція, детерміновані пристрої, фінітний комбінаторний процес Поста, абстрактна машина Тьюрінга, нормальні алгоритми Маркова.*

Поняття алгоритму належить до фундаментальних понять математики, кібернетики та інформатики.

Під алгоритмом розуміють чіткі інструкції про виконання в певній послідовності деякої системи операцій для рішення задач певного класу. Звичайно, запропоноване визначення швидше роз'яснює слово “алгоритм”, а не дає точного математичного визначення поняття алгоритм. Незважаючи на це, в науці довгий час використовувалися саме інтуїтивні визначення цього поняття про що свідчать і наступні приклади.

Алгоритм - чітко визначений опис способу рішення задачі у вигляді кінцевої послідовності дій.

Алгоритм - правило, що сформульоване на деякій мові та визначає процес переробки вихідних даних в необхідні результати.

Алгоритм - сукупність правил, що визначає ефективну процедуру рішення будь-якої задачі деякого заданого класу задач.

Алгоритм - вказівки, що однозначно визначають процес перетворення вихідної інформації у вигляді послідовності елементарних дискретних кроків, які дозволяють за кінцеву їх кількість отримати необхідний результат.

2.1.1.1. Властивості алгоритмів

До основних властивостей алгоритму відносять наступні характеристики:

Детермінованість - визначення кроків алгоритму, тобто після кожного кроку або зазначається, який крок слід роботи далі, або дається команда на зупинку. Ця властивість означає, що застосування алгоритму до тих самих даних має призвести до одного і того самого результату.

Масовість-алгоритм може бути використаний для розв'язання цілого класу задач одного типу.

Результативність - виконання алгоритму має або закінчитися результатом, або інформацією про те, чому не може бути одержаний результат.

Зрозумілість-алгоритм має бути зрозумілим конкретному виконавцю, який повинен виконати кожну команду алгоритму у суворій послідовності з її призначенням.

Дискретність-можливість розбиття алгоритму на скінчену кількість етапів, причому результати попереднього етапу є вхідними для наступного.

2.1.1.2. Вимоги до алгоритмів

Серед вимог, що подані до алгоритму, необхідно вирізняти наступні:

- будь-який алгоритм застосовується для початкових даних і видає результат; маючи за мету в подальшому уточнити поняття алгоритму, необхідно уточнити також і поняття даних, тобто зазначити, які вимоги задовольняють об'єкти, щоб з ними могли працювати алгоритми;
- дані та алгоритм розміщуються в пам'яті, яка вважається однорідною та дискретною, іноді навіть нескінченною;
- у якості виконавця може виступати людина, різні технічні пристрої.

Слід зазначити, що навіть при виконанні всіх зазначених вимог не всяка послідовність дій є алгоритмом.

2.1.2. Підходи до визначення алгоритму

З метою формалізації поняття алгоритму, починаючи з 30-років ХХ ст., було здійснено цілий ряд досліджень, які мали за мету виявити засоби, що фактично б залучалися для побудови алгоритмів. Проблема полягала в тому, щоб на цій основі дати визначення поняття алгоритм, яке було б досконалим не лише з точки зору формальної точності, але й з точки зору фактичного існування сутності даного поняття. При цьому дослідники виходили з точки зору різних технічних та логічних уявлень, внаслідок чого було запропоновано декілька алгоритмічних моделей, що на сьогодні використовуються для формалізації поняття алгоритм.

2.1.2.1. Алгоритм як обчислювальна функція.

Історично першою формалізацією поняття алгоритму були рекурсивні функції. Основана дана модель на тому положенні, що будь-який алгоритм чи алгоритмічну проблему можна звести до обчислення значення деякої цілочисельної функції цілочисельних аргументів.

В даній теорії алгоритмом прийнято називати систему обчислень, що для деякого класу задач Z із умови A дозволяє за допомогою однозначно визначеної послідовності операцій, що здійснюються “механічно”, отримати результат B .

Для доведення, подамо всі початкові умови задачі Z у вигляді послідовності:

$$\alpha_0, \alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n,$$

а розв’язки як:

$$\beta_0, \beta_1, \beta_2, \beta_3, \dots, \beta_m.$$

Якщо ж позначити через F множину номерів n тих умов α_n , які алгоритм може перетворити в рішення, то результат роботи алгоритму, що здійснює перетворення $\alpha_n \rightarrow \beta_m$ однозначно визначається заданою на F числовій функції $m=f(n)$.

Таким чином, довільний алгоритм можна звести до обчислення значення деякої числової функції. І, навпаки, якщо для функції f існує алгоритм, який призводить до стандартного запису значення функції $m=f(n)$, то функція f називається *алгоритмічно обчислювальною* або просто *обчислювальною*.

Як було зазначено раніше, у теорії алгоритмів, зокрема і в теорії обчислювальних функцій, прийнято конструктивний, фінітний підхід, основною рисою якого є те, що вся множина об'єктів (у нашому випадку – функцій) будується зі скінченного числа початкових об'єктів за допомогою простих операцій та операторів, ефективна виконуваність яких досить очевидна.

2.1.2.2. Алгоритмічні моделі на основі детермінованих пристроїв

Другий підхід щодо уточнення поняття алгоритму базується на уявленні про алгоритм як деякий детермінований пристрій, що здатний виконувати в кожний окремий момент лише чітко фіксовані елементарні операції. Основними теоретичними моделями даного типу є фінітний процес Поста та машини Тьюрінга. Дані алгоритмічні моделі суттєво вплинули на розуміння логічної природи майбутніх ЕОМ. Іншою моделлю даного типу є машини з довільним доступом, що була запропонована в 70-х роках ХХ ст. і використовувалася для моделювання реальних ЕОМ та одержання оцінки складності обчислень.

2.1.2.3. Нормальні алгоритми Маркова

Алгоритмічними моделями третього типу є перетворення слів у довільному алфавіті за допомогою операцій підстановки, тобто заміни однієї частини слова на іншу. Основною теоретичною моделлю даного типу є нормальні алгоритми Маркова. Вони є строгими означеннями алгоритму, який уточнює інтуїтивне уявлення на основі алфавітно-словарного підходу.

Слід зазначити, що наведені алгоритмічні моделі, які були запропоновані для формалізації поняття алгоритму, математично еквівалентні. Але на практиці вони суттєво різняться з точки зору складності, що виникає при їх реалізації.

Незважаючи на це, всі алгоритмічні моделі, так сама як і вся теорія алгоритмів, мають великий вплив на розвиток ЕОМ та практику програмування. Адже саме на основі цих моделей

були реалізовані різні напрямки в програмуванні. Так, мікропрограмування базується на ідеї машини Тьюрінга, структурне програмування запозичило свої конструкції з теорії рекурсивних функцій, а мови символічної обробки інформації (ПРОЛОГ, РЕФАЛ) беруть початок від нормальних алгоритмів Маркова.

2.2. АЛГОРИТМІЧНІ МОДЕЛІ

2.2.1. Обчислювальні функції

Література: [8,116-117;8,118-121;12,365-369;15,30-49;16,23-29]

Ключові поняття: *обчислювальна функція, нуль-функція, функція наступності, функція проєкції, оператор суперпозиції, оператор примітивної рекурсії, оператор мінімізації, примітивно-рекурсивні функції, частково-рекурсивні функції, теза Черча.*

2.2.1.1. Поняття про обчислювальну функцію.

Алгоритмом прийнято називати систему обчислень, що для деякого класу задач Z із умови A дозволяє за допомогою однозначно визначеної послідовності операцій, що здійснюються “механічно”, отримати результат B .

Для доведення, подамо всі початкові умови задачі Z у вигляді послідовності:

$$\alpha_0, \alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n,$$

а розв’язки як:

$$\beta_0, \beta_1, \beta_2, \beta_3, \dots, \beta_m.$$

Якщо ж позначити через F множину номерів n тих умов α_n , які алгоритм може перетворити в рішення, то результат роботи алгоритму, що здійснює перетворення $\alpha_n \rightarrow \beta_m$ однозначно визначається заданою на F числовою функцією $m=f(n)$.

Таким чином, довільний алгоритм можна звести до обчислення значення деякої числової функції. І, навпаки, якщо для функції f існує алгоритм, який призводить до стандартного

запису значення функції $m=f(n)$, то функція f називається *алгоритмічно обчислювальною* або просто *обчислювальною*.

Як було зазначено раніше, у теорії алгоритмів, зокрема і в теорії обчислювальних функцій, прийнято конструктивний, фінітний підхід, основною рисою якого є те, що вся множина об'єктів (у нашому випадку – функцій) будується зі скінченного числа початкових об'єктів за допомогою простих операцій та операторів, ефективна виконуваність яких досить очевидна.

При цьому слід зазначити, що у теорії обчислювальних функцій визначають множину натуральних чисел $N=0,1,2,3,\dots$ і розглядають тільки числові функції, розуміючи під ними функції n змінних (n -місні функції), аргументи і значення яких належать N .

Виходячи з вищесказаного, надамо означення обчислювальної функції:

Обчислювальна функція – це числова функція $f(x_1, x_2, x_3, \dots, x_n)$, значення якої можна обчислювати за допомогою деякого алгоритму на підставі відомих значень аргументу.

Найпростіші функції

Розглянемо клас числових функцій, що використовуються в якості базису для побудови обчислювальних функцій.

$O(x)=0$ (нуль-функція)

$S(x)=x+1$ (функція наступності, але не додавання одиниці)

$\Gamma_m^n(x_1, x_2, x_3, \dots, x_m, \dots, x_n) = x_m$ (функція проєкції або введення фіктивних змінних або вибору аргументу).

Оператори, що застосовуються до найпростіших функцій

В якості операторів, застосування яких до базисних функцій призводить до утворення нових функцій оберемо наступні три оператори:

- оператор суперпозиції
- оператор примітивної рекурсії
- оператор мінімізації або найменшого кореня.

Оператор суперпозиції (S_m^n)

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

Нехай задана m -місна функція $F(x_1, x_2, x_3, \dots, x_m)$ та m -на кількість n -місних функцій $f_1(x_1, x_2, x_3, \dots, x_n)$, $f_2(x_1, x_2, x_3, \dots, x_n)$, $f_3(x_1, x_2, x_3, \dots, x_n), \dots, f_m(x_1, x_2, x_3, \dots, x_n)$. Тоді говорять, що n -місна функція $\varphi(x_1, x_2, x_3, \dots, x_n)$ утворилася в результаті підстановки у функцію F замість її аргументів m функцій $f_1, f_2, f_3, \dots, f_m$.

Така підстановка називається суперпозицією S_m^n .

$$\begin{aligned} S_m^n(F, f_1, f_2, \dots, f_m) &= F(f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \\ &\dots, f_m(x_1, x_2, \dots, x_n)) = \\ &= \varphi(x_1, x_2, \dots, x_n) \end{aligned}$$

Приклад: здійснюючи операцію суперпозиції функцій $f(x)=0$ та $g(x)=x+1$ отримуємо $h(x)=g(f(x))=0+1=1$.

Оператор примітивної рекурсії (R^n)

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

$$\begin{aligned} f(x_1, x_2, x_3, \dots, x_n, 0) &= \varphi(x_1, x_2, \dots, x_n) \\ f(x_1, x_2, x_3, \dots, x_n, y+1) &= \psi(x_1, x_2, \dots, x_n, y, f(x_1, x_2, x_3, \dots, x_n, y)) \end{aligned}$$

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

Для правильного 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:

$$\begin{aligned} f(x) &= a \\ f(x, y+1) &= \psi(x, y, f(y)), \text{ де } a - \text{ константа.} \end{aligned}$$

Схеми примітивної рекурсії визначають функцію f рекурсивно не 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) .

Наприклад, необхідно побудувати 2-місну функцію $f(x, y)=x+y$ з елементарних функцій за допомогою оператора примітивної рекурсії.

Теоретичний курс

Функція $f(x,y) = x+y$ визначається функцією проєкції $I^1_1(x) = x$ та функцією слідування $S(x,y,z) = z+1$.

Таким чином,

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

$$f(x,1) = S(x,0,x) = x+1$$

$$f(x,2) = S(x,1,x+1) = x+2$$

.....

$$f(x,y-1) = S(x,y-2,x+y-2) = x+y-1$$

$$f(x,y) = S(x,y-1,x+y-1) = x+y$$

або

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

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

Таким чином, $f(x,y) = x+y = R^1(I^1_1(x), g(x,y,z))$, де $g(x,y,z) = S(z) = z+1$

Приклад: побудувати 2-місну функцію $f(x,y) = x*y$ з елементарних функцій за допомогою оператора примітивної рекурсії.

$$f_{xy}(3,2) = 6$$

$$f_{xy}(3,2) = F_{xy}(3,1) + 3 = 6$$

$$f_{xy}(3,1) = F_{xy}(3,0) + 3 = 3$$

$$f_{xy}(3,0) = 0 = O(x)$$

Приклад: побудувати 2-місну функцію $f(x,y) = x^y$ з елементарних функцій за допомогою оператора примітивної рекурсії.

$$f_x^y(3,2) = 9$$

$$f_x^y(3,2) = f_x^y(3,1) * 3 = 9$$

$$f_x^y(3,1) = f_x^y(3,0) * 3 = 3$$

$$f_x^y(3,0) = 1 = S(O(x))$$

Оператор мінімізації (μ)

Розглянемо обчислювальну n -місну числову функцію $f(x_1, x_2, x_3, \dots, x_n)$. Зафіксуємо деякі значення $x_1, x_2, x_3, \dots, x_{n-1}$ її перших $(n-1)$ аргументів та розглянемо рівняння $f(x_1, x_2, x_3, \dots, x_{n-1}, y) = x_n$

Значення виразу для заданої функції f залежить від значення $x_1, x_2, x_3, \dots, x_{n-1}, x_n$, тому вираз дає часткову функцію змінних параметрів $x_1, x_2, x_3, \dots, x_{n-1}, x_n$, що визначає μ -оператор.

Наприклад, отримаємо за допомогою μ -оператора функцію $d(x,y)$:

$$d(x,y)=\mu z[y+z=x]=\mu z[S((I_2^3(x,y,z), I_3^3(x,y,z)))] = I_1^3(x,y,z).$$

Обчислимо функція $d(7,2)$. Для цього необхідно задати у значення 2 та встановити змінній Z послідовно значення 0,1,2..., кожного разу обчислюючи суму $y+z$. Як тільки вона дорівнюватиме 7, то відповідне значення прийняти за значення $d(7,2)$.

Проведемо обчислення:

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

Таким чином $d(7,2)=5$.

2.2.1.2. Примітивно-рекурсивні функції

Функція називається примітивно-рекурсивною, якщо вона бути утворена з найпростіших функцій за допомогою скінченного числа застосувань операторів суперпозиції S_m^n та примітивної рекурсії R^n

2.2.1.3. Частково-рекурсивні функції

Функція називається частково-рекурсивною, якщо вона бути утворена з найпростіших функцій за допомогою скінченного числа застосувань операторів суперпозиції S_m^n , примітивної рекурсії R^n та мінімізації μ_y .

2.2.1.4. Теза Черча

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):

ТЕЗА ЧЕРЧА - клас алгоритмічно обчислювальних часткових числових функцій збігається з класом усіх частково-рекурсивних функцій.

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

2.2.2. Алгоритмічні моделі на основі детермінованих пристроїв

Література: [8, 116-118; 8, 124-129; 8, 118-124; 8, 137-140; 12, 402-408;

15, 204-254; 18, 60-66; 19, 7-36; 19, 83-89]

Ключові поняття: *фінітний комбінаторний процес Поста, нескінчена стрічка, пристрій керування, стан машини Поста, програма машини Поста, абстрактна машина Тьюрінга, функціональна схема машини Тьюрінга, конфігурація машини Тьюрінга, універсальна машина Тьюрінга, машини з довільним доступом, команди машин з довільним доступом.*

2.2.2.1. Фінітний комбінаторний процес Поста

Основні положення та означення фінітного комбінаторного процесу Поста

Фінітний комбінаторний процес Поста (тут та надалі - машина Поста) складається із стрічки та пристрою керування (каретки).

Стрічка нескінчена та складається з комірок однакового розміру (див. 2.1.).

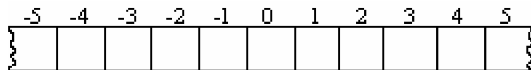


Рис. 2.1. Нескінчена стрічка

Послідовність, у якій розташовані комірки, відповідає послідовності, у якій розташовані натуральні числа.

В кожній комірці стрічки може бути записано або символ мітки \blacktriangledown або нічого (відповідна комірka називається або поміченою або пустою).

Інформація про заповнення стрічки називається її *станом*.

Каретка може переміщуватися вздовж стрічки праворуч та ліворуч. Коли ж вона нерухома, то стоїть проти однієї з комірок.

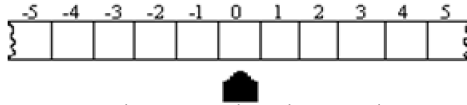


Рис. 2.2. Нескінчена стрічка і пристрій керування

Інформація про стан стрічки та місце розташування каретки називається *станом машини Поста*.

Робота машини Поста відбувається у дискретному часі. За одиницю часу каретка може виконати одну з операцій:

- зміститися праворуч
- зміститися ліворуч
- встановити мітку
- знищити мітку
- визначити наявність мітки.

Програма Поста

Кожна програма машини Поста складається з команд.

Кожна команда програми машини Поста складається з *номеру команди, операції та переходу*.

Наприклад,

- команда зміщення праворуч
 $i \rightarrow j$
- команда зміщення ліворуч
 $i \leftarrow j$
- команда встановлення мітки
 $i \blacktriangledown j$
- команда знищення мітки
 $i \varepsilon j$
- команда передачі керування
 $i ? \rightarrow j_1, j_2$
- команда зупинки
 $i \text{ стоп або } i !$ (знак оклику)

Відповідно, *програма машини Поста* – це скінчений перелік команд, що має наступні властивості:

- на n -му місці записується команда з номером N ;
- передача керування повинна відбуватися тільки до існуючого номеру команди.

Необхідні умови роботи машини Поста:

- визначеність стану машини Поста (місцезоташування каретки і міток);
- наявність програми машини Поста.

Зауваження щодо роботи машини Поста:

- виконання команди встановлення/знищення мітки не призводить до переміщення каретки і можливе тільки за умови порожньої / відміченої комірки;
- виконання команди передачі керування з верхнім та нижнім індексом не змінює стан машини Поста. Верхній перехід відбувається у випадку, коли комірка, яку визначає каретка, пуста, і навпаки.

Наприклад,

Машина Поста
 $i \rightarrow j1, j2$

Мова програмування
Pascal
If “ ” then $j1$ else $j2$

Результат виконання програми машини Поста:

- в ході виконання програми машина Поста зустрічається із командою зупинки, що призводить до результативної зупинки.
- в ході виконання програми машина Поста зустрічається із некоректною командою, що призводить до безрезультативної зупинки.¹
- в ході виконання програми машина Поста не зустрічається ні з однією з вищевказаних команд, що призводить до “зациклювання”.

Зауваження: різні програми, що опрацьовують один і той же стан, можуть призводити до всіх трьох результатів, і навпаки, одна і та ж програма може давати різні результати для різних початкових станів.

2.2.2.2. Абстрактна обчислювальна машина Тьюрінга

28 травня 1936 року у журналі “Праці Лондонського математичного товариства” з’явилася стаття А.М.Тьюрінга «Про обчислювальні числа», в якій автор уточнював поняття алгоритму через абстрактний пристрій, що надалі отримав назву “*абстрактної машини Тьюрінга*”.

Формальний опис машини Тьюрінга

Машина Тьюрінга (МТ) складається з:

- нескінченної стрічки, що поділена на комірки. В кожній комірці може бути записаний один із символів кінцевого алфавіту $A = \{a_0, a_1, a_2, \dots, a_m\}$, що називається зовнішнім алфавітом (ЗА). Для кожної машини Тьюрінга можна задати власний ЗА.
- керуючого пристрою, що може знаходитися в одному із внутрішнього станів $Q = \{q_1, q_2, \dots, q_n\}$. Кількість елементів Q визначає об’єм “внутрішньої пам’яті” машини Тьюрінга. У множині Q вирізняють два спеціальні стани: початковий q_1 та кінцевий q_z (або ! – знак оклику), де z – не числовий індекс, а мнемонічна ознака кінця. Таким чином, машина Тьюрінга починає роботу в стані q_1 та, потрапивши в q_z зупиняється.
- каретки, що переміщуючись вздовж стрічки, може:
 - *записувати в комірку символ зовнішнього алфавіту;*
 - *зміщуватися на комірку праворуч/ліворуч чи залишатися на місці*

Функціонування машини Тьюрінга можна описати так: в залежності від внутрішнього стану машини Тьюрінга (q_i) та символу зовнішнього алфавіту на стрічці (a_j) відбувається запис нового символу зовнішнього алфавіту (a'_j), зміщення каретки (d) та перехід до нового внутрішнього стану (q'_i).

Таким чином, робота машини Тьюрінга визначається системою команд вигляду:

$$q_i a_j \rightarrow q'_i a'_j d \quad (1)$$

Функціональна схема машини Тьюрінга

Враховуючи те, що для кожної пари $q_i a_j$ необхідна команда вигляду (1), програму машини Тьюрінга зручно записувати у вигляді прямокутної таблиці, стовпчики якої відповідають знакам внутрішніх станів машини Тьюрінга, а рядки – знакам зовнішнього алфавіту (див. *Таблиця 2.1.*). На перетині стовпчиків та рядків записана трійка знаків. Таку таблицю називають *функціональною схемою* машини Тьюрінга.

Таблиця 2.1.

Функціональна схема машини Тьюрінга

A \ Q	q_1	q_2	...	q_n
a_0	$q/a/d$			
a_1				
...				
a_m				

Конфігурація машини Тьюрінга

Повним станом машини Тьюрінга або конфігурацією машини Тьюрінга, за якою можна однозначно визначити поведінку машини, називається сукупність внутрішнього стану, стану стрічки та положення каретки.

Конфігурацію машини Тьюрінга надалі будемо записувати у вигляді

$a_1 q_i a_2$ (2), де

- q_i - поточний внутрішній стан,
- a_1 - слово ліворуч від каретки,
- a_2 - слово, що утворене символом, який

визначається кареткою, та символами праворуч від нього.

Наприклад, конфігурація K з внутрішнім станом, словом $abcde$ на стрічці та кареткою під символом C , записується як **abq_icde**.

Стандартною початковою конфігурацією назвемо конфігурацію вигляду $q_1 a$, тобто конфігурація, що містить початковий стан, в якому каретка розглядає крайній лівий символ слова, записаного на стрічці.

Якщо машина Тьюрінга, почавши роботу з деяким словом, записаним на стрічці, прийде в заключний стан, то вона називається застосованою для цього слова. Результатом її роботи вважається слово, записане на стрічці в заключний

момент. Якщо ж машина в жодний момент не прийде в заключний стан, то вона називається незастосовною до слова, і результат її роботи не визначений.

Приклад програми машини Тьюрінга

Написати програму для машини Тьюрінга для збільшення числа n на 1. Число записане у вигляді послідовності одиниць. Каретка знаходиться десь зліва від числа (див. рис. 2.3.).

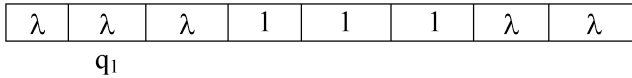


Рис. 2.3. Конфігурація машини Тьюрінга для задачі збільшення числа n на 1

Пояснення до рішення задачі: на першому такті визначається символ у комірці. Якщо це знак λ , то відбувається зміщення праворуч (символ в комірці та стан машини Тьюрінга залишаються незмінними). У випадку знаходження лівої позиції заданого числа (каретка зустріла в комірці перший символ 1), відбувається зміщення каретки ліворуч та перехід до стану q_2 (символ в комірці залишається без змін). Зустрівшись зі символом λ у стані q_2 , машина Тьюрінга змінює його на символ 1 та припиняє роботу (переходить до стану q_z).

Таблиця 2.2.

Функціональна схема машини Тьюрінга для задачі збільшення числа n на 1

A \ Q	q_1	q_2
1	$q_2 1L$	
λ	$q_1 \lambda R$	$q_z 1E$

Універсальна машина Тьюрінга

На сьогодні ми дотримувались точки зору, за якою різні алгоритми виконувалися в різних машинах Тьюрінга, що різнилися, звичайно, своїми функціональними схемами.

Але виникає природне запитання: *а чи можна побудувати універсальну машину Тьюрінга, яка б була здатна виконувати будь-який алгоритм, тобто була здатна виконувати роботу будь-якої машини Тьюрінга?*

Для відповіді на це запитання уявимо ситуацію, коли ми задали деякому виконавцю (людині) завдання, яке б полягало у демонстрації роботи машини Тьюрінга по опрацюванню деякої початкової інформації.

Звичайно, додатковою умовою буде повідомлення функціональної схеми даної машини Тьюрінга. Ми неодноразово, складаючи програми для машини Тьюрінга, самі виступали в якості виконавців, відтворюючи алгоритм машини.

Складемо словесний опис цього алгоритму:

Вказівка1: визначити на стрічці комірку, під якою підписана буква.

Вказівка2: знайти у функціональній схемі стовпчик, який позначений буквою, що написана під початковою коміркою.

Вказівка3: у стовпчику, що визначений у вказівці 2, знайти трійку символів, що розташована на перехресті із рядком, який позначений тим символом, що вписаний у комірку.

Вказівка4: змінити символ у комірці на перший символ трійки.

Вказівка5: якщо в трійці другим символом є символ !, то зупинити роботу. Якщо ж в трійці другим символом є символ E, то зміни символ, що підписаний під визначеною коміркою, третім символом трійки.

Вказівка7: якщо в трійці другим символом є символ L/R, то зітри символ під коміркою та ліворуч/праворуч запиши третій символ із трійки.

Вказівка 8: перейти до вказівки 1.

Але на місці виконавця може бути і сама машина Тьюрінга. А це означає, що замість словесного алгоритму за допомогою 7 вказівок, алгоритм відтворення може бути заданий у вигляді функціональної схеми.

Вихідними даними до нашого алгоритму буде функціональна схема та початкова конфігурація деякої машини Тьюрінга.

Але при цьому є два зауваження:

- безпосередня подача функціональної схеми деякої машини Тьюрінга та відповідної конфігурації на стрічку універсальної машини Тьюрінга в якості вихідної інформації неможлива! Адже функціональна схема задається двомірною таблицею, конфігурація у вигляді одномірної таблиці з підписаним нижче символом стану;
- універсальна машина Тьюрінга може оперувати тільки фіксованим скінченим зовнішнім алфавітом. Але при цьому вона повинна бути пристосована до прийому в якості функціональної схеми та конфігурації з будь-якою кількістю будь-яких символів.

Таким чином постає **завдання**: знайти спосіб “одномірного” подання інформації та кінцевого алгоритму.

Вирішення проблеми

Замість подання функціональної схеми у вигляді двомірної таблиці запишемо на стрічці послідовність п’ятірок, де перший символ – позначення стовпчика, другий – рядку, наступні три – символи команди.

Таким чином, замість функціональної схеми програми для збільшення натурального числа N , поданого послідовністю одиниць на 1, виникає рядок (див. рис. 2.4.).

q_1	1	q_2	1	L	q_1	λ	q_1	λ	R	q_2	λ	q_z	1	E
-------	---	-------	---	---	-------	-----------	-------	-----------	---	-------	-----------	-------	---	---

Рис. 2.4. Лінійний запис функціональної схеми машини Тьюрінга

Теоретичний курс

Для запису конфігурації машини Тьюрінга будемо дотримуватися такого правила: символ, що відповідає стану машини Тьюрінга, записуємо не під коміркою, а безпосередньо ліворуч від неї.

Наприклад, на рис. 2.5. зображена конфігурація машини Тьюрінга у звичному вигляді, на рис. 2.6. – її лінійне подання.

λ	λ	λ	1	1	1	λ	λ
-----------	-----------	-----------	---	---	---	-----------	-----------

q_1

Рис. 2.5 . Конфігурація машини Тьюрінга

Для однозначного визначення символів, що відповідають внутрішнім станам універсальної машини Тьюрінга ($q_1, q_2, q_3, \dots, q_m$), зміщенню каретки (L,E,R) та зовнішньому алфавіту ($a_1, a_2, a_3, a_4, \dots, a_k$), змінємо ці символи на послідовності з 1 та 0, які надалі називатимемо кодовою групою.

λ	q_1	λ	λ	1	1	1	λ	λ
-----------	-------	-----------	-----------	---	---	---	-----------	-----------

Рис. 2.6. Лінійний запис конфігурації машини Тьюрінга

За такого підходу важливо дотримуватися наступних правил:

- різні символи замінюються різними кодовими групами, але один і той самий символ завжди замінюється однією і тією ж кодовою групою.
- кодові групи повинні мати розподільні символи.

Ці правила можуть бути виконанні за умови спеціального кодування:

- в якості кодових груп беруться $3+k+m$ різновиди слів вигляду $10\dots 01$, де між символами 1 – послідовності 0.

Співвідношення кодових груп до вихідних букв здійснюється відповідно до таблиці 2.3.

Співвідношення кодових груп до символів машини Тьюрінга

Тип групи	Символ	Кодова група	Примітка	
символи зміщення каретки	L E R	101 1001 10001		
символи зовнішнього алфавіту	a_0 a_1 ... a_k	100001 10000001 ... 10.....01	4 нулі 6 нулів 2(k+1) нулів	парна кількість нулів, більша 2
символи внутрішнього стану	q_1 q_2 ... q_m	1000001 100000001 ... 10.....01	5 нулів 7 нулів 2(m+1)+1 нулів	непарна кількість нулів, більша 5

За такої схеми кодування матимемо:

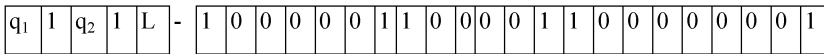


Рис. 2.7. Приклад кодування символів універсальної машини Тьюрінга

Такий рядок називатимемо *шифром* функціональної схеми машини Тьюрінга або шифром конфігурації машини Тьюрінга.

Звернемо увагу на те, як змінився алгоритм відтворення роботи універсальної машини Тьюрінга:

Вказівка1: визначити в шифрі конфігурації кодову групу, що розташована безпосередньо праворуч від кодової групи з непарною кількістю нулів.

Вказівка2-6: [див. 18, 87-88]

Вказівка7: якщо в трійці кодових груп з шифром схеми другою є група 1001, то шифрі конфігурації зміни кодову групу з непарною кількістю нулів третьою кодовою групою трійки.

Таким чином, алгоритм відтворення є описом деякої функціональної схеми машини Тьюрінга. Ця схема і є універсальною машиною Тьюрінга.

Практична значущість поняття універсальної машини Тьюрінга полягає у тому, що сучасні ЕОМ функціонують за таким же принципом як і вона, тобто: до запам'ятовуючого

пристрою поряд із вихідними даними певної задачі вводиться також і програма її рішення.

Порівняйте сказане із фрагментом тексту підручника А.Ф.Верлань, Н.В.Апатової. Інформатика 10-11 кл. ст.13.

«...щоб розв'язати задачу, до пам'яті комп'ютера потрібно ввести програму, яка визначає процес розв'язування, а також необхідні дані...» «...Дж. фон Нейман (1947 р.) запропонував кілька принципів роботи комп'ютерів, названих принципами програмного керування. Головний з них – це принцип єдності лінійної пам'яті. “Єдиної” означає, що і програма, і дані зберігаються в одній пам'яті. “Лінійної” означає, що всі комірки пам'яті пронумеровані від 0 до деякого числа N . Одна і та сама комірка пам'яті для однієї задачі може зберігати команду, а для іншої дані».

2.2.2.3. Машини з довільним доступом

Машина з довільним доступом (МДД) як алгоритмічна модель була запропонована в 70-х роках ХХ ст. з метою моделювання реальних обчислювальних машин та аналізу складності обчислень.

Конфігурація та команди машин з довільним доступом.

Машина з довільним доступом складається з нескінченної кількості регістрів $R_1, R_2, R_3, \dots, R_n$, в кожному з яких може бути записане натуральне число N . Надалі позначатиме через r_n число, що записане в регістрі R_n .

Станом машини з довільним доступом або конфігурацією називається послідовність $(r_1, r_2, r_3, \dots, r_n)$.

Функціонування машини полягає у зміні конфігурації шляхом виконання команд в порядку їх написання.

Машина з довільним доступом має наступні типи команд:

- *Команда обнулення ($Z(n)$).*

Дія команди $Z(n)$ полягає у заміні змісту регістру R_n на 0. Зміст інших регістрів не змінюється.

Умовне позначення команди: $r_n := 0$

- *Команда додавання одиниці ($S(n)$).*

Дія команди $S(n)$ полягає у збільшенні змісту регістру R_n на 1. Зміст інших регістрів не змінюється.

Умовне позначення команди: $r_n := r_n + 1$

- *Команда переадресації ($T(m, n)$).*

Дія команди $T(m,n)$ полягає у заміні змісту регістру R_n числом r_m , що зберігається в регістрі R_m . Зміст інших регістрів (включно і R_m) не змінюється.

Умовне позначення команди: $r_n := r_m$ або $r_m \rightarrow R_n$.

- Команда умовного переходу ($J(m,n,q)$).

Дія команди $J(m,n,q)$ полягає в наступному:

- порівнюється зміст регістрів R_n і R_m
- якщо $r_m = r_n$, то МДД переходить до виконання команди з номером q в переліку команд
- якщо $r_m < > r_n$, то МДД переходить до виконання наступної команди з переліку команд.

Кінцева впорядкована послідовність команд даних типів складає програму машини з довільним доступом.

Особливості функціонування машини з довільним доступом

Нехай зафіксована початкова конфігурація $K_0=(a_1, a_2, a_3, \dots, a_n)$ чисел і програма $P=I_1, I_2, I_3, \dots, I_s$. Тоді однозначно визначена послідовність конфігурацій $K_0, K_1, K_2, \dots, K_s$, де K_1 конфігурація, що отримана з K_0 застосуванням I_1 . Нехай на деякому кроці виконана команда I_t та отримана конфігурація K_t . Тоді, якщо I_t не є командою умовного переходу, то наступною командою K_{t+1} є конфігурація, що отримана з K_t застосуванням I_{t+1} . Якщо I_t – команда умовного переходу, тобто $I_t = J(m,n,q)$, то K_{t+1} отримана з K_t застосуванням команди I_q , якщо $r_m = r_n$ в конфігурації K_{t+1} і команда I_{t+1} , якщо $r_m < > r_n$.

Робота машини з довільним доступом припиняється, якщо:

- виконана остання команда, тобто $t=s$ і I_t не команда умовного переходу;
- якщо $I_t = J(m,n,q)$, $r_m=r_n$ в конфігурації K_t і $q>s$;
- якщо $I_t = J(m,n,q)$, $r_m < > r_n$ в конфігурації K_t і $t=s$.

Якщо обчислення припинилося, то послідовність $r_1, r_2, r_3, \dots, r_n$ змісту регістрів $R_1, R_2, R_3, \dots, R_n$ називається заключною конфігурацією.

Програми машин з довільним доступом

Обчислення функції $f(x,y)=x+y$.

Ця функція може бути обчислення наступної програмою при початковій конфігурації $(x,y,0,0)$.

$P=I_1 I_2 I_3 I_4$, де

$I_1=J(3,2,5)$

$I_2=S(1)$

$I_3=S(3)$

$I_4=J(1,1,1)$.

Дана програма додає 1 до x до тих пір, доки r_3 не стане дорівнювати y .

Обчислення функції $f(x,y)=x*y$.

Нехай H – програма, що обчислює функцію $x+y$. Тоді $f(x,y)=x*y$ обчислюється програмою:

$J(2,3,p)$

$S(3)$

$H[1,4 \rightarrow 5]$

$T(5,4)$

$J(1,1,1)$

$I_p T(5,1)$

2.2.3. Теорія нормальних алгоритмів Маркова

Література: [8,137-140;12,362-365;18,23-37]

Ключові поняття: *нормальні алгоритми Маркова, підстановка нормальних алгоритмів Маркова, схема орієнтованих підстановок нормальних алгоритмів Маркова, означення нормальних алгоритмів Маркова.*

Основні положення та означення теорії нормальних алгоритмів

Теорія нормальних алгоритмів була розроблена радянським математиком **А.А. Марковим** (1903 р. – 1979 р.) наприкінці 40-х – на початку 50-х років ХХ ст. і є ще однією алгоритмічною моделлю для уточнення поняття алгоритму.

Алгоритмом (запропоноване Марковим поняття для позначення алгоритму) в теорії нормальних алгоритмів є правила по перетворенню слів в довільному алфавіті. Нормальні алгоритми оперують словами скінченої довжини, перетворюючи їх одне в одне за допомогою підстановок, тобто змін однієї частини слова на іншу.

Цей процес нагадує процедуру виконання завдання типу: “Шляхом поступової зміни літер слово “слон” перетворити на “муху””.

Поняття алфавіту нормального алгоритму

Алфавітом (як і випадку будь-якої формальної системи) називається будь-яка скінчена множина деяких символів.

Будь-яка скінчена послідовність N літер деякого алфавіту – це *слова* завдовжки N у цьому алфавіту. Наприклад, в алфавіту A з трьох літер $\{a,b,c\}$ словами є послідовності $a,b,c,ab,bac,aaabacc$.

Порожнє слово, що не містить жодного символу, позначається як λ .

Якщо слово α є частиною слова β , то кажуть, що слово α входить у слово β .

Наприклад, у слові $d = abcbcababaa$ є 4 підслова a , 2 підслова bc , одне слово sba .

Поняття про підстановку

Підстановкою називається операція над словами, що задається за допомогою впорядкованої пари (α, β) та полягає у наступному: у довільному слові S знаходять перше входження слова α та, не змінюючи інших частин слова S , змінюємо в ньому це входження словом β .

Отримане слово називають **результатом підстановки** застосування підстановки (α, β) до слова S .

Якщо ж першого входження α в слово β немає (і відповідно немає ні одного входження α в S), то вважається, що підстановка (α, β) не застосована до слова S .

Для позначення підстановки (α, β) надалі будемо використовувати запис

$\alpha \rightarrow \beta$, який називається формулою підстановки (α, β) .

Деякі підстановки будемо вважати заключними ($\alpha \rightarrow * \beta$ або $\alpha \rightarrow \beta !$).

Схеми орієнтованих підстановок

Впорядкований скінчений перелік формул підстановок в алфавіті А

$$\left\{ \begin{array}{l} \alpha_1 \rightarrow \beta_1 // ! \\ \alpha_2 \rightarrow \beta_2 // ! \\ \dots\dots\dots \\ \alpha_n \rightarrow \beta_n // ! \end{array} \right.$$

називається *схемою орієнтованих підстановок* або *схемою нормальних алгорифмів*. Дана схема і визначає алгоритм перетворення слів, який називається нормальним алгорифмом Маркова.

Означення нормального алгоритму Маркова

Нехай задано алфавіт А і зафіксовано впорядковану (задану в певному порядку) систему орієнтованих підстановок Р. Виходячи з довільного слова α в алфавіті А розглядаються підстановки в тому порядку, в якому їх задано. Перша підстанова, що зустрілася, з лівою частиною, яка є підсловом α , використовується для перетворення α , в яке замість першого входження лівої частини підстановки підставляється її права частина, внаслідок чого утворюється слово α_1 . Далі, виходячи з слова α_1 , процес повторюється, поки він не зупиниться.

Ознаками зупинки процесу перетворення слова α є два випадки:

- коли утворюється таке слово α_n , що жодне з лівих частин допустимих підстановок не є його словами;
- коли при утворенні слова α_n використано останню підстановку (підстановку із знаком !).

Пара (А,Р) визначає нормальний алгорифм Маркова.

Приклади нормальних алгоритмів Маркова

Приклад 1. Нехай задано алфавіт $A=\{a,b,v,g,\dots\}$ і систему орієнтованих підстановок P

$y \rightarrow u$	$p \rightarrow r$	застосуємо алгоритм до слова “слон” та прослідкуємо перетворення:
$l \rightarrow u$	$t \rightarrow r!$	слон \rightarrow суон \rightarrow муон \rightarrow мухн \rightarrow муха
$c \rightarrow m$	$o \rightarrow x$	застосуємо алгоритм до слова “ветер” та прослідкуємо перетворення:
$v \rightarrow b$	$n \rightarrow a$	ветер \rightarrow бетер \rightarrow бетет \rightarrow берет

Приклад 2. Нехай задано алфавіт $A=\{1,+ \}$ і систему орієнтованих підстановок $P=\{+ \rightarrow \lambda, 1 \rightarrow 1 \}$.

Слово $1+11+1111+1$ алгоритм перетворює наступним чином:

$1+11+1111+1$
 $111+1111+1$
 $1111111+1$
 11111111

Еквівалентним є алгоритм із $P=\{1+ \rightarrow +1, +1 \rightarrow 1, 1 \rightarrow 1 \}$.

Приклад 3. Нехай задано алфавіт $A=\{1,0 \}$ і систему орієнтованих підстановок $P=\{1 \rightarrow \lambda, 0 \rightarrow 0 \}$.

Слово 0101 алгоритм перетворює в слово 00 .

Приклад 4. Нехай задано алфавіт $A=\{0,1,2,3,4,5,6,7,8,9 \}$ і систему орієнтованих підстановок P

$0b \rightarrow 1!$	$a0 \rightarrow 0a$	$0a \rightarrow 0b$
$1b \rightarrow 2!$	$a1 \rightarrow 1a$	$1a \rightarrow 1b$
$2b \rightarrow 3!$	$a2 \rightarrow 2a$	$2a \rightarrow 2b$
$3b \rightarrow 4!$	$a3 \rightarrow 3a$	$3a \rightarrow 3b$
$4b \rightarrow 5!$	$a4 \rightarrow 4a$	$4a \rightarrow 4b$
$5b \rightarrow 6!$	$a5 \rightarrow 5a$	$5a \rightarrow 5b$
$6b \rightarrow 7!$	$a6 \rightarrow 6a$	$6a \rightarrow 6b$
$7b \rightarrow 8!$	$a7 \rightarrow 7a$	$7a \rightarrow 7b$
$8b \rightarrow 9!$	$a8 \rightarrow 8a$	$8a \rightarrow 8b$
$9b \rightarrow b0$	$a9 \rightarrow 9a$	$9a \rightarrow 9b$
$b \rightarrow 1!$		$\lambda \rightarrow a$

Застосуємо алгоритм до слова “499” та прослідкуємо перетворення:

499 → a499 → 4a99 → 49a9 → 499a → 499b → 49b0 → 4b00 → 500

Як видно із прикладу, даний алгоритм виконує збільшення числа на 1.

2.2.4. Еквівалентність алгоритмічних моделей

Різноманітні підходи до уточнення (формалізації) поняття алгоритму привели до чітких алгоритмічних моделей.

В 1936 році Алонзо Черч сформулював свою тезу. Незабаром, після формулювання тези Черча, Алан Тьюрінг показав, що клас обчислювальних за Тьюрінгом функцій співпадає з класом частково-рекурсивних функцій та запропонував свій тезис, що формулюється так: “Класи обчислювальних та обчислювальних за Тьюрінгом функцій збігаються”. В свою чергу Марков сформулював свій тезис: “Будь-яка обчислювальна в інтуїтивному сенсі функція обчислювальна за Марковим”.

Загальні риси всіх алгоритмічних моделей.

- обчислювальна функція задається скінченим переліком елементарних інструкцій – програмою (програми машини Тьюрінга, схеми орієнтовних підстановок, схема примітивної рекурсії тощо);
- обчислення значень функції за вказаною програмою проводиться за чітко зафіксованому кінцевому переліку простих правил (опис підстановок нормальних алгоритмів чи кроків машини Тьюрінга);
- визначається єдиний спосіб подання вхідної та вихідної інформації (значення аргументів та функцій).

Всі вказані спільні риси обчислювальних функцій є доказом справедливості тези Черча в широкому сенсі: всі алгоритмічні моделі, що використовуються для точного визначення інтуїтивного поняття алгоритму – еквівалентні між собою.

Подамо більш чітке означення еквівалентності через визначення спільних ознак у різних алгоритмічних моделей.

Зв'язок частково-рекурсивних функцій з машиною Тьюрінга.

Для доведення еквівалентності частково-рекурсивних функцій з машиною Тьюрінга достатньо показати, що базисні функції та оператори суперпозиції S_m^n , примітивної рекурсії R^n та мінімізації μ_y можуть бути реалізовані на машині Тьюрінга .

Таблиця 2.4.

Функціональна схема функції наступності

A \ Q	q ₁
1	q ₁ 1R
λ	q ₂ 1E

Нуль-функція.

Побудуємо програму для машини Тьюрінга із зовнішнім алфавітом $A=\{\lambda,1\}$ та внутрішнім станом $\{q_1\}$ і програмою, що записана у табл. 2.5.

Таблиця 2.6.

Функціональна схема функції проекції

A \ Q	q ₁
1	q ₁ λR
λ	q ₂ λE

Функція наступності.

Побудуємо програму для машини Тьюрінга із зовнішнім алфавітом $A=\{\lambda,1\}$ та внутрішнім станом $\{q_1\}$ і програмою, що записана у табл. 2.4.

Таблиця 2.5.

Функціональна схема нуль-функції

A \ Q	q ₁
1	q ₁ λR
λ	q ₂ λE

Функція проекції.

Побудуємо програму для машини Тьюрінга із зовнішнім алфавітом $A=\{\lambda,1\}$ та внутрішнім станом $\{q_1\}$ і програмою, що записана у табл. 2.6.

Зв'язок частково-рекурсивних функцій з машинами з довільним доступом.

Оператор суперпозиції.

Нехай функція $g(y_1, y_2, \dots, y_n)$, $f_1(x_1, x_2, x_3, \dots, x_m)$, $f_2(x_1, x_2, x_3, \dots, x_m)$, ..., $f_n(x_1, x_2, x_3, \dots, x_m)$ обчислювальні на МДД.

Тоді обчислювальна і функція $h(x_1, x_2, x_3, \dots, x_m) = S(g, f_1, \dots, f_n)$.

Доказ: нехай G, F_1, \dots, F_n – програми стандартного вигляду для обчислення функцій $g(y_1, y_2, \dots, y_n)$, $f_1(x_1, x_2, x_3, \dots, x_m)$, ..., $f_n(x_1, x_2, x_3, \dots, x_m)$ відповідно. Необхідно написати програму H для обчислення h .

Покладемо $t = \max(n, m, r(G), r(F_1), \dots, r(F_n))$.

Запам'ятаємо $x_1, x_2, x_3, \dots, x_m$ в регістрах $R_{t+1}, R_{t+2}, \dots, R_{t+m}$, а $F_1(x_1, x_2, x_3, \dots, x_m), \dots, F_n(x_1, x_2, x_3, \dots, x_m)$ в регістрах $R_{t+m+1}, \dots, R_{t+m+n}$.

Вказані регістри не використовуються для обчислення за програмами G, F_1, \dots, F_n .

Програма H матиме вигляд:

$T(1, t+1)$

$T(2, t+2)$

...

$T(m, t+m)$

$F_1[t+1, \dots, t+m \rightarrow t+m+1]$

...

$F_n[t+1, \dots, t+m \rightarrow t+m+n]$

$G[t+m+1, \dots, t+m+n \rightarrow 1]$

Оператор примітивної рекурсії.

Нехай функції $g(x_1, x_2, x_3, \dots, x_n)$ та $h(x_1, x_2, x_3, \dots, x_n, y, z)$ – обчислювальні на МДД. Тоді функція $f = R(g, h)$ – обчислювальна.

Доказ: нехай G та H – програми стандартного вигляду для обчислення функцій g та h відповідно. Необхідно написати програму F для обчислення функції $f = R(g, h)$.

За початковою конфігурацією $(x_1, x_2, x_3, \dots, x_n, 0)$ за програмою G обчислюється $f(x_1, x_2, x_3, \dots, x_n, 0) = g(x_1, x_2, x_3, \dots, x_n)$. Тепер, якщо $y < 0$, то застосовуємо (багаторазово) програму H для обчислення функцій $f(x_1, x_2, x_3, \dots, x_n, 1)$, $f(x_1, x_2, x_3, \dots, x_n, 2)$, ..., $f(x_1, x_2, x_3, \dots, x_n, y)$.

Нехай $m = \max(n+2, r(G), r(H))$.

Запам'ятемо $x_1, x_2, x_3, \dots, x_n, y$ в регістрах $R_{m+1}, R_{m+2}, \dots, R_{m+n}, R_{m+n+1}$.

Номер циклу k , де $k=0, 1, 2, 3, \dots$ у заносимо R_{m+n+2} .

Тимчасове значення $f(x_1, x_2, x_3, \dots, x_n, k)$ розташуємо в регістрі R_{m+n+3} .

Програма F матиме вигляд:

$T(1, m+1)$

$T(2, m+2)$

...

$T(n+1, m+n+1)$

$T(n+2, m+n+2)$

...

$G[1, 2, 3, \dots, n \rightarrow m+n+3]$

$I_q: J(t+2, t+1, p)$

$H[m+1, m+2, \dots, m+n, t+2, t+3 \rightarrow t+3]$

$S(t+2)$

$J(1, 1, q)$

$I_p: T(t+3, 1)$

2.3. АЛГОРИТМІЧНО НЕРОЗВ'ЯЗУВАНІ ПРОБЛЕМИ

Література: [14, 208-209; 14, 210-217; 16, 62-70; 18, 89-94]

Ключові поняття: *алгоритмічно нерозв'язувані проблеми, теорема Райса, проблема самозастосованості, проблема зупинки*.

Алгоритмічно нерозв'язувані проблеми – не невдача, а науковий факт. Знання можливої алгоритмічної нерозв'язуваності має бути таким самим елементом наукової культури, як для фізика знання про неможливість створення вічного двигуна.

Якщо ж важливо мати справу з розв'язуваною задачею, то потрібно чітко уявляти дві обставини:

- відсутність загального алгоритму, який вирішує певну проблему не означає, що в кожному конкретному випадку не можна досягти успіху;

- поява нерозв'язаності – це, зазвичай, результат надмірної загальності задачі.

Теорема М.Райса.

Теорема Райса є однією з найбільш загальних теорем теорії алгоритмів, що пояснює природу багатьох проблем в практиці програмування.

Для пояснення теореми дамо деякі визначення. Нехай існує деяка множина A натуральних чисел.

Характерною функцією множини A будемо називати функцію

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

Множина A називається рекурсивною, якщо її характерна функція рекурсивна.

Наведемо змістовне формулювання теореми Райса.

Нехай Q – деяка властивість одномісних частково-рекурсивних функцій. Властивість Q називається нетривіальною, якщо є функції, яким характерна дана властивість Q , і яким вона не властива.

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

У відповідності до тез Черча і Тьюрінга задача є алгоритмічно розв'язуваною тоді і тільки тоді, коли існує деяка машина Тьюрінга M^0 , що вирішує дану задачу. Нехай необхідно визначити, чи характерна функції $\varphi_M(x)$, що реалізується машиною M , властивістю Q . На стрічці машини M^0 повинна бути записана інформація про програму машини M . Цю інформацію задамо у вигляді шифру $\Pi(M)$. Результатом роботи машини M^0 повинна бути відповідь “так” чи “ні”, в залежності від того, характерна функції $\varphi_M(x)$ властивість Q чи ні. Першому випадку домовимося співставляти символ 1, другому – 0. Таким чином, машина Тьюрінга M^0 розпізнає властивість Q одномісних частково-рекурсивних функцій, якщо конфігурацію

$q_1\text{Ш}(M)$ вона перетворює в q_11 , якщо функції $\varphi_M(x)$ характерна властивість Q , та в q_10 в іншому випадку.

Із сказаного робимо наступний висновок: якою б не було нетривіальна властивість Q одномісних частково-рекурсивних функцій, задача розпізнання цієї властивості алгоритмічно нерозв'язувана, тобто не існує машини M^0 , яка вирішує дану задачу (теорема Райса). Для доказу даного висновку покажемо, що з існування машини M^0 , яка розпізнає властивість Q , витікає рекурсивність множини номерів F_q функцій, яким властивість Q також характерна.

За наявності машини M^0 обчислення функції $\varphi_{N_{F_q}}(n)$ може бути здійснено наступним чином. Спочатку запис числа n переводиться в шифр $\text{Ш}(M_n)$ машини з номером n , потім застосовується машина M^0 , яка видає 1 , якщо функція $\varphi_{N_{F_q}}(n) = \varphi_M(x)$ має властивість Q , і 0 – в іншому випадку. В результаті значення $\varphi_{N_{F_q}}(n)$ обчислюється в алфавіті $\{0,1\}$.

Потім 0 перетворюється в $|$, в 1 – в $\|$.

З обчислювальності функції $\varphi_{N_{F_q}}(n)$ витікає її рекурсивність. Але в силу нетривіальності властивості Q множина F_q не пуста і відмінна від сукупності всіх одномісних частково-рекурсивних функцій, ось чому у відповідності до теореми Райса функція $\varphi_{N_{F_q}}(n)$ не може бути рекурсивна.

Отримане протиріччя доводить теорему.

Аналогічні результати можуть бути встановлені і для інших програм, за допомогою яких можна подати частково-рекурсивні функції, зокрема і для програм алгоритмічних мов програмування. Звідси витікає нерозв'язуваність багатьох програм, що пов'язані з програмуванням. Наприклад, якщо є деяка програма, то за її допомогою неможливо визначити функцію, яку дана програма реалізує. За двома програмами неможливо встановити, чи реалізують вони одну і ту ж функцію, а це призводить до нерозв'язуваності багатьох задач, що пов'язані з еквівалентними перетвореннями і мінімізацією програм.

Приклади алгоритмічно нерозв'язуваних проблем

Проблема самозастосованості.

Проблема самозастосованості одна з найактуальніших алгоритмічно нерозв'язуваних проблем.

Зміст проблеми в наступному. Будемо розглядати машини Тьюрінга, у зовнішньому алфавіті яких присутні, поряд з іншими, символи 1 та 0. Нехай на стрічку машини M записаний її шифр $\Pi(M)$ і машина запущена в початковому стані q_1 . Якщо після деякого скінченого числа кроків машина M прийде в заключний стан, то таку машину будемо називати самозастосованою, в іншому випадку – несамозастосованою. Проблема самозастосованості полягає в тому, щоб на основі машини M визначити, чи є вона самозастосованою. Машина Тьюрінга M^0 вирішує проблему самозастосованості, якщо для будь-якої машини M конфігурацію $q_1\Pi(M)$ вона перетворює в q_11 , якщо M самозастосована, і в q_10 – якщо несамозастосована.

Звідси, проблема самозастосованості алгоритмічно нерозв'язувана, тобто не існує машини Тьюрінга, щоб вирішувала дану задачу.

Для доведення уявимо протилежне – машина M , що вирішує проблему самозастосованості існує. На її основі побудуємо нову машину M' . Для цього стан q_0 зробимо не заключними, та введемо новий заключний стан q_0' та додамо до програми M^0 дві нові команди:

$$q_01 \rightarrow q_01E$$

$$q_00 \rightarrow q_0'0E$$

Машина M' застосована до шифрів несамозастосованих машин і не застосована до шифрів самозастосованих машин. Дійсно, якщо деяка машина M несамозастосована, то на початку M' , працюючи так як M^0 , перейде в конфігурацію q_00 , а потім зупиниться у відповідності до команди $q_00 \rightarrow q_0'0E$. Якщо ж M само застосована, то M' перейде в конфігурацію q_01 і ця конфігурація буде повторюватися нескінченно у відповідності до команди $q_01 \rightarrow q_01E$.

Сама машина M' є або само застосованою або несамозастосованою. У першому випадку вона застосована до власного шифру, тобто до шифру само застосованих машин, що неможливо з побудови M' . В іншому випадку вона не

застосована до власного шифру, тобто до шифру несамозастосованих машин, що також неможливо. Зазначене протиріччя виникло із припущення існування машини M^0 , що вирішує проблему самозастосованості.

Проблема зупинки

Серед загальних вимог до алгоритмів відзначалася вимога результативності.

Найрадикальнішим формулюванням тут була б вимога, щоб за будь-яким алгоритмом A і даними α можна було б визначити, чи призведе робота A з початковими аргументами α до результату чи ні? Іншими словами, треба побудувати такий алгоритм B , щоб $B(A,\alpha)=I$, якщо $A(\alpha)$ дає результат, та $B(A,\alpha)=X$, в іншому випадку.

З урахуванням тези Тьюрінга цю задачу можна сформулювати як задачу про побудову машини Тьюрінга: побудувати машину T_0 таку, що для будь-якої машини Тьюрінга T і будь-яких початкових даних для машини $T_0(\xi_T, \alpha)=I$, якщо машина $T(\alpha)$ зупиняється, та $T_0(\xi_T, \alpha)=X$, якщо вона не зупиняється.

Ця задача називається проблемою зупинки, а її формулювання нагадує задачу побудови універсальної машини Тьюрінга.

2.4. СКЛАДНІСТЬ АЛГОРИТМІВ

2.4.1. Поняття про складність алгоритмів

Література: [3, 11-14; 3, 47-54; 3, 93-122; 4, 22; 4, 222-264; 5, 74-108; 11, 85-421; 12, 374-385; 13, 11-23; 13, 148-166]

Ключові поняття: *складність алгоритмів, часова і емкісна складності алгоритмів, псевдокод, методика визначення часової складності алгоритмів, алгоритми впорядкування, впорядкування обміном, впорядкування вибором, впорядкування вставками, впорядкування підрахунком.*

Поняття алгоритмічно нерозв'язуваних проблем звичайно важливе і практично значуще. Розв'язування багатьох задач, як це не парадоксально, пов'язане саме з алгоритмічною нерозв'язуваністю.

Але з розвитком обчислювальної техніки все більше уваги стали приділяти не просто наявності алгоритму рішення деякого класу задач, а й ефективності цих алгоритмів.

Використовуючи алгоритмічні моделі (фінітний комбінаторний процес машини Поста, абстрактну машину Тюрінга чи машини з довільним доступом) ми не замислювалися над обмеженням ресурсів (стрічка була нескінчена, а час необмежений). Але в реальних ЕОМ і пам'ять і час обмежені. Ось чому замало знати про існування того чи іншого алгоритму – необхідно мати уявлення про необхідні ресурси, а саме:

- чи може певна програма (алгоритм) розміститися в пам'яті ЕОМ;
- чи дасть вона результат за належний час.

Дослідженням цих питань і займається розділ теорії алгоритмів – аналіз складності алгоритмів.

Складність алгоритмів – кількісна характеристика, яка визначає час, що необхідний для виконання алгоритму (часова складність), і об'єм пам'яті, необхідний для його розміщення (ємкісна складність).

Складність розглядається, за звичай, для машинних алгоритмічних моделей (ЕОМ), оскільки в них час і пам'ять присутні у явному вигляді.

Часова характеристика (фізичний час виконання) складності алгоритму – це величина $\tau \cdot t$, де t – кількість дій алгоритму (елементарних команд), а τ – середній час виконання однієї операції (команди).

Кількість команд t визначається описом алгоритму в певній алгоритмічній моделі і не залежить від фізичної реалізації цієї моделі.

Середній час τ – величина фізична і залежить від швидкості обробки сигналів в елементах і вузлах ЕОМ. Ось

чому об'єктивною математичною характеристикою складності алгоритму в певній моделі є кількість команд t .

Ємкісна характеристика складності алгоритмів визначається кількістю комірок пам'яті, що використовуються в процесі його обчислення. Ця величина не може перевищувати кількість дій t , що перемножена на певну константу (кількість комірок, які використовуються при виконанні однієї команди). В свою чергу, кількість дій t може перевищувати об'єм пам'яті (за рахунок використання повторень в одних і тих же комірках). До того ж проблема пам'яті технічно долається легше, ніж проблема швидкодії, яка має фізичне обмеження – швидкість розповсюдження фізичних сигналів (300 км/с). Ось чому часова складність вважається більш суттєвою характеристикою алгоритму, і в подальшому увагу ми зосередимо саме на ній.

Слід зазначити, що часова складність алгоритму не є постійною величиною і залежить від розмірності задачі (об'єм пам'яті для зберігання даних) – кількість комірок для різних даних. Отже, складність алгоритму – функція, значення якої залежить від розмірності n даних задачі.

Зазвичай говорять, що *час виконання алгоритму* або його *часова складність* має порядок $T(n)$ від вихідних даних розмірністю n . Наприклад, деякий алгоритм має часову складність $T(n)=cn^2$, де c – деяка константа. Одиниця виміру $T(n)$ точно не визначена, проте ми будемо розуміти її як кількість кроків, що виконані на ідеалізованому комп'ютері.

У більшості випадків, при визначенні часової складності алгоритму $T(n)$, розуміють максимальний час виконання алгоритму по всім вхідним даним, так звану часову складність алгоритму $T_{max}(n)$ у *найгіршому випадку*. Також розглядають і $T_{avg}(n)$ як середній (в статистичному сенсі) час виконання алгоритму на всіх вихідних даних розмірністю n . Хоча $T_{avg}(n)$ є досить об'єктивною мірою визначення часової складності, проте часто неможливо стверджувати рівнозначність всіх вхідних даних. Таким чином, на практиці середній час виконання алгоритму знайти складніше, ніж час у найгіршому випадку. Ось чому, зазвичай, використовують час у найгіршому випадку як міру часової складності алгоритму $T(n)$. Обчислення часової складності алгоритму $T_{min}(n)$ у *найсприятливішому випадку* хоча

і буде предметом нашої уваги, проте ми повинні розуміти, що значення цієї функції не повинно суттєво впливати на вибір того чи іншого алгоритму. Це пояснюється декількома причинами:

- по-перше, знаючи час роботи в *найгіршому випадку*, можна гарантувати, що виконання алгоритму закінчиться за якийсь час, навіть не знаючи, якого вигляду буде вихідна послідовність;

- по-друге, на практиці «погані» входи (для яких час роботи близький до максимуму) можуть часто потраплятися. Наприклад, для бази даних поганим запитом може бути пошук відсутнього елемента (досить поширена ситуація).

Виконуючи аналіз алгоритмів використовують різні способи їх подання: словесний опис, блок-схеми чи запис однією з мов програмування (машинно-орієнтованої чи високого рівня). Кожен із вказаних способів має свої переваги і оптимальнішим було б їх комплексне використання. Проте, ми скористаємося підходом, за яким спочатку пояснимо ідею того чи іншого методу, що необхідно реалізувати алгоритмом, а потім запишемо алгоритм *псевдокодом*.

Псевдокод – штучна неформальна мова, яка використовується для подання алгоритмів. Псевдокод - зручна і досить проста мова, що нагадує повсякденну мову і не є справжньою мовою програмування. Ретельно підготовлена на псевдокодї реалізація алгоритму може бути легко перетворена на програму на тій чи іншій мові програмування – для цього досить змінити оператори псевдокоду на їх еквіваленти мови програмування.

Слід зазначити, що ідея використання спрощеної моделі мови програмування високого рівня з'явилася в роботі [3, с.47-54]. Так, А. Ахо, Дж. Хопкрофт, Дж. Ульман для більш наочного подання алгоритмів вводять так званій *Спрощений Алгол*. У більш пізніх роботах автори розуміють під псевдокодом композицію мови *Pascal* і менш формальних та узагальнених операторів „людською” мовою [4, с.22]. Псевдокод використовує традиційні конструкції математики і мов процедурного програмування, зокрема такі як вирази, умови, оператори і процедури. Специфічним для цієї мови є те, що вона дозволяє використовувати будь-який тип математичних записів.

Псевдокод не має фіксованого набору типів даних. Змінними можуть бути цілі числа, слова і масиви.

Програма на *псевдокоді* – це один із наведених нижче операторів:

1. *змінна* ← вираз
2. *if* умова *then* оператор1 *else* оператор2
3. *while* умова *do* оператор
4. *repeat* оператор *until* умова
5. *for* *змінна* ← початкове значення *step* розмір кроку *until* кінцеве значення *do* оператор
6. *begin*
 оператор
 оператор
 ...
 оператор
 оператор
end
7. підпрограми
procedure ім'я (перелік параметрів) :
 оператор
function ім'я (перелік параметрів) : тип результату
 оператор
8. *comment* коментар
 { коментар }

Дамо короткий опис основних операторів псевдокоду.

1. Оператор присвоювання.

Оператор присвоювання *змінна* ← *вираз* означає, що необхідно обчислити вираз праворуч від стрілки і його значення привласнити змінній, що розташована ліворуч від стрілки. Часова складність оператора присвоювання визначається часом, що затрачується на обчислення значення виразу і присвоювання цього значення змінній.

2. Оператор умови.

В операторі умови *if* умова *then* оператор1 *else* оператор2 умовою, що вказана за *if*, може бути будь-який вираз, що приймає значення *true* або *false*. Якщо ця умова має

значення *true*, то треба виконувати *оператор1*, що слідує за *then*. В іншому випадку треба виконувати *оператор2*, що стоїть за *else* (якщо *else* присутнє). Часова складність оператора умови дорівнює сумі складностей, необхідних для обчислення значення і перевірки його, і складності *оператора1*, що знаходиться відразу за *then*, або *оператора2*, що стоїть за *else*, у залежності від того, який з них виконується.

3. Оператори повторення.

Оператори

- *while* умова *do* оператор
- *repeat* оператор *until* умова
- *for* змінна ← початкове значення *step* розмір кроку *until* кінцеве значення *do* оператор

призначені для організації повторень.

У *операторі повторення з передумовою*

while умова *do* оператор

обчислюється значення умови, що йде після *while*. Якщо воно істинне (приймає значення *true*), то виконується *оператор*, що стоїть після *do*. Цей процес повторюється доти, поки умова не стане хибною. Якщо спочатку ця умова була істинною, то виконання *оператора* повинно зрештою привести цю умова до значення *false*, щоб закінчилося виконання *while*-оператора. Для обчислення часової складності *while*-оператора додаються складності всіх перевірок умови і усіх виконаних *операторів*.

Оператор повторення з післяумовою

repeat оператор *until* умова

тракується аналогічно, але тільки тепер оператор, що знаходиться після *repeat*, виконується перед перевіркою умови.

У *оператора повторення з параметром*

for змінна ← початкове значення *step* розмір кроку *until* кінцеве значення *do* оператор

„початкове значення”, „розмір кроку” і „кінцеве значення” є виразами. У випадку коли значення кроку позитивне, змінна приймає значення, що відповідає значенню виразу для початкового значення. Якщо воно більше кінцевого значення, то виконання *оператора* закінчується. В іншому випадку виконується *оператор*, що стоїть після *do*, значення змінної

збільшується на *розмір кроку* і порівнюється із *кінцевим значенням*. Процес повторюється доти, поки значення *змінної* менше, ніж *кінцеве значення*. Випадок, коли *розмір кроку* від'ємний, трактується аналогічно з тією лише різницею, що закінчення відбувається, коли значення *змінної* стає меншим за *кінцеве значення*. Часова складність оператора повторення з параметром аналогічна часовій складності *while*-оператора. Проте у такому випадку враховуються і часова складність обчислення значення *змінної* при кожній ітерації.

4. Підпрограми.

Фрагменти програмного коду, що часто використовуються, можна оформлювати в якості підпрограм. Підпрограми визначаються і надалі використовуються в алгоритмах.

Підпрограми використовуються одним з двох способів.

Перший спосіб полягає у використанні підпрограми за допомогою оператора виклику підпрограми-процедури

procedure ім'я (перелік параметрів):
оператор

Цей оператор є, по суті, іменем процедури, за яким вказується перелік параметрів. Завершується виконання підпрограми-процедури завершенням виконання останнього оператора підпрограми.

Другий спосіб полягає у тому, що використовується оператор виклику підпрограми-функції

function ім'я (перелік параметрів) : тип результату
оператор,

який дозволяє обчислити деяке значення і присвоїти його значення функції. Наприклад, вираз $c \leftarrow \text{MIN}(a, b)$ означає, що значенню *змінної c* присвоюється значення мінімальної серед *змінних a* і *b*.

На увагу заслуговує аналіз складності виклику **підпрограм**. Для алгоритмів, що містять кілька процедур (серед яких немає рекурсивних), можна підрахувати загальний час виконання алгоритму шляхом послідовного знаходження часу виконання процедур, починаючи з тієї, яка не має викликів інших процедур. Потім необхідно визначити час виконання

процедур, що викликають цю процедуру, використовуючи вже обчислений час виконання цієї процедури. Продовжуючи цей процес можна знайти час виконання всіх процедур i , нарешті, час виконання всього алгоритму.

Якщо ж в алгоритмі присутні рекурсивні підпрограми, то не можна упорядкувати всі підпрограми таким чином, щоб кожна з них викликала тільки підпрограми, час виконання яких підраховано на попередньому кроці. У цьому випадку з кожною рекурсивною підпрограмою пов'язують часову складність $T(n)$, де n визначає кількість аргументів підпрограми. Потім одержують рекурентне співвідношення для $T(n)$, тобто рівняння (або нерівність) для $T(n)$, де беруть участь значення $T(k)$ для різних значень k .

Техніка рішення рекурентних співвідношень залежить від виду цих співвідношень.

Пояснимо сказане на прикладі аналізу підпрограми обчислення факторіалу $n!$.

function *factorial* (n): цілочисельна змінна

begin

(1) **if** $n \leq 1$ **then**

(2) *factorial* $\leftarrow 1$

else

(3) *factorial* $\leftarrow n * \text{factorial}(n-1)$

end

Природною мірою кількості вхідних даних для функції *factorial* є значення n . Позначимо через $T(n)$ часову складність алгоритму. Часова складність рядків (1) і (2) має значення 1, а рядку (3) – $(1+1)+T(n-1)$. Таким чином, для деяких констант c і d маємо:

$$T(n) = \begin{cases} c + T(n-1), & \text{якщо } n > 1, \\ d, & \text{якщо } n \leq 1. \end{cases}$$

Припустивши, що $n > 2$ і підставивши у співвідношення (1) $n-1$ замість n , отримаємо $T(n) = 2c + T(n-2)$. Аналогічно, якщо $n > 3$, отримаємо $T(n) = 3c + T(n-3)$. Продовжуючи цей процес, в загальному випадку для деякого i , $n > 1$, маємо $T(n) = ic + T(n-i)$. Поклавши в останньому виразу $i = n-1$, отримуємо

$$T(n) = c(n-1) + T(1) = c(n-1) + d.$$

Таким чином, загальний метод рішення рекурентних співвідношень, полягає у послідовному розкритті виразу $T(k)$ у правій частині рівняння (шляхом підстановки у вихідне співвідношення k замість n) до тих пір, поки не вийде формула, в якій у правій частині відсутнє $T(n)$.

Оскільки для різних конкретних даних однакової розмірності n алгоритм може витрачати різну кількість команд, функція складності $T(n)$ визначається різними способами. Розглянемо деякі з них на прикладі *алгоритмів впорядкування*.

Під *впорядкуванням* розуміється процес перестановки об'єктів деякої множини у певному порядку [5, с. 74]. Методи впорядкування є блискучою ілюстрацією ідеї аналізу складності алгоритмів, тобто ідеї, що дозволяє оцінювати робочі характеристики алгоритмів, і відповідно, зважено підходити до вибору адекватного алгоритму рішення конкретної задачі.

Д. Кнут в третьому томі „*Сортировка и поиск*” своєї монографії „*Искусство программирования для ЭВМ*” наводить наступний факт: „Постачальники обчислювальних машин вважають, що в середньому більше 25% машинного часу систематично витрачається на впорядкування. В багатьох обчислювальних системах на неї (операцію впорядкування) припадає більше половини машинного часу. Виходячи з цієї статистики, можна стверджувати, що або (1) впорядкування має багато важливих галузей застосування, або (2) нею користуються без потреби, або (3) використовуються неефективні алгоритми впорядкування” [14, с.8]. Як зазначає Д. Кнут, в кожному із вказаних тверджень є доля істини. І якщо в першому і другому випадках вплинути на ситуацію складно (сфера застосування алгоритмів, що в якості проміжного етапу використовують принцип впорядкування, об'єктивно значна, а необхідність використання того чи іншого алгоритму суб'єктивна і залежить від „людського фактору”), то виправданим буде зосередження уваги на ефективності цих алгоритмів.

Існують декілька алгоритмів впорядкування (саме такий термін для позначення процедури ранжування використовує Д. Кнут, хоча і визнає, що термін „сортування” набув більшого розповсюдження серед програмістів), зокрема: впорядкування

обміном („бульбашкове” впорядкування), впорядкування вибором, впорядкування вставками, швидке впорядкування, шейкер-впорядкування, впорядкування Шелла, пірамідалне і порозрядне впорядкування.

Введемо основну термінологію і позначення, що використовуються в алгоритмах впорядкування.

Дана послідовність A , що складається із n елементів:

$a[1], a[2], \dots, a[n]$

Впорядкування передбачає розміщення цих елементів у порядку зростання (спадання) відповідно до лінійного відношення порядку, такому як „ \leq ” („ \geq ”) для чисел.

Звичайно, впорядкування може застосовуватися не лише до елементів з „чисельною” природою. Так, в узагальненому випадку операцію впорядкування можна сформулювати наступним чином: „Впорядкувати послідовність записів таким чином, щоб значення ключових полів цих записів склали зростаючу (спадаючу) послідовність”. Іншими словами, записи $a[1], a[2], \dots, a[n]$ зі значеннями ключів $k[1], k[2], \dots, k[n]$ необхідно розташувати у такому порядку, щоб $k[1] \leq k[2] \leq \dots \leq k[n]$.

Існує декілька основних алгоритмів впорядкування (Д.Кнут наводить 25 основних алгоритмів [11, с.85]) та велика кількість їх модифікацій. Така кількість алгоритмів впорядкування пояснюється перевагами і недоліками кожного з алгоритмів для деяких конфігурацій даних і апаратури. Корисно розглянути характеристики хоча б основних з них, щоб для конкретного випадку можна було зробити розумний вибір. Проте, алгоритми впорядкування будуть використані нами не лише для розгляду методики оцінки часової складності алгоритмів, а й для ознайомлення з основними методами розробки ефективних алгоритмів.

Для пояснення методики оцінки часової складності алгоритмів $T(n)$ скористаємося так званими „простими” алгоритмами. До цієї групи відносять алгоритми впорядкування обміном, упорядкування вибором та впорядкування вставками. Для демонстрації методів удосконалення „простих” алгоритмів з метою зниження їх часової складності можна скористатися алгоритмами швидкого впорядкування, шейкер-впорядкування та алгоритмом впорядкування Шелла.

Слід зазначити, що поділ алгоритмів впорядкування на алгоритми впорядкування обміном-вставками-вибором (тим більше на групи „простих” і „складних”) досить умовний. Адже всі алгоритми використовують операцію обміну елементів місцями, яка передбачає як вибір елементів, так і їх вставку. Проте, саме така класифікація є найбільш вживаною в літературі, що присвячена аналізу алгоритмів впорядкування, а отже саме її ми і будемо притримуватися.

Впорядкування обміном

Алгоритм впорядкування обміном базується на принципі порівняння пари сусідніх елементів до тих пір, доки не будуть впорядковані всі елементи. Щоб описати основну ідею цього методу, який іноді називають методом „бульбашки”, уявимо, що елементи зберігаються в послідовності (масиві), розташованому вертикально. Елементи, що мають малі значення, є більш „легкішими” і „спливають” нагору подібно бульбашкам. При першому проході уздовж масиву (перегляд починається знизу), береться перший елемент послідовності і його значення по черзі порівнюється зі значеннями наступних елементів. Якщо зустрічається елемент з більш „важким” значенням, то ці елементи міняються місцями. При зустрічі з елементом, що має більш „легше” значення, цей елемент стає „еталоном” для порівняння, і всі наступні елементи порівнюються з цим новим, більш „легшим” елементом. У результаті елемент з найменшим значенням опиниться вгорі послідовності. Під час другого проходу уздовж масиву знаходиться елемент із другим по величині значенням, що міститься під елементом, який було знайдено при першому перегляді послідовності. Процес повторюється до тих пір, доки не будуть впорядковані всі елементи послідовності. Відзначимо, що під час другого і наступних переглядів послідовності немає необхідності переглядати елементи, знайдені за попередні перегляди, адже вони мають значення менші за значення елементів, що залишилися. Іншими словами, під час i -го перегляду не перевіряються елементи, що знаходяться на позиціях вище i .

Проілюструємо роботу алгоритму впорядкування обміном наступним прикладом, що записаний на псевдокодi.

- (1) **for** $i \leftarrow n-1$ **step** -1 **until** 1 **do**
- (2) **for** $j \leftarrow 1$ **step** 1 **until** i **do**
- (3) **if** $a[j] > a[j+1]$ **then**
- (4) swap($a[j]$, $a[j+1]$)

Звернемо увагу на те, що підпрограма-процедура *swap* рядку (4) використовується в багатьох алгоритмах впорядкування для перестановки елементів місцями. Код цієї процедури наведено нижче:

procedure *swap* (x, y)

begin

 temp \leftarrow x

 x \leftarrow y

 y \leftarrow temp

end

Для визначення часової складності алгоритму виконаємо наступні дії. Біля кожного рядку алгоритму зазначимо його вартість (число операцій) і кількість разів, за яку виконується цей рядок. Зауважимо, що рядки усередині циклу виконуються на один раз менше, ніж перевірка, оскільки остання перевірка виводить з циклу. Для кожного j від 1 до i підрахуємо, скільки разів буде виконаний рядок (3), і позначимо це число через k_j . Аналогічну дію виконаємо і для рядку (4).

Таблиця 2.7.

Визначення часової складності алгоритму впорядкування обміном

№	Команда	Вартість	Кількість виконань
(1)	for $i \leftarrow n-1$ step -1 until 1 do	c_1	n
(2)	for $j \leftarrow 1$ step 1 until i do	c_2	n-1
(3)	if $a[j] > a[j+1]$ then	c_3	$\sum_{j=1}^i k_j$
(4)	swap ($a[j]$, $a[j+1]$)	c_4	$\sum_{j=1}^i t_j$

Рядок вартістю c , що повторений m разів, дає внесок cm в загальну кількість операцій. Склавши внески всіх рядків, одержимо вираз, що позначає часову складність алгоритму впорядкування обміном:

$$T(n) = c_1 n + c_2 (n-1) + c_3 \sum_{j=2}^i k_j + c_4 \sum_{j=2}^i t_j$$

Обчислимо суму кількості виконань для рядку (3).

$$\sum_{j=1}^i k_j = \frac{1+n-1}{2} (n-1) = \frac{n(n-1)}{2}$$

Таким чином, часова складність алгоритму $T(n)$ дорівнює:

$$\begin{aligned} T(n) &= c_1 n + c_2 (n-1) + c_3 \left(\frac{n(n-1)}{2} \right) + c_4 \sum_{j=2}^i t_j = \\ &= c_1 n + c_2 (n-1) + c_3 \left(\frac{n^2 - n}{2} \right) + c_4 \sum_{j=2}^i t_j = \\ &= c_1 n + c_2 (n-1) + c_3 \left(\frac{1}{2} (n^2 - n) \right) + c_4 \sum_{j=2}^i t_j = \\ &= c_1 n + c_2 n - c_2 + \frac{c_3 n^2}{2} - \frac{c_3 n}{2} + c_4 \sum_{j=2}^i t_j = \\ &= \left(\frac{c_3}{2} \right) n^2 + \left(c_1 + c_2 - \frac{c_3}{2} \right) n - c_2 + c_4 \sum_{j=2}^i t_j \end{aligned}$$

Як бачимо, функція $T(n)$ – квадратична, тобто має вигляд $T(n) = an^2 + bn + c$, де константи a , b і c визначаються значеннями c_1, \dots, c_4 .

Слід зауважити, що час роботи алгоритму залежить не тільки від n , але і від того, який саме масив розмірністю n поданий їй на вхід. Для алгоритму впорядкування обміном найбільш сприятливий випадок, коли послідовність уже впорядкована.

Процедура обміну *swap* у такому разі не буде виконана жодного разу, а часова складність алгоритму $T_{min}(n)$ дорівнюватиме:

$$T_{min}(n) = \left(\frac{c_3}{2}\right)n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2$$

Таким чином, у найбільш сприятливому випадку час $T_{min}(n)$, необхідний для обробки масиву розміру n , залишається бути квадратичною функцією від n .

Якщо ж масив розташований у зворотному (спадному) порядку, час роботи алгоритму буде максимальним: кожен елемент $a[j]$ прийдеться міняти місцями з елементом $a[j+1]$. При цьому $t_j = j$.

Обчислимо суму кількості виконань для рядку (4).

$$\begin{aligned} T_{max}(n) &= \left(\frac{c_3}{2}\right)n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2 + c_4 \sum_{j=2}^i t_j = \\ &= \left(\frac{c_3}{2}\right)n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2 + c_4 \left(\frac{n^2 - n}{2}\right) = \\ &= \left(\frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + c_2 - \frac{c_3}{2} - \frac{c_4}{2}\right)n - c_2 \end{aligned}$$

Як бачимо, у гіршому випадку час роботи алгоритму $T_{max}(n)$ також є квадратичною функцією від n .

Аналіз середнього значення часової складності алгоритму впорядкування обміном $T_{avg}(n)$ залежить від обраного розподілу ймовірностей, і на практиці реальний розподіл може відрізнятись від передбачуваного, який, зазвичай, вважають рівномірним. Іноді рівномірний розподіл моделюють, використовуючи генератори випадкових чисел. Проте, можна припустити, що в середньому обмін відбувається приблизно у $i/2$ випадках і його загальна кількість приблизно дорівнює значенню $(1+2+...+n)/2 \approx n^2/4$. У такому випадку середнє значення часової складності алгоритму впорядкування обміном $T_{avg}(n)$ можна показати формулою:

$$\begin{aligned} T_{\text{avg}}(n) &= \left(\frac{c_3}{2}\right)n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2 + c_4\left(\frac{n^2 - n}{4}\right) = \\ &= \left(\frac{c_3}{2} + \frac{c_4}{4}\right)n^2 + (c_1 + c_2 - \frac{c_3}{2} - \frac{c_4}{4})n - c_2 \end{aligned}$$

Звичайно, алгоритм впорядкування обміном можна легко модифікувати. Один із шляхів полягає у тому, що можна позбавитися великої кількості непотрібних порівнянь, запам'ятавши чи відбувався обмін на попередньому кроці. Якщо обміну не було, то послідовність вважається впорядкованою, і алгоритм закінчує роботу. Цей процес удосконалення алгоритму можна продовжити, якщо запам'ятовувати не тільки сам факт обміну, але і місце (індекс) останнього обміну. Адже зрозуміло, що всі пари елементів з індексом, меншим від j , вже упорядковані, і наступні перегляди можна закінчувати на цьому індексі.

Проте, всі запропоновані нами вдосконалення жодним чином не впливають на кількість обмінів – вони лише зменшують кількість надлишкових повторних перевірок. Нажаль, обмін двох елементів – більш ресурсоемна операція, ніж порівняння, тому всі наші вдосконалення дають лише незначний ефект.

Таким чином, аналіз алгоритму впорядкування обміном показав, що ніяких переваг, окрім легкозапам'ятовуючої назви, цей алгоритм не має. Проте, саме на прикладі цього алгоритму ми розглянули методику визначення часової складності алгоритмів, яку надалі застосуємо при аналізі інших алгоритмів.

Впорядкування вибором

Ідея методу впорядкування вибором полягає у тому, що на i -му кроці вибирається найменший елемент серед елементів послідовності $a[i]..a[n]$, який міняється місцями з елементом $a[i]$.

У залежності від результату пошуку елемент $a[i]$ або залишається в i -ій позиції або вони міняються місцями і процес повторюється.

Визначення часової складності алгоритму впорядкування вибором

№	Команда	Вартість	Кількість виконань
(1)	<i>for</i> $i \leftarrow 1$ <i>step</i> 1 <i>until</i> $n-1$ <i>do</i>	c_1	n
	<i>begin</i> {for}		
(2)	$min \leftarrow a[i]$	c_2	$n-1$
(3)	$index \leftarrow i$	c_3	$n-1$
(4)	<i>for</i> $j \leftarrow i+1$ <i>step</i> 1 <i>until</i> n <i>do</i>	c_4	$\sum_{j=i+1}^n t_j$
	<i>begin</i> {for}		
(5)	<i>if</i> $a[j] < min$ <i>then</i>	c_5	$\sum_{j=i+1}^n (t_j - 1)$
	<i>begin</i> {if}		
(6)	$min \leftarrow a[j]$	c_6	$\sum_{j=i+1}^n k_j$
(7)	$index \leftarrow j$	c_7	$\sum_{j=i+1}^n k_j$
	<i>end</i> {if}		
(8)	<i>swap</i> ($a[i]$, $a[index]$)	c_8	$n-1$
	<i>end</i> {for}		
	<i>end</i> {for}		

Склавши внески всіх рядків, одержимо вираз, що позначає часову складність алгоритму впорядкування вибором:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=i+1}^n t_j + c_5 \sum_{j=i+1}^n (t_j - 1) + c_6 \sum_{j=i+1}^n k_j + c_7 \sum_{j=i+1}^n k_j + c_8 (n-1)$$

Обчислимо суму кількості виконань оператора повторення рядка (4) і оператора умови, що знаходиться в рядку (5).

$$\sum_{j=i+1}^n t_j = \frac{2+n}{2}(n-1) = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=i+1}^n t_j - 1 = \frac{1+n-1}{2}(n-1) = \frac{n(n-1)}{2}$$

Таким чином, часова складність алгоритму $T(n)$ дорівнює:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n^2+n}{2} - 1\right) + c_5\left(\frac{n^2-n}{2}\right) + c_6 \sum_{j=i+1}^n k_j + c_7 \sum_{j=i+1}^n k_j + c_8(n-1)$$

Слід зауважити, що, як і у випадку впорядкування обміном, час роботи алгоритму впорядкування вибором залежить не тільки від n , але і від того, яка саме вихідна послідовність розмірністю n . Обчислення часових складностей $T_{max}(n)$ і $T_{avg}(n)$ алгоритму впорядкування вибором відбувається аналогічним до алгоритму впорядкування обміном чином.

У випадку, коли послідовність уже впорядкована оператори в рядках (6)-(7) виконуватися не будуть. Таким чином, часова складність алгоритму $T_{min}(n)$ дорівнюватиме:

$$T_{min}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n^2+n}{2} - 1\right) + c_5\left(\frac{n^2-n}{2}\right) + c_8(n-1)$$

Як бачимо, навіть у найбільш сприятливішому випадку час $T_{min}(n)$, необхідний для обробки послідовності, залишається бути квадратичною функцією від n .

Слід зазначити, що, незважаючи на відсутність значної різниці в значенні функції $T(n)$ алгоритму впорядкування

вибором у порівнянні з алгоритмом впорядкування обміном, впорядкування вибором є більш придатнішим. Пов'язано це, зокрема, з таким моментом. Кожен з алгоритмів використовує процедуру обміну $swap(x,y)$. В алгоритмі впорядкування обміном ця процедура виконується не менш, ніж $n(n-1)/2$ разів. Але, оскільки процедура $swap(x,y)$ виконується після оператора умови, то можна очікувати, що кількість обмінів буде дещо меншим за $n(n-1)/2$. Однак, в алгоритмі впорядкування вибором процедура $swap(x,y)$ знаходиться поза внутрішнім оператором повторення і тому виконується рівно $n-1$ раз для послідовності розмірністю n . А це значно менше за розглянуту в алгоритмі впорядкування обміном кількість.

Впорядкування вставками

Ідея методу полягає у тому, що розглядаються дії алгоритму на його i -му кроці. При цьому, послідовність розділена на дві частини: впорядковану $a[1]...a[i]$ і неупорядковану $a[i+1]...a[n]$. На кожному наступному, $(i+1)$ -му кроці алгоритму береться $a[i+1]$ елемент і вставляється на потрібне місце в упорядковану частину послідовності.

Пошук потрібного місця для чергового елемента вхідної послідовності здійснюється шляхом послідовних порівнянь з елементом, що знаходиться перед ним. У залежності від результату порівняння елемент або залишається на поточному місці (вставка закінчена), або вони міняються місцями і процес повторюється.

Таким чином, у процесі вставки елемент x якби „просівається” до початку масиву. При цьому зупинка відбувається у випадку коли:

- знайдено елемент, менший від x або
- досягнуто початок послідовності.

Біля кожного рядку процедури зазначимо його вартість (число операцій) і кількість разів, за яку виконується цей рядок. Для кожного i від 2 до n підрахуємо, скільки разів буде виконаний рядок (4), і позначимо це число через t_i .

Визначення часової складності алгоритму впорядкування вставками

№	Команда	Вартість	Кількість виконань
(1)	<i>for</i> $i \leftarrow 2$ <i>step</i> 1 <i>until</i> n <i>do</i>	c_1	n
	<i>begin</i> { <i>for</i> }		
(2)	$key \leftarrow A[i]$	c_2	$n-1$
(3)	$j \leftarrow i-1$	c_3	$n-1$
(4)	<i>while</i> ($j > 0$) <i>and</i> ($A[j] > key$) <i>do</i>	c_4	$\sum_{i=2}^n t_i$
	<i>begin</i> { <i>while</i> }		
(5)	$A[j+1] \leftarrow A[j]$	c_5	$\sum_{j=2}^n (t_j - 1)$
(6)	$j \leftarrow j-1$	c_6	$\sum_{i=2}^n (t_i - 1)$
	<i>end</i> { <i>while</i> }		
(7)	$A[j+1] \leftarrow key$	c_7	$n-1$
	<i>end</i> { <i>for</i> }		

Склавши внески всіх рядків, одержимо вираз, що позначає часову складність алгоритму впорядкування вставками:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{i=2}^n t_i + c_5 \sum_{i=2}^n (t_i - 1) + c_6 \sum_{i=2}^n (t_i - 1) + c_7 (n-1)$$

Як і у випадку аналізу алгоритму обміном, зауважити, що час роботи процедури залежить не тільки від n , але і від того, який саме масив розмірністю n поданий їй на вхід.

Якщо вихідна послідовність вже упорядкована у зворотному (спадному) порядку, час часова складність роботи алгоритму $T_{max}(n)$ буде максимальною: кожен елемент $A[j]$ прийдеться порівнювати з усіма елементами $A[j-1].. A[1]$. При цьому $t_i = i$.

Обчислимо суми

$$\sum_{i=2}^n i = \frac{2+n}{2}(n-1) = \frac{n(n+1)}{2} - 1$$

$$\sum_{i=2}^n (i-1) = \sum_{i=2}^{n-1} i = \frac{1+n-1}{2}(n-1) = \frac{n(n-1)}{2}$$

Як бачимо, у найгіршому випадку часова складність роботи алгоритму $T_{max}(n)$ дорівнює

$$\begin{aligned} T_{max}(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + \\ &+ c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) = \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \end{aligned}$$

$$+ (c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Функція $T_{max}(n)$ - квадратична, тобто має вигляд багаточлена (полінома) $T_{max}(n) = an^2 + bn + c$, де константи a , b і c визначаються значеннями c_1, \dots, c_7 . Середнє значення часової складності роботи алгоритму $T_{avg}(n)$ буде доволі близьким до $T_{max}(n)$, адже в середньому біля половини елементів послідовності $a[1]..a[j-1]$ більше за $a[j]$. Це означає, що t_i в середньому можна вважати рівним $i/2$, отже і функція часової складності роботи алгоритму $T_{avg}(n)$ буде квадратичною:

$$T_{avg}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{4} - 2\right) +$$

$$+ c_5\left(\frac{n(n-1)}{4}\right) + c_6\left(\frac{n(n-1)}{4}\right) +$$

$$+ c_7(n-1) = \left(\frac{c_4}{4} + \frac{c_5}{4} + \frac{c_6}{4}\right)n^2 +$$

$$+ (c_1 + c_2 + c_3 + \frac{c_4}{4} + \frac{c_5}{4} + \frac{c_6}{4} + c_7)n - (c_2 + c_3 + 2c_4 + c_7)$$

Цікаво, що у найбільш сприятливішому випадку (послідовність впорядкована), рядок (4) завершується після першої ж перевірки, так що всі t_i рівні 1, і часова складність $T_{min}(n)$ дорівнює

$$\begin{aligned} T_{min}(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + \\ &+ c_7(n-1) = \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

Таким чином, у найбільш сприятливішому випадку час $T_{min}(n)$, необхідний для впорядкування послідовності розмірністю n , є лінійною функцією від n , тобто має вигляд $T_{min}(n)=an+b$ для деяких констант a і b (ці константи знову визначаються обраними значеннями c_1, \dots, c_7).

Наведений приклад показує, що час роботи у найгіршому та найбільш сприятливішому випадках може значно відрізнятись.

Підводячи підсумок, порівнюємо ефективність розглянутих нами алгоритмів. Для цього скористаємося наведеними Н.Віртом аналітичними формулами [5, с.106], які дозволяють визначити часові складності $T_{min}(n)$, $T_{avg}(n)$ і $T_{max}(n)$ (див. *Таблиця 2.10.*). Зазначимо, що Н.Вірт зосереджує увагу лише на операціях порівняння (С) і обміну (М).

Аналітичні формулами для визначення часової складності алгоритмів

	С/М	$T_{min}(n)$	$T_{avg}(n)$	$T_{max}(n)$
Впорядкування обміном	С	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	М	0	$(n^2-n)*0.75$	$(n^2-n)*1.5$
Впорядкування Вибором	С	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	М	$3(n-1)$	$n(\ln n+0.57)$	$n^2/4+3(n-1)$
Впорядкування Вставками	С	$n-1$	$(n^2+n-2)/4$	$(n^2-n)/2 - 1$
	М	$2(n-1)$	$(n^2+9n-10)/4$	$(n^2+3n-4)/2$

Як бачимо, для кожного із розглянутих нами алгоритмів функція визначення часової складності в середньому і найгіршому випадках є квадратичною. Існують більш ефективні алгоритму, які іноді називають „логарифмічними” за тієї

причини, що функція визначення часової складності в середньому і найгіршому випадках є $T(n)=an\log n+b+c$. Різницю між „квадратичними” і „логарифмічними” алгоритмами наочно демонструє приклад, наведений у роботі [13,с.29-30]. Нехай потрібно впорядкувати послідовність розмірністю в один мільйон елементів. Виникає питання: що швидше – впорядковувати його алгоритмом вставок з часовою складністю, щодо відповідає квадратичній функції $T(n)=c_1n^2$ на комп’ютері, який виконує 100 мільйонів операцій за секунду, чи алгоритмів впорядкування злиттям з часовою складністю $T(n)=c_2n\log n$. При цьому алгоритм впорядкування вставками написаний надзвичайно економно і для сортування n чисел потрібно лише $2n^2$ операцій. У той час алгоритм впорядкування злиттям написаний без особливої турботи про ефективність і вимагає $50n\log n$ операцій.

Для впорядкування мільйона елементів послідовності одержуємо для потужного комп’ютера:

$$\frac{2(10^6)^2 \text{ операцій}}{10^8 \text{ операцій за секунду}} = 20000 \text{ сек} \approx 5,56 \text{ год}$$

натомість для повільного лише

$$\frac{50(10^6) \log(10^6) \text{ операцій}}{10^6 \text{ операцій за секунду}} \approx 1000 \text{ сек} \approx 17 \text{ хв.}$$

Як бачимо, перевага більш ефективного алгоритму очевидна.

Алгоритми впорядкування, які були нами використанні для ознайомлення з методикою визначення складності алгоритмів, базувалися на операціях порівняння та обміну елементів. Проте, існують алгоритми впорядкування які не тільки не використовують операцію порівняння елементів, а й мають часову складність у найгіршому випадку $T_{\max}(n)=an+b$. Серед цих алгоритмів виділимо *алгоритм впорядкування підрахунком* і проаналізуємо його.

Нехай дана послідовність розмірністю n елементів, кожен з яких має цілочисельне додатне значення, і відоме значення k найбільшого елемента з всієї послідовності. Ідея впорядкування підрахунком полягає у записі до допоміжної послідовності в

позицію, що відповідає значенню елемента вихідної послідовності, кількості таких елементів.

Проілюструємо роботу алгоритму впорядкування підрахунком наступним прикладом, що записаний на псевдокодї.

- (1) *for* $i \leftarrow 1$ *step* 1 *until* k *do* $c[i] \leftarrow 0$
- (2) *for* $i \leftarrow 1$ *step* 1 *until* n *do* $c[a[i]] \leftarrow c[a[i]] + 1$
- (3) *for* $i \leftarrow 2$ *step* 1 *until* k *do* $c[i] \leftarrow c[i] + c[i-1]$
- (4) *for* $i \leftarrow n$ *step* -1 *until* 1 *do*
 begin
- (5) $b[c[a[i]]] \leftarrow a[i]$
- (6) $c[a[i]] \leftarrow c[a[i]] - 1$
 end

Проаналізуємо роботу алгоритму впорядкування підрахунком, в якому використовуються вихідна $a[1..n]$, результуюча $b[1..n]$ і допоміжна $c[1..k]$ послідовності. Після ініціювання допоміжної послідовності $c[1..k]$ оператором повторення в рядку (1), до i позиції цієї послідовності записується кількість елементів послідовності $a[1..n]$, значення яких дорівнює i (рядок 2). В рядку (3) алгоритму відбувається обчислення часткових сум послідовності $c[1], c[2], \dots, c[k]$, що відповідають кількості елементів, значення яких не перевищує i . В рядках (4)-(5) кожен елемент послідовності $a[1..n]$ записується в необхідну позицію послідовності $b[1..n]$. Дійсно, якщо всі n вихідної послідовності $a[1..n]$ відмінні один від одного, то в упорядкованій послідовності $b[1..n]$ елемент $a[i]$ повинен бути записаним в позицію $c[a[i]]$. У випадку, коли в вихідній послідовності $a[1..n]$ зустрічаються елементи з однаковими значеннями, то після кожного запису елемента $a[i]$ в послідовність $b[1..n]$ (рядок 5) значення $c[a[i]]$ елемента зменшується на 1 (рядок (6)). Це дозволить на наступному етапі записати елемент, значення якого дорівнюватиме $a[i]$, в позицію зліва. Такий підхід демонструє одну із властивостей алгоритмів впорядкування, що називається *стійкістю*. Саме стійкість алгоритму впорядкування вказує на необхідність запису елементів з однаковими значеннями в тому порядку, в якому

вони були записані у вихідній послідовності. Така властивість особливо актуальна за умови впорядкування послідовностей елементів, кожен з яких є записом, що разом із чисельними значеннями зберігає і додаткову інформацію.

Аналіз складності алгоритму впорядкування підрахунком показує, що час $T_{max}(n)$ у найгіршому випадку є лінійною функцією від n і k , тобто має вигляд $T_{max}(n)=an+bk$ для деяких констант a і b .

2.4.2. Асимптотична часова складність алгоритмів

Література: [4,29-35;4,265-275;5,105-108;13,26-30;13,811-837;17,352-416]

Ключові поняття: *асимптотична часова складність, асимптотично точна оцінка складності, швидкість росту часової складності, Θ -символіка, поліноміальна й експоненціальна складність, класи P і NP .*

Раніше було зазначено, що одиниця виміру $T(n)$ точно не визначена, проте розумілася нами як кількість кроків, що виконані на ідеалізованому комп'ютері.

Справді, застосувати такі стандартні одиниці виміру часу як секунди і мілісекунди ми не можемо, адже на час виконання програми, що реалізовує той чи інший алгоритм, впливають ще й такі фактори як кількість і якість скомпільованого коду, архітектура і набір внутрішніх інструкцій ЕОМ тощо.

Час виконання також може суттєво залежить від вибраної множини тестових вхідних даних. Звичайно, як зазначає Н.Вірт, для практичних цілей корисно мати експериментальні дані, що можуть „пролити світло” на коефіцієнти c_i , тим самим провести подальшу оцінку різних методів впорядкування. Наведені у роботі [5, с.105-108] експериментальні результати (в мілісекундах) (див. *Таблиця 2.11.*) базуються на обчисленнях, що були проведені на системою CDC 6400 (в якості мови програмування для реалізації алгоритмів була, звичайно ж, обрана мова Pascal).

Експериментальні дані роботи алгоритмів (в мілісекундах)

	Упорядкована послідовність		Випадковий розподіл		Послідовність упорядкована у зворотному порядку	
	256	512	256	512	256	512
Впорядкування обміном	540	2165	1026	4054	1492	5931
Впорядкування вибором	489	1907	509	1956	695	2675
Впорядкування вставками	12	23	366	1444	704	2836

В якості критерію визначення ефективності алгоритмів можуть використовуватися різні показники. Зокрема, Д.Кнут наводить в якості одного з поширених тестів для порівняльної оцінки різних алгоритмів задачу впорядкування послідовності розмірності n 100-символьних записів. Так, автор зазначає, що терабайтова послідовність – 10^{10} записів по 100 символів – була впорядкована у вересні 1997 року за 2,5 години. При цьому використовувалася система Silicon Graphics Origin2000, що складалася з 32 процесорів, 8 Гбайт оперативної пам'яті, 559 дисків по 4 Гбайти, та програмне забезпечення *NSort*, в основі якого лежить неопублікований алгоритм впорядкування [11, с.420].

Наведені приклади, як і більш сучасні результати експериментального порівняння алгоритмів впорядкування, не спростовують жодної з оцінок складності, що були отримані теоретично. Тим більше, що ця залежність стає очевидною при реалізації одного і того ж алгоритму на різних комп'ютерах, компіляторах, при використанні різних вхідних даних.

Ось чому у більшості випадків роблять висновки на зразок: „часова складність того чи іншого алгоритму пропорційна n^2 ”. Константи пропорційності у більшості випадків також не можна точно визначити. Пов'язано це з тим фактом, що у більшості випадків алгоритми реалізуються

програмами, які записані операторами мов програмування високого рівня. Отже, виникає необхідність запису цих алгоритмів в термінах „кроків”, кожен з яких повинен виконуватися за скінчений фіксований час після трансляції їх у машинну мову будь-якої ЕОМ. Проте, точно визначити час виконання будь-якого кроку алгоритму дуже важко, адже цей час залежить не тільки від „природи” самого кроку, а й від процесу трансляції і машинних інструкцій певного комп’ютера. Ось чому замість точної оцінки ефективності алгоритму, що придатна (актуальна, валідна) для певної обчислювальної системи, прийнято використовувати менш точну, але більш загальну *асимптотичну часову складність* як основну міру ефективності виконання алгоритму.

Для опису *швидкості росту* часової складності алгоритмів використовується Θ -символіка („тета-символіка”). Наприклад, коли говорять що час виконання $T(n)$ деякого алгоритму має порядок $\Theta(n^2)$, тобто $T(n) = \Theta(n^2)$, то вважають, що існують такі константи $c_1, c_2 > 0$ і таке число n_0 , що $c_1 n^2 \leq T(n) \leq c_2 n^2$ для всіх $n \geq n_0$. Оскільки, як було зазначено вище, складність алгоритму – це функція, значення якої залежить від розмірності n даних задачі, то запис $T(n) = \Theta(n^2)$ можна подати у вигляді $f(n) = \Theta(n^2)$. Враховуючи різні значення пропорційності для n різних алгоритмів, точнішим буде використання запису $f(n) = \Theta(g(n))$, де $g(n)$ – деяка функція від n . Зміст цього запису полягає у тому, що існують такі константи $c_1, c_2 > 0$ і таке число n_0 , що $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ для всіх $n \geq n_0$. Якщо для деякого алгоритму $f(n) = \Theta(g(n))$, то функцію $g(n)$ вважають *асимптотично точною оцінкою* його складності.

Для пояснення скористаємося прикладом. Нехай часова складність деякого алгоритму $T(n) = (1/2)n^2 - 3n$. Перевіримо, що цей алгоритм має швидкість росту $\Theta(n^2)$, тобто $(1/2)n^2 - 3n = \Theta(n^2)$. Для цього зазначимо константи c_1, c_2 і число n_0 , так щоб нерівність $c_1 n^2 \leq (1/2)n^2 - 3n \leq c_2 n^2$ виконувалася для всіх $n \geq n_0$. Розділимо нерівність на n^2 і отримаємо $c_1 \leq 1/2 - 3/n \leq c_2$. Таким чином, для виконання другої нерівності достатньо прийняти $c_2 = 1/2$. Перша ж є нерівність виконується, наприклад, при $n_0 = 7$ і $c_1 = 1/14$.

Значимо, що запис $f(n) = \Theta(g(n))$ включає в себе дві оцінки росту швидкості: верхню $O(g(n))$ („о від же від ен“) і нижню $\Omega(g(n))$ („омега від же від ен“).

Вважають, що:

- $f(n) = O(g(n))$, якщо знайдеться така константа $c > 0$ і таке число n_0 , що $0 \leq f(n) \leq cg(n)$ для всіх $n \geq n_0$;
- $f(n) = \Omega(g(n))$, якщо знайдеться така константа $c > 0$ і таке число n_0 , що $0 \leq cg(n) \leq f(n)$ для всіх $n \geq n_0$.

Із наведеного випливає, що для функції $f(n)$ властивість $f(n) = \Theta(g(n))$ виконується тоді і тільки тоді, коли $f(n) = O(g(n))$ і $f(n) = \Omega(g(n))$.

Спрощення, яких ми припускалися при оцінці складності алгоритму $T(n)$, базуються на правилах виконання операцій додавання та множення в Θ -символіці.

Правило сум. Нехай $T_1(n)$ і $T_2(n)$ - час виконання двох програмних фрагментів P_1 і P_2 . $T_1(n)$ має швидкість росту $O(f(n))$, $T_2(n) - O(g(n))$. Тоді $T_1(n) + T_2(n)$, тобто час послідовного виконання фрагментів P_1 і P_2 , має швидкість росту $O(\max(f(n), g(n)))$. Для доказу цього нагадаємо, що існують константи c_1, c_2, n_1 і n_2 такі, що за умови $n \geq n_1$ виконується нерівність $T_1(n) \leq c_1 f(n)$, та аналогічно $T_2(n) \leq c_2 g(n)$, якщо $n \geq n_2$. Нехай $n_0 = \max(n_1, n_2)$. Якщо $n \geq n_0$, то, очевидно, що і $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. Звідси випливає, що при $n \geq n_0$ справедлива нерівність $T_1(n) + T_2(n) \leq (c_1 + c_2) \max(f(n), g(n))$. Остання нерівність і означає, що $T_1(n) + T_2(n)$ має порядок росту $O(\max(f(n), g(n)))$. Наприклад, скористаємося правилом сум, для обчислення часу послідовного виконання програмних фрагментів з повторенням і розгалуженнями. Нехай є три фрагменти з часами виконання відповідно $O(n^2)$, $O(n^3)$ і $O(n \log n)$. Тоді час послідовного виконання перших двох фрагментів має порядок $O(\max(n^2, n^3))$, тобто $O(n^3)$. Час виконання всіх трьох фрагментів має порядок $O(\max(n^3, n \log n))$, тобто знову $O(n^3)$. У загальному випадку час виконання кінцевої послідовності програмних фрагментів, без врахування констант, має порядок фрагмента з найбільшим часом виконання.

З правила сум також випливає, що якщо $g(n) < f(n)$ для всіх n , що перевищують n_0 , то вираз $O(f(n)+g(n))$ еквівалентний $O(f(n))$. Наприклад, вираз $O(n^2+n)$ еквівалентний виразу $O(n^2)$.

Правило добутку. Якщо $T_1(n)$ і $T_2(n)$ мають швидкість росту $O(f(n))$ і $O(g(n))$ відповідно, то добуток $T_1(n)*T_2(n)$ має швидкість росту $O(f(n)*g(n))$. З правила добутку випливає, що $O(n^2/2)$ еквівалентно $O(n^2)$.

Як було показано раніше, часову складність алгоритмів можна оцінити за допомогою функцій часу виконання, зневажаючи при цьому константами пропорційності. За такої умови, алгоритм з часом виконання $\Theta(n^2)$, наприклад, ефективніше алгоритму з часом виконання $\Theta(n^3)$. Константи пропорційності залежать не тільки від компілятора і комп'ютера, але і від властивостей самого алгоритму. Нехай при заданій комбінації „компілятор-комп'ютер” один алгоритм виконується за $100n^2$ мілісекунд, а інший - за $5n^3$ мілісекунд. Чи може другий алгоритм бути ефективнішим, ніж перша?

Відповідь на це питання залежить від розміру вхідних даних n . При розмірі вхідних даних $n < 20$ алгоритм з часом виконання $5n^3$ завершиться швидше, ніж алгоритм з часом виконання $100n^2$. З цього робимо висновок про те, що, коли алгоритм в основному опрацьовує вхідні дані незначної розмірності, перевагу необхідно віддати алгоритму з часом виконання $\Theta(n^3)$. Однак при зростанні n відношення часу виконання $5n^3/100n^2 = n/20$ також росте. Тому при значних n алгоритм з часом виконання $\Theta(n^2)$ стає переважнішим алгоритму з часом виконання $\Theta(n^3)$. Якщо навіть при порівняно невеликих n , коли час виконання обох програм приблизно однаково, вибір ефективнішого алгоритму є дещо складним, то для більшої надійності роблять вибір на користь алгоритму з меншою швидкістю росту складності. Ще однією причиною, що змушує віддавати перевагу алгоритмам з найменшої швидкістю росту складності, є той факт, що чим меншою є швидкість росту складності алгоритму, тим більший розмір задачі, яку можна обчислити на ЕОМ. Іншими словами, якщо збільшується швидкість обчислень комп'ютера, то росте також і розмір задач, розв'язуваних на комп'ютері. Однак незначне збільшення

швидкості обчислень комп'ютера приводить тільки до невеликого збільшення розмірності задач, розв'язуваних протягом фіксованого проміжку часу.

Для пояснення скористаємося прикладом, наведеним в [3, с. 12-14]. Нехай є п'ять алгоритмів $A_1 \dots A_5$ з часовими складностями $n, n \log_2 n, n^2, n^3$ і 2^n відповідно (див. Таблиця 2.12.). Припустимо, що одиниця інформації перероблюється за одну мілісекунду. Тоді алгоритм A_1 може обробити за одну секунду вхідну інформацію розмірністю 1000 одиниць, A_1 -140, в той час як алгоритм A_5 – лише 9 одиниць. Складемо залежність можливих розмірностей задачі від складності алгоритму.

Таблиця 2.12.

Залежність можливих розмірів задачі від складності алгоритму

Алгоритм	Часова складність	Розмірність задачі		
		1 с	1 хв	1 год
A_1	n	1000	$6 \cdot 10^4$	$3,6 \cdot 10^6$
A_2	$n \log_2 n$	140	4893	$2 \cdot 10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Розглянемо, як вплине збільшення швидкодії обчислювальної техніки на можливості використання алгоритмів $A_1 \dots A_5$.

Враховуючи, що алгоритм обчислюється на одній ЕОМ, вважаємо час виконання алгоритму дорівнює $T_a = f_a(n)$. Зафіксуємо максимально припустимий час T , на протязі якого працювати ЕОМ. Тоді, вирішуючи рівняння $f_a(n) = T$ відносно n , можна знайти найбільшу розмірність N , для якої ще можна застосувати алгоритм A , не перевищуючи час T . Наприклад, $f_a(n) = n^2$, то $N = \sqrt{T}$.

Тепер, використовуємо для виконання цих алгоритмів ЕОМ із збільшеною в 10 разів швидкодією (див. Таблиця 2.13.). Здавалося, що такий підхід призведе до збільшення в 10 разів максимального часу роботи старої ЕОМ, тобто $f_a(n) = 10T$. Дійсно, розмірність n (стара) і N (нова) пов'язані відношенням $f_a(N) = 10 f_a(n)$. Зокрема, якщо $f_a(n) = n^2$, то $(N^2) = 10(n^2)$, звідки

$N = n\sqrt{10} \approx 3.16n$. Проте, у випадку $f_a(n)=2^n$, то $2^N=10*2^n$, звідки $N=n+\log_2 10 \approx n+3.3$.

Як видно з табл. 2.13., десятикратне збільшення швидкодії ЕОМ призвело до збільшення розмірності, що може бути обчислена алгоритмом A_3 , у три рази, а алгоритмом A_5 - всього на три одиниці. Якщо як основу для порівняння взяти 1 хвилину, то, замінюючи алгоритм A_4 алгоритмом A_3 , можна вирішити у 6 разів більшу задачу, а замінюючи A_4 на A_2 – більшу в 125 разів. Ці результати роблять набагато більше враження, ніж дворазове поліпшення, що досягається за рахунок десятикратного збільшення швидкості. Якщо за основу для порівняння взяти 1 годину, то розходження виявляється ще значніше.

З табл. 2.13. видно, що у випадку поліноміальних алгоритмів розмірність задачі із збільшенням швидкодії ЕОМ збільшується на мультиплікатну константу, тоді як для експоненціальних алгоритмів має місце збільшення на адитивну константу.

Таблиця 2.13.

Залежність можливих розмірів задачі від складності алгоритму

Алгоритм	Часової складності	Розмірність задачі		
		x1	x10	x100
A_1	n	a_1	$10*a_1$	$100*a_1$
A_2	$n\log_2 n$	a_2	$10*a_2$	$10*a_2$
A_3	n^2	a_3	$3,16*a_3$	$10*a_3$
A_4	n^3	a_4	$2,15*a_4$	$4,64*a_4$
A_5	2^n	a_5	$a_5+3,3$	$a_5+6,64$

Дослідження складності алгоритмів привели до усвідомлення важливого факту: подібно до того як існують алгоритмічно нерозв'язувані проблеми, існують і задачі об'єктивно складні (такі, складність яких суттєво не зменшується із покращенням швидкодії ЕОМ).

Підвищення фізичної швидкодії зменшує час фізичного часу в константне число раз, однак не змінює ступінь поліному у функції складності. Зміна архітектури ЕОМ іноді може дещо

зменшити цей ступінь. Якщо ж задача має складність вище поліноміальної (2^n , 3^n тощо), то складність цієї задачі зберігається при будь-якому прогресі обчислювальної техніки.

Слід зазначити, що множина проблем (задач), для якої існують алгоритми з поліноміальною складністю, називається класом P . Звідси виходить, що проблеми з класу P є *поліноміально розв'язуваними*. Клас проблем P прийнято ототожнювати з класом „практично розв'язуваних” проблем. На відміну від класу P , існують проблеми експоненціальної складності (клас NP), які відповідають повному перебору підмножини n – елементів множини або навіть перевищують його за складністю. Скорочення NP походить від англійських слів „*Nondeterministic Polynomial (time)*”, що перекладається як „недетермінований поліноміальний час”. Назва класу пов'язана з тим фактом, що історично клас NP задач вперше був описаний у термінах так званих недетермінованих машин Тьюрінга.

Вивчення питання співвідношення класів P і NP призвело

?

до постановки проблеми $P=NP$, яка на сьогодні лишається однією з найскладніших проблем теорії обчислень. Вважається, що NP задачі не можуть бути вирішені за поліноміальний час, а,

?

отже, проблему $P=NP$ можна сформулювати гіпотезою $P \neq NP$. Найбільш вірогідне співвідношення між класами P і NP в такому випадку можна зобразити рис. 2.8.

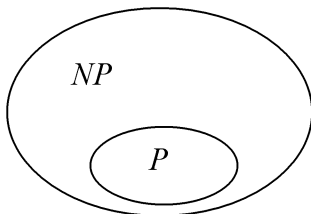


Рис. 2.8. Співвідношення між класами P і NP

Звичайно, хоча експоненціальна функція (така, як 2^n) росте швидше будь-якої поліноміальної функції від n , для невеликих значень n алгоритм, що вимагає $\Theta(2^n)$ часу, може виявитися ефективніше багатьох алгоритмів із поліноміально обмеженим часом роботи. Наприклад, сама функція 2^n не перевищує n^{10} до значення n , рівного 59. Однак швидкість росту експоненціальної функції настільки

стрімка, що такі задачі називають „важко розв'язними”, якщо у всіх алгоритмів, які її вирішують, складність щонайменше експоненціальна. Ще одна перевага поліноміальних алгоритмів полягає в тому, що, коли на практиці вдається вирішити задачу за поліноміальний час, то ступінь поліному в ході експериментальних досліджень часто зменшується. Це пов'язано з тим, що різні дослідники вдосконалюють алгоритм, іноді покращуючи його часову складність до $\Theta(n^3)$ чи ще нижче. Експоненціальні алгоритми, навпаки, на практиці вимагають стільки ж часу як і в теорії. Ось чому від таких алгоритмів відмовляються як тільки розроблено алгоритм з поліноміальною часовою складністю. На сьогодні відомо багато задач класу NP , пов'язаних із різними областями математики та інформатики: математичною логікою, теорією графів, комп'ютерними мережами, множинами, математичним програмуванням, алгеброю і теорією чисел, оптимізацією програм тощо. Наявність цих задач, практична значущість яких не дозволяє обмежитись простим віднесенням їх до задач класу NP , вказує на необхідність перегляду стратегій їх виконання. Серед підходів, що є менш безнадійними, аніж спроби виконання задач класу NP , слід виділити:

1. *Наближені* алгоритми. Такі алгоритми дозволяють одержувати не оптимальні рішення, а рішення, що відрізняються від дійсного оптимуму не більше ніж на фіксовану частку цього оптимуму. На практиці таке, близьке до оптимального, рішення може бути достатнім.

2. *Ймовірнісні* алгоритми. Іноді виявляється можливим побудувати алгоритми, які дуже часто щодо деякого ймовірнісного розподілу на індивідуальних задачах працюють досить непогано – як в сенсі якості одержаних рішень, так і часу, витраченого на їх одержання.

3. *Евристичні* алгоритми. Алгоритми, математична основа яких часто дуже сумнівна, дозволяють на практиці отримувати рішення, що відрізняються на декілька відсотків від оптимального.

Звісно, запропоновані підходи не виключають використання експоненціальних алгоритмів, тим більше що такі алгоритми при незначних n можуть, зрештою, бути досить

оптимальними. Крім того, використання методів розробки ефективних алгоритмів (наприклад, методу розгалужень і меж чи динамічного програмування) у поєднанні з евристичними ідеями (наприклад, „жадібна” стратегія чи генетичний алгоритм) дозволяють проектувати алгоритми, результати роботи яких на практиці є досить прийнятними.

2.5. МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ

На сьогодні відомо декілька методів побудови ефективних алгоритмів. Серед них слід виділити такі як декомпозиції, метод розгалужень і меж, динамічне програмування, „жадібні” методи, пошук з поверненням і локальний пошук. Слід зазначити, що жоден із вказаних методів не є універсальним, адже вибір того чи іншого методу дуже часто залежить від природи даних, які необхідно опрацювати. Проте знання цих методів є необхідною умовою при розробці ефективних алгоритмів.

2.5.1. Декомпозиція

Література: [3, 75-81; 4, 276-280; 13, 15-23]

Ключові поняття: *декомпозиція, лінійний перегляд, впорядкування злиттям, збалансоване і незбалансоване розбиття.*

Одним з найбільш широко застосованим методом проектування ефективних алгоритмів є *метод декомпозиції (метод розбивки або метод "розділяй і пануй")*. Цей метод припускає таку декомпозицію (розбивку) задачі на більш дрібні задачі, що на основі рішень цих більш дрібних задач можна легко одержати рішення вихідної задачі. Продемонструємо принцип роботи методу декомпозиції на прикладі декількох алгоритмів.

Спочатку скористуємося задачею пошуку k -елемента у послідовності розмірністю n . Найбільш прості алгоритми базуються на послідовному перегляді кожного елемента на

відповідність певній властивості, зокрема, чи дорівнює k -елемент шуканому. Такий метод називають *лінійним* переглядом (пошуком), а його часова складність відповідає $\Theta(n)$.

Задача пошуку істотно спрощується, якщо послідовність упорядкована. У такому випадку пошук полягає у поділі послідовності розмірністю n на дві рівні частини і перевірки приналежності шуканого k -елемента лівій ($1..m$) чи правій ($m+1..n$) частині. Далі процес рекурсивно продовжується вже над однією з половин розмірності до тих пір, доки не буде знайдено шуканий елемент. Описаний підхід, в основі якого лежить метод декомпозиції, називається „бінарним („логіфічним”) пошуком”. Часова складність алгоритму „бінарного пошуку” відповідає $\Theta(\log_2 n)$.

Метод декомпозиції можна використати для впорядкування послідовності. Справді, застосування методу декомпозиції до алгоритму впорядкування вставками дозволяє зменшити його часову складність до $\Theta(n \log_2 n)$. Пояснюється це тим, що при впорядкуванні методом простих вставок j -ий елемент порівнюється в середньому близько з $j/2$ раніше впорядкованих елементів. Ось чому загальна кількість порівнянь алгоритму дорівнює приблизно $(1+2+\dots+n)/2 \approx n^2/4$, що досить багато, навіть при незначних n . Проте, алгоритм впорядкування вставками можна вдосконалити, скориставшись „бінарним пошуком”: позиція вставки j -ого елемента буде визначена за $\log_2 n$ операцій порівняння з відповідним чином вибраними елементами впорядкованої послідовності. Наприклад, при необхідності визначення позиції елемента $a[64]$, його значення спочатку порівнюється елементом $a[32]$, потім, якщо це значення менше, воно порівнюється з $a[16]$, якщо ж більше – то з $a[48]$ тощо. Таким чином, позиція елемента $a[64]$ буде знайдена за 6 порівнянь. Загальна кількість порівнянь для послідовності розмірністю n приблизно дорівнює $n \log_2 n$. Наведений алгоритм називається *алгоритмом впорядкування бінарними вставками*, а його часова складність відповідає $T(n) = \Theta(n \log_2 n)$

Проілюструємо роботу алгоритму впорядкування бінарними вставками наступним прикладом, що записаний на псевдокоді.

```
for  $i \leftarrow 2$  step 1 until  $n$  do  
  begin{for}  
     $key \leftarrow a[i]$   
    If  $b < a[i-1]$  then  
      begin  
         $l \leftarrow 1$ ;  
         $r \leftarrow i$ ;  
        repeat  
           $m \leftarrow (l + r) \setminus 2$ ; { „\” - цілочисельне ділення}  
          if  $b < a[m]$  then  
             $r \leftarrow m$   
          else  $l \leftarrow m + 1$ ;  
        until  $(l = m)$ ;  
      for  $l \leftarrow i-1$  step -1 until  $m$  do  $a[l+1] \leftarrow a[l]$ ;  
       $a[m] \leftarrow b$ ;  
    end;  
  end{for}
```

Алгоритм впорядкування бінарними вставками неєдиний алгоритм, що використовує метод декомпозиції. Розглянемо принцип роботи так званого *алгоритму впорядкування злиттям*.

У алгоритмі впорядкування злиттям послідовність спочатку поділяється на дві половини меншої розмірності. Потім відбувається впорядкування кожної з половин окремо. Після цього дві упорядковані послідовності половинного розміру зливаються в одну. Рекурсивне розбиття задачі на менші відбувається доти, поки розмірність послідовності не дорівнюватиме 1 (будь-яку послідовність розмірністю в 1 елемент можна вважати упорядкованим).

Підпрограму впорядкування злиттям можна записати на псевдокодi у наступному виглядi:

```
procedure Merge_Sort (i,j)
if i<j then
  begin
    m←(i+j)\2
    Merge(Merge_Sort(i,m), Merge_Sort(m+1,j))
  end
```

Виконаємо аналіз складності цієї підпрограми. Як було сказано раніше, метод декомпозиції базується на рекурсивному виклику деякої підпрограми для вирішення задачі меншої розмірності. Таким чином, ми повинні враховувати час, витрачений на рекурсивні виклики. Внаслідок цього одержуємо рекурентне співвідношення, яке і дозволяє оцінити часову складність алгоритму. Припустимо, що алгоритм розбиває задачу розмірністю n на a підзадач, кожна з яких має в b разів меншу розмірність. Будемо вважати, що розбиття вимагає часу $D(n)$, а з'єднання отриманих рішень — часу $C(n)$. Тоді одержуємо співвідношення для часу роботи $T(n)$ на задачах розмірністю n (у найгіршому випадку): $T(n)=aT(n/2)+D(n)+C(n)$. Це співвідношення виконане для досить великих n , коли задачу має сенс розбивати на підзадачі. Для малих n , коли така розбивка неможлива або не потрібно, застосовується якийсь прямий метод рішення задачі. Припустимо, що розмірність послідовності є ступінь двійки (хоча це обмеження не дуже істотне). Тоді на кожному кроці послідовність поділяється навпіл. Розбиття на частини вимагає часу $\Theta(1)$, а злиття – часу $\Theta(n)$. Одержуємо співвідношення:

$$T(n) = \begin{cases} \Theta(n), & \text{якщо } n = 1, \\ 2T(n/2) + \Theta(n), & \text{якщо } n > 1. \end{cases}$$

Це співвідношення приводить до значення $T(n) = \Theta(n \log_2 n)$.

Таким чином, часова складність алгоритму впорядкування злиттям є $T(n) = \Theta(n \log_2 n)$, що виявляється значно ефективнішою за інші алгоритми зі часовою складністю в $T(n) = \Theta(n^2)$.

Рекурентні рівняння часто зустрічаються при аналізі складності алгоритмів. Тому наведемо рішення таких рівнянь в узагальненому вигляді.

Нехай a, b і c – додатні константи. Рішення рекурентного рівняння:

$$T(n) = \begin{cases} b, & \text{якщо } n=1, \\ aT(n/c) + bn, & \text{якщо } n > 1 \end{cases}$$

де n – ступінь числа c , має вигляд

$$T(n) = \begin{cases} \Theta(n), & \text{якщо } a < c, \\ \Theta(n \log n), & \text{якщо } a = c, \\ \Theta(n^{\log_c a}), & \text{якщо } a > c. \end{cases}$$

В якості доведення покажемо, що коли n – ступінь числа c , то $T(n) = bn \sum_{i=0}^{\log_c n} r^i$, де $r=a/c$. Якщо $a < c$, то $\sum_{i=0}^{\log_c n} r^i$

сходиться, а відповідно $T(n) = \Theta(n)$. Якщо $a=c$, то кожним членом ряду буде 1, а всього в ньому $\Theta(\log n)$ членів. Ось чому

$$T(n) = \Theta(n \log n). \text{ І якщо } a > c, \text{ то } bn \sum_{i=0}^{\log_c n} r^i = bn \frac{r^{\log_c n} - 1}{r - 1}, \text{ що}$$

складає $\Theta(n^{\log_c a})$.

Слід зазначити, що при використанні методу декомпозиції бажано, щоб підзадачі були однакової розмірності. Наприклад, алгоритм впорядкування вставками можна розглядати як поділ послідовності на дві половини – одна розмірністю в 1, а інша – в $n-1$ елементів (див.рис.2.9.).

Оскільки процедура поділу займає час $\Theta(n)$, для часу роботи $T(n)$ одержуємо співвідношення $T(n) = T(n-1) + \Theta(n)$. Оскільки $T(1) = \Theta(1)$, то

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^n \Theta(k) = \Theta\left(\sum_{k=1}^n (k)\right) = \Theta(n^2)$$

Як бачимо, за умови максимально незбалансованого розбиття час роботи алгоритму вставками складає $\Theta(n^2)$. Найкращим випадком вважається ситуація, коли послідовність розподіляється на дві рівні частини (див. рис. 2.10.).

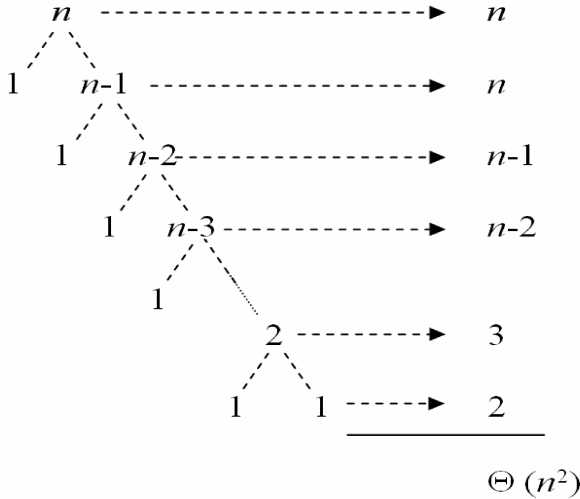


Рис. 2.9. Приклад незбалансованого розбиття

Рекурентне співвідношення в такому випадку має наступний вигляд $T(n) = 2T(n/2) + \Theta(n)$, що відповідає $T(n) = \Theta(n \log_2 n)$.

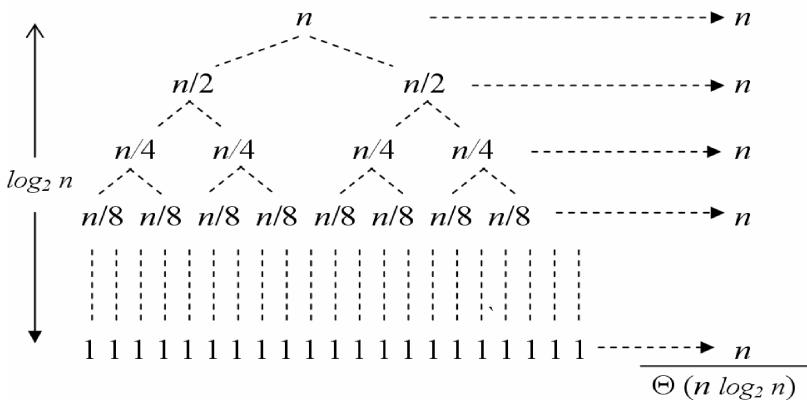


Рис. 2.10. Приклад збалансованого розбиття

2.5.2. Метод розгалужень і меж

Література:[4,296-298;17,446-461]

Ключові поняття: *метод розгалужень і меж, оптимізаційні задачі, задача комівояжера (traveling-salesman problem), повний перебір, дерево рішень, нижня оцінка вартості, правила прийняття рішень.*

Для пояснення принципу роботи *методу розгалужень і меж* серед множини задач класу NP виділимо клас так званих *дискретних оптимізаційних задач*.

Всі задачі цього класу мають перелік характерних властивостей, зокрема:

1. В кожній задачі є лише кінцева кількість варіантів, з яких необхідно зробити вибір.
2. Кожному з варіантів зіставлена деяка чисельна характеристика.
3. Необхідно обрати варіант, чисельна характеристика якого досягає екстремуму.

Однією із перших таких задач, що привернули увагу математиків, є так звана *задача комівояжера (traveling-salesman problem)*. Зміст задачі полягає у наступному. Нехай є n міст $A_1, A_2, A_3, \dots, A_n$ із заданими відстанями між ними $d_{ij}(i, j=1, 2, \dots, n)$. Необхідно, вирушаючи з міста A_1 , обрати такий маршрут переміщення $A_1, A_{i,1}, A_{i,2}, \dots, A_{1,j}, A_{2,j}, \dots, A_{1,j}$, за якого комівояжер, побувавши в кожному з міст, повернувся б в A_1 , пройшовши мінімально можливий сумарний шлях.

У загальному випадку задача комівояжера формулюється в термінах *теорії графів* і полягає в пошуку в неорієнтованому *графі* з ваговими значеннями *ребер* такого маршруту (простого циклу, що включає всі вершини), у якому сума всіх ваг ребер буде мінімальною.

В якості основи для задачі комівояжера скористаємося графом з п'ятьма вершинами (див. рис.2.11.). Вага (вартість) кожного ребра, що зазначена цифрами на рис. 2.11., не обов'язково дорівнює довжині ребра. Це може бути, наприклад, час шляху чи плата за проїзд.

Теоретичний курс

Найбільш очевидним методом рішення задачі комівояжера, як і інших дискретних оптимізаційних задач, є повний перебір всіх варіантів. Проте, алгоритм знаходження оптимального маршруту шляхом повного перебору всіх маршрутів з n вузлами має часову складність $\Theta(n!)$, оскільки необхідно переглянути $(n-1)!$ різних перестановок, а оцінка кожної з перестановок займає час $\Theta(n)$, тобто загальна кількість маршрутів $n! = 1 * 2 * 3 * 4 * \dots * (n-1) * n$.

Цей процес можна зобразити наступним фрагментом дерева рішень (див. рис. 2.12.)

Таким чином, виникає об'єктивна необхідність в пошуку методів побудови більш ефективних алгоритмів для рішення задачі комівояжера, що дозволять скоротити повний перегляд, відкидаючи заздалегідь непридатну множину варіантів.

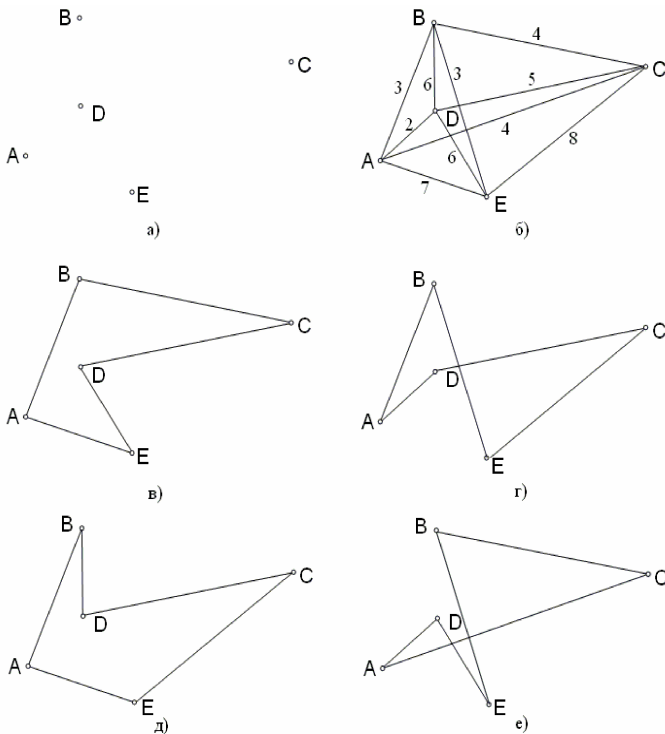


Рис. 2.11. Приклади побудови маршрутів для задачі комівояжера

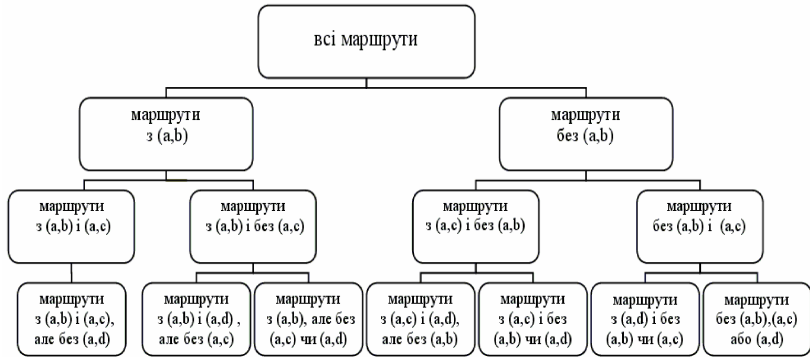


Рис. 2.12. Фрагмент дерева рішень повним переглядом для задачі комівояжера

Метод розгалужень і меж базується на ідеї „розумного” перегляду всіх припустимих варіантів комбінаторної задачі оптимізації. Застереження „розумного” має важливе значення, оскільки відбувається не просто перегляд всіх варіантів, як у випадку повного перегляду, а будується доказ оптимальності деякого рішення на основі послідовного розбиття простору рішень.

Слово „розгалужень” у назві методу відноситься саме до цього процесу розбиття. Термін „меж” відноситься до процесу визначення так званих *нижніх оцінок* (меж) вартості (наприклад, вартості того чи іншого маршруту), що використовуються при побудові доказу оптимальності без повного перегляду всіх варіантів.

Як і у випадку рішення задачі комівояжера повним перебором всіх маршрутів з n вузлами, при використанні методу розгалужень і меж також будується дерево рішень. При цьому кожна вершина (вузол) має два розгалуження (сина): те, що містить певне ребро і те, у якого такого ребра немає.

Ребра дерева рішень переглядаються в лексикографічному порядку, тобто: (a,b) , (a,c) ,..., (a,f) , (b,c) ,..., (b,f) , (c,d) тощо.

Рішення про необхідність включення чи вилучення того чи іншого ребра приймається на основі наступних *правил*:

правило 1: вибір ребра не призводить до появи вершини зі ступенем три чи більше, тобто кількість ребер, що виходять з цієї вершини не повинна перевищувати два ребра;

правило 2: вибір ребра не призводить до утворення замкнутого контуру (циклу) з ребрами, що були прийняті раніше;

правило 3: якщо вилучення ребра (x,y) призводить до того, що у вершини x чи y немає інших ребер на даному маршруті, тоді ребро (x,y) необхідно включити;

правило 4: якщо включення ребра (x,y) призводить до того, що у вершини x чи y буде більше двох ребер на даному маршруті чи утворюється цикл з раніше включеними ребрами, тоді ребро (x,y) необхідно вилучити.

На наступному етапі обчислюється нижня межа вартості для всіх рішень задачі. При цьому приймається до уваги, що вартість будь-якого маршруту можна обчислити як половину суми по всім вузлам n вартості пар ребер цього маршруту. Таким чином, сума вартості двох ребер, що входять до маршруту, не може мати вартість меншу, ніж половина суми по всім вершинам n вартості двох ребер найменшої вартості. Отже, жоден з маршрутів не може мати вартість меншу, ніж половина суми по всім вершинам вартості двох ребер найменшої вартості, що виходять з цієї вершини.

У випадку задачі комівояжера, поданої мал. 2.13, нижня межа вартості маршруту становить 17.5 і обчислюється наступним чином. Спочатку обирають два ребра вершини A з найменшою вартістю. Це ребра (a,b) і (a,d) з сумарною вартістю 5. Для вершини B такими ребрами є (a,b) і (b,e) із сумарною вартістю, що дорівнює 6. Аналогічно, сумарна вартість двох ребер з найменшою вартістю вершин C , D і E дорівнює відповідно 8, 7 і 9. Отже, нижня межа вартості маршруту складатиме $(5+6+3+7+9)/2=17.5$. Проте, прийняті правила включення і вилучення ребер, зокрема правило 3 і 4, вимагають включити в маршрут ребро (a,e) і виключити ребро (b,c) . Таким

чином, двома ребрами для вершини A будуть ребра (a, d) і (a, e) , загальна вартість яких дорівнює 9. Для вершини b залишаються ребра (a,b) і (b,e) із сумарною вартістю, що дорівнює 6. Для вершини C вибір ребра (b, c) неможливий, отже обираються ребра (a, c) і (c, d) , загальна вартість яких дорівнює 9.

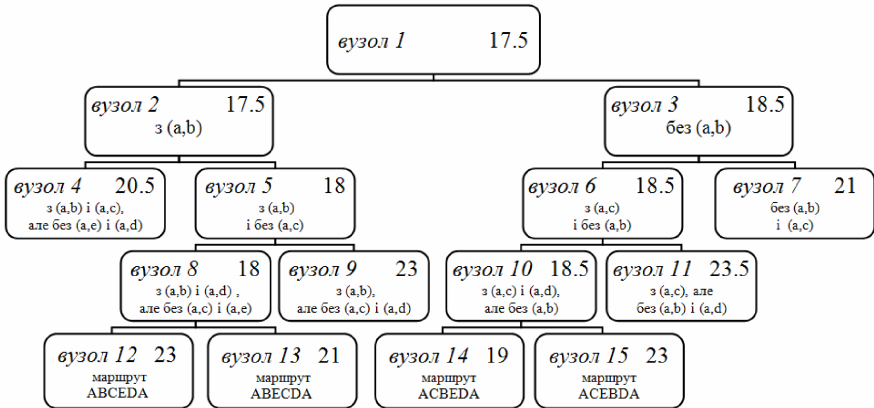


Рис. 2.13. Фрагмент дерева рішень методом розгалужень і меж для задачі комівояжера

Для вершини D обраними залишаються ребра (a,d) і (c,d) , тоді як для вершини E необхідно вибрати ребра (a,e) і (b,e) . Таким чином, нижня межа маршруту з урахуванням прийнятих обмежень дорівнює $(9+6+9+7+10)/2=20.5$.

При побудові дерева рішень необхідно враховувати той факт, що коли нижня межа для деякого розгалуження буде вищою за найменшу вартість знайденого на даний момент маршруту, то розгалуження вилучається і його нащадки не розглядаються. Якщо жодного з розгалужень вилучити неможливо, то розглядається спочатку розгалуження з меншою нижньою межею, з надією швидше досягти рішення, що буде мати нижчу вартість за вартість найкращого на даний момент рішення. Розглянувши одне розгалуження, можна ще раз перевірити, чи не можливо вилучити інше розгалуження, адже можливо знайдено нове „найкраще” рішення. Таким чином, одержане нове дерево рішень демонструє принцип роботи методу розгалужень і меж (див. рис. 2.13.).

Приймаємо за вихідну точку маршруту вершину A і позначаємо її як вузол 1. Першим ребром в лексикографічному порядку буде ребро (a,b) . Далі розглядаються сині вершини A - вузли 2 і 3, до маршрутів яких включено і виключено ребро (a,b) відповідно. Зауважимо, що виключене ребро зазначається із знаком заперечення (інверсії), тобто $\overline{a,b}$. У відповідності до прийнятих правил включення ребра (a,b) в маршрут не підвищує його нижню межу, проте виключення цього ребра призводить до підвищення нижньої межі до 18.5, адже тепер вартість найменших двох ребер для вершин A і B становить 6 і 7 відповідно. Отже, спочатку аналізуються нащадки вузла 2.

Наступним ребром у лексикографічному порядку буде ребро (a,c) . Таким чином, ми переходимо до вузлів 4 і 5, що відповідають маршрутам, в яких ребро (a,c) відповідно включається або виключається. Стосовно вузла 4 можна зробити висновок, що ні ребро (a,d) , ні ребро (a,e) не можуть входити в маршрут (правило 1 прийняття ребер). Відповідно розглядаються спочатку вузли 5, а потім 4 і переходимо до ребра (a,d) . Нижні межі для вузлів 8 і 9 дорівнюють відповідно 18 і 23. Для кожного з цих вузлів нам відомі три ребра з ребер, що виходять з вершини A , тому ми можемо зробити висновок стосовно ребра (a,e) . Далі розглянемо синів вузла 8. Першим ребром, що залишилося, у лексикографічному порядку буде ребро (b,c) . Якщо ми включимо в маршрут це ребро, то не зможемо включити ребро (b,d) чи (b,e) , тому що уже виключено ребро (a,b) . Оскільки ми уже виключили ребра (a,e) і (b,c) , у нас повинні бути ребра (c,e) і (d,e) . Ребро (c,d) не може входити в маршрут, інакше у вершин C і D три прилеглі до них ребра входили б у маршрут. Залишається один маршрут (a,b,c,e,d,a) , вартість якого дорівнює 23. Аналогічно можна довести, що вузол 13 з виключеним ребром (b,c) є маршрутом (a,b,e,c,d,a) , вартість якого дорівнює 21. Далі повертаємося до вузла 5 і розглядаємо його другого сина, вузол 9. Проте, нижня межа цього вузла дорівнює 23, що перевершує найкращу на даний момент вартість – 21. Отже, ми видаляємо вузол 9 і повертаємося до вузла 2, де виконуємо аналіз його другого сина, вузла 4. Нижня межа для вузла 4 дорівнює 20.5, проте її

цілочисельне значення - 21- вказує на те, що жоден з маршрутів, що містить вузол 4, не може мати вартість меншу за 21. Але, оскільки у нас вже є маршрут з вартістю 21, нам не прийдеться розглядати нащадків вузла 4, а отже, вилучаємо вузол 4.

На наступному етапі повертаємося у вузол 1 і розглядаємо його другого сина – вузол 3. На рівні вузла 3 ми розглянули тільки одне ребро (a,b) . Вузли 6 і 7 є синами вузла 3. Вузол 6 відповідає тим маршрутам, що містять ребро (a,c) , але не містять ребро (a,b) . Нижня межа вузла 6 становить 18,5. Вузол 7 відповідає тим маршрутам, що не містять ні ребро (a,c) , ні ребро (a,b) , звідси випливає висновок, що ці маршрути містять ребра (a,d) і (a,e) . Нижня межа для вузла 7 дорівнює 21, тому можемо вилучити його, оскільки уже відомий маршрут з такою вартістю.

Продовжується аналіз над синами вузла 6. Це вузли 10 і 11. Вузол 11 вилучається, оскільки його нижня межа перевершує вартість найкращого зі знайдених на даний момент маршрутів. Синами вузла 10 є вузли 14 і 15, що відповідають маршрутам, що включають і вилучають ребро (b,c) . Проте, у відповідності до правила 1 і 2 прийняття ребер робити висновок, що вузли 14 і 15 є окремими маршрутами. Один з них (a,c,b,e,d,a) має найменшу серед усіх маршрутів вартість – 19, а це означає, що він і є самим оптимальним маршрутом комівояжера.

Експериментальні дані дозволяють стверджувати, що метод розгалужень і меж для рішення задачі комівояжера є розумною альтернативою повному перебору, а його часова складність дорівнює $\Theta(1.26^n)$.

2.5.3. Динамічне програмування

Література: [3,83-85;4,280-288;13,299-326;17,461-465]

Ключові поняття: *динамічне програмування, принцип оптимальності Р.Белмана, табличний підхід.*

Рекурсивна техніка, яка була використана при розгляді метода декомпозиції, корисна, якщо задачу можна розбити на

підзадачі за розумний час, а сумарна кількість підзадач буде незначною. Якщо, наприклад, сума розмірностей підзадач дорівнює an для деякої константи $a > 1$, то рекурсивний алгоритм матиме поліноміальну часову складність. За умови, коли розподіл задачі розмірністю n зводить до появи n задач розмірністю $n-1$, то рекурсивний алгоритм, ймовірніше, матиме експоненціальну складність. У такому випадку часто можна одержати більш ефективні алгоритми за допомогою табличної техніки, яку часто називають *динамічним програмуванням*.

Слід зазначити, що *динамічне програмування* не обов'язково пов'язане з процесом написання програм. Сам термін „*динамічне програмування*” походить від теорії керування і означає процес покрокового рішення задач, коли на кожному кроці вибирається одне рішення з множини допустимих (на цьому кроці) рішень, причому таке, що оптимізує задану цільову функцію або функцію критерію. В основі теорії динамічного програмування лежить *принцип оптимальності Р.Белмана*.

Подібно методу декомпозиції, динамічне програмування вирішує задачу, розбиваючи її на підзадачі і поєднуючи їх рішення. Проте, існує важлива відмінність. Метод декомпозиції поділяє задачу на незалежні підзадачі, в свою чергу підзадачі - на більш дрібні підзадачі і так далі, а потім збирає рішення основної задачі «знизу вгору». Динамічне програмування ж застосовують тоді, коли підзадачі мають спільні „*підпідзадачі*”. За такої умови, рекурсивний алгоритм виконує зайві обчислення, вирішуючи ті самі підпідзадачі по декілька разів. Алгоритм, що заснований на динамічному програмуванні, розв'язує кожну з підзадач один раз і запам'ятовує відповіді в спеціальній таблиці. Такий підхід дозволяє не обчислювати заново відповідь до підзадач, які вже зустрічалися.

Застосуємо метод динамічного програмування до вирішення задачі комівояжера. Зазначимо, що алгоритм знаходження мінімального маршруту комівояжера був запропонований *Л.Фордом* (1956) і *Р.Белманом* (1958). Розробка власного алгоритму вирішення задачі комівояжера і його програмна реалізація методом динамічного програмування

була виконана О.Г. Тюріним в процесі підготовки даного видання. Саме цим алгоритмом ми з вдячністю і скористаємося.

Спочатку подамо у табличній формі – таблиця X - значення всіх фрагментів можливих маршрутів, тобто ребер графу з рис. 2.11.

Таблиця 2.14.

Таблиця X. Значення ребер графу з рис. 2.11.

	A	B	C	D	E
A	0	3	4	2	7
B	3	0	4	6	3
C	4	4	0	5	8
D	2	6	5	0	6
E	7	3	8	6	0

Для визначення найкоротшого шляху комівояжера, що розпочинається з вершини A і цією вершиною закінчується, будемо будувати варіанти його переміщення послідовно, вершина за вершиною. Для цього нам знадобиться ще дві таблиці. Перша таблиця - таблиця Y (див. Таблиця

2.15.) - буде містити значення мінімальних маршрутів з вершини A у вершину n, що складаються з мінімальних маршрутів з вершини A у вершину n-1 та маршрутів з вершинами n-1 у вершину n. Таким чином, обчислення мінімального маршруту A..n базується на раніше обчисленому значенні маршруту A..n-1, що зберігається в таблиці. Наприклад, комірка Y[B,3] (перехрестя рядку B і стовпчика 3) таблиці Y містить мінімальне значення маршруту з вершини A у вершину B, що проходить через 3 вершини, наприклад, ACB, ADB чи AEB, і обчислюється наступним чином:

$$Y[B,3] = \min \begin{cases} A-C+c-b=Y[C,2]+X[c,b]=4+4=8 \\ A-D+d-b=Y[D,2]+X[d,b]=2+6=8 \\ A-E+e-b=Y[E,2]+X[e,b]=7+3=10 \end{cases}$$

Наведена формула означає, що у вершину B можна пройти маршрутами ACB , ADB чи AEB , мінімальна вартість яких становить 8, 8 і 10 відповідно.

Таблиця 2.15.

Таблиця Y . Значення мінімальних маршрутів

	1	2	3	4	5
A	0	0	0	0	0
B	0	3	8	15	∞
C	0	4	7	14	∞
D	0	2	9	12	17
E	0	7	6	11	18

Зрозуміло, що маршрут з вершини A у вершину B , який складається з 3 вершин і проходить через вершину B , неможливий, відповідно його значення не враховується. Це дає право стверджувати, що у вершину B можна пройти двома мінімальними маршрутами, що містять 3 вершини (два ребра), і це маршрути ACB і ADB . Зазначимо, що у відповідності до

принципу оптимальності Р.Белмана доцільніше скористатися саме маршрутом ACB , вибір якого базується на оптимальнішому значенні ребра $(c-b)$ у порівнянні з ребром $(d-b)$.

Зауважимо, що нулі в рядку A таблиці Y означаються неможливість прокладання маршруту з вершину A у вершину A через 1,2,3,... n вершин, адже у такому випадку вершина A мала б 3 ребра, що виходять з неї (з врахуванням необхідно повернення у вершину A). Нулі ж стовпчика 1 вказують на те, що пройти з вершину A у вершину B, C, D, \dots, n маршрутом, який би містив лише одну вершину, також неможливо. Знак „ ∞ ” в комірках $Y[B,5]$ і $Y[C,5]$ означає, що вершини B і C , якими повинні скінчитися маршрути з 5 вершин, вже були задіяні у маршрутах, а їх повторне використання призведе до наявності у вершин B і C третього ребра. Отже, такі маршрути також неможливі.

Третя таблиця - таблиця Z - містить безпосередньо маршрут, поданий послідовністю чисел, що відповідають номеру вершини.

Наприклад, рядок E складається з елементів, що утворюють послідовність 1,3,4,2,5.

Ця послідовність - чисельне позначення маршруту $ACDBE$. Зауважимо, що всі маршрути починаються з вершини A , отже стовпчик 1 таблиці Z міститиме одиниці.

Значення, записані у другому стовпчику, відповідають значенню самої вершини, на яку повинен закінчитися маршрут.

Дійсно, маршрут з вершини A у вершину B буде закінчений у вершині B , чисельне значення якої дорівнює 2. Слід зазначити, що всі маршрути, наведені у таблиці Z , не включають фрагмент, що вказує повернення у вершину A . Це несуттєво, адже перехід з передостанньої вершини $n-1$ у вершину A однозначно визначений ребром $((n-1)-a)$.

Прослідкуємо визначення послідовності з 3 вершин для маршруту, що повинен закінчитися у вершині B . Зрозуміло, що ми опрацьовуємо рядок B . Послідовність для маршруту з 3 вершин вказує на те, що ми повинні до послідовності, що визначає мінімальний маршрут для 3 вершин, дописати значення вершини B , тобто 2. Мінімальний маршрутом для маршруту, що включає 3 вершини і закінчується у вершині B , є сума з маршруту, який включає 2 вершини і закінчується у вершині C , та ребра $(c-b)$. А це означає, що необхідно до послідовності 1-3, що визначає маршрут $A-C$, дописати значення вершини 2. Нову послідовність 1-3-2 маршруту $A-C-B$ зберігаємо до тих пір, доки не визначенні всі значення

Таблиця 2.16.

Таблиця Z . Чисельне позначення маршрутів

	1	2	3	4	5
A	0	0	0	0	0
B	1	3	4	2	0
C	1	2	5	3	0
D	1	3	2	5	4
E	1	3	4	2	5

мінімальних маршрутів з 3 вершин, і потім записуємо у рядок B таблиці Z .

Прослідкуємо за роботою алгоритму вирішення задачі комівояжера для графа, що поданий на рис.2.11.

- (1) *for* $i \leftarrow 1$ *step* 1 *until* n *do*
- (2) *for* $j \leftarrow 1$ *step* 1 *until* n *do*
begin
- (3) *if* $j=2$ *then*
- (4) $Y[i,j] \leftarrow X[l,i]$
- (5) *else* $Y[i,j] \leftarrow 0$
- (6) *if* $j=1$ *then*
- (7) $Z[i,j] \leftarrow 1$
- (8) *else if* $j=2$ *then*
- (9) $Z[i,j] \leftarrow i$
- (10) *else* $Z[i,j] \leftarrow 0$
- (11) $TEMP[i,j] \leftarrow 0$
end
- (12) *for* $j \leftarrow 3$ *step* 1 *until* n *do*
begin
- (13) *for* $i \leftarrow 2$ *step* 1 *until* n *do*
begin
- (14) $Y[i,j] \leftarrow \infty$
- (15) *for* $k \leftarrow 2$ *step* 1 *until* n *do*
begin
- (16) $y \leftarrow 1$
- (17) *for* $l \leftarrow 1$ *step* 1 *until* $j-1$ *do*
- (18) *if* $Z[k,l]=i$ *then*
- (19) $y \leftarrow 0$
- (20) *if* $y=1$ *then*
begin
- (21) *if* $Y[k,j-1]+X[k,i] < Y[i,j]$ *then*
begin
- (22) $Y[i,j] \leftarrow Y[k,j-1]+X[k,i]$
- (23) *for* $l \leftarrow 1$ *step* 1 *until* $j-1$ *do*
- (24) $TEMP[i,l] \leftarrow Z[k,l]$
- (25) $TEMP[i,j] \leftarrow i$

```
(26)      s ← X[k,i]
           end
(27)      else if (Y[k,j-1]+X[k,i]= Y[i,j]) and (X[k,i]<s) then
           begin
(28)          Y[i,j] ← Y [k,j-1]+X[k,i]
(29)          for l←1 step 1 until j-1 do
(30)              TEMP[i,l] ← Z [k,l]
(31)              TEMP[i,j] ← i
(32)          s ← X[k,i]
           end
           end
           end
(33) for i←2 step 1 until n do
(34)     for l←1 step 1 until j do
(35)         Z[i,l] ← TEMP[i,l]
           end
(36) for i←1 step 1 until n do
           begin
(37)     for j←1 step 1 until n do
           begin
(38)         write(X[i,j])
           end
(39)     writeln
           end
(40) for i←1 step 1 until n do
           begin
(41)     for j←1 step 1 until n do
           begin
(42)         write(Y[i,j])
           end
(43)     writeln
           end
(44) for i←1 step 1 until n do
           begin
(45)     for j←1 step 1 until n do
```

```
begin
(46) write(Z[i,j])
end
(47) writeln
end
```

У рядках (1)-(11) відбувається встановлення початкових значень для елементів таблиць Y і Z . Враховуючи те, що мінімальний маршрут з 2 вершин відповідає ребру цих вершин, то до стовпчика 2 рядків $B-E$ таблиці Y записуємо значення ребер $(a-b)-(a-e)$ відповідно. До стовпчика 1 рядків $B-E$ таблиці Z записуємо 1, а до стовпчика 2 – чисельні значення вершин, в яких повинні закінчитися відповідні маршрути. Таблиця $TEMP$, що зберігатиме тимчасові значення послідовностей мінімальних маршрутів, заповнюється 0.

Основна робота алгоритму відбувається в рядках (12)-(35). Оператори повторення, що записані у рядках (12) і (13), дозволяють організувати опрацювання всіх стовпчиків і рядків таблиць X і Y відповідно. Оператор повторення, що записаний у рядках (15) необхідний для використання попередніх значень з таблиць X і Y . Комірку $Y[i,j]$, яка використовується для зберігання мінімального значення маршруту, що складається з j вершин і закінчується у вершині i , на початку роботи оператору повторення рядку (13) заповнюється максимально можливим значенням. Саме з цим значенням на першому етапі порівнюється маршрут, що складається з мінімального маршруту з $j-1$ вершинами ($Y[k,j-1]$) і ребром з вершини k у вершину i ($X[k,i]$). Зазначимо, що значення змінної u і відповідні операції над нею в рядках (16)-(19) дозволяють уникнути перевірки маршруту з вершини, наприклад, B у вершину B . Оператор умови, що записаний у рядку (21) встановлює у поточну комірку $Y[i,j]$ значення мінімального з маршрутів $Y[k,j-1]+X[k,i]$, де k змінюється від 2 до n . Оператор повторення рядку (23) дозволяє запам'ятати у таблицю $TEMP$ значення попереднього оптимального маршруту, що складається з $j-1$ вершин і закінчується у вершині i . Рядок (25) доповнює цей оптимальний маршрут вершиною i , в якій і повинен цей маршрут закінчуватися. Зміна s у рядку (26) отримує значення

ребра $X[k,i]$ і використовується в якості додаткової умови у рядку (27). Саме вона дозволяє обрати оптимальний маршрут, виходячи із значення поточного ребра $X[k,i]$, у випадку коли є декілька попередніх маршрутів з однаковими значеннями. Рядки (27)-(32) дозволяють зробити оптимальний вибір. Наприклад, визначення оптимального маршруту, що проходить через 3 вершини і закінчується у вершині B , відбувається на основі пошуку мінімального значення маршрутів, що проходять через 2 вершини і закінчується у вершинах C, D чи E з врахуванням ребер $(c-b)$, $(d-b)$ і $(e-b)$ відповідно.

За умови двох однакових значень мінімальних маршрутів саме умова рядку (27) і значення змінної s дозволяють обрати оптимальний маршрут.

Оператори повторення рядків (33) і (34) після аналізу всіх попередніх маршрутів записують послідовності оптимальних маршрутів з таблиці TEMP у таблицю Y.

Оператори повторення рядків (36) і (47) дозволяють подати у табличній формі значення ребер між всіма вершинами (таблиця X), мінімальних маршрутів (таблиця Y) і самі послідовності, що відповідають цим маршрутам (таблиця Z).

В результаті роботи алгоритму одержуємо значення двох мінімальних маршрутів $A-C-B-E-D(1-3-2-5-4)$ і $A-C-D-B-E(1-3-4-2-4)$, значення яких 17 і 18 відповідно. Враховуючи те, що повний маршрут повинен закінчуватися вершиною A , до значення маршрутів $A-C-B-E-D$ і $A-C-D-B-E$ додаємо значення ребер $(d-a)$ і $(e-a)$. Одержуємо значення 19 ($17+2=19$) і 25 ($18+7=25$) відповідно. Отже, робимо висновок, що маршрут $A-C-B-E-D-A$ із значенням 19 є оптимальним маршрутом комівояжера.

Враховуючи те, що основна робота алгоритму зосереджена в операторах повторення (рядки (12), (13) і (15)), часова складність алгоритму буде $\Theta(n^3)$. Незважаючи на показник ступеня, функція часової складності алгоритму, що побудований на основі методу динамічного програмування, росте набагато повільніше, ніж функція часової складності алгоритму повного перебору всіх варіантів. І якщо при $n \leq 5$ часова складність алгоритму динамічного програмування вища за складність алгоритму повного перебору

($n!=5!=120 < n^3=5^3=125$), то вже при $n=6$ ситуація змінюється ($n!=6!=720 > n^3=6^3=216$), а за умови наявності 50 вершин ця перевага ще значніша ($n!=50!=3,04 \cdot 10^{64} \gg n^3=50^3=125000$).

2.5.4. Евристичні алгоритми

Література: [4,288-291; 13,326-346]

Ключові поняття: *евристичний алгоритм, „жадібна” стратегія, найкращий вибір, критерії прийняття рішення.*

Для рішення багатьох оптимізаційних задач є більш прості і швидкі евристичні алгоритми, ніж динамічне програмування. Зокрема, це так звані „жадібні” алгоритми. „Жадібний” алгоритм на кожному кроці робить локально оптимальний вибір з надією, що остаточне рішення також виявиться оптимальним. Таким чином, принцип „жадібного” вибору застосовується для оптимізаційних задач у тому випадку, якщо послідовність локально оптимальних (жадібних) виборів дає глобально оптимальне рішення. Різницю між жадібними алгоритмами і динамічним програмуванням можна пояснити так: на кожному кроці жадібний алгоритм бере самий „жирний” шматок, а потім уже намагається зробити *найкращий вибір* серед тих, що залишилися; алгоритм динамічного програмування навпаки приймає рішення тільки після того, як прорахуються задалегідь усі варіанти.

Пояснимо принцип „жадібного” метода побудови алгоритмів наступним прикладом. Нехай у нас є монети номіналом в 25, 10 і 1 копійки. Нам потрібно здійснити покупку вартістю 88 копійок, при цьому скориставшись мінімальною кількістю монет. Оптимальним вибором буде три монети номіналом 25 копійок, одна монета в 10 копійок і три монети номіналом в 1 копійку. Наш вибір можна пояснити наступним описом алгоритму. Спочатку ми спробували скористатися монетою найбільшого номіналу (25 копійок), яка є, звичайно, менша за необхідну суму. Потім ми обчислили різницю ($88-25=63$) і дійшли до висновку про можливість використання ще двох монет номіналом в 25 копійок, тобто ($63-25=38$) і ($38-25=13$). Після третього вибору ми розуміємо, що скоритися

монетами в 25 копійок неможливо ($13 < 25$) і переходимо до використання монет меншим номіналом тощо. Описаний алгоритм побудований за „жадібним” методом. Слід зазначити, що „жадібні” алгоритми зазвичай працюють швидше, ніж алгоритми, що базуються на динамічному програмуванні. Проте „жадібні” методи не завжди дають оптимальне рішення. Зокрема, для вирішення аналогічної задачі з монетами для суми в 30 копійок „жадібним” методом потрібно шість монет (одна номіналом в 25 копійок і п’ять в 1 копійку). Проте оптимальним рішенням є використання трьох монет номіналом в 10 копійок. Незважаючи на це, „жадібні” методи дозволяють отримувати досить оптимальні рішення, що відрізняються від дійсно оптимальних лише на декілька відсотків.

Застосуємо „жадібну” стратегію для побудови алгоритму вирішення задачі комівояжера. Як і випадку розгляду попередніх методів використовуємо рис. 2.11. „Жадібний” алгоритм розглядає спочатку найкоротші ребра. *Критерієм прийняття* ребра є виконання двох умов:

а) вибір не призводить до появи вершини зі ступенем три чи більше, тобто кількість ребер, що виходять з цієї вершини не повинна перевищувати два ребра;

б) вибір не призводить до утворення замкнутого контуру (циклу) з ребрами, що були прийняті раніше.

Сукупність ребер, що були обрані у відповідності до вказаного критерію, утворює сукупність не з’єднаних шляхів. Таке положення зберігається до виконання останнього кроку, коли останнє ребро приєднується до першої вершини, утворюючи маршрут.

Прослідкуємо процес утворення маршруту за „жадібним” алгоритмом. На першому етапі обирається ребро (a,d) , довжина якого дорівнює 2. Далі обираємо два ребра (a,b) і (b,e) , довжина яких дорівнює 3 і також приймаємо їх. На наступному етапі розглядаємо ребра (a,c) і (b,c) , довжина яких дорівнює 4. Проте, відмовляємося від них, так як їх прийняття призводить до підвищення ступенів вершин a і b до трьох, що суперечить критерію прийняття ребер. Ребро (c,d) , довжиною 5 приймаємо на наступному кроці. Два ребра (b,d) і (d,e) довжиною 6 відхиляємо відповідно до першої умови критерію прийняття.

Ребро (a,c) , довжиною 7 відхиляємо у відповідності до другої умови критерію, адже його прийняття призведе до утворення замкнутого контуру (циклу) з ребрами (a,d) і (c,d) . Ребро (c,e) , довжиною 8 приймаємо на останньому етапі. Таким чином, з використанням „жадібного” алгоритму був одержаний маршрут $(a \rightarrow b \rightarrow e \rightarrow c \rightarrow d \rightarrow a)$, вартість якого дорівнює 21. Як бачимо „жадібна” стратегія дозволила отримати маршрут, що відрізняється всього на 9 відсотків від оптимального.

Ми розглянули основні методи, що використовуються при проектуванні ефективних алгоритмів. Зрозуміло, ці методи не є універсальними і абсолютними. Проте, саме їх потужна техніка дозволяє проектувати елегантні і природні алгоритми. Адже вони спонукають до вирішення таких завдань як оптимальне подання даних і знаходження розумного порядку виконання операцій. Проте, як слушно зазначають А. Ахо, Дж. Хопкрофт і Дж. Ульман [3, с.86], найважливішим принципом майстерного розробника алгоритмів повинне бути почуття незадоволеності. Розробнику необхідно вивчати задачу з різних точок зору, поки він не переконається, що одержав алгоритм, який найбільш підходить для досягнення його мети.

ПРАКТИЧНИЙ КУРС

МАТЕМАТИЧНА ЛОГІКА

1.1. Скласти таблиці істинності для наступних формул і визначити, які з них будуть здійсненні, тотожно істинні чи тотожно хибні:

- а) $(P \rightarrow Q) \rightarrow ((P \rightarrow \bar{Q}) \rightarrow \bar{P})$
- б) $((P \rightarrow Q) \rightarrow P) \rightarrow Q$
- в) $(P \wedge (Q \vee \bar{P})) \wedge ((\bar{Q} \rightarrow P) \vee Q)$
- г) $((P \wedge \bar{Q}) \rightarrow Q) \rightarrow (P \rightarrow Q)$
- д) $((P \rightarrow Q) \rightarrow Q) \rightarrow Q$

Приклад. а) Складемо таблицю для наступної формули $(P \rightarrow Q) \rightarrow ((P \rightarrow \bar{Q}) \rightarrow \bar{P})$. Спочатку спростимо її за допомогою рівносильностей алгебри висловлювань:

$$\begin{aligned} (P \rightarrow Q) \rightarrow ((P \rightarrow \bar{Q}) \rightarrow \bar{P}) &\equiv (\bar{P} \vee Q) \rightarrow ((\bar{P} \vee \bar{Q}) \rightarrow \bar{P}) \equiv \\ &\equiv (\bar{P} \vee Q) \rightarrow ((\bar{P} \vee \bar{Q}) \vee \bar{P}) \equiv (\bar{P} \vee Q) \rightarrow ((\bar{P} \wedge \bar{Q}) \vee \bar{P}) \equiv \\ &\equiv (\bar{P} \vee Q) \rightarrow (\bar{P} \vee Q) \equiv \overline{\bar{P} \vee Q} \vee \bar{P} \vee Q \equiv P \wedge \bar{Q} \vee \bar{P} \vee Q \equiv \\ &\equiv \bar{Q} \vee \bar{P} \vee Q \equiv \bar{P} \vee 1 \equiv 1. \end{aligned}$$

Тобто, отримуємо тотожно істинну формулу.

Перевіримо результат за допомогою таблиці істинності.

P	Q	\bar{P}	\bar{Q}	$P \rightarrow Q$	$(P \rightarrow \bar{Q})$	$(P \rightarrow \bar{Q}) \rightarrow \bar{P}$	$(P \rightarrow Q) \rightarrow ((P \rightarrow \bar{Q}))$
0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	1
1	0	0	1	0	1	0	1
1	1	0	0	1	0	1	1

Практичний курс

1.2. Визначити значення змінних X, Y і Z, при яких формула стає здійсненою:

а) $((X \rightarrow (Y \wedge Z)) \rightarrow (\bar{Y} \rightarrow \bar{X})) \rightarrow \bar{Y}$

б) $((X \vee Y) \vee Z) \rightarrow ((X \vee Y) \wedge (X \vee Z))$

Приклад. а) Для цього побудуємо таблицю істинності для формули $((X \rightarrow (Y \wedge Z)) \rightarrow (\bar{Y} \rightarrow \bar{X})) \rightarrow \bar{Y}$

X	Y	Z	\bar{X}	\bar{Y}	$Y \wedge Z$	$X \rightarrow (Y \wedge Z)$	$\bar{Y} \rightarrow \bar{X}$	$((X \rightarrow (Y \wedge Z)) \rightarrow (\bar{Y} \rightarrow \bar{X}))$	$((X \rightarrow (Y \wedge Z)) \rightarrow (\bar{Y} \rightarrow \bar{X})) \rightarrow \bar{Y}$
0	0	0	1	1	0	1	1	1	1
0	0	1	1	1	0	1	1	1	1
0	1	0	1	0	0	1	1	1	0
0	1	1	1	0	1	1	1	1	0
1	0	0	0	1	0	0	0	1	1
1	0	1	0	1	0	0	0	1	1
1	1	0	0	0	0	0	1	1	0
1	1	1	0	0	1	1	1	1	0

Отже, отримуємо, що формула буде здійсненою при наступних значеннях X, Y і Z: (0,0,0), (0,0,1), (1,0,0), (1,0,1).

1.3. Використовуючи рівносильні перетворення, привести наступні формули до більш простіших:

а) $\overline{(P \vee Q)} \rightarrow ((P \vee Q) \rightarrow P)$

б) $\overline{(\bar{P} \wedge \bar{Q})} \vee ((P \rightarrow Q) \wedge P)$

в) $(P \rightarrow Q) \wedge (Q \rightarrow P) \wedge (P \vee Q)$

г) $(P \rightarrow Q) \wedge (Q \rightarrow \bar{P}) \wedge (R \rightarrow P)$

д) $(P \wedge R) \vee (P \wedge \bar{R}) \vee (Q \wedge R) \vee (\bar{P} \wedge Q \wedge R)$

е) $\overline{((P \rightarrow Q) \wedge (Q \rightarrow \bar{P}))}$

Приклад. а) Скориставшись рівносильними перетвореннями, отримуємо

$$\begin{aligned} \overline{(P \vee Q)} \rightarrow ((P \vee Q) \rightarrow P) &\equiv (P \vee \overline{Q}) \rightarrow ((\overline{P} \wedge \overline{Q}) \vee P) \equiv \\ &\equiv (\overline{P} \vee Q) \vee P \vee \overline{Q} \equiv (\overline{P} \vee P) \vee (Q \vee \overline{Q}) \equiv 1 \vee 1 \equiv 1. \end{aligned}$$

Отримуємо тотожно істинну формулу. Для перевірки правильності спростування даної формулу побудуємо таблицю істинності.

P	Q	\overline{P}	$\overline{P \vee Q}$	$\overline{(\overline{P \vee Q})}$	$(P \vee Q)$	$(P \vee Q) \rightarrow P$	$\overline{(\overline{P \vee Q})} \rightarrow ((P \vee Q) \rightarrow P)$
0	0	1	1	0	0	1	1
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	1	0	1	0	1	1	1

Отже, отримуємо тотожно істинну формулу і за таблицею істинності, тому що ця формула приймає значення 1 при всіх значеннях вхідних змінних P та Q .

1.4. Наступні формули виразити рівносильними їм формулами, що містять лише операції заперечення і кон'юнкції:

- а) $(X \vee Y) \rightarrow (\overline{X} \rightarrow Z)$
- б) $(\overline{X} \rightarrow Y) \vee (\overline{X} \rightarrow Y)$
- в) $((X \vee Y \vee Z) \rightarrow X) \vee Z$
- г) $((X \rightarrow Y) \rightarrow Z) \rightarrow \overline{X}$

Приклад. а) Щоб виразити дану формулу рівносильною, яка містить лише операції заперечення і кон'юнкції, скористаємось рівносильними перетвореннями:

$$\begin{aligned} (X \vee Y) \rightarrow (\overline{X} \rightarrow Z) &\equiv \overline{X \vee Y} \vee (\overline{X} \vee Z) \equiv \overline{X} \wedge \overline{Y} \vee X \vee Z \equiv \\ &\equiv X \vee \overline{Y} \vee Z \equiv \overline{\overline{X} \wedge Y \wedge \overline{Z}} \end{aligned}$$

Отже, отримуємо формулу, що містить тільки операції заперечення і кон'юнкції.

1.5. Наступні формули виразити рівносильними їм формулами, що містять лише операції заперечення і диз'юнкції:

а) $(X \rightarrow Y) \rightarrow (Y \wedge Z)$

б) $(\overline{X} \wedge \overline{Y}) \rightarrow (X \wedge Y)$

в) $((\overline{X} \wedge Y) \vee Z) \rightarrow (Z \wedge \overline{Y})$

г) $((X \rightarrow (Y \wedge Z)) \rightarrow (\overline{Y} \rightarrow \overline{X})) \rightarrow \overline{Y}$

Приклад. а) Щоб виразити дану формулу рівносильною, яка містить лише операції заперечення і диз'юнкції, скористаємось рівносильними перетвореннями. Отримаємо:

$$\begin{aligned} (X \rightarrow Y) \rightarrow (Y \wedge Z) &\equiv (\overline{X} \vee Y) \rightarrow (Y \wedge Z) \equiv \overline{(\overline{X} \vee Y)} \vee (Y \wedge Z) \equiv \\ &\equiv \overline{(\overline{X} \vee Y)} \vee (\overline{Y} \vee \overline{Z}) \end{aligned}$$

1.6. Наступні формули виразити рівносильними їм формулами, що містять лише операції заперечення, кон'юнкції і диз'юнкції:

а) $((X \rightarrow Y) \wedge (Y \rightarrow X)) \rightarrow (X \vee Y)$

б) $((X \rightarrow Y) \wedge (Y \rightarrow \overline{X})) \rightarrow (Z \rightarrow X)$

в) $((X \leftrightarrow Y) \wedge (\overline{X} \leftrightarrow \overline{Y})) \rightarrow ((X \vee Y) \wedge (\overline{X} \vee \overline{Y}))$

г) $((X \leftrightarrow \overline{Y}) \rightarrow Z) \rightarrow (X \leftrightarrow \overline{Z})$

Приклад. а) Для того щоб виразити дану формулу рівносильними їм формулами, що містять лише операції заперечення, кон'юнкції і диз'юнкції, необхідно, використовувати основні еквівалентності алгебри логіки.

$$\begin{aligned} ((X \rightarrow Y) \wedge (Y \rightarrow X)) \rightarrow (X \vee Y) &\equiv ((\overline{X} \vee Y) \wedge (\overline{Y} \vee X)) \rightarrow (X \vee Y) \equiv \\ &\equiv \overline{(\overline{X} \vee Y) \wedge (\overline{Y} \vee X)} \vee (X \vee Y) \equiv \\ &\equiv X \wedge \overline{Y} \vee Y \wedge \overline{X} \vee (X \vee Y) \equiv \\ &\equiv X \wedge \overline{Y} \vee Y \vee Y \wedge \overline{X} \vee X \equiv Y \vee X \vee X \vee Y \equiv X \vee Y. \end{aligned}$$

1.7. Рівносильними перетвореннями кожен з наступних формул привести до диз'юнктивної та кон'юнктивної нормальних форм:

а) $\overline{(X \vee Z)} \wedge (X \rightarrow Y)$

$$\text{б) } (X \leftrightarrow Y) \wedge (\overline{Z \rightarrow T})$$

$$\text{в) } (X \vee (Y \wedge \overline{Z})) \wedge (\overline{X \vee Z})$$

$$\text{г) } ((X \rightarrow Y) \rightarrow (Z \rightarrow \overline{X})) \rightarrow (\overline{Y} \rightarrow \overline{Z})$$

$$\text{д) } (X \rightarrow (Y \rightarrow Z)) \rightarrow ((X \rightarrow \overline{Z}) \rightarrow (X \rightarrow \overline{Y}))$$

Приклад. а) Щоб привести формулу до диз'юнктивної нормальної форми, необхідно представити її у вигляді диз'юнкції елементарних кон'юнкцій будь-якої кількості різних незалежних змінних. Отже,

$$\begin{aligned} (\overline{X \vee Z}) \wedge (X \rightarrow Y) &\equiv \overline{X} \wedge \overline{Z} \wedge (\overline{X} \vee Y) \equiv \overline{X} \wedge \overline{Z} \wedge \overline{X} \vee \overline{X} \wedge \overline{Z} \wedge Y \equiv \\ &\equiv (\overline{X} \wedge \overline{Z}) \vee (\overline{X} \wedge Y \wedge \overline{Z}) \end{aligned}$$

А щоб привести ту саму формулу до кон'юнктивної нормальної форми, треба її подати у кон'юнкції елементарних диз'юнкцій також будь-якої кількості різних незалежних змінних. Отримуємо:

$$(\overline{X \vee Z}) \wedge (X \rightarrow Y) \equiv (\overline{X} \wedge \overline{Z}) \wedge (\overline{X} \vee Y) \equiv (\overline{X}) \wedge (\overline{Z}) \wedge (\overline{X} \vee Y).$$

1.8. Рівносильними перетвореннями кожен з наступних формул привести до досконалої диз'юнктивної нормальної форми (ДДНФ):

$$\text{а) } \overline{(X \vee Z)} \wedge (X \rightarrow Y)$$

$$\text{б) } (X \leftrightarrow Y) \wedge (\overline{Z \rightarrow T})$$

$$\text{в) } (X \vee (Y \wedge \overline{Z})) \wedge (\overline{X \vee Z})$$

$$\text{г) } ((X \rightarrow Y) \rightarrow (Z \rightarrow \overline{X})) \rightarrow (\overline{Y} \rightarrow \overline{Z})$$

$$\text{д) } (X \rightarrow (Y \rightarrow Z)) \rightarrow ((X \rightarrow \overline{Z}) \rightarrow (X \rightarrow \overline{Y}))$$

$$\text{е) } (\overline{X} \wedge Y) \vee (Y \wedge Z \wedge T) \vee (\overline{X} \wedge Y \wedge Z \wedge T)$$

$$\text{ж) } ((X \rightarrow Y) \rightarrow \overline{Y}) \rightarrow (X \rightarrow (Y \wedge X))$$

$$\text{з) } ((X \wedge Y) \rightarrow \overline{X}) \wedge ((X \wedge Y) \rightarrow \overline{Y})$$

Приклад. а) Скориставшись рівносильними перетвореннями, отримемо:

$$\begin{aligned} (\overline{X \vee Z}) \wedge (X \rightarrow Y) &\equiv \overline{X} \wedge \overline{Z} \wedge (\overline{X} \vee Y) \equiv \overline{X} \wedge \overline{Z} \wedge \overline{X} \vee \overline{X} \wedge \overline{Z} \wedge Y \equiv \\ &\equiv (\overline{X} \wedge \overline{Z}) \vee (\overline{X} \wedge Y \wedge \overline{Z}) \equiv (\overline{X} \wedge \overline{Z} \wedge (Y \vee \overline{Y})) \vee (\overline{X} \wedge Y \wedge \overline{Z}) \equiv \\ &\equiv (\overline{X} \wedge \overline{Z} \wedge Y) \vee (\overline{X} \wedge \overline{Z} \wedge \overline{Y}) \vee (\overline{X} \wedge Y \wedge \overline{Z}) \equiv (\overline{X} \wedge Y \wedge \overline{Z}) \vee (\overline{X} \wedge \overline{Y} \wedge \overline{Z}). \end{aligned}$$

1.9. Рівносильними перетвореннями кожен з наступних формул привести до досконалої кон'юнктивної нормальної форми (ДКНФ):

- а) $\overline{(X \vee Z)} \wedge (X \rightarrow Y)$
- б) $(X \leftrightarrow Y) \wedge \overline{(Z \rightarrow T)}$
- в) $(X \vee (Y \wedge \overline{Z})) \wedge \overline{(X \vee Z)}$
- г) $((X \rightarrow Y) \rightarrow (Z \rightarrow \overline{X})) \rightarrow (\overline{Y} \rightarrow \overline{Z})$
- д) $(X \rightarrow (Y \rightarrow Z)) \rightarrow ((X \rightarrow \overline{Z}) \rightarrow (X \rightarrow \overline{Y}))$
- е) $(X \vee Y) \wedge (Y \vee Z) \wedge (Z \vee T)$
- ж) $X \wedge (Y \vee \overline{Z}) \wedge (X \vee Y \vee Z)$
- з) $\overline{(X \wedge (Y \vee Z))} \rightarrow ((X \wedge Y) \vee Z)$

Приклад. а) Скористаємось рівносильними перетвореннями та отримаємо:

$$\begin{aligned} (\overline{X \vee Z}) \wedge (X \rightarrow Y) &\equiv \overline{X} \wedge \overline{Z} \wedge (\overline{X} \vee Y) \equiv \\ &\equiv (\overline{X} \vee (Y \wedge \overline{Y})) \vee (\overline{Z} \wedge \overline{X}) \wedge (\overline{Z} \vee (X \wedge \overline{X})) \vee (Y \wedge \overline{Y}) \wedge (\overline{X} \vee Y \vee (Z \wedge \overline{Z})) \equiv \\ &\equiv \overline{(X \vee Y \vee Z)} \wedge \overline{(X \vee \overline{Y} \vee Z)} \wedge \overline{(X \vee Y \vee \overline{Z})} \wedge \overline{(X \vee \overline{Y} \vee \overline{Z})} \wedge \overline{(X \vee Y \vee \overline{Z})} \wedge \\ &\wedge \overline{(X \vee Y \vee \overline{Z})} \wedge \overline{(X \vee \overline{Y} \vee \overline{Z})} \wedge \overline{(X \vee \overline{Y} \vee \overline{Z})} \wedge \overline{(X \vee Y \vee \overline{Z})} \wedge \overline{(X \vee Y \vee \overline{Z})} \equiv \\ &\equiv \overline{(X \vee Y \vee Z)} \wedge \overline{(X \vee \overline{Y} \vee Z)} \wedge \overline{(X \vee Y \vee \overline{Z})} \wedge \overline{(X \vee \overline{Y} \vee \overline{Z})} \wedge \overline{(X \vee Y \vee \overline{Z})} \wedge \\ &\wedge \overline{(X \vee \overline{Y} \vee \overline{Z})}. \end{aligned}$$

1.10. Знайти за допомогою таблиць істинності досконалі диз'юнктивну і кон'юнктивну нормальні форми для наступних формул:

- а) $\overline{(X \wedge Y)} \rightarrow \overline{(X \vee Y)}$ з) $X \vee (Y \rightarrow (Z \leftrightarrow (X \wedge Y)))$
 б) $X \rightarrow Y$ и) $X \wedge \overline{(Y \wedge (Z \rightarrow (X \leftrightarrow Y)))}$
 в) $X \leftrightarrow Y$ к) $(X \wedge Y \wedge Z) \vee T$
 г) $(X \wedge Y) \vee Z$ л) $((X \wedge \bar{Y}) \vee Z) \wedge T$
 д) $(X \vee Y) \wedge Z$ м) $(X \wedge ((Y \wedge Z) \vee T)) \vee \bar{T}$
 е) $(X \leftrightarrow Z) \rightarrow (X \wedge \bar{Y})$ н) $(X \wedge Y) \vee (Y \wedge Z) \vee (Z \wedge T)$
 ж) $\overline{(\bar{X} \vee \bar{Z})} \wedge (X \rightarrow (Y \wedge Z))$

Приклад. а) побудуємо таблицю істинності

X	Y	$X \wedge Y$	$X \vee Y$	$\overline{(X \wedge Y)}$	$\overline{(X \vee Y)}$	$(X \wedge Y) \rightarrow \overline{(X \vee Y)}$
0	0	0	0	1	1	1
0	1	0	1	1	0	0
1	0	0	1	1	0	0
1	1	1	1	0	0	1

Для того щоб за таблицею істинності побудувати ДДНФ, вписуємо ті значення змінних, що дають конститuentу одиниці

X	Y	$F(X, Y)$	Одержуємо ДДНФ за допомогою з'єднання операцією диз'юнкції записаних конститuent одиниці: $(\bar{X} \wedge \bar{Y}) \vee (X \wedge Y).$
0	0	1	
1	1	1	

Для того щоб за тією ж таблицею істинності побудувати ДКНФ, вписуємо ті значення змінних, що дають конститuentу нуля.

X	Y	$F(X, Y)$	Одержуємо ДКНФ за допомогою з'єднання операцією кон'юнкції записаних конститuent нуля: $(X \vee \bar{Y}) \wedge (\bar{X} \vee Y).$
0	1	0	
1	0	0	

ТЕОРІЯ АЛГОРИТМІВ

Алгоритмічні моделі на основі обчислювальних функцій

2. Використовуючи частково-рекурсивні функції виконати наступні задачі:

2.1. Довести, що функція $f(x,y)=x+y$ задовольняє схему примітивної рекурсії.

2.2. Довести, що функція $f(x,y)=x*y$ задовольняє схему примітивної рекурсії.

2.3. Довести, що функція $f(x,y)=x^y$ задовольняє схему примітивної рекурсії.

2.4. Довести, що наступні функції є примітивно рекурсивними:

- $\left[\frac{x}{y} \right]$ - залишок від ділення x на y , де $[x/0]=x$;
- $\text{rest}(x,y)$ - залишок від ділення x на y , де $\text{rest}(x,0)=x$;
- $k(x,y)$ – найменше спільне ділене чисел x та y , де $k(x,0)=k(0,y)=0$;
- $d(x,y)$ – найбільший спільний дільник чисел x та y , де $d(0,0)=0$
- $\left[\sqrt{x} \right]$

2.5. Довести, що функція, яка перелічує числа Фібоначчі

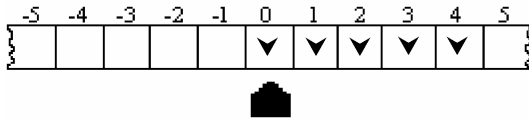
$$\begin{cases} f(0) = 0, & f(1) = 1, \\ f(n+2) = f(n) + f(n+1) \end{cases}$$

є примітивно рекурсивною.

Алгоритмічні моделі на основі детермінованих пристроїв

3. Написати програму машини Поста для рішення наступної задачі:

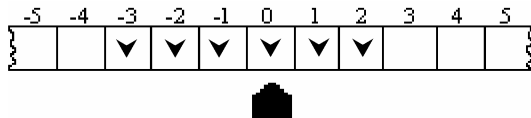
3.1. Яким би не було натуральне число N , якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться проти самої лівої секції запису, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число $N+1$. Каретка після виконання програми може знаходитися в будь-якому положенні.



Приклад. 3.1.

1 → 2	переходимо праворуч, не перевіряючи зміст секції, бо машина знаходиться проти заповненої самої лівої секції
2 ? 3,1	перевіряємо зміст секції. Якщо вона порожня переходимо до команди 3, якщо наповнена, повертаємось до команди 1.
3 ▼ 4	ставимо мітку в порожній секції та переходимо до команди 4.
4 !	зупиняємось.

3.2. Яким би не було натуральне число N , якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться проти однієї секції запису, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число $N+1$. Каретка після виконання програми може знаходитися в будь-якому положенні.



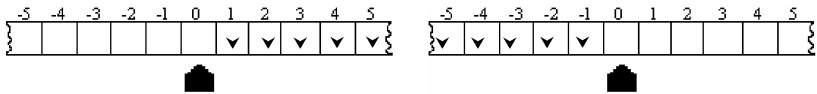
- Знайдіть додаткові 2 рішення задачі 3.2, програма яких складається з 4 команд. Перевірте, щоб програма містила

Практичний курс

команди зміщення вліво, встановлення мітки, передачі керування та зупинки.

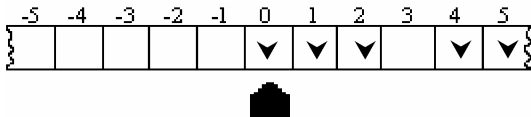
- Довести, що існує ще 11 рішень задачі 3.2., що складаються з 4 команд та містять команду зміщення вліво.

3.3. Яким би не було натуральне число N , якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться зліва / справа від запису, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число $N+1$. Каретка після виконання програми може знаходитися в будь-якому положенні.

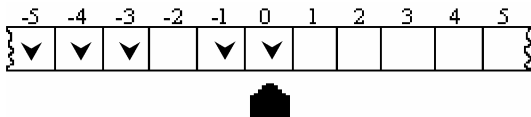


3.4. Яким би не було натуральне число N , якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться в довільному місці стрічки, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число $N+1$. Каретка після виконання програми може знаходитися в будь-якому положенні. Запропонуйте декілька варіантів рішення задачі.

3.5. Додати два числа, що записані на відстані в 1 комірку одне від одного. Каретка знаходиться проти: самої лівої комірки лівого числа

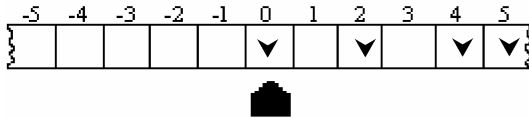


самої правої комірки правого числа.



3.6. Додати два числа, що записані на довільній відстані одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.

3.7. Додати довільну кількість чисел, що записані на відстані в 1 комірку одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.



3.8. Знайти найбільше з двох натуральних чисел N і M , що записані на відстані в одну пугу комірку одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.

3.9. Перемножити два числа, що записані на довільній відстані одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.

3.10. Виконати ділення числа, що записане на стрічці на задане число K .

Примітка: у випадку “нульового” ділення програма не повинна призводити до результативної зупинки.

4. Написати програму машини Тьюрінга для рішення наступної задачі:

4.1. Збільшити число p , записане у двійковій системі числення, на 1.

Приклад. 4.1. Вхідним алфавітом буде $A = \{0, 1, \lambda\}$. Отже,

A \ Q	q ₁	q ₂	q ₃
0	q ₁ 0R	q ₃ 1L	q ₃ 0L
1	q ₁ 1R	q ₂ 0L	q ₃ 1L
λ	q ₂ λ L	q ₃ 1L	q ₂ λ R

Практичний курс

У стані q_1 каретка знаходиться проти лівої цифри числа p та зсувається праворуч за командами q_10q_10R та q_11q_11R до тих пір, доки не зчитає пустий символ λ . Команда $q_2\lambda L$ переведе стан машини на молодший розряд числа p . У стані q_2 відбувається перехід ліворуч, якщо у черговому розряді числа міститься одиниця. Це забезпечується командами q_21q_20L . Якщо ж міститься нуль, то команда q_20q_31L додає одиницю та забезпечує перехід до стану q_3 . Команда $q_2\lambda q_31L$ ставить на місце першої прогалини зліва від числа, якщо у результаті додавання одиниці виникає перенесення з старшого розряду вихідного числа p . У стані q_3 повертає машину на перший символ зліва результату – числа $p+1$.

4.2. Додати два числа, що записані у вигляді послідовності одиниць між якими стоїть знак “+”. Каретка знаходиться проти самої лівої комірки лівого числа.

λ	1	1	+	1	λ
-----------	---	---	---	---	-----------

q_1

4.3. Знайти найбільше з двох натуральних чисел N і M , що записані у вигляді послідовностей одиниць між якими стоїть знак “=”.

4.4. Яким би не було натуральне число N , подане в десятковій системі числення, на стрічці записати $N+1$. Каретка встановлена проти розряду одиниць (в самій правій позиції числа).

λ	5	8	λ	λ	λ
-----------	---	---	-----------	-----------	-----------

q_1

4.5. Скопіювати натуральне число N , що записані у вигляді послідовності одиниць зі знаком “*” наприкінці.

λ	1	1	1	*	λ
-----------	---	---	---	---	-----------

q_1

4.6. Перемножити два натуральні числа, що записані у вигляді послідовності одиниць та розподілені знаком “*”.

λ	λ	1	1	1	*	1	1	λ	λ	λ	λ	λ	λ
-----------	-----------	---	---	---	---	---	---	-----------	-----------	-----------	-----------	-----------	-----------

q_1

4.7. Додати два натуральні числа, що подані в десятковій системі числення.

λ	5	8	+	3	λ
-----------	---	---	---	---	-----------

q_1

4.8. Відняти два натуральні числа, що подані в десятковій системі числення.

λ	5	8	-	3	λ
-----------	---	---	---	---	-----------

q_1

4.9. Нескінчено збільшувати на одиницю натуральне число N , що подане в десятковій системі числення.

4.10. Знайти найбільший спільний дільник двох натуральних чисел, що подані в десятковій системі числення.

4.11. Виконати ділення числа N , що записане у вигляді послідовності одиниць, на задане число K (послідовність *).

5. Скласти схеми орієнтовних підстановок для нормальних алгоритмів Маркова для рішення наступної задачі:

5.1. Перетворити будь-яке слово в алфавіті $A=(a_0, a_1, a_2, \dots, a_n)$ в порожнє слово.

Приклад. 5.1. За умовами задачі є алфавіт $A = (a_0, a_1, a_2, \dots, a_n)$. Нехай у загальному вигляді слово P цього алфавіту має вигляд $a_0a_1a_2\dots a_n$, де $a_0, a_1, a_2, \dots, a_n$ – букви з алфавіту A . Якщо усі букви цього алфавіту замінити їхніми номерами отримаємо новий алфавіт $B = (0, 1, 2, \dots, n)$. Будь-яке слово, складене з елементів утвореного алфавіту B буде порожнім, бо не буде містити букв.

5.2. Перетворити будь-яке парне число в алфавіті $A=(1)$ в порожнє слово, а непарне – в слово 1.

5.3. Перетворити довільне слово в алфавіті $A=(1)$ в 1, якщо слово ділиться на 3, в іншому випадку – записати порожнє слово.

5.4. Збільшити довільне число в алфавіті $A=(0, 1, 2, \dots, 9)$ на 1.

5.5. Зменшити довільне число в алфавіті $A=(0, 1, 2, \dots, 9)$ на 1.

Складність алгоритмів

6.1. Розташуйте наступні функції в порядку зростання швидкості росту: а) n , б) \sqrt{n} , в) $\log n$, г) $\log \log n$, д) $\log^2 n$, е) n^2 , ж) $(1/3)^n$, з) $(3/2)^n$.

Приклад. 6.1. Порядку зростання швидкості росту функції залежить від розмірності n даних задачі. Отже, порядок буде наступним

ж) $(1/3)^n$ – степінь числа, меншого одиниці, буде найменшою

г) $\log \log n$ - подвійний логарифм більше, ніж степінь малого числа

б) \sqrt{n} - корінь з числа n менший за саме число

в) $\log n$ – логарифм числа більший, ніж його корінь

д) $\log^2 n$ - квадрат логарифма більший за сам логарифм

а) n - саме число більше, ніж його корінь, логарифм

з) $(3/2)^n$ – ступінь числа, більшого за одиницю більше самого числа

е) n^2 – квадрат числа більший, ніж саме число, його кореня чи логарифма.

6.2. Побудуйте таблицю залежності максимально допустимих розмірностей задачі для розв'язання за одну добу для алгоритмів з часовими складностями n , $\log n$, n^2 , 2^n .

6.3. Запишіть вираз $n^3/1000-100n^2-10n+1$ використовуючи Θ -символіку.

6.4. При якому найменшому значенні n алгоритм, що виконує $10n^2$ операцій, ефективніше алгоритму, що виконує 2^n операцій?

6.5. Використовуючи Θ -символіку, визначте час (в найгіршому випадку) виконання наступної підпрограми як функції від n .

```
(1) procedure A
(2)   for  $i \leftarrow 1$  step 1 until  $n$  do
(3)     for  $j \leftarrow 1$  step 1 until  $n$  do
           begin
(4)        $C[i,j] \leftarrow 0$ 
(5)     for  $k \leftarrow 1$  step 1 until  $n$  do
            $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$ 
(6)     end
```


МОДУЛЬНО-ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ

МОДУЛЬ №1. АЛГЕБРА ЛОГІКИ

Визначте варіанти правильних відповідей на наступні запитання

1. Складне висловлення, яке істинне тоді і тільки тоді, коли одне з простих висловлень є істинним, а інше – обов’язково хибним, називається:
 - a) *диз’юнкцією*
 - b) *кон’юнкцією*
 - c) *з’єднувальною диз’юнкцією*
 - d) *антикон’юнкцією*
 - e) *правильна відповідь відсутня*
2. Підстава, наявність якої обов’язково спричиняє певний наслідок, називається:
 - a) *достатньою*
 - b) *необхідною*
 - c) *правильна відповідь відсутня*
3. Формула, що задана кон’юнкцією різних повних елементарних диз’юнкцій будь-якої кількості різних незалежних змінних (при цьому рівність диз’юнкції розуміється з точністю до порядку змінних), називається:
 - a) *диз’юнктивною нормальною формою*
 - b) *кон’юнктивною нормальною формою*
 - c) *досконалою диз’юнктивною нормальною формою*
 - d) *досконалою кон’юнктивною нормальною формою*
 - e) *правильна відповідь відсутня*
4. Логічний оператор, який описує відношення між суб’єктом та предикатом і несе кількісну характеристику виразу, називається:
 - a) *функтором*
 - b) *квантором*
 - c) *правильна відповідь відсутня*
5. Характеристиками будь-якої формальної аксіоматичної теорії є :
 - a) *множина висловлень і предикатів*

- b) правила виведення нових аксіом даної теорії
 c) алфавіт для побудови формул теорії
 d) система аксіом
 e) всі вищевказані варіанти
6. Нехай P означає: “число a ділиться на число b ”, Q означає: “число a ділиться на число c ”, а R - “число a ділиться на добуток b і c ”. Встановіть відповідність між наведеними висловленнями і їх формулами:
- a) a ділиться на b , і a ділиться на c a) $\overline{P \wedge Q}$
 b) a ділиться на b , але не ділиться на c b) $P \wedge Q \rightarrow R$
 c) a не ділиться на b , і a не ділиться на c c) $P \wedge \overline{Q}$
 d) неправда, що a ділиться на b і c d) $P \vee Q \rightarrow R$
 e) якщо a ділиться на b і на c , то a ділиться на добуток чисел b і c e) $\overline{P} \wedge \overline{Q}$
 f) якщо a ділиться на b або на c , то a ділиться на добуток чисел b і c f) $P \vee Q \rightarrow \overline{R}$
 g) якщо a ділиться на b або на c , то a не ділиться на добуток чисел b і c g) $P \wedge Q$
7. Визначте формулу, що відповідає висловленню “якщо сума двох чисел, і перше з доданків діляться на деяке число d , то і другий доданок ділиться на це ж число”
- a) $D(a,b) \vee D(b,a) \rightarrow E(a,b)$
 b) $S(a,b,c) \wedge D(a,d) \wedge D(c,d) \rightarrow D(b,d)$
 c) $S(a,b,c) \wedge D(a,d) \vee D(c,d) \rightarrow D(b,d)$
 d) правильна відповідь відсутня
8. Встановіть відповідність між елементами алфавіту числень предикатів:
- a) символи логічних операцій a) $\forall x, \exists x.$
 b) предметні змінні b) $\wedge, \vee, \rightarrow, -.$
 c) одномісні предикатні змінні c) $P(\cdot), Q(\cdot), F(\cdot),$
 d) символи кванторних операцій d) x, y, z

9. Нехай предикат $P(x,y)$ заданий на деякій множині таблицею істинності. Визначте формули, значення яких буде істинним:

a) $\forall xR(x,a)$

b) $\exists xR(x,a)$

c) $\forall yR(a,y)$

d) $\exists yR(a,y)$

e) $\forall xR(x,b)$

f) $\forall xyR(x,y)$

x	y	$P(x,y)$
a	a	0
a	b	1
b	a	1
b	b	1

10. Визначте значення істинності висловлень X, Y, Z і R , якщо висловлення a) і b) – істинні, а висловлення c) і d) – хибні:

a) якщо 4 - парне число, то X

b) якщо Y , то 4 – непарне число

c) якщо 4 - парне число, то Z

d) якщо R , то 4 – непарне число

11. Які послідовності символів можна вважати формулами алгебри висловлень:

a) $((P \wedge Q) \wedge R) \rightarrow \bar{S}$

b) $((P \leftrightarrow Q) \wedge R) \rightarrow (P \vee R)$

c) $(P \leftrightarrow Q) \rightarrow R) \rightarrow (P \vee R)$

d) $((P \wedge (Q \rightarrow R)) \vee ((P \leftrightarrow R) \wedge Q))$

12. В структурі формальної мови систему знакових засобів, сукупність різноманітних форму, що фіксує в знаковій формі логічну структуру міркувань, логічні властивості складових елементів міркувань, називають:

a) *об'єкт-мовою*

b) *метамовою*

c) *правильна відповідь відсутня*

13. Визначте значення істинності наступних висловлень:

a) *якщо 12 ділиться на 6, то 12 ділиться на 5*

- b) якщо 11 ділиться на 6, то 11 ділиться на 3*
- c) якщо 15 ділиться на 3, то 15 ділиться на 6*
- d) 15 ділиться на 6 тоді і тільки тоді, коли 15 ділиться на 3*
- e) 15 ділиться на 5 тоді і тільки тоді, коли 15 ділиться на 4*

14. Формула $(F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$ є :

- a) здійсненою*
- b) тотожньо-істинною*
- c) тотожньо-хибною*
- d) правильна відповідь відсутня*

15. Під виразом $\forall x P(x)$ розуміють висловлення, істинне тоді, коли:

- a) $P(x)$ істинне для кожного елемента x деякої множини, і хибне в іншому випадку*
- b) $P(x)$ істинне хоча б для одного елемента x деякої множини*
- c) правильна відповідь відсутня*

МОДУЛЬ №2. ОБЧИСЛЮВАЛЬНІ ФУНКЦІЇ

I. Визначте варіанти правильних відповідей на наступні запитання

1. Частково-рекурсивні функції як алгоритмічна модель для уточнення поняття алгоритму були запропонована в:
 - a) 20-х роках XX ст.
 - b) 30-х роках XX ст.
 - c) 40-х роках XX ст.
 - d) 50-х роках XX ст.
 - e) 60-х роках XX ст.
 - f) 70-х роках XX ст.
 - g) правильна відповідь відсутня

2. До класу найпростіших функцій, що використовуються в якості базису для побудови обчислювальних функцій, відносяться:
 - a) нуль-функція
 - b) функція суперпозиції
 - c) функція примітивної рекурсії
 - d) всі вищевказані варіанти
 - e) правильна відповідь відсутня

3. В якості операторів, що застосовуються до базисних функцій для створення нових обчислювальних функцій, використовують:
 - a) оператор наступності
 - b) оператор проекції
 - c) оператор введення фіктивних аргументів
 - d) всі вищевказані варіанти
 - e) правильна відповідь відсутня

4. Підстановка, в результаті якої n -місна функція утворюється з підстановки у функцію F замість її аргументів m функцій f_1, f_2, \dots, f_m , називається:
 - a) оператором примітивної рекурсії
 - b) оператором мінімізації

- c) *оператором суперпозиції*
 - d) *оператором проєкції*
 - e) *правильна відповідь відсутня*
5. Функція називається частково-рекурсивною, якщо вона утворена з базисних функцій та:
- a) *оператора примітивної рекурсії*
 - b) *оператора мінімізації*
 - c) *оператора проєкції*
 - d) *оператора суперпозиції*
 - e) *всі вищевказані варіанти*
 - f) *правильна відповідь відсутня*
6. Оператор примітивної рекурсії дозволяє будувати $n+1$ -місну арифметичну функцію $G(x_1, x_2, y)$ з двох заданих функцій:
- a) *$n+2$ -місної функції $Q(x_1, x_2, y, z)$ та $n+1$ -місної функції $W(x_1, x_2, y)$*
 - b) *$n+2$ -місної функції $Q(x_1, x_2, y, z)$ та $n+2$ -місної функції $W(x_1, x_2, y, z)$*
 - c) *$n+1$ -місної функції $Q(x_1, x_2, y)$ та $n+1$ -місної функції $W(x_1, x_2, y)$*
 - d) *$n+1$ -місної функції $Q(x_1, x_2, y)$ та $n+2$ -місної функції $W(x_1, x_2, y, z)$*
 - e) *правильна відповідь відсутня*
7. Функція називається примітивно-рекурсивною, якщо вона утворена з базисних функцій та:
- a) *оператора примітивної рекурсії*
 - b) *оператора мінімізації*
 - c) *оператора проєкції*
 - d) *оператора суперпозиції*
 - e) *всі вищевказані варіанти*
 - f) *правильна відповідь відсутня*
8. Оператор примітивної рекурсії позначається як:
- a) R^n
 - b) I_m^n
 - c) $S(x)$

- d) $M(z)$
e) S_m^n
f) *правильна відповідь відсутня*
9. Оператор суперпозиції позначається як:
a) R^n
b) I_m^n
c) $S(x)$
d) $M(z)$
e) S_m^n
f) *правильна відповідь відсутня*
10. Функція введення фіктивних аргументів позначається як:
a) R^n
b) I_m^n
c) $S(x)$
d) $M(z)$
e) S_m^n
f) *правильна відповідь відсутня*
11. Тезис, стосовно того, що клас алгоритмічно обчислювальних часткових числових функцій збігається з класом усіх частково-рекурсивних функцій, запропонував:
a) *Пост*
b) *Тьюрінг*
c) *Марков*
d) *Гільбер*
e) *правильна відповідь відсутня*
12. В результаті виконання функції проєкції $\Gamma_m^n(x_1, x_2, \dots, x_m, \dots, x_n)$ отримується значення:
a) x_1
b) x_2
c) x_m
d) x_n
e) *правильна відповідь відсутня*

13. Двомісна функція $f_+(x,y) = x+y$ задовольняє наступну схему примітивної рекурсії:

- a) $x+0=S(x)$ та $x+(y+1)=(x+y)+1=S(x+y)$
- b) $x+0=S(x)$ та $x+(y+1)=(x+y)+1=S(y+1)$
- c) $x+0=I^1_1(x)$ та $x+(y+1)=(x+y)+1=S(x+y)$
- d) $x+0=I^1_1(x)$ та $x+(y+1)=(x+y)+1=S(y+1)$
- e) правильна відповідь відсутня

14. Двомісна функція $f_*(x,y) = x*y$ задовольняє наступну схему примітивної рекурсії:

- a) $x+0=O(x)$ та $x^*(y+1)=x*y+x=f_*(x,y)+x=f_+(x, f_*(x,y))$
- b) $x+0=S(x)$ та $x^*(y+1)=x*y+x=f_*(x,y)+x=f_+(x, f_*(x,y))$
- c) $x+0=S(0)$ та $x^*(y+1)=x*y+x=f_*(x,y)+x=f_+(y, f_*(x,y))$
- d) $x+0=S(x)$ та $x^*(y+1)=x*y+x=f_*(x,y)+x=f_+(y, f_*(x,y))$
- e) правильна відповідь відсутня

15. Функція наступності позначається як:

- a) R^n
- b) I^n_m
- c) $S(x)$
- d) $M(z)$
- e) S^n_m
- f) правильна відповідь відсутня

**МОДУЛЬ №3. АЛГОРИТМІЧНІ МОДЕЛІ НА ОСНОВІ
ДЕТЕРМІНОВАНИХ ПРИСТРОЇВ**

Визначте варіанти правильних відповідей на наступні запитання

1. Машина з довільним доступом як алгоритмічна модель для уточнення поняття алгоритму була запропонована в:
 - a) 20-х роках ХХ ст.
 - b) 30-х роках ХХ ст.
 - c) 40-х роках ХХ ст.
 - d) 50-х роках ХХ ст.
 - e) 60-х роках ХХ ст.
 - f) правильна відповідь відсутня

2. Фінітний комбінаторний процес Поста був:
 - a) результатом паралельного, у відношенні до машин Тьюрінга, дослідженням
 - b) теоретичною основою абстрактної машини Тьюрінга
 - c) всі вищевказані варіанти
 - d) правильна відповідь відсутня

3. До алгоритмічних моделей на основі детермінованих пристроїв відносять:
 - a) машини з довільним доступом
 - b) нормальні алгоритми Маркова
 - c) фінітний комбінаторний процес Поста
 - d) частково-рекурсивні функції
 - e) абстрактну машину Тьюрінга
 - f) правильна відповідь відсутня

4. Команда програми машини Поста складається з:
 - a) операції машини Поста
 - b) символу внутрішнього стану
 - c) переходу
 - d) символу зовнішнього алфавіту
 - e) номеру команди

5. Верхній перехід команди передачі керування в машині Поста відбувається у випадку, коли комірка, яку визначає каретка,:
 - a) *відмічена міткою*
 - b) *пуста*
 - c) *правильна відповідь відсутня*

6. В ході виконання програми машина Поста не зустрічається ні з командою зупинки, ні з некоректною командою, що призводить до:
 - a) *результативної зупинки*
 - b) *безрезультативної зупинки*
 - c) *“за циклювання” програми*
 - d) *правильна відповідь відсутня*

7. Абстрактна машина Тьюрінга складається з:
 - a) *каретки*
 - b) *керуючого пристрою*
 - c) *команди машини Тьюрінга*
 - d) *нескінченної кількості регістрів*
 - e) *нескінченної стрічки*
 - f) *всі вищевказані варіанти*
 - g) *правильна відповідь відсутня*

8. Прямокутна таблиці, що використовується для запису програм машини Тьюрінга, називається:
 - a) *функціональною схемою машини Тьюрінга*
 - b) *конфігурацією машини Тьюрінга*
 - c) *програмою машини Тьюрінга*

9. Конфігурацією машини Тьюрінга називається сукупність:
 - a) *функціональної схеми, внутрішнього стану та положення каретки машини Тьюрінга*
 - b) *внутрішнього стану, програми та стану стрічки машини Тьюрінга*
 - c) *стану стрічки, функціональної схеми та положення каретки*
 - d) *правильна відповідь відсутня*

10. Конфігурація K з внутрішнім станом q_1 , словом **алгоритм** на стрічці та кареткою під символом p записується як:
- a) *а₁лгоритм*
 - b) *алгоq₁ритм*
 - c) *алгор q₁итм*
 - d) *алгори q₁тм*
 - e) *алг q₁оритм*
 - f) *алгоритq₁м*
 - g) *правильна відповідь відсутня*
11. Мащини з довільним доступом мають наступні типи команд:
- a) *команду обнуління*
 - b) *команду безумовного переходу*
 - c) *команду умовного переходу*
 - d) *команду віднімання одиниці*
 - e) *команду додавання одиниці*
 - f) *команду встановлення мітки*
 - g) *команду зміщення вправо/вліво*
 - h) *всі вищевказані варіанти*
12. Символам, що використовують для позначення символів зміщення каретки машини Тьюрінга, відповідають наступні кодові групи універсальної машини Тьюрінга:
- a) *11*
 - b) *101*
 - c) *1001*
 - d) *10001*
 - e) *100001*
 - f) *1000001*
 - g) *10000001*
 - h) *правильна відповідь відсутня*

13. Вкажіть варіанти правильних записів програм машини Поста:
- | | | | |
|-----------|-----------|-----------|-----------|
| a) $1 E3$ | b) $1 E2$ | c) $1 E2$ | d) $1 V2$ |
| $3 V 2$ | $2 V4$ | $2 V3$ | $2 стоп$ |
| $2 стоп$ | $3 стоп$ | $3 стоп$ | $3 E1$ |
14. Робота машини з довільним доступом з конфігурацією $K=(a_1, a_2, \dots, a_s)$ припиняється після виконання команди I_t , якщо:
- | | |
|---|----------|
| a) $I_t=J(m, n, q), r_n=r_m \ i \dots$ | a) $t=s$ |
| b) $I_t \neq J(m, n, q) \ i \dots$ | b) $q>s$ |
| c) $I_t=J(m, n, q), r_n \neq r_m \ i \dots$ | c) $t=s$ |
15. Інформація про стан стрічки та місцерозташування каретки машини Поста називається:
- a) станом стрічки машини Поста
 - b) станом каретки машини Поста
 - c) функціональною схемою машини Поста
 - d) командою машини Поста
 - e) правильна відповідь відсутня

МОДУЛЬ №4. СКЛАДНІСТЬ АЛГОРИТМІВ

1. При якому найменшому значенні n алгоритм, що виконує $100n^2$ операцій, ефективніше алгоритму, що виконує 2^n операцій?
(2 балів)
2. Алгоритми сортування вставками і злиттям мають складність $8n^2$ і $64n\log n$ відповідно. Для яких значень n алгоритм сортування вставками є більш ефективнішим?
(3 балів)
3. Визначте складність алгоритму пошуку заданого елемента у впорядкованому масиві розмірністю n .
(4 балів)
4. Алгоритм впорядкування вирішує задачу розмірністю n за $f(n)$ мікросекунд. Визначте максимальну розмірність задачі впорядкування для функцій і часу, наведених у таблиці.

	1 сек.	1 хв.	1 год.	1 доба
$\log n$				
n				
$n\log n$				
n^2				
n^3				
2^n				
$n!$				

(6 балів)

5. Визначте складність алгоритму (в найкращому і найгіршому випадках), сутність роботи якого полягає у наступному:

І етап: вихідний масив рекурсивно поділяється на дві частини меншої розмірності до тих пір, доки розмірність частини не становитиме 1 (будь-

який масив розмірністю в один елемент можна вважати впорядкованим);

II етап: окремо виконується впорядкування кожної із частин;

III етап: здійснюється поєднання двох впорядкованих частин масиву в одну і процес продовжується етапом I.

(10 балів)

Правильні відповіді до модульно-тестових завдань для самоконтролю

№	Модуль 1.	Модуль 2.	Модуль 3.	Модуль 4.
1	e	b	f	
2	a	a	a	
3	d	e	a,c,e	
4	b	c	a,b,c	
5	b,c,d	a,b,d	b	
6	a-d,b-c,c-e,d-a, e-b	e	c	
7	b	a,d	a,b,e	
8	a-b,b-d,c-c,d-a	a	a	
9	b,d,e	e	d	
10	$x=1,y=0,z=0,r=1$	b	b	
11	b	e	a,c,e	
12	a	e	b,c,d	
13	a-0,b-1,c-0,d-0, e-0	c	c,d	
14	b	a	a-b,b-a,c-c	
15	a	c	e	

ЛІТЕРАТУРА

Основна:

1. Emil L. Post Finite combinatory processes – formulation 1 // The Journal of Symbolic Logic. - 1936. –№4, P. 122-134.
2. Turing A.M. On Computable numbers with an application to the Entscheidungsproblem //Proc. London Math. Soc., Ser.2. – 1936. –V.42. - №3-4, P.230-265.
3. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1979. – 536 с.
4. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы.: Пер. с англ.:Уч. пос. – М.: Издательский дом «Вильямс», 2000.-384 с.:ил.
5. Вирт. Н Алгоритмы + структуры данных = программы.- М.: Мир, 1985.-406с.
6. Грин Д., Кнут Д. Математические методы анализа алгоритмов.- М.: Мир, 1987.- 120 с.
7. Гуц А.К. Математическая логика и теория алгоритмов: Учебное пособие. – Омск: Диалог – Сибирь, 2003. – 108 с.
8. Дискретна математика: Підручник / Ю.М. Бардачов, Н.А. Соколова, В.Є Ходаков; За ред. В.Є. Ходакова. – К.: Вища шк., 2002. –287 с.: іл.
9. Игошин В.И. Задачник-практикум по математической логике. – М.: Просвещение, 1986. – 159 с.
10. Игошин В.И. Математическая логика и теория алгоритмов. - Саратов: Изд-во Саратовского ун-та, 1991. – 256 с.
11. Кнут Д. Искусство программирования для ЭВМ. – В 3–х т. Т.3 Сортировка и поиск. – М.:Мир, 1977.
12. Комп'ютерна дискретна математика: Підручник / М.Ф. Бондаренко, Н.В. Білоус, А.Г. Руткас. – Харків: Компанія СМІТ, 2004.-480 с.
13. Кормен Т и др. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р.Ривест. – М.:МЦНМО, 2001. - 960 с.

14. Лісовик Л.П., Шкільняк С.С. Теорія алгоритмів: Навч. посібник.- К.: Видавничий поліграфічний центр “Київський університет”, 2003.-163 с.
15. Мальцев А. И. Алгоритмы и рекурсивные функции.- М.:Наука, 1986/-368 с.
16. Носов В.А. Основы теории алгоритмов и анализа их сложности. Курс лекций. М.: 1992. – 112 с.
17. Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. – М.: Мир, 1984. – 510 с.
18. Трахтенброт Б.А. Алгоритмы и машинное решение задач. М.: Наука, 1957.- 98 с.
19. Успенский В.А. Машина Поста. 2-е изд., испр. М.: Наука, 1988.- 96 с.
20. Эдельман С.Л. Математическая логика. Уч. пос. для интов. –М.: Высшая школа, 1975. – 176 с.

Додаткова:

1. Алферова З.В. Теория алгоритмов.- М.: Статистика, 1973.- 164 с.
2. Асанов М.О., Баранский В.А., Расин В.В. Дискретная математика: графы, матроиды, алгоритмы. – Ижевск, НИЦ «Регулярная и хаотическая динамика», 2001.-288с.
3. Гильберт Д., Бернайс П. Основания математики. Т.1, Т.2. – М.: Наука, 1982.
4. Глушков В.М. Введение в кибернетику. Киев: Изд-во АН Укр ССР. 1964. 324 с.
5. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. - М. Мир. 1982.
6. Евстигнеев В.А. Применение теории графов в программировании. - М. Наука. 1986.
7. Зубенко В.В., Шкільняк С.С. Теорія алгоритмів у прикладах та задачах. – К.: Інтелектуальні системи, 1993. – 84 с.
8. Иванов Б.Н. Дискретная математика. Алгоритмы и программы. Уч. пос. - М.: Лаборатория Базовых Знаний, 2001.

9. Игошин В.И. Задачник-практикум по математической логике. – М.: Просвещение, 1986. – 159 с.
10. Катленд Н. Вычислимость. Введение в теорию рекурсивных функций: Пер. с англ. М.: Мир, 1983.- 256 с.
11. Клини С. Введение в метаматематику. – М.: Иностранная литература, 1957. – 526 с.
12. Клини С. Математическая логика. – М.: Мир, 1973. – 480 с.
13. Колмогоров А.Н., Драгалин Ф.Г. Математическая логика. Дополнительные главы. – М.: МГУ, 1984. – 120 с.
14. Кристофидес Н. Теория графов. Алгоритмический подход. - М. Мир. 1980.
15. Кузьмин И.В., Березюк И.Т., Фурманов К.К., Шаронов В.Б. Синтез вычислительных алгоритмов управления. - Л.: Техника, 1975. - 248 с.
16. Лавров И.А., Максимова Л.Л. Задачи по теории множеств, математической логике и теории алгоритмов. М.: Наука, 1984. - 224 с.
17. Лісовик Л.П., Редько В.Н. Алгоритми и формальные системи. – К.: КГУ, 1981. – 112 с.
18. Лісовик Л.П., Шкільняк С.С. Основи теорії алгоритмів. – К.: Інтелектуальні системи, 1993. – 94 с.
19. Лісовик Л.П., Шкільняк С.С. Теорія алгоритмів. – Київ, 2001. – Деп. в ДНТБ України 23.07.2001, № 132-Ук2001. – 148 с.
20. Мальцев А.И. Алгоритмы и рекурсивные функции. – М.: Наука, 1965. – 392 с.
21. Манин Ю.И. Вычислимое и невычислимое. – М.: Советское радио, 1980. – 128 с.
22. Марков А. А. Теория алгорифмов. Труды математического института им. В. А. Стеклова, – Т.42. – М. – Л.: Издательство Академии наук СССР, 1954. – 375 с.
23. Марков А.А., Нагорный Н.М. Теория алгорифмов. – М.: Наука, 1984. – 432 с.
24. Математические методы построения и анализа алгоритмов. – Л.: Наука, 1990. – 241 с.

25. Машины Тьюринга и рекурсивные функции / Г. Д. Эббинхауз, К. Якобе, Ф. К. Ман, Г. Хермес.— М.: Мир, 1972.—264 с.
26. Мендельсон Э. Введение в математическую логику. – М.: Наука, 1976. – 320 с.
27. Новиков В.А. Дискретная математика для программистов. Санкт-Петербург.: Питер, 2001.
28. Прийма С.М. Теорія алгоритмів: Навч. посібник. - Мелітополь: МДПУ, 2004. - 48 с.:іл.
29. Роджерс Х. Теория рекурсивных функций и эффективная вычислимость. – М.: Мир, 1972. – 624 с.
30. Роджерс Х. Теория рекурсивных функций і ефективна вычислимость. М., 1972.
31. Справочная книга по математической логике (под ред. Дж.Барвайса). Ч.1–Ч.4. – М.: Наука, 1982–1983.
32. Тарьян Р.Э. Сложность комбинаторных алгоритмов. Киб. Сборник. Н.с. вып. 17. - М. Мир. 1980.
33. Теорія алгоритмів (конспект лекцій) / Укладач: С.М. Прийма - Мелітополь: МДПУ, 2004.- 32 с.:іл.
34. Трахтенброт Б.А. Сложность алгоритмов и вычислений: спецкурс для студентов НГУ, Новосибирск, 1967.
35. Тьюринг А. Может ли машина мыслить? Пер.и примечания Ю.В.Данилова. М.: ГИФМЛ, 1960.
36. Успенський В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения. – М.: Наука, 1987. – 288 с.
37. Шенфилд Дж. Математическая логика. – М.: Наука, 1975. – 528 с.
38. Шкільняк С.С. Исследование программных алгебр функций натуральных аргументов і значений. – Модели и системы обработки информации. Вып. 8. – К., 1989. – С. 9–16.
39. Шкільняк С.С. Математична логіка: приклади і задачі. – Київ: ВПЦ "Київський університет", 2002. – 56 с.
40. Яворський Б.І. Теорія алгоритмів / Конспект лекцій. - Тернопіль: ТДТУ імені Івана Пулюя, 2000. - 32 с.

ПРЕДМЕТНИЙ ПОКАЗЧИК

А

- аксіома, 7, 8, 56, 58, 60, 61, 64
- аксіоматичний метод, 7, 54, 57
- алгебра висловлень, 11, 18, 19, 20, 25, 31, 35, 37, 38, 40, 49, 50, 54, 59, 62, 64
 - формула алгебри висловлень, 19
 - функція алгебри висловлень, 37
- алгоритм, 9, 35, 53, 57, 67, 69, 70, 71, 82, 83, 85, 90, 99, 101, 102, 107, 113, 120, 122, 123, 124, 126, 127, 128, 129, 131, 132, 133, 134, 135, 138, 144, 152, 153, 154, 168, 179, 181
 - властивості, 68
 - переходу від довільної формули алгебри висловлень до ДДНФ, 33
 - переходу від довільної формули алгебри висловлень до ДДНФ через таблицю істинності, 31
 - переходу від довільної формули алгебри висловлень до ДКНФ, 35
 - переходу від довільної формули алгебри висловлень до ДКНФ через таблицю істинності, 32
- алгоритмічні моделі
 - еквівалентність алгоритмічних моделей, 92
 - машина Поста, 76, 177
 - машина Тьюрінга, 79, 177
 - машини з довільним доступом, 86, 177
 - нормальні алгорифми Маркова, 89
- В
 - висловлення, 11, 13
 - просте (елементарне), 11, 41, 130
 - складне (складене), 11

Д

- диз'юнкція, 14
 - виключна, 16, 22, 27, 28, 38
 - з'єднувальна, 15
- сурова, 15

Е

- еквіваленція (еквівалентність), 17

З

- закони логічних операцій, 20
 - додаткові закони, 22
 - закон асоціативності, 22
 - закон Блейка - Порещького, 22
 - закон виключення третього, 21
 - закон де Моргана, 21
 - закон дистрибутивності, 22
 - закон згортання логічного виразу, 22
 - закон ідемпотентності, 20, 33
 - закон комутативності, 22
 - закон контрапозиції, 22
 - закон нуля та одиниці, 21
 - закон поглинання, 21
 - закон подвійного заперечення, 21
 - закон протиріччя, 21
 - закон розкриття імплікації та еквівалентності, 21
 - закон склеювання, 22
- заперечення, 12
 - подвійне заперечення, 13

І

- імплікація, 16, 17, 60

К

- квантор, 44
 - квантор загальності, 45, 66
 - квантор існування, 45
- кон'юнкція, 13
- конституента нуля, 27
- конституента одиниці, 27

Л

- логіка, 15, 40, 56, 59, 62, 187
 - математична логіка, 5

символічна логіка, 5
формальна логіка, 5
логічні числення, 8, 56
числення предикатів, 65
числення висловлень, 59

М

метаматематика, 8
методи розробки алгоритмів
динамічне програмування, 131,
143, 144, 152
метод декомпозиції, 131, 132,
133, 134
метод розгалуджень і меж, 131,
137, 139, 141

Н

нормальна форма, 28
диз'юнктивна нормальна
форма, 28
досконала диз'юнктивна
нормальна форма, 29
досконала кон'юнктивна
нормальна форма, 29
кон'юнктивна нормальна
форма, 28

О

обчислювальна функція, 69, 72, 75
нуль-функція, 71, 72, 173
примітивно-рекурсивна
функція, 75, 174
функція наступності, 71, 72
функція проєкції, 71, 72
частково-рекурсивна функція,
75, 174
оператор
мінімізації, 74

примітивної рекурсії, 73, 94,
174
суперпозиції, 73, 94, 175

П

предикат, 39, 40, 41, 42, 43, 44, 45,
48, 49, 52
двомісний предикат, 42
одномісний предикат, 41
пропозиційні зв'язки, 12
протиріччя, 8, 33, 35, 36, 97, 99

С

складність алгоритму, 100, 107,
108, 113, 120, 124, 128
асимптотична часова
складність, 124
ємкісна характеристика
складності, 101
часова характеристика
складності, 100
швидкість росту складності,
124, 168

Т

таблиця істинності, 13
тавтологія, 20, 40
теза Черча, 76
теорія алгоритмів
теорія алгоритмів, 8, 57, 70, 72,
96, 100, 186

Ф

функтатор, 12

Ш

штрих Шеффера, 16, 22

Прийма Сергій Миколайович

МАТЕМАТИЧНА ЛОГІКА І ТЕОРІЯ АЛГОРИТМІВ

Навчальний посібник
для студентів вищих педагогічних навчальних закладів

Друкується в авторській редакції

Підписано до друку 02.06.2008. Формат 60x84/16.
Друк ризографічний. Папір офсетний. Гарнітура Times New
Roman.
Ум. друк. арк. 7,79. Наклад 300 прим.
Зам. № 1276.

Видавець і виготівник
ТОВ „Видавничий будинок Мелітопольської міської друкарні”
72316, м. Мелітополь, вул. К.Маркса, 21
т. (0619) 44-01-43, тел./факс (06192) 6-74-43
e-mail:mmdprint@mail.ru

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виробників і
розповсюджувачів видавничої продукції
від 26.09.2003 р., серія ДК №1509