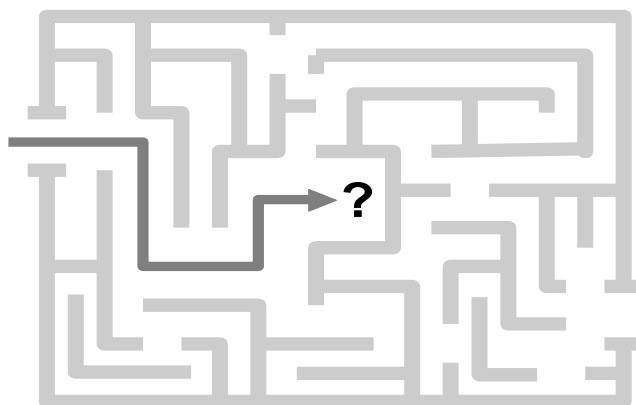


Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

О. В. Костів, С. А. Ярошко

МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ

Тексти лекцій



Львів
Видавничий центр ЛНУ імені Івана Франка
2002

Костів О. В., Ярошко С. А. Методи розробки алгоритмів: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002. – 96 с.

У текстах лекцій розглянуто алгоритми розв'язування типових задач з програмування та подано основні методи розробки алгоритмів. Розв'язки типових задач допоможуть читачеві здобути початкові навички у проектуванні алгоритмів та у програмуванні мовою Pascal. Докладне висвітлення методів розробки алгоритмів та пояснення прикладів їхнього застосування стане у пригоді тим, хто збирається створювати власні ефективні програми розв'язування складних задач.

Для студентів факультету прикладної математики та інформатики, механіко-математичного факультету і всіх, хто цікавиться програмуванням.

Рекомендовано до друку науково-методичною радою факультету прикладної математики та інформатики
Протокол № 11 від 25.09.01

Рецензенти: Г. Г. Цегелик, доктор фіз.-мат. наук (Львівський національний університет);
П. В. Вагін, кандидат фіз.-мат. наук (Львівський національний університет)

Відповідальний за випуск В. В. Черняхівський

Редактор І. М. Лоїк

© Костів О. В., Ярошко С. А., 2002

ЗМІСТ

1.	Вступ	5
2.	Задачі цілочислової арифметики	6
2.1.	Середнє арифметичне цифр числа	6
2.2.	Чи є число паліндромом?	7
2.3.	Гіпотеза Безу	8
2.4.	Запис числа у шістнадцятковій системі	9
2.5.	Розкладання числа на прості множники	12
3.	Програми з простим повторенням	14
3.1.	Покрокове введення даних	14
3.2.	Покрокове виведення даних	16
3.3.	Обчислення за рекурентними формулами	17
4.	Поєднання повторення з галуженням	18
4.1.	Скільки є «правильних» серед усіх заданих?	18
4.2.	Максимальний елемент послідовності	20
4.3.	Перевірка впорядкованості	21
4.4.	Пошук місця елемента послідовності	22
5.	Вкладені цикли в матричних задачах	24
5.1.	Побудова матриць	24
5.2.	Дії матричної алгебри	28
5.3.	Порівняння та переміщення елементів матриці	30
6.	Основні алгоритми сортування	32
6.1.	Сортування вставками	33
6.2.	Сортування вибором	34
6.3.	Сортування обмінами	35
7.	Сортування структур даних	36
7.1.	Впорядкування рядків матриці	37
7.2.	Впорядкування файлу за допомогою списку	39
7.3.	Впорядкування файлу бінарним деревом	41
7.4.	Впорядкування файлу злиттям	42
8.	Опрацювання текстової інформації	47
8.1.	Розпізнавання чисел	47
8.2.	Форматування виведення числової інформації	48
9.	Обчислення з заданою точністю	51
9.1.	Сумування рядів	52
9.2.	Обчислення кореня алгебричного рівняння	54
9.3.	Числове інтегрування	56
10.	Покрокова розробка програм	59
11.	Рекурсія	64
11.1.	Задача про Ханойські вежі	65
11.2.	Алгоритм швидкого сортування	67
11.3.	Обхід двійкового дерева	70
12.	Програмування з поверненням назад	72
12.1.	Тур коня	72
12.2.	Тур коня без рекурсії	76
13.	Метод часткових цілей	78

14. Метод підйому. Евристики	84
15. Генетичні алгоритми.....	87
15.1. Сфера застосування.....	87
15.2. Підготовчі кроки	88
15.3. Програмна реалізація	90
16. Список літератури	96

1. Вступ

Сучасні технології програмування переживають період бурхливого розвитку. Причинами цього явища є зростання потужностей комп'ютерної техніки, її здешевлення, створення всесвітньої мережі Internet, виникнення все нових і нових сфер застосування комп'ютерів і потреба у різноманітному програмному забезпеченні.

З метою полегшення життя програміста створено середовища візуального програмування. Тепер насправді «одним пальцем» (та за допомогою миші) можна будувати надскладний інтерфейс майбутньої програми: наділяти його різноманітними меню, панелями діалогу, піктограмами та хитромудрими перемикачами. І дуже часто за зовнішнім блиском усіх цих вікон, кнопок та лінійок прокрутки початкуючий програміст втрачає головне.

Що ж головне у програмі? Вікна та меню – це лише засіб обміну інформацією між програмою і зовнішнім світом. Вони прийшли на зміну перфокартам і значно спростили «спілкування» з комп'ютером, зробили його наочнішим і доступнішим широкому загалу користувачів. Вікна та меню – це лише форма, а зміст програми – в алгоритмах, які вона реалізує. Саме алгоритми визначають ефективність (а, отже, корисність) програми, саме алгоритми є найважливішим продуктом і надбанням комп'ютерної науки. Саме вони, вбрані у шати машинних команд чи об'єктно-орієнтованих програм, керують роботою комп'ютерів у всьому світі.

Навчитися складати алгоритми – одне з перших і головних завдань майбутнього програміста. На жаль, не існує простої відповіді на запитання, як це зробити? Майже кожна задача з програмування вимагає індивідуального підходу, недарма програмування порівнюють з мистецтвом. Однак як у живописі чи музиці є основні прийоми і навички, якими зобов'язаний володіти кожен митець, так і в програмуванні є класи задач, для розв'язування яких створено типові алгоритми. Відомі також певні підходи до розробки алгоритмів розв'язування нових задач, чи *методи розробки алгоритмів*.

Кожному випускникові Львівського національного університету імені Івана Франка, що вивчав програмування, необхідно навчитися класифікувати задачі, застосовувати до їхнього розв'язування відомі алгоритми, враховуючи особливості конкретної задачі, необхідно оволодіти основними методами розробки алгоритмів. Саме описові алгоритмів та методів їхньої розробки присвячено цей посібник. У його першій частині (р. 1–8) наведено розв'язки окремих типових задач, докладно описано процес створення та удосконалення кожного алгоритму. Її можна використовувати і як своєрідний «розв'язник» задач з програмування, і для того, щоб набути початкового досвіду проектування алгоритмів. Друга частина (р. 9–14) розрахована на читачів, що зважилися іти далі. Її присвячено описові та ілюстрації основних методів розробки алгоритмів. Вони стануть у пригоді тим, кому доведеться створювати програми для зовсім нових, ще не розв'язаних раніше задач.

Усі приклади програм у посібнику написано мовою Free Pascal. Сподіваємося, що їх зможуть легко зрозуміти ті читачі, які вивчають будь-яку

іншу мову програмування високого рівня. Побудова алгоритму значною мірою залежить від вибору структур даних для відображення у програмі даних задачі. У кожному прикладі це питання буде обговорено зокрема, проте сподіваємось, що читач володіє знаннями щодо таких структур даних як масив, рядок, файл, список, стек, черга, дерево. Ні один з прикладів не претендує на досконалість. Автори розраховують на те, що читачі спробують запропонувати власні варіанти алгоритмів чи модифікації описаних.

2. Задачі цілочислової арифметики

Чи доводилось Вам виконувати такі завдання: обчислити середнє арифметичне цифр заданого натурального числа; перевірити, чи задане натуральне число є паліндромом; розкласти задане натуральне число на прості множники? (*Паліндром* – число, величина якого не зміниться, якщо порядок цифр у його записі змінити на обернений.) На перший погляд може здатися, що програми для розв'язування цих задач є дуже складними, потребують використання масивів чи файлів. Насправді ж для їхнього створення достатньо вміло використати операції цілочислової арифметики. Давайте подивимось, як це зробити.

Надалі під час розв'язування задач ми будемо старатися виконувати такі дії: формулювати допоміжні запитання до умови задачі, щоб краще її зрозуміти, і давати на них відповіді; проектувати прості змінні та структури даних для потреб майбутнього алгоритму, давати його словесний опис.

2.1. Середнє арифметичне цифр числа

Задача 1. Дано натуральне число. Обчислити і надрукувати середнє арифметичне цифр у записі цього числа.

Розв'язування. Відомо, що для обчислення середнього арифметичного деяких величин необхідно знати їхню суму та кількість. Як обчислити суму цифр, з яких складається запис числа n ? Для цього доведеться якимось чином отримати кожен з них. Найпростіше отримати цифру з наймолодшого розряду – розряду одиниць. Вона дорівнює остачі від ділення n на 10. Наприклад, для $n = 1976$ легко отримати $1976 \bmod 10 = 6$. Як дізнатися кількість десятків у числі n ? Спробуємо так: $(1976 \bmod 100 - 6) \operatorname{div} 10 = 7$. Однак, поміркувавши трохи, вдається досягти такого ж результату і простішим способом: $1976 \operatorname{div} 10 \bmod 10 = 7$. Тут операція ділення вилучає уже враховану цифру з розряду одиниць, а операція взяття остачі повертає наступну – цифру з розряду десятків. Очевидно, що послідовно вилучаючи з запису числа молодші цифри, можна перебрати їх усі.

Як знайти k кількість цифр у записі числа n в десятковій системі числення? Наприклад, за формулою $k = \lfloor \lg n \rfloor + 1$. Або перелічити їх по одній, адже ми вже придумали, як отримати кожен з них.

Які змінні необхідні для побудови алгоритму? Очевидно, такі: n – задане число, c – наймолодша цифра, s – сума цифр, k – лічильник цифр, a – шукане середнє арифметичне.

Тепер запишемо алгоритм у вигляді програми на Free Pascal:

```

program digitsAverage;
  var n,s:word;
      k,c:byte; a:real;
begin write('Введіть натуральне число: '); readln(n);
      s:=0; k:=0;      {початкові значення суми та лічильника}
      while n>0 do
        begin c:=n mod 10;      {отримали наймолодшу цифру}
            s:=s+c; k:=k+1;    {врахували її}
            n:=n div 10      {вилучили її}
        end; {while}
      a:=s/k;      {обчислили середнє арифметичне}
      writeln('n=',n,' a=',a)      {і видрукували його}
end.

```

Виконаємо ручну прокрутку цієї програми, щоб краще зрозуміти, як працює алгоритм. Для цього заповнимо таблицю прокрутки. З її допомогою покажемо, як змінюються значення змінних і умова продовження циклу під час виконання програми для деякого конкретного значення n . Нехай дано число 1976, тоді

c	s	k	n	$n>0?$	a
	0	0	1976	\Rightarrow +	
6	6	1	197	+	
7	13	2	19	+	
9	22	3	1	+	
1	23	4	0	- \Leftarrow	5.75

Тут символами “ \Rightarrow ” і “ \Leftarrow ” позначають відповідно початок і кінець циклу. З таблиці видно, як з n послідовно вилучаються цифри, які стають значеннями змінної c .

Задачу розв’язано.

2.2. Чи є число паліндромом?

Задача 2. Дано натуральне число. Перевірити, чи воно є паліндромом.

Розв’язування. Запис паліндрома симетричний: перша цифра дорівнює останній, друга – передостанній і т. д. Проте перевірити, чи так воно є у заданого числа, не так уже й просто: отримати першу цифру набагато складніше, ніж останню. А чи не можна пристосувати якийсь алгоритм з попередньої задачі для того, щоб отримати обернений запис числа? Це легко зробити, якщо відповідним чином враховувати послідовні значення змінної c і будувати з них нове число. Адже кожне число можна записати у вигляді полінома за степенями 10, коефіцієнтами якого є цифри числа.

Розглянемо другий рядок таблиці прокрутки попередньої задачі. Бачимо, що цифра 6 уже перемістилась зі змінної n до змінної s . Як зробити, щоб на наступному кроці циклу змінна s отримала значення 67, а не 13? Наприклад, за допомогою оператора $s:=s*10+c$ (замість $s:=s+c$). Легко переконатися, що виконання цього оператора на кожному кроці циклу дає змогу отримати в s обернений запис заданого числа. Проте є ще одна перешкода для отримання розв'язку задачі: під час виконання циклу змінна n отримала значення нуль, і ми не маємо з чим порівнювати побудоване нами число. Цю перешкоду можна легко подолати: використаємо в циклі не саме значення n , а його копію.

Тепер легко записати весь алгоритм, уточнивши призначення змінних: n – задане число; m – його копія; c – наймолодша цифра; s – нове число:

```

program isPolyndrome;
  var n,m,s:longint;
      c:byte;
begin write('Введіть натуральне число: '); readln(n);
      s:=0; m:=n; {початкове значення нового числа та копія n}
      while m>0 do
        begin c:=m mod 10; s:=s*10+c; {врахували наймолодшу цифру}
            m:=m div 10 {i вилучили її}
        end; {while}
      if s=n then writeln(n,'- паліндром')
        else writeln(n,'- не паліндром')
end.

```

Отже, ми успішно пристосували попередню програму до нової задачі. Голівудські продюсери давно зрозуміли, що хороший сценарій можна використати і для зйомок фільму, і для серіалу, і для публікації роману чи хоча б збірника коміксів. Це розуміння вони перейняли від програмістів: адже один з методів розробки алгоритмів і полягає у повторному використанні старого перевіреного алгоритму (чи його частини) у нових умовах.

2.3. Гіпотеза Безу

Розглянемо деяке натуральне число n . Якщо воно не паліндром, то побудуємо нове число, змінивши порядок цифр у записі n на обернений, і додамо його до n . Якщо отримана сума не паліндром, то повторимо з нею описані дії, поки не отримаємо паліндром. Гіпотеза Безу стверджує, що описаний процес скінченний для будь-якого натурального n . Цю гіпотезу досі не доведено.

Задача 3. Дано натуральні числа a , b , l ($a \leq b$). Перевірити, чи виконується гіпотеза Безу для кожного натурального числа з проміжку $[a, b]$ не більше, як за l кроків процесу.

Розв'язання. Щоб побудувати алгоритм розв'язування цієї задачі, спробуємо краще зрозуміти її умову та сформулювати головні етапи отримання розв'язку. Отже, у задачі йдеться про перевірку гіпотези Безу для послідовності чисел $a, a+1, \dots, b$. Якщо б ми вміли виконувати таку перевірку для одного числа n , то перевірити усю послідовність теж змогли б, послідовно надаючи змінній n значення $a, a+1, \dots, b$ (наприклад, за допомогою оператора циклу **for**). Як

виконати перевірку для n ? Очевидно, що описаний процес перетворення є циклічним. Цикл необхідно виконувати до отримання паліндрома або до вичерпання заданої кількості повторень. Розв'язуючи попередню задачу, ми описали алгоритм отримання оберненого запису заданого числа. Оформимо тепер цей алгоритм у вигляді підпрограми, яку використаємо з метою перевірки на кожному кроці циклу.

Тепер уже можна записати алгоритм. Призначення змінних пояснено у коментарях програми:

```

program hypothesisBezu;
  var a,b,l:word;      {вхідні дані задачі}
      n:word;          {параметр зовнішнього циклу, який перебирає
                        всі натуральні числа заданого діапазону}
      k:word;          {лічильник кроків внутрішнього циклу}
      m:longint;       {на першому кроці циклу - копія n,
                        на наступних - сума прямого і оберненого запису}
      s:longint;       {обернений запис числа m}
  function reverse(m:word):longint;
  {повертає обернений запис числа m}
    var s:longint;
  begin s:=0;
    while m>0 do
      begin s:=s*10+m mod 10; m:=m div 10;
      end; {while} reverse:=s
  end; {reverse}
Begin {головної програми}
  write('Введіть дані a,b,l: '); readln(a,b,l);
  for n:=a to b do
    begin k:=0; s:=0; m:=n;
      repeat m:=m+s; s:=reverse(m); k:=k+1
      until (s=m) or (k>1);
      write('Для числа ',n,' за ',k-1,' кроків ');
      if k>1 then writeln('гіпотеза не виконується')
      else writeln('отримано паліндром ',s)
    end{for}
End.

```

У цій програмі початкові значення для змінних s і m підбрано так, щоб перевірку (чи є паліндромом n) здійснити на першому кроці циклу. Результати перевірки гіпотези друкують для кожного числа з проміжку $[a, b]$.

Розв'язок цієї задачі демонструє, як у складній програмі можна використати алгоритм – розв'язок простішої задачі. Таблицю прокрутки цієї програми пропонуємо читачеві побудувати самостійно.

2.4. Запис числа у шістнадцятковій системі

Безперечно, більшість із нас розпочинала вивчати програмування з опанування двійкової, вісімкової та шістнадцяткової систем числення. Пригадуєте, чи не найгроміздкішими були обчислення під час переведення чисел з однієї системи в іншу. От би мати таку програму, яка б швидко і безпомилково виконувала такі переведення! Звичайно, кожен компілятор вміє робити переведення з десяткової системи у двійкову, і навпаки. Проте виконує

він це для внутрішніх потреб програми і майже ніколи не демонструє користувачеві результати переведення.

Задача 4. Дано натуральне число. Отримати його запис у двійковій системі числення.

Розв'язування. Пригадаємо собі алгоритм переведення цілого числа у нову систему: потрібно обчислити остачу від ділення цього числа на нову основу, отриману частку знову поділити на нову основу і обчислити остачу і так далі, аж доки чергова частка дорівнюватиме нулю. Знайдені остачі, записані в оберненому до черговості отримання порядку, утворюють запис числа в новій системі.

Розв'яжемо спочатку простішу від поставленої задачу: переведемо задане число у двійкову систему. Щоб послідовно обчислити згадані частки й остачі, нам стане в пригоді досвід розв'язування попередніх задач (використаємо в циклі операції **mod** і **div**). Як побудувати двійковий запис числа з обчислених остач? Щоб відповісти на це запитання, доведеться спочатку вирішити, змінну якого типу ми використаємо для зберігання цього запису. Наприклад, його можна змодельювати змінною цілого типу. Двійковий запис числа буває досить довгим, тому доцільно використати тип `longint`, що має найширший серед цілих типів діапазон можливих значень. Щоб чергова остача під час побудови двійкового запису займала відповідний розряд, будемо домножувати її на відповідний степінь десяти. Цей степінь можна зберігати у додатковій робочій змінній. Отже, використаємо такі змінні: n – задане число; c – чергова остача; p – степінь десяти; s – число, що моделює двійковий запис числа n .

```

program binaryForm1;
  var n:word; c:byte; p,s:longint;
begin write('Введіть натуральне число: '); readln(n);
      s:=0; p:=1;      {двійковий запис спочатку порожній; p=10^0}
  while n>0 do
  begin c:=n mod 2;           {обчислили чергові остачу}
        n:=n div 2;          {і частку}
        s:=s+c*p;           {остачу помістили в двійковий запис}
        p:=p*10 {підготували степінь 10 для наступної ітерації}
  end; {while}
  writeln('Запис у двійковій системі: ',s)
end.

```

Перевіримо за допомогою ручної прокрутки, чи буде ця програма працювати правильно, наприклад, для $n = 10$:

c	n	s	p	$n>0?$
	10	0	1	\Rightarrow +
0	5	0	10	+
1	2	10	100	+
0	1	10	1000	+
1	0	1010	10000	- \Leftarrow

Бачимо, що величини s і p зростають дуже швидко. Їхнього розміру вистачить для утворення двійкового запису довжиною до 10 цифр, тобто для $n \in [0; 1023]$. Щоб мати змогу переводити у двійкову систему і більші числа, використаємо для зображення їхнього запису рядок. Перед початком циклу він порожній. У циклі на початку цього рядка необхідно дописувати '0' або '1', залежно від парності частки (за такого підходу нам навіть не потрібно буде обчислювати остачу).

Порівняйте таку програму з попередньою:

```

program binaryForm2;
  var n:word; s:string;
begin write('Введіть натуральне число: '); readln(n);
  s:=''; {двійковий запис спочатку порожній}
  while n>0 do
    begin if odd(n) then s:='1'+s else s:='0'+s;
      {остача від ділення непарного числа на 2 дорівнює 1}
      n:=n div 2
    end; {while}
  writeln('Запис у двійковій системі: ',s)
end.

```

Програма binaryForm2 виглядає привабливіше за binaryForm1: вона потребує менше змінних і обчислень, працює для ширшого діапазону вхідних даних. Пристосуємо її щодо отримання шістнадцяткового запису числа. З цією метою нам необхідно вирішити, як перетворювати остачі c від ділення на 16 у літери – шістнадцяткові цифри. Для значень $c = 0, 1, \dots, 9$ таке перетворення можна виконати за допомогою виразу $\text{chr}(\text{ord}('0')+c)$. (Тут використано такі властивості: $\text{chr}(\text{ord}('0'))='0'$ і $\text{ord}('5')-\text{ord}('0')=5$). А для значень $c = 10, \dots, 15$ – за допомогою виразу $\text{chr}(\text{ord}('A')-10+c)$.

```

program hexaForm;
  const a=ord('0'); b=ord('A')-10; {ці константи використаємо
    для того, щоб не робити в циклі зайвих обчислень}
  var n:longint; c:byte; s:string;
begin write('Введіть натуральне число: '); readln(n);
  s:=''; {шістнадцятковий запис спочатку порожній}
  while n>0 do
    begin c:=n mod 16; n:=n div 16;
      if c<10 then s:=chr(a+c)+s
      else s:=chr(b+c)+s
    end; {while}
  writeln('Запис у шістнадцятковій системі: ',s)
end.

```

У програмуванні часто буває так, що доводиться спочатку формулювати і розв'язувати деякі допоміжні задачі, вибирати кращий варіант з кількох можливих, далі удосконалювати його і доводити до завершення. Такий процес побудови алгоритму ми і хотіли проілюструвати цим прикладом.

2.5. Розкладання числа на прості множники

На завершення цього параграфу опишемо розв'язування ще однієї цікавої задачі.

Задача 5. Дано натуральне число n ($n > 1$). Розкласти його на прості множники.

Розкладом є послідовність простих чисел – дільників n – з урахуванням їхньої кратності. Наприклад: $84 = 2 \times 2 \times 3 \times 7$. Як отримати такий розклад? Перше, що спадає на думку, – це перебрати всі числа k з проміжку від 2 до n і для кожного з них визначити, чи воно просте та чи є воно дільником n . Якщо так, то надрукувати знайдене значення k потрібну кількість разів. Як перевірити, чи ділиться n на k ? Скільки разів? Просто перевірити остачу від ділення n на k , k^2 , k^3 , ... Наприклад, використавши змінну m для зберігання степеня k це можна зробити так:

```
m:=k;
while n mod m=0 do
begin write(m); m:=m*k end;
```

Як перевірити, чи число k є простим? Необхідно визначити, чи має k інші дільники, крім 1 і k . З цією метою перебираємо усі числа 2, 3, ..., $k-1$, перевіряючи, чи k ділиться на ці числа. Проте неважко здогадатися, що кожному дільникові j числа k , більшому за \sqrt{k} , відповідає дільник $k \operatorname{div} j$, менший за \sqrt{k} , тому достатньо буде перебрати числа 2, 3, ..., $[\sqrt{k}]$ (для великих чисел k це суттєво скорочує перебір). Нехай змінна $kSimple$ набуває значення «істина», якщо k – просте, і значення «хиба» у протилежному випадку. Перевірку того, чи k – просте, можна записати так:

```
kSimple:=true;
for j:=2 to round(sqrt(k)) do
if k div j=0 then begin kSimple:=false; Break end;
```

Відомо, що результат обчислення кореня має тип `real` і може бути обчислений з недостачею. Тому у цьому фрагменті для перетворення `sqrt(k)` до цілого типу ми використали стандартну функцію `round`, щоб взяти значення з надлишком, а не `trunc`, яка повертає ціле значення з недостачею. Тепер можна записати всю програму:

```
program decomposition1;
var n:longint; {задане число}
    k:longint; {кандидати в дільники}
    m:longint; {ступінь k}
    j:longint; {параметр внутрішнього циклу}
    kSimple:Boolean;
begin write('Введіть натуральне число: '); readln(n);
      write(n, '='); {починаємо друкувати розклад}
      for k:=2 to n-1 do
        begin kSimple:=true; {перевіряємо k на простоту}
              for j:=2 to round(sqrt(k)) do
```

```

    if k div j=0 then begin kSimple:=false; Break end;
  if kSimple then
  begin m:=k;                               {скільки разів k ділить n?}
    while n mod m=0 do
      begin write(m, 'x'); m:=m*k
        end; {while}
    end; {if}
  end; {for} writeln('1')                   {друк розкладу завершено}
end.

```

У цій програмі кожне з чисел 2, 3, ..., $n-1$ необхідно перевірити, чи воно просте. Зауважимо, що така перевірка потребує виконання циклу. Після уважного аналізу програми `decomposition1` можна дійти висновку, що набагато вигідніше перевіряти на простоту виключно дільники числа n : обчислити остачу легше, ніж виконати згаданий цикл. Після нескладної переробки отримаємо:

```

program decomposition2;
  var n,k,m,j:longint;
      kSimple:Boolean;
begin write('Введіть натуральне число: '); readln(n);
      write(n, '=');                               {починаємо друкувати розклад}
  for k:=2 to n-1 do
  if n mod k=0 then                                {якщо k - дільник n, то}
  begin kSimple:=true;                             {перевіряємо k на простоту}
    for j:=2 to round(sqrt(k)) do
      if k div j=0 then begin kSimple:=false; Break end;
    if kSimple then
    begin m:=k;                                     {скільки разів k ділить n?}
      while n mod m=0 do
        begin write(m, 'x'); m:=m*k
          end; {while}
      end; {if}
    end; {if&for} writeln('1')                     {друк розкладу завершено}
  end.

```

Ця програма відрізняється від попередньої лише одним умовним оператором, однак працює набагато швидше. Проаналізуємо її. Вона розпочинає роботу з того, що знаходить серед чисел 2, 3, ..., $n-1$ перше, яке ділить n . Далі відбувається перевірка, чи це число є простим, хоча ця перевірка є зайвою: якщо n не ділиться на 2, 3, ..., $k-1$ і ділиться на k , то k – просте число. Справді, якщо б k було складним, то воно ділилося б на котресь з чисел 2, 3, ..., $k-1$, але тоді б і n ділилось на це число, що суперечить способів побудови k .

Наведені міркування засвідчують, що і `decomposition2` не є досконалою. Очевидно, можна суттєво зменшити кількість дорогих перевірок чисел на простоту. Адже перший вибраний з чисел 2, 3, ..., $n-1$ дільник k буде простим. Якщо тепер надрукувати його і поділити на нього n , то часткою буде число, набір простих дільників якого збігається з набором ще неврахованих дільників n . Тобто описані дії можна повторити для частки $n \text{ div } k$. Процес завершиться, коли чергова частка дорівнюватиме одиниці.

Враховуючи все сказане, складемо програму, яка не міститиме явних перевірок на простоту і не виконуватиме перебору зайвих значень k :

```

program decomposition3;
  var n,k:longint;
begin write('Введіть натуральне число: '); readln(n);
  write(n, '=');           {починаємо друкувати розклад}
  k:=2;
  while n>1 do
    if n mod k=0 then
      begin write(k, 'x'); n:=n div k
      end{надрукували черговий дільник і вилучили його з n}
    else k:=k+1;
  writeln('1')           {друк розкладу завершено}
end.

```

Ми розглянули декілька варіантів програми розв'язування однієї задачі. Процес побудови програм демонструє, як, відштовхуючись від програми, яка спочатку видалася досить доброю, ми, використовуючи очевидні міркування, отримали програму, що мало схожа на початкову і, що важливо, набагато краща за неї.

3. Програми з простим повторенням

У цьому параграфі йтиме мова про задачі, у яких необхідно виконати послідовний перебір членів деякої числової послідовності. На прикладах програм попереднього параграфа ми уже демонстрували читачеві, що для побудови алгоритму важливе значення має вибір структури даних. Чи завжди для опрацювання заданої послідовності значень a_1, a_2, \dots, a_n у програмі використовують масив? Підсвідомо хочеться відповісти «так», адже в умові задачі записано змінні з індексами. Проте поспішати з висновками не будемо, бо у постановці задач здебільшого використовують систему математичних позначень, яка не збігається з «системою позначень» програми. Математик пише $y(x)$, щоб вказати функціональну залежність *математичних* величин y та x , а програміст використовує *прості змінні*: x – для зберігання заданої величини та y – для запису результату обчислень. А дужки після імені змінної у мовах програмування мають спеціальне синтаксичне значення. Так і масиви у програмах використовують далеко не в кожному випадку, коли в умові задачі є змінні з індексами. Наприклад, якщо кількість членів заданої послідовності наперед невідома, то оголосити масив просто не вдасться. Для перебору значень звичайно застосовують відповідні цикли. Якщо кожне значення враховується тільки на одній ітерації циклу, то для його зберігання доцільно використати просту змінну, а не елемент масиву.

3.1. Покрокове введення даних

Часто умову задачі формують так, що для отримання результату необхідно певним чином опрацювати кожен член заданої послідовності. Наприклад:

Задача 6. Дано натуральне число n і дійсні a_1, a_2, \dots, a_n . Обчислити $\sqrt{a_1^2 + a_2^2 + \dots + a_n^2}$.

Щоб отримати розв'язок задачі, необхідно знайти відповіді на кілька запитань. Що є шуканим значенням? Корінь з суми квадратів членів послідовності. Для її накопичення використаємо змінну s . Що необхідно зробити з кожним значенням послідовності $\{a_i\}$? Піднести до квадрату і додати до загальної суми. Ці дії можна виконати протягом однієї ітерації, тому для тимчасового зберігання a_i використаємо просту змінну, наприклад, a . Кількість членів послідовності задано значенням n . Тому для їх перебору зручніше застосувати цикл з параметром – змінною i , оскільки кількість повторень буде задано значенням n . Які дії необхідно виконати перед циклом? Тільки ініціалізувати змінну s .

Тепер можемо записати програму:

```

program squareNorm;
  var i,n:integer; a,s:real;
begin write('Введіть значення n: '); readln(n);
  s:=0;
  for i:=1 to n do
  begin write('Введіть ',i,'-е число: '); readln(a);
    s:=s+sqr(a)
  end; {for} s:=sqrt(s);
  writeln('s=',s)
end.

```

Іноді в задачах вказують не кількість членів послідовності, а умову її закінчення: послідовність чисел закінчується нулем, послідовність літер закінчується крапкою тощо. У таких випадках перебір виконують за допомогою циклу з перед- або постумовою.

Задача 7. Дано послідовність чисел, яка містить хоча б одне нульове значення, і перший член якої відмінний від нуля. Обчислити середнє арифметичне членів послідовності, що передують першому нулеві.

Тут умовою закінчення обчислень є рівність нулю чергового члена послідовності. Як і в попередній задачі для зберігання значень послідовності використаємо просту змінну a . Для обчислення суми значень і їхньої кількості використаємо змінні s і k . Ці змінні необхідно ініціалізувати перед початком циклу.

```

program average;
  var a,s:real; k:integer;
begin s:=0; k:=0;
  write('Введіть число: '); read(a);
  while a<>0 do
  begin s:=s+a; inc(k);
    write('Введіть число: '); read(a)
  end; {while}
  s:=s/k; writeln('average=',s)
end.

```

За умовою задачі послідовність містить ненульові значення, тому внаслідок виконання програми отримаємо $k > 0$ і оператор $s := s/k$; виконається без помилок. Для більшої надійності програми можна виконати додаткову перевірку значення k і замість цього оператора присвоєння використати такий:

```
if k=0 then writeln('Послідовність починається нулем')
  else s:=s/k;
```

Зверніть увагу на спосіб введення даних у програмі *average*. Перший член зчитується ще перед початком циклу (щоб було що порівнювати з нулем в умові циклу). Наступні значення зчитуються в циклі вже після обробки попереднього значення. Отже, буде враховано кожен член послідовності, крім останнього – нуля, як і потрібно за умовою задачі.

3.2. Покрокове виведення даних

До окремої категорії задач належать ті, в яких йде мова не про обробку конкретної послідовності, а про обчислення і виведення на друк певним чином заданих послідовностей. Розглянемо кілька прикладів.

Задача 8. Протабулювати функції $\sin(x)$ та $\cos(x)$ на проміжку від 0 до π з кроком $\pi/12$. (Це означає отримати таблицю значень функцій для $x = 0, \pi/12, \pi/6, \dots, \pi$.)

У цій задачі необхідно обчислити три послідовності: $x_i = i \pi/12$, $s_i = \sin x_i$, $c_i = \cos x_i$ для $i = 0, 1, \dots, 12$. Для тимчасового зберігання членів цих послідовностей використаємо прості змінні x , s та c відповідно. Межі зміни індексу послідовностей відомі, тому застосуємо цикл з параметром:

```
program table;
  const n=12;
  var i:integer; x,s,c,h:real;
begin writeln('  x      sin x      cos x'); {друкуємо шапку}
      writeln('-----'); {таблиці}
  h:=pi/n; {крок зміни аргумента досить порахувати один раз}
  for i:=0 to n do {друкуємо саму таблицю}
  begin x:=h*i; s:=sin(x); c:=cos(x);
        writeln(x:6:2,s:10:4,c:10:4)
  end{for}
end.
```

Чергові значення x_i , s_i , c_i нема сенсу зберігати після їхнього надрукування, тому в програмі можна обійтися без масивів, використовуючи лише прості змінні.

Задача 9. Нехай послідовність $\{a_i\}$ задано співвідношеннями: $a_0 = 1$;

$$a_k = \frac{a_{k-1}}{k} + 0,2k - 1, \quad k = 1, 2, \dots. \text{ Дано } m \ (m > 1). \text{ Надрукувати } a_0, \dots, a_{n-1}, \text{ де } n - \text{ номер першого члена послідовності, для якого виконується умова } a_n > m.$$

Число n наперед невідомо, тому масив оголосити і використати не вдасться, проте він і не потрібен: для обчислення членів послідовності достатньо

однієї простої змінної a . Обчислення рекурентної формули з умови задачі можна задають таким оператором:

```
a:=a/k+0.2+k+1;
```

Змінна a в правій частині оператора містить значення a_{k-1} . У результаті виконання обчислень отримаємо нове значення, яке є значенням a_k . На наступному кроці воно замінить значення a_{k-1} у змінній a :

```
program succession;
  var a,m:real; k:integer;
begin write('Введіть m: '); readln(m);
  k:=0; a:=1; {задали значення A0}
  repeat writeln('a(',k,')=',a);
    inc(k); a:=a/k+0.2*k-1 {обчислили Ak}
  until a>m
end.
```

3.3. Обчислення за рекурентними формулами

Попередній приклад уже містив такі обчислення. Розглянемо складніші випадки.

Задача 10. Числа Фібоначчі задано рекурентними співвідношеннями:
 $f_0 = f_1 = 1; f_n = f_{n-1} + f_{n-2}, n = 2, 3, \dots$. Обчислити k -е число Фібоначчі ($k > 0$).

На відміну від попередньої задачі, тут для обчислення чергового члена послідовності потрібні значення не одного, а двох попередніх. Щоб побудувати алгоритм, використаємо такі змінні: n – для номера чергового числа Фібоначчі; a – для зберігання значення f_{n-2} ; b – для f_{n-1} і c – для f_n . Очевидно, що для $n = 2$ необхідно виконати

```
a:=1; b:=1; c:=a+b;
```

Тут a і b містять «старі» значення, а c – «нове». Після збільшення n значення c стане «старим», а значення, що міститься в a , – непотрібним для наступних обчислень. Щоб відобразити в алгоритмі «старіння» значень, виконують переприсвоєння:

```
a:=b; b:=c;
```

Тепер запишемо цілу програму:

```
program Fibonacci;
  var a,b,c:longint; n,k:integer;
begin write('Введіть номер числа: '); readln(k);
  b:=1; c:=1; {c=F1}
  for n:=2 to k do
  begin a:=b; b:=c; {значення a і b «постаріли»}
    c:=a+b {обчислили нове}
  end; {for}
```

```
writeln('f(',k,')=',c)
end.
```

У цій програмі використано так званий механізм «старе-нове» для реалізації обчислень за рекурентними формулами. Наведемо ще один приклад його використання.

Задача 11. Дано дійсні числа a, b, ε ($a > b > 0, \varepsilon > 0$). Послідовності $\{x_i\}, \{y_i\}$

$$\text{утворені за правилом } x_1 = a, \quad y_1 = b, \quad x_k = \frac{1}{2}(x_{k-1} + y_{k-1}), \\ y_k = \sqrt{x_{k-1}y_{k-1}}, \quad k = 2, 3, \dots. \text{ Знайти перше } x_n \text{ таке, що } |x_n - y_n| < \varepsilon.$$

Для зберігання значень x_k і y_k використаємо змінні u і v , а для значень x_{k-1}, y_{k-1} – змінні p, q . Тоді для $k = 1$ значення u і v необхідно прочитати, бо їх задано. Усі наступні обчислити в циклі:

```
program aveVSgeo;
var u,v,p,q,eps:real; k:integer;
begin k:=1; {початкові члени послідовностей треба прочитати}
write('Введіть a, b, eps: '); readln(u,v,eps);
while abs(u-v) >= eps do
begin inc(k); p:=u; q:=v; {«старіння» значень}
u:=(p+q)/2; v:=sqrt(p*q) {обчислення нових}
end; {while} writeln('x(',k,')=',u)
end.
```

Змінну k в цій програмі ми використали не як параметр циклу, а для обчислення номера члена послідовності $\{x_i\}$, для якого виконується умова задачі, і з метою унаочнення виведення результату.

4. Поєднання повторення з галуженням

Поєднання циклу з розгалуженням в одному алгоритмі необхідне для розв'язування задач, що вимагають перевірки певної умови для кожного члена заданої послідовності значень. Наприклад, для обчислення кількості парних чисел серед усіх заданих, для відшукування найбільшого чи найменшого члена послідовності тощо.

4.1. Скільки є «правильних» серед усіх заданих?

У задачах, описаних у цьому параграфі, необхідно обчислити кількість членів заданої послідовності, які задовільняють певний критерій. Першу задачу можна розв'язати за допомогою покрокового введення і перевірки даних з використанням простих змінних, а для отримання розв'язку другої задачі необхідно використати масив і зберігати в пам'яті усю послідовність.

Задача 12. Дано натуральні числа n, a_1, \dots, a_n . Обчислити кількість членів a_i послідовності $\{a_i\}$, що задовільняють умову $2^i < a_i < i! + 3$.

Щоб розв'язати задачу, необхідно обчислити степені двійки і факторіали. Очевидно, що 2^i легко обчислити, якщо відомо 2^{i-1} , бо $2^i = 2 \times 2^{i-1}$. Так само $i! = i \times (i-1)!$. Степінь двійки будемо накопичувати у змінній p , а факторіал – у

змінній f . Кожне значення a_i розглядають тільки один раз, тому для його зберігання використаємо просту змінну a . Перед початком циклу необхідно ініціалізувати змінні p , f і лічильник «правильних» членів послідовності – змінну k :

```

program condition;
var n,a,i,k:integer; p,f:longint;
begin write('Введіть кількість чисел: '); readln(n);
    p:=1; f:=1; k:=0;
    for i:=1 to n do
      begin write('Введіть ',i,'-е число: '); readln(a);
        p:=p*2; f:=f*i;
        if (p<a) and (a<f+3) then inc(k)
      end; {for} writeln('k=',k)
end.

```

Задача 13. Дано натуральне число n ($2 \leq n \leq 20$) і дійсні числа r, a_1, \dots, a_n . Скільки серед точок $(a_1, a_n), (a_2, a_{n-1}), \dots, (a_n, a_1)$ є таких, що належать колові радіуса r з центром у початку координат?

Позначимо відстань від точки (a_i, a_{n+1-i}) до початку координат через d_i .

Відомо, що $d_i = \sqrt{a_i^2 + a_{n+1-i}^2}$. Очевидно, що $d_i = d_{n+1-i}$. Тому якщо точка (a_i, a_{n+1-i}) належить колові, то й точка (a_{n+1-i}, a_i) теж йому належить. Отже, умову $d_i \leq r$ перевіряють не для усіх n точок, а для $n/2$ точок, якщо n парне, чи для $[n/2]+1$, якщо воно непарне. Зауважимо також, що для зменшення кількості обчислень можна перевіряти не умову $d_i \leq r$, а $d_i^2 \leq r^2$. Тоді відпаде необхідність щоразу добувати корінь, а величину r^2 можна обчислити один раз перед циклом.

Для обчислення d_1 нам необхідні значення a_1 і a_n , тому для зберігання значень послідовності $\{a_i\}$ використаємо масив. Як оголосити розмір цього масиву? Очевидно, необхідно використати найбільше можливе значення n (за умовою дорівнює 20), щоб пам'яті вистачило для будь-яких вхідних даних:

```

program inCircle;
type vector=array[1..20] of real;
var a:vector; r:real; k,i,n,n1:byte;
begin write('Введіть кількість чисел: '); readln(n);
    write('Введіть радіус: '); readln(r);
    write('Введіть ',n,' чисел: ');
    for i:=1 to n do read(a[i]); {введення даних закінчено}
    n1:=n+1; r:=sqr(r); k:=0;
    for i:=1 to n div 2 do {перевіримо першу половину точок}
      if sqr(a[i])+sqr(a[n1-i])<=r then inc(k);
    k:=k*2;
    if odd(n) then {перевіримо середню точку}
      if 2*sqr(a[n div 2+1])<=r then inc(k);
    writeln('Точок в колу є ',k)
end.

```

4.2. Максимальний елемент послідовності

Відшукування найбільшого чи найменшого елемента послідовності є складовою частиною багатьох задач, тому докладно розглянемо його алгоритм.

Уявіть, що ви стоїте біля початку довгого столу, на якому рядочком лежать чудові яблука. Вам дозволено взяти собі одне. Яке? Звичайно ж найбільше! Як його знайти, якщо кінця столу і яблук, які лежать там, добре не видно? Беріть до рук перше, яке вам сподобалось, і гайда вздовж столу: якщо знайдеться більше, то заміняєте!

Приблизно так само можна сформулювати алгоритм відшукування найбільшого серед чисел a_1, a_2, \dots, a_n . Для зберігання найбільшого значення використаємо додаткову змінну. Назвемо її, наприклад, b . Початковим значенням для b може бути значення будь-якого члена послідовності, проте найзручніше вибрати перший. Далі перебираємо усі інші a_i (з другого по n -ий) і перевіряємо, чи не знайдеться елемент, значення якого більше за b . Якщо так, то про старе значення b можна забути, а замість нього запам'ятати знайдене і продовжити порівняння. Операторами мови Pascal це можна записати так:

```
b:=a[1];
for i:=2 to n do if a[i]>b then b:=a[i];
```

Чи обов'язково початковим значенням для b має бути a_1 ? Ні. Можна шукати найбільший елемент і так:

```
b:=a[n];
for i:=1 to n-1 do if a[i]>b then b:=a[i];
```

або взяти за початкове будь-яке інше значення a_i , проте тоді параметр циклу мав би змінюватись від 1 до n .

Чи можна схожим чином шукати найменший елемент послідовності? Звичайно! Для цього достатньо змінити в умові знак «>» на знак «<».

Чи можна цей алгоритм пристосувати для розв'язування складнішої задачі: знайти не тільки значення, але й номер найбільшого члена послідовності? Так. Для цього необхідно кожного разу, коли ми запам'ятовуємо якесь значення a_i в змінній b , запам'ятовувати і його номер i (наприклад, у змінній k):

```
b:=a[1]; k:=1;
for i:=2 to n do
if a[i]>b then begin b:=a[i]; k:=i end;
```

Послідовність $\{a_i\}$ може містити однакові значення, тому може статись так, що найбільше значення не буде єдиним. Номер якого з них знайде описаний алгоритм? Зміна значень b і k відбудеться тільки тоді, коли $a_i > b$, і не відбудеться, коли $a_i = b$. Тому, очевидно, змінна k міститиме номер *першого* за порядком найбільшого елемента. З метою отримання номера *останнього* найбільшого елемента можна почати перегляд послідовності з кінця або в умові замінити знак «>» на « \geq ».

Задача 14. Дано дійсні a_1, a_2, \dots, a_{50} . Замінити місцями перший і максимальний члени послідовності, надрукувати перетворену послідовність.

Ми вже майже все вміємо робити для розв'язання цієї задачі: завантажувати дані в елементи масиву, знаходити значення і номер найбільшого. Залишилось тільки відповісти на запитання, як замінити місцями значення двох змінних a і b ? Оператор $a:=b$ чи $b:=a$ одразу ж витре значення однієї зі змінних, тому для здійснення обміну використовують третю змінну c :

```
c:=a; {тимчасово зберегли значення a в змінній c}
a:=b; {значення b уже на своєму новому місці}
b:=c; {обмін завершено}
```

Згідно з описаним раніше алгоритмом номер максимального елемента послідовності буде записано в змінну k . Щоб замінити місцями a_1 і a_k , нам у пригоді стане змінна b : вона містить копію значення a_k . Остаточо отримаємо програму:

```
program change;
  const n=50;
  var a:array[1..n]of real; b:real;
      i,k:byte;
begin write('Введіть ',n,' чисел: ');
  for i:=1 to n do read(a[i]);
  b:=a[1]; k:=1;
  for i:=2 to n do
    if a[i]>b then begin b:=a[i]; k:=i end;
  a[k]:=a[1]; a[1]:=b;
  writeln('Змінена послідовність:');
  for i:=1 to n do write(a[i]:10:4); writeln
end.
```

У попередній задачі ми не знали точно кількість членів заданої послідовності, тому пам'ять для неї у програмі *inCircle* виділяли з запасом. У програмі *change* кількість членів послідовності (і відповідний розмір масиву) задано константою, бо в умові задачі 14 зазначено, що їхня кількість дорівнює 50.

4.3. Перевірка впорядкованості

Іноді початківця може поставити у скрутну ситуацію така проста задача:

Задача 15. Дано дійсні a_1, a_2, \dots, a_{50} . Перевірити, чи утворюють вони зростаючу послідовність.

Справді, задача має особливість: щоб дати ствердну відповідь, необхідно пересвідчитись, що для кожної пари сусідів виконано $a_{i-1} < a_i$, а для отримання негативної відповіді достатньо, щоб ця умова порушилась хоча б один раз. Тому цикл з перевітками можна завершувати, коли у черговій парі сусідів порушено умову впорядкованості, або коли перебрали всю послідовність. Тоді відповідь на

запитання задачі легко отримати, перевіривши, чи досягнуто кінець послідовності:

```

program growth;
const n=50;
var a:array[1..n]of real; i:byte;
begin write('Введіть ',n,' чисел: ');
  for i:=1 to n do read(a[i]);
  i:=2;
  while (a[i-1]<a[i])and(i<=n) do inc(i);
  if i>n then writeln('Послідовність зростаюча')
  else writeln('Умову впорядкованості порушено після a',i)
end.

```

Якщо послідовність не є зростаючою, то цикл завершиться ще до того, як значення i перевищить n , і на друк буде виведено повідомлення про номер елемента послідовності, після якого вперше порушено умову впорядкованості.

Очевидно, що перевірити впорядкованість заданої послідовності за спаданням можна за допомогою такого ж алгоритму. Досить лише замінити умову $a_{i-1} < a_i$ на $a_{i-1} > a_i$.

4.4. Пошук місця елемента послідовності

Впорядкованість послідовності значень за зростанням (чи неспаданням) є досить корисною властивістю. Якщо до такої послідовності доводиться долучати нові значення, що часто трапляється на практиці, то робити це треба так, щоб не порушити впорядкованості.

Задача 16. Дано впорядкований за незростанням масив цілих або дійсних чисел $a_1 \leq a_2 \leq \dots \leq a_n$ і деяке число b (відповідно ціле або дійсне), для якого необхідно знайти таке місце серед чисел a_1, a_2, \dots, a_n , щоб після вставлення числа b на це місце впорядкованість не порушилася.

Цю задачу називають задачею пошуку місця елемента. Для $b \in n+1$ можливість: $b \leq a_1, a_1 < b \leq a_2, \dots, a_{n-1} < b \leq a_n, a_n < b$, і розв'язком задачі пошуку місця елемента b буде відповідно одне з чисел $1, 2, \dots, n, n+1$, яке вказує індекс для розташування доданого елемента. Для розв'язування цієї задачі використовують різні алгоритми.

Лінійний пошук. Переглянемо масив a , починаючи з його першого елемента, і перевіримо, чи $a_i < b$. Перше значення i , для якого вона не виконується, і є номером елемента масиву, в який необхідно записати b . Для b потрібно «звільнити» місце, перемістивши значення a_i, \dots, a_n на один елемент далі. Такі переміщення зручно починати з кінця масиву. Якщо ж умову виконано для усіх елементів масиву, то b необхідно дописати після останнього елемента масиву:

```

program forwardInsert;
const n=10;
type vector=array[1..n]of integer;
var a:vector; b:integer; i,j:byte;

```

```

begin write('Введіть послідовність з ',n-1,' чисел: ');
  for i:=1 to n-1 do read(a[i]);
  write('Введіть нове число: '); readln(b);
  i:=1; {шукаємо місце для b}
  while (a[i]<b)and(i<n) do inc(i);
  for j:=n-1 downto i do a[j+1]:=a[j]; {посуваємо «хвіст»}
  a[i]:=b; {вставляємо b в масив}
  for i:=1 to n do write(a[i]:5); writeln
end.

```

Якщо перевірку умови $a_i < b$ почати з останнього елемента масиву, то пошук і звільнення місця можна об'єднати в одному циклі:

```

program simpleInsert;
const n=10;
type vector=array[1..n]of integer;
var a:vector; b:integer; i:byte;
begin write('Введіть послідовність з ',n-1,' чисел: ');
  for i:=1 to n-1 do read(a[i]);}
  write('Введіть нове число: '); readln(b);
  i:=n-1; {шукаємо і звільняємо місце}
  while (a[i]>b)and(i>=1) do
    begin a[i+1]:=a[i]; dec(i)
    end; {while}
  a[i+1]:=b; {вставляємо b в масив}
  for i:=1 to n do write(a[i]:5); writeln
end.

```

Бінарний пошук. Для відшукування місця кожен з описаних вище алгоритмів виконає n порівнянь. Для пришвидшення пошуку можна використати метод поділу масиву навпіл. За цим методом спочатку задають межі пошуку місця для b рівними 1 і $n+1$, далі ці межі крок за кроком стискають так: порівнюють b з a_s , де s – ціла частина середнього арифметичного меж пошуку; якщо $a_s < b$, то замінити нижню межу на $s+1$, у протилежному випадку замінити на s верхню межу; коли межі збігаються, то їхнє значення і буде результатом пошуку місця. Число порівнянь, яких потребує цей алгоритм, не перевищує $\lceil \log_2(n+1) \rceil + 1$:

```

program binaryInsert;
const n=10;
type vector=array[1..n]of integer;
var a:vector; b:integer; i,p,q,s:byte;
begin write('Введіть послідовність з ',n-1,' чисел: ');
  for i:=1 to n-1 do read(a[i]);
  write('Введіть нове число: '); readln(b);
  p:=1; q:=n; {початкові межі пошуку}
  while (p<>q) do
    begin s:=(p+q)div 2; {шукаємо середній елемент}
      if a[s]<b then p:=s+1 else q:=s {і порівнюємо з ним}
    end; {while}
  for i:=n-1 downto p do a[i+1]:=a[i]; {посуваємо «хвіст»}
  a[p]:=b; {вставляємо елемент}
  for i:=1 to n do write(a[i]:5); writeln
end.

```

5. Вкладені цикли в матричних задачах

В алгоритмах попереднього параграфу для обробки послідовностей значень ми використовували прості цикли. Одного циклу було достатньо, бо послідовність, вектор чи масив a_1, a_2, \dots, a_n є одновимірними об'єктами. Якщо ж у задачі необхідно перебрати елементи деякої матриці – двовимірного об'єкта, – то одного циклу замало. Алгоритм, який перебирає елементи матриці, повинен використати два цикли: один – для перебору рядків (чи стовпців) матриці, а другий – для перебору елементів рядка (стовпця). Такі алгоритми необхідні, наприклад, для побудови матриці за заданим правилом, для відшукування її максимального елемента, для виконання дій над матрицями тощо.

5.1. Побудова матриць

На практиці трапляються випадки, коли матриці задають за допомогою правила обчислення їхніх елементів. Наприклад, $a_{ij} = i + j - 1$, або $a_{ij} = \sin i + \cos j$ тощо. Щоб використати такі матриці для обчислень, необхідно спочатку сформувати їх у пам'яті комп'ютера. Наведемо кілька прикладів такого формування.

Задача 17. Дано натуральне n . Побудувати і надрукувати одиничну матрицю n -го порядку.

У мові Pascal розподіл пам'яті для змінних, у тому числі для масивів, виконують статично, тому задане число n для спрощення зображають у програмі константою. (Константи можна легко надати необхідне значення.) Пригадаємо, що на головній діагоналі одиничної матриці розташовано одиниці, а всі інші елементи є нулями. Як довідатися, чи елемент належить головній діагоналі? Просто: його індекси рівні між собою:

```

program buildUnite;
  const n=10;
  var a:array[1..n,1..n]of integer;
      i,j:byte;
begin {формування матриці}
  for i:=1 to n do
    for j:=1 to n do
      if i=j then a[i,j]:=1           {головна діагональ}
        else a[i,j]:=0;
        {виведення матриці}
    for i:=1 to n do
      begin for j:=1 to n do write(a[i,j]);
            writeln      {закінчили друкувати рядок матриці}
      end {for i}
end.

```

Цей алгоритм досить очевидний, однак не найкращий: будуючи за ним матрицю, доведеться виконати n^2 перевірок індексів. Як уникнути зайвих перевірок? Спробуємо підійти до отримання одиничної матриці по-іншому: заповнимо спочатку її головну діагональ, а потім – верхній та нижній

трикутники, враховуючи їхню взаємну симетрію. Діагональ – це одна лінія (одновимірний об’єкт), тому для перебору її елементів використаємо простий цикл. Верхній трикутник містить елементи рядків з 1-го по $n-1$ -ий – переберемо ці рядки за допомогою одного циклу. У i -му рядку є елементи a_{ij} , де $j = i+1, \dots, n$ – переберемо їх за допомогою вкладеного циклу:

```

program simmetricBuildUnite;
  const n=10;
  var a:array[1..n,1..n]of integer;
      i,j:byte;
begin {формування матриці}
  for i:=1 to n do a[i,i]:=1;    {головна діагональ}
  for i:=1 to n-1 do
  for j:=i+1 to n do
  begin a[i,j]:=0;                {трикутники над i}
      a[j,i]:=0 end;           {під діагоналлю}
    {виведення матриці}
  for i:=1 to n do
  begin for j:=1 to n do write(a[i,j]);
    writeln      {закінчили друкувати рядок матриці}
  end {for i}
end.

```

Задача 18. Дано натуральне n . Отримати квадратну матрицю

$$\text{порядку } n \text{ вигляду } \begin{pmatrix} 1 & 2 & \dots & n-1 & n \\ 2 & 3 & \dots & n & 0 \\ \dots & \dots & \dots & \dots & \dots \\ n-1 & n & \dots & 0 & 0 \\ n & 0 & \dots & 0 & 0 \end{pmatrix}.$$

У такій матриці нижче від побічної діагоналі розташовано нулі. Як розпізнати побічну діагональ? Для індексів її елементів виконується рівність $i + j = n + 1$. Щоб не виконувати зайвих перевірок, сформуємо цю матрицю частинами, як у попередній програмі:

```

program buildMatrix;
  const n=10; n1=n+1;
  var a:array[1..n,1..n]of integer;
      i,j:byte;
begin {формування матриці}
  for i:=1 to n do a[i,n1-i]:=n; {побічна діагональ}
  for i:=1 to n-1 do
  for j:=1 to n-i do
  begin a[i,j]:=i+j-1;           {трикутники над i}
      a[j,i]:=0 end;         {під діагоналлю}
    {виведення матриці}
  for i:=1 to n do
  begin for j:=1 to n do write(a[i,j]);
    writeln      {закінчили друкувати рядок матриці}
  end {for i}
end.

```

Задача 19. Дано дійсні числа a_1, \dots, a_{64} . Отримати дійсну квадратну матрицю порядку 8, елементами якої є ці числа, розташовані за схемою, зображеною на рис. 1.

Цю задачу можна розв'язати принаймні двома способами: прочитати послідовність a_1, \dots, a_{64} , і перебрати елементи матриці, присвоївши їм відповідні значення цієї послідовності; перебрати елементи послідовності і вставити їх у відповідні місця матриці. Опишемо оба способи.

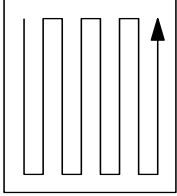


Рис. 1

Щоб діяти за першим способом, для зберігання членів заданої послідовності використаємо одновимірний масив a . Матрицю, яку будемо будувати, назвемо b . Який елемент a_k відповідає елементові b_{ij} ? Легко переконатися, що для непарних стовпців матриці виконується співвідношення $k = (j-1)*8+i$, а для парних – співвідношення $k = j*8-i+1$. Тепер можна сформулювати матрицю, перебираючи її за стовпцями.

```

program matrixFromSequence;
  const n=8; m=sqr(n);
  var a:array[1..m]of real;
      b:array{1..n,1..n}of real;
      l,i,j:byte;
begin write('Введіть',m,'чисел: ');
  for i:=1 to m do read(a[i]); readln;
  for j:=1 to n do                                {формуємо матрицю}
    if odd(j) then                                  {непарний стовпець}
      begin l:=(j-1)*n;
        for i:=1 to n do b[i,j]:=a[l+i]
      end else                                       {парний стовпець}
      begin l:=j*n+1;
        for i:=1 to n do b[i,j]:=a[l-i]
      end;{if & for j}
  for i:=1 to n do                                {виведення матриці}
    begin for j:=1 to n do write(a[i,j]);
      writeln      {закінчили друкувати рядок матриці}
    end {for i}
end.

```

Щоб реалізувати другий спосіб, необхідно встановити обернені співвідношення між порядковим номером члена заданої послідовності та індексами елемента матриці. Щоб уникнути складних обчислень, простежимо, як рухається «змійка», зображена на рис. 1, матрицею. Очевидно, що в непарних стовпцях вона опускається, у парних – підіймається. Тобто напрям руху змінюється тоді, коли заповнено черговий стовпець з восьми елементів і «змія» переходить до наступного. Заповнення розпочинається з елемента b_{11} , а закінчується елементом b_{18} , коли розташовано усі члени заданої послідовності. Для послідовного зчитування заданих чисел використовують просту змінну.

```

program sequenceIntoMatrix;
  const n=8; m=sqr(n);
  var a:real;
      b:array{1..n,1..n}of real;

```

```

    k,i,j,step:byte;
begin write('Введіть',m,'чисел: ');
    i:=1; j:=1;           {координати початкового елемента}
    step:=1;              {спочатку напрям руху - додатній}
    for k:=1 to m do      {формуємо матрицю}
    begin read(a); b[i,j]:=a; {розташували чергове число}
        if k mod 8=0 then {стовпець заповнено}
            begin inc(j);   {перейшли до нового стовпця}
                step:=-step end {і змінили напрям}
            else i:=i+step {просуваємся стовпцем}
        end; {for k}
    for i:=1 to n do      {виведення матриці}
    begin for j:=1 to n do write(a[i,j]);
        writeln           {закінчили друкувати рядок матриці}
    end {for i}
end.

```

Ця програма виконує більше перевірок, ніж попередня, проте вимагає меншого обсягу пам'яті для даних.

Задача 20. Дано квадратну матрицю A розміру $n \times n$, побудовану з дійсних чисел. Отримати дійсну матрицю B такого ж розміру, кожен елемент b_{ij} якої дорівнює сумі елементів матриці A , розташованих в області, визначеній індексами i та j , як вказано на рис. 2 (область заштриховано, включаючи межі).

Умова цієї задачі складніша від попередніх, і побудова алгоритму може викликати певні труднощі. Щоб полегшити собі завдання, умовно розкладемо цю задачу на складові. Отримати матрицю – отримати кожен її елемент. Перебрати матрицю поелементно ми вже вміємо: для цього необхідно використати два вкладені цикли. Як отримати елемент b_{ij} ? Треба просумувати ті елементи матриці A , які належать області, схематично зображеній на рис. 2. Ця область є прямокутною частиною матриці A – двовимірним об'єктом. Отже необхідно знову ж таки два вкладені цикли, щоб перебрати елементи області. А які її межі? Очевидно, що перший індекс елементів області змінюється від 1 до i , а другий – від j до n . Так для b_{1n} відповідна область містить тільки один елемент a_{1n} , а для b_{n1} – матрицю A загалом. Обміркувавши все це, запишемо програму:

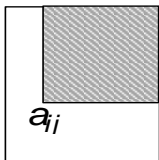


Рис. 2

```

program regionsSumming;
    const n=10;
    var a,b:array[1..n,1..n] of real;
        i,j,k,l:byte; s:real;
    begin writeln('Введіть матрицю A');
        for i:=1 to n do
        for j:=1 to n do read(a[i,j]);
            readln;           {закінчили введення матриці A}
        for i:=1 to n do     {перебираємо елементи матриці B}
        for j:=1 to n do
        begin s:=0;           {сумуємо елементи відповідної області}
            for k:=1 to i do
            for l:=j to n do s:=s+a[k,l];
        end
    end.

```

```

    b[i,j]:=s
end; {for j&i}
for i:=1 to n do           {виведення матриці}
begin for j:=1 to n do write(a[i,j]);
    writeln                {закінчили друкувати рядок матриці}
end {for i}
end.

```

Можна тішитися написаною програмою: все-таки чотири вкладені цикли подужали! Проте через деякий час зауважимо, що ця програма виконує досить багато зайвих обчислень: для кожного b_{ij} вона додає всі a_{kl} «як вперше», починаючи від 0. А чи не можна якось використати уже обчислені суми для отримання нових елементів матриці B ? Очевидно, у цьому випадку починати заповнювати її необхідно з правого верхнього кута. Легко переконатися, що для елементів останнього стовпця матриці B виконуються співвідношення $b_{1n} = a_{1n}$, $b_{in} = b_{i-1,n} + a_{in}$, $i = 2, \dots, n$, а для усіх інших – співвідношення $b_{ij} = b_{i,j+1} + s_{ij}$, $j = n-1, \dots, 1$, де $s_{1j} = a_{1j}$, $s_{ij} = s_{i-1,j} + a_{ij}$, $i = 1, \dots, n$. Економічний щодо обчислень алгоритм реалізує така програма:

```

program economySumming;
const n=10;
var a,b:array[1..n,1..n]of real;
    i,j,k,l:byte; s:real;
begin writeln('Введіть матрицю A');
    for i:=1 to n do
    for j:=1 to n do read(a[i,j]);
        readln;                {закінчили введення матриці A}
b[1,n]:=a[1,n];                {формуємо останній стовпець}
    for i:=2 to n do b[i,n]:=b[i-1,n]+a[i,n];
    for j:=n-1 downto 1 do
begin s:=0;                    {сумуємо елементи чергового стовпця}
    for i:=1 to n do {i отримуємо "нове" зі "старого"}
        begin s:=s+a[i,j]; b[i,j]:=b[i,j+1]+s
        end {for i}
    end; {for j}
    for i:=1 to n do           {виведення матриці}
begin for j:=1 to n do write(a[i,j]);
    writeln                {закінчили друкувати рядок матриці}
end {for i}
end.

```

Виявляється, що розумне використання раніше здобутого досвіду дає змогу значно спростити життя!

5.2. Дії матричної алгебри

У матричній алгебрі визначено операції додавання та віднімання матриць, множення матриці на скаляр або на іншу матрицю, транспонування тощо. Як ці операції реалізувати програмно? Продемонструємо це на прикладах. Щоб не загромождувати тексти програм практично стандартними циклами з поелементним введенням-виведенням матриць, обмежимося лише

інформативною частиною алгоритмів. Використовуватимемо в них такі оголошення:

```
const n=10; m=12; t=9;
type matr1=array[1..n,1..m]of real;
      matr2=array[1..m,1..t]of real;
      matr3=array[1..n,1..t]of real;
      square=array[1..n,1..n]of real;
var a,b,c:matr1; d:square;
    p:matr2; q:matr3;
    i,j,k:byte; s:real;
```

Задача 21. Дано дійсні матриці A , B . Обчислити матрицю C , якщо $C = A + B$.

Відомо, що матриці додаються поелементно, тому цю задачу легко розв'язати так:

```
for i:=1 to n do
for j:=1 to n do c[i,j]:=a[i,j]+b[i,j];
```

Легко обчислити також різницю матриць $A - B$: достатньо у цьому фрагменті програми замінити знак «+» на знак «-»; чи множення матриці A на скаляр s : необхідно замінити оператор присвоєння у вкладеному циклі на оператор

```
c[i,j]:=a[i,j]*s;
```

Задача 22. Дано дійсні матриці A , P . Обчислити матрицю Q , якщо $Q = A \times P$.

Якщо матриця A має розмір $n \times m$, а матриця $P - m \times t$, то їхнім добутком

$$q_{ij} = \sum_{k=1}^m a_{ik} p_{kj}$$

буде $n \times t$ матриця Q , елементи якої обчислюють за формулою $q_{ij} = \sum_{k=1}^m a_{ik} p_{kj}$, $i = 1, \dots, n, j = 1, \dots, t$. Отже, для обчислення матриці Q доведеться використати три вкладені цикли: два – для перебору q_{ij} і третій – для накопичення суми:

```
for i:=1 to n do
for j:=1 to t do
begin s:=0;
  for k:=1 to m do s:=s+a[i,k]*p[k,j];
  q[i,j]:=s
end; {for j&i}
```

Задача 23. Транспонувати дану квадратну матрицю D .

Операція транспонування полягає в заміні рядків матриці відповідними стовпцями, і навпаки. Тобто внаслідок транспонування значення елементів d_{ij} та d_{ji} повинні помінятися місцями. Таку заміну можна було б виконати поелементно. Однак **помилково** використовувати з цією метою такий алгоритм:

```
for i:=1 to n do
for j:=1 to n do
```

```
begin s:=d[i,j]; d[i,j]:=d[j,i]; d[j,i]:=s
end; {for j&i}
```

Спробуйте, і ви пересвідчитесь, що матриця D залишиться без змін. Адже кожна пара елементів d_{ij} та d_{ji} візьме участь в обмінах двічі! Під час транспонування матриці її головна діагональ повинна залишатися без змін, а обміни необхідно виконати тільки для відповідних елементів трикутників, розташованих над і під діагоналлю. Тому правильно транспонувати матрицю можна так:

```
for i:=1 to n-1 do
for j:=i+1 to n do
begin s:=d[i,j]; d[i,j]:=d[j,i]; d[j,i]:=s
end; {for j&i}
```

5.3. Порівняння та переміщення елементів матриці

Про відшукування максимального елемента послідовності ми докладно говорили у п. 3.2. Для матриці можна використати той самий підхід: взяти значення котрогось з елементів матриці за початкове значення найбільшого і порівняти його з усіма іншими, виконуючи переприсвоєння, якщо знайдеться більше. Це можна зробити, наприклад, так (тут a – $n \times m$ матриця):

```
b:=a[1,1];
for i:=1 to n do
for j:=1 to m do
if a[i,j]>b then b:=a[i,j];
```

Все – як і раніше, тільки збільшилась кількість індексів і циклів. До речі, поміркуйте, чому не можна починати цикл по i чи цикл по j від 2?

Задача 24. Дано матрицю A розміру $n \times m$. Переставити її рядки і стовпці так, щоб максимальний за модулем елемент став у її лівий верхній кут.

Таку задачу доводиться розв'язувати на практиці, наприклад, у під час відшукування розв'язку системи лінійних алгебричних рівнянь методом Гауса з вибором головного елемента. Щоб виконати необхідні перестановки, потрібно знати індекси k та l максимального за модулем елемента матриці та поміняти місцями елементи першого та k -го рядків і першого та l -го стовпців. Якщо використати допоміжний одновимірний масив, то зможемо поміняти місцями рядки матриці як значення простих змінних. Обмін стовпців виконаємо поелементно:

```
program moveMax;
const n=10; m=12;
type vector=array[1..m] of real;
matrix=array[1..n] of vector;
var a:matrix; b,c:real;
v:vector; i,j,k,l:byte;
begin writeln('Введіть матрицю A');
```

```

for i:=1 to n do
for j:=1 to m do read(a[i,j]);
    readln;                                {закінчили введення матриці A}
b:=abs(a[1,1]);                            {початкове значення максимального}
k:=1; l:=1;                                {та його координати}
for i:=1 to n do                          {переглядаємо всю матрицю}
for j:=1 to m do
    if abs(a[i,j])>b then                  {знайшли більше}
    begin b:=abs(a[i,j]); k:=i; l:=j
    end; {if & for j&i}
if k<>l then                              {міняємо місцями рядки}
    begin v:=a[l]; a[l]:=a[k]; a[k]:=v end;
if l<>1 then                              {стовпці міняємо місцями}
    for i:=1 to n do                        {поелементно}
    begin c:=a[i,l]; a[i,l]:=a[i,1]; a[i,1]:=c end;
for i:=1 to n do                          {виведення матриці}
begin for j:=1 to m do write(a[i,j]);
    writeln                                {закінчили друкувати рядок матриці}
end {for i}
end.

```

Разом з максимальними чи мінімальними елементами матриці часто на практиці доводиться знаходити так звані сідлові елементи. Сідловим називають найменший серед максимальних елементів рядків матриці (або найбільший серед мінімальних елементів рядків матриці). Як його відшукати? Про що, взагалі, йдеться, коли стоїть завдання «знайти значення (і, можливо, координати) сідлового елемента заданої матриці»? За означенням, сідловий елемент є найменшим з чисел певної послідовності. Отже, мова йде про відшукування найменшого. Таку задачу ми уже розв'язували! Які ж числа є в тій послідовності, з якої необхідно вибрати найменше? Кожне з них є найбільшим елементом відповідного рядка матриці. Отже, щоб отримати цю послідовність, необхідно просто n разів розв'язати задачу з відшукування найбільшого. Це ми також вміємо робити, тому задачу можна розв'язати! Наведемо змістовну частину алгоритму відшукування сідлового елемента матриці (використано змінні, оголошення яких такі, як у попередній програмі).

```

{початковим значенням сідлового елемента є максимальний елемент}
b:=a[1,1];                                {першого рядка матриці - знайдемо його!}
for j:=2 to m do
    if a[1,j]>b then b:=a[1,j];
for i:=2 to n do                            {а тепер переглянемо всі інші рядки матриці}
begin c:= a[i,1];                            {і знайдемо їхні максимальні елементи}
    for j:=2 to m do
        if a[i,j]>c then c:=a[i,j];
        {та порівнюємо їх з поточним значенням сідлового елемента}
    if c<b then b:=c
end;
{b містить остаточне значення сідлового елемента}

```

Сподіваємося, що розв'язки задач, наведені у цьому параграфі, допоможуть читачеві навчитися знаходити просте в складному, застосовувати

знайомі прості алгоритми для розв'язування складніших задач і не губитися перед заплутаними на перший погляд умовами.

6. Основні алгоритми сортування

У цьому параграфі ми розглянемо питання, яке часто виникає в програмуванні: перерозміщення елементів заданого масиву у зростаючому чи спадному порядку. Уявіть, як важко було б користуватися словником, якщо б слова у ньому не розташовувалися в алфавітному порядку. Так само від порядку, в якому зберігаються дані в пам'яті комп'ютера, багато в чому залежить швидкодія та простота алгоритмів, що використовуються для їхньої обробки.

За тлумачним словником слово «*сортування*» – це «розподіл, відбір за сортами; поділ на категорії, сорти, розряди», але програмісти традиційно використовують це слово у набагато вужчому значенні, позначаючи ним розташування предметів у зростаючому або спадному порядку. Іноді такий процес називають *впорядкуванням*. Якщо послідовність даних може містити однакові значення, то говорять про перерозподіл даних за неспаданням або незростанням.

Важко переоцінити значення алгоритмів сортування, особливо сьогодні. Передусім їх використовуються для впорядкування даних та їхніх ключів у всеможливих базах даних для забезпечення швидкого доступу до даних. Сортування використовують з метою спрощення формул у алгоритмах комп'ютерної алгебри, пришвидшення роботи оптимізуючих компіляторів та в багатьох інших галузях програмування.

Методи сортування чудово ілюструють ідею *аналізу алгоритмів*, тобто ідею, яка дає змогу оцінити робочі характеристики алгоритмів, а, отже, свідомо вибирати серед, здавалося б, рівноцінних методів найоптимальніший. Якщо розмір бази даних, яка підлягає сортуванню, вимірюється десятками чи сотнями тисяч записів, то на перше місце стає швидкодія алгоритму та його вимоги до додаткової пам'яті. Сьогодні відомо близько десяти основних алгоритмів сортування і до сотні їхніх модифікацій та видозмін. Така різноманітність зумовлена тим, що неможливо придумати один алгоритм, найкращий на всі випадки життя, а впорядковувати дані доводиться за найрізноманітніших умов. Часто доводиться враховувати особливості зберігання та початкового розподілу невідсортованих даних, жорсткі обмеження на кількість порівнянь чи переміщень даних тощо.

У цьому параграфі ми розглянемо таку задачу:

Задача 25. Дано масив цілих чисел a_1, a_2, \dots, a_n . Переставити його елементи так, щоб виконувалась умова $a_1 \leq a_2 \leq \dots \leq a_n$.

Ми наведемо три основні алгоритми впорядкування числового масиву: *сортування вибором*, *сортування обмінами* та *сортування вставками*. Ці алгоритми належать до «абетки програміста». Ми докладно обговоримо принципи їхньої роботи та їхні властивості, дамо рекомендації щодо використання.

6.1. Сортування вставками

У п. 3.4 ми вже розглядали задачу щодо відшукування місця вставлення нового елемента у впорядковану послідовність. Ідею вставки можна використати для побудови алгоритму впорядкування масиву. Якщо б початок масиву містив впорядковані значення, то решту значень можна було б вставити у цю частину, підтримуючи порядок, і, отже, отримати відсортований масив.

Для початку обчислень вважатимемо, що впорядкована частина складається тільки з першого елемента масиву. Далі необхідно перебрати усі інші елементи – від другого до останнього – і вставити їх на відповідне місце у впорядковану частину. Зрозуміло, що у цьому випадку ця частина видовжуватиметься щоразу на один елемент, аж поки не займе весь масив.

Ми навели кілька алгоритмів відшукування місця елемента: два варіанти лінійного пошуку та бінарний пошук. Залежно від того, який з них використано для впорядкування масиву, отримують різні модифікації алгоритму сортування: прості вставки, бінарні вставки чи інші.

Використаємо лінійний пошук, у якому в одному циклі об'єднано порівняння і переміщення елементів (такий алгоритм пошуку в п. 3.4 було реалізовано програмою *simpleInsert*) і отримуємо алгоритм *сортування простими вставками*. Оформимо його у вигляді процедури *sortInsert*:

```

program sortBySimpleInsert;
  const n=10;
  type vector=array[1..n]of integer;
  var a:vector; i:byte;

procedure sortInsert(var a:vector);
{      ВПОРЯДКУВАННЯ МАСИВУ а ПРОСТИМИ ВСТАВКАМИ      }
  var b:integer; i,j:byte;
{спочатку впорядкованим є лише перший елемент послідовності}
{переберемо всі інші і кожен з них вставимо на відповідне місце}
begin for j:=2 to n do           {шукаємо місце для j-го елемента}
  begin b:=a[j]; i:=j-1;
    while (i>=1)and(a[i]>b) do {посуваєм впорядковані елементи}
      begin a[i+1]:=a[i]; dec(i)
      end;{while}
    a[i+1]:=b   {вставляєм j-ий елемент у впорядковану частину}
  end;{for j}
end;{sortInsert}

begin write('Введіть послідовність з ',n,' чисел: ');
  for i:=1 to n do read(a[i]); readln;
  sortInsert(a);                               {впорядковуємо масив}
  for i:=1 to n do write(a[i]:5); writeln
end.

```

За цим алгоритмом кожен з членів масиву ніби проникає на відповідний йому рівень, або «занурюється» у впорядковану частину масиву. Тому сортування простими вставками ще називають *методом занурення*.

Які затрати цього алгоритму для впорядкування масиву з n елементів? Затратами у цьому випадку є порівняння елементів та виконання переприсвоєнь. Позначимо A – кількість порівнянь і B – кількість присвоєнь. Знайдемо найменші та найбільші значення величин A і B .

Порівняння виконують в умові внутрішнього циклу. Це означає, що на кожному кроці зовнішнього циклу виконується принаймні одне порівняння: якщо масив a уже впорядковано за зростанням (найкращий варіант вхідних даних), то елемент a_{i+1} необхідно вставити після останнього елемента впорядкованої частини (фактично, він уже стоїть на цьому місці), і буде виконано тільки одне порівняння; якщо ж масив a впорядковано за спаданням (найгірший варіант вхідних даних), то елемент a_{i+1} необхідно вставити перед першим елементом впорядкованої частини, і буде виконано $1+2+\dots+(n-1)$ порівнянь. Отже, $n-1 \leq A \leq (n^2-n)/2$.

Переміщення елементів за допомогою переприсвоєнь виконуються і в зовнішньому, і у внутрішньому циклах. У зовнішньому буде виконано $2 \times (n-1)$ присвоєнь, а у внутрішньому – ні одного, якщо масив уже впорядковано за зростанням, або $(n^2-n)/2$, якщо масив впорядковано за спаданням. Отже, $2 \times (n-1) \leq B \leq n^2/2 + 3/2n - 2$.

Бачимо, що описаний алгоритм має оцінку $O(n^2)$, тобто є квадратичним. На практиці для зменшення затрат на сортування використовують бінарні вставки та інші модифікації алгоритму сортування вставками.

6.2. Сортування вибором

У впорядкованому за зростанням масиві на останньому місці стоїть найбільший елемент, на передостанньому – другий за величиною і т. д. Щоб досягти такого розташування значень у масиві, можна діяти так: знайти максимальний елемент масиву і поміняти його місцями з останнім, далі знайти найбільший серед всіх крім останнього і поміняти його місцями з передостаннім і т. д. Впорядкування буде завершено, коли більший з першого та другого елементів займе друге місце в масиві, а менший – перше.

Як реалізувати описаний спосіб впорядкування? Нам щоразу необхідно знаходити найбільший елемент даної послідовності і міняти його місцями з останнім. Схожу задачу розв'язано у п. 3.2 за допомогою програми *change*. Використаємо її алгоритм для відшукування значення та індексу максимального елемента невпорядкованої частини масиву. А щоб впорядкувати весь масив, пошук найбільшого і обмін повторимо $n-1$ раз.

Алгоритм сортування оформимо у вигляді процедури. Оголошення та текст головної програми можуть залишатися такими ж, як у попередньому підрозділі:

```

const n=10;
type vector=array[1..n] of integer;

procedure sortFMax(var a:vector);
{   ВПОРЯДКУВАННЯ МАСИВУ а ЗА ДОПОМОГОЮ ВИБОРУ НАЙБІЛЬШОГО   }
  var b,k:integer; i,j:byte;
begin

```

```

        {шукатимемо найбільший елемент невідсортованої частини}
for j:=n downto 2 do           {і ставити його на j-е місце}
begin b:=a[1]; k:=1; {початкові значення і номер найбільшого}
    for i:=2 to j do           {перевіряємо всі решту}
        if a[i]>b then begin b:=a[i]; k:=i
            end; {if & for i}
        a[k]:=a[j]; a[j]:=b      {мінємо a(j) з найбільшим}
    end{for j}
end; {sortFMax}

```

Легко бачити, що за описаним алгоритмом для довільних даних завжди виконується однакова кількість порівнянь: $A = (n^2 - n) / 2$. Тому за затратами на порівняння він є гіршим від алгоритму сортування вставками. Проте алгоритм сортування вибором набагато економніший за переміщеннями елементів масиву: вони відбуваються $3 \times (n - 1)$ раз у зовнішньому циклі. Тому сортування вибором особливо ефективно тоді, коли переміщення є затратними, наприклад, якщо сортувати необхідно масив ключів, і разом з ними переміщувати великі масиви інформації. Приклади таких програм наведемо пізніше.

6.3. Сортування обмінами

Алгоритм сортування обмінами є одним з найочевидніших. За цим алгоритмом переглядають увесь масив і перевіряють кожну пару сусідніх елементів на впорядкованість (як ми це робили у п. 3.3). Якщо виявиться, що сусіди впорядковані, то просто перевіряють наступну пару, а у протилежному випадку міняють їх місцями. Очевидно, що внаслідок одного перегляду найбільший за величиною елемент у результаті послідовних обмінів переміститься в кінець масиву і займе своє остаточне місце. Початок масиву буде містити, швидше за все, невідсортовані елементи, тому перевірки та переміщення необхідно виконати повторно, не зачіпаючи при цьому останнього елемента. Внаслідок другого перегляду другий за величиною елемент займе передостаннє місце. Процес сортування завершиться після $n - 1$ перегляду, коли востаннє ми порівняємо перший і другий елементи масиву.

Під час сортування за цим алгоритмом найбільші елементи ніби впливають на поверхню масиву, тому метод сортування обмінами називають ще *методом бульбашки*. Наведемо текст процедури, що реалізує цей метод:

```

const n=10;
type vector=array[1..n]of integer;

procedure sortBubble(var a:vector);
    var b:integer; i,j:byte;
begin for j:=n downto 2 do      {повторимо перегляд n-1 раз і}
    for i:=1 to j-1 do {порівняємо «невідсортованих» сусідів}
        if a[i]>a[i+1] then      {потрібно поміняти їх місцями!}
            begin b:=a[i]; a[i]:=a[i+1]; a[i+1]:=b
                end; {if & for i&j}
    end; {sortBubble}

```

Все зовсім просто, але якщо трошки поекспериментувати з цим алгоритмом на різноманітних вхідних даних, то виявиться, що він виконує багато зайвих порівнянь. Наприклад, алгоритм не розпізнає ситуації, коли задано впорядкований за зростанням масив. У цьому випадку вистачило б одного перегляду, щоб пересвідчитись: усі пари сусідів розташовано у правильному порядку, а *sortBubble* вперто продовжує виконувати усі ітерації зовнішнього циклу! Як удосконалити процедуру і позбавити її такого очевидного недоліку? Необхідно використати додаткову змінну для зберігання інформації щодо того, чи відбувались перестановки елементів: якщо ні, то масив уже впорядковано і перевірки можна припинити. Як засвідчують експерименти, досить часто в результаті одного перегляду масиву не тільки найбільший елемент займає своє місце, а й декілька інших (якщо вони відразу були розташовані у «правильному» порядку в заданому масиві). З метою відстеження такої ситуації, будемо заносити в додаткову змінну не просто ознаку того, що переміщення відбулися, а номер останнього елемента, для якого їх виконано. Тоді така змінна міститиме номер останнього елемента неупорядкованої частини масиву. З урахуванням цих міркувань отримаємо процедуру:

```

procedure sortReplace(var a:vector);
{      ВПОРЯДКУВАННЯ МАСИВУ a ЗА ДОПОМОГОЮ ОБМІНІВ      }
var b,k:integer; i,j:byte;
begin j:=n;           {спочатку весь масив - неупорядкований}
  while j>0 do
  begin k:=0; {припустили, що всі пари є у правильному порядку}
    for i:=1 to j-1 do {переглядаємо «неупорядкованих» сусідів}
      if a[i]>a[i+1] then {знайшли порушення, тому виконаємо}
        begin b:=a[i]; a[i]:=a[i+1]; a[i+1]:=b; {переміщення i}
          k:=i           {i запам'ятаємо місце, де це сталося}
        end; {if & for i}
      j:=k              {змінити межу неупорядкованої частини}
    end; {while}
  end; {sortReplace}

```

Кількості порівнянь і переміщень за цим алгоритмом оцінюють так: $n-1 \leq A \leq (n^2-n)/2$, $0 \leq B \leq 3n^2/2-3n/2$. Бачимо, що оцінка кількості порівнянь не гірша, ніж в алгоритмові сортування простими вставками, проте кількість переміщень у гіршому випадку є втричі більшою. Це найбільший недолік методу бульбашки: він виконує занадто багато переміщень елементів масиву. На практиці цей метод доцільно використовувати для «майже впорядкованих» вхідних даних, що потребують перестановки лише кількох великих значень, що «заблукали» на початку чи посередині майже впорядкованого масиву. Використовують також різноманітні покращені модифікації алгоритму сортування обмінами.

7. Сортування структур даних

Тепер, коли ми вміємо кількома способами впорядковувати послідовність чисел, розглянемо складніші випадки сортування даних. Наприклад,

сортування рядків заданої матриці та різні способи впорядкування записів файла.

7.1. Впорядкування рядків матриці

Задача 26. Дано матрицю $A:10 \times 12$. Впорядкувати за зростанням кожен рядок матриці.

Розв'язок цієї задачі є простим і очевидним узагальненням алгоритму впорядкування одновимірного масиву: з метою впорядкування кожного рядка, необхідно просто десять разів виконати процедуру сортування. Який з описаних раніше методів впорядкування вибрати? Враховуючи оцінки ефективності, наведені у попередньому параграфі, зупинимо свій вибір на сортуванні простими вставками і використаємо оголошену раніше процедуру *sortInsert*:

```

program sortEachLine;
  const n=10; m=12;
  type vector=array[1..m]of real;
        matrix=array[1..n]of vector;
  procedure sortInsert(var a:vector);
  {      ВПОРЯДКУВАННЯ МАСИВУ а ПРОСТИМИ ВСТАВКАМИ      }
  var b:integer; i,j:byte;
begin for j:=2 to m do          {шукаємо місце для j-го елемента}
  begin b:=a[j]; i:=j-1;
        while (i>=1)and(a[i]>b) do {посуваєм впорядковані елементи}
        begin a[i+1]:=a[i]; dec(i)
        end; {while}
        a[i+1]:=b          {вставляєм j-й елемент у впорядковану частину}
  end; {for j}
end; {sortInsert}

var a:matrix; i,j:byte;
begin writeln('Введіть матрицю A');
  for i:=1 to n do
  for j:=1 to m do read(a[i,j]);
    readln;          {закінчили введення матриці A}
    { ВПОРЯДКУВАННЯ РЯДКІВ МАТРИЦІ }
  for i:=1 to n do          {перебираємо рядки матриці}
    sortInsert(a[i]);      {і сортуємо кожен з них}
    { ВИВЕДЕННЯ МАТРИЦІ }
  for i:=1 to n do
  begin for j:=1 to m do write(a[i,j]);
        writeln          {закінчили друкувати рядок матриці}
  end {for i}
end.

```

Задача 27. Дано матрицю $A:10 \times 12$. Впорядкувати рядки матриці за зростанням їхніх перших елементів.

Щоб розв'язати цю задачу, необхідно переставити місцями рядки матриці так, щоб їхні перші елементи були впорядковані за зростанням. Тобто кожен рядок розглядають як єдиний запис, елемент структури даних, а його перший елемент – як його ключ. Переміщення рядків матриці є доволі затратною

операцією, тому використаємо найекономніший щодо переміщень алгоритм – сортування вибором. Удосконалимо його так, щоб разом з переміщенням ключів відбувалось переставляння рядків.

Введення та виведення матриці у цій задачі нічим не відрізняється від описаного в попередній програмі, тому наведемо тільки той фрагмент програми, який відповідає власне за впорядкування:

```

type vector=array[1..m]of real;
      matrix=array[1..n]of vector;
var a:matrix; v:vector;
      c:real; i,j,k:byte;
begin      { ... введення матриці }           {ВПОРЯДКУВАННЯ}
      {ПЕРШОГО СТОВПЦЯ МАТРИЦІ ЗА ДОПОМОГОЮ ВИБОРУ НАЙБІЛЬШОГО}
      {будемо знаходити найбільший елемент невпорядкованої частини}
for j:=n downto 2 do           {і ставити його на j-е місце}
begin c:=a[1,1]; k:=1; {початкові значення, номер найбільшого}
      for i:=2 to j do           {перевіряємо всі решту}
          if a[i,1]>c then begin c:=a[i,1]; k:=i
              end; {if & for i}
          v:=a[k]; a[k]:=a[j]; a[j]:=v      {мінємо a(j) з найбільшим}
      end; {for j}
      { виведення матриці ... }
end.

```

У цій програмі рядки матриці міняємо місцями за допомогою додаткового масиву v , елементи першого стовпця змінюють місце разом зі своїм рядком.

Задача 28. Дано матрицю $A:10 \times 12$. Впорядкувати рядки матриці за зростанням сум модулів їхніх елементів.

Ця задача, як і попередня, передбачає впорядкування рядків матриці за зростанням певних ключів, однак, на відміну від попередньої, тут ключі рядків наперед невідомі. Їх необхідно обчислити і зберігати під час сортування. Використаємо для зберігання ключів одновимірний масив b . Його довжина дорівнює кількості рядків матриці:

```

type vector=array[1..m]of real;
      matrix=array[1..n]of vector;
var a:matrix; v:vector;
      b:array[1..n]of real;
      c:real; i,j,k:byte;
begin      { ... введення матриці }
      { ФОРМУВАННЯ МАСИВУ КЛЮЧІВ }
      for i:=1 to n do
          begin s:=0;
              for j:=1 to m do s:=s+abs(a[i,j]);
              b[i]:=s
          end; {for i}
      { ВПОРЯДКУВАННЯ РЯДКІВ МАТРИЦІ }
      for j:=n downto 2 do
          begin c:=b[1]; k:=1;           {початковий ключ}
              for i:=2 to j do       {перевіряємо решту ключів}
                  if b[i]>c then begin c:=b[i]; k:=i

```

```

        end; {if & for i}
        b[k]:=b[j]; b[j]:=c;           {мінємо ключі}
        v:=a[k]; a[k]:=a[j]; a[j]:=v   {i відповідні рядки}
    end; {for j}
{ виведення матриці ... }
end.

```

Впорядкування стовпців матриці можна виконувати схожим чином. Відповідні алгоритми будуть відрізнятися тільки індексами біля елементів матриці та способом виконання перестановок: для стовпців їх потрібно виконувати поелементно, як у п. 4.3, у програмі *moveMax*.

7.2. Впорядкування файла за допомогою списку

На відміну від масиву – структури даних з безпосереднім доступом до даних – файл є структурою з послідовним доступом. Тому сортування записів файла суттєво відрізняється від сортування елементів масиву. Передусім тому, що виконати доступ до конкретного запису файлу є складніше і суттєво довше, ніж до елемента масиву. З описаних раніше алгоритмів хіба що впорядкування вибором можна було б адаптувати для сортування файлів. На практиці ж для цього використовують інші алгоритми. Про них ми поговоримо далі.

Вибір способу сортування записів файлу залежить також і від його розміру: якщо файл можна повністю завантажити в оперативну пам'ять, то його можна впорядкувати за допомогою лінійного списку, чи дерева. У протилежному випадку здебільшого використовують впорядкування злиттям.

Задача 29. Дано файл, кожен запис якого містить унікальний цілочисловий ключ. Впорядкувати записи файлу за зростанням ключів.

Припустимо спочатку, що розмір файлу не перевищує розмір доступної динамічної пам'яті і всі його записи можна завантажити в деяку динамічну структуру даних, наприклад, в лінійний однонаправлений список. Якщо в процесі завантаження ми збережемо початковий взаємний порядок розташування записів, то далі доведеться розв'язувати задачу сортування списку. Зручніше використати інший підхід: створити за даним файлом впорядкований список і тоді елементи списку записати назад у файл. Як створити впорядкований список? Для цього можна перший запис файлу одразу завантажити у першу ланку списку, а кожен нову ланку з наступним записом вставляти у список так, щоб не порушити впорядкування ключів – як в описаному раніше алгоритмі впорядкування простими вставками.

У п. 3.4 ми розглянули кілька варіантів алгоритму відшукування місця нового елемента у впорядкованій послідовності значень. З метою впорядкування масиву простими вставками ми використали пошук, що починав перегляд масиву з кінця і одночасно виконував порівняння й переміщення елементів (програма *simpleInsert*). Для побудови впорядкованого списку такий спосіб пошуку місця не підійде: лінійний список можна переглядати тільки від першої ланки до останньої, а не у зворотньому порядку. До того ж для нових ланок списку не потрібно звільняти місце, адже зв'язок між окремими ланками здійснюють за допомогою вказівників, яким байдуже, чи розташовано ланки в

пам'яті підряд, чи ні. Тому доцільно виконувати пошук місця для нової ланки, як було описано програмою *forwardInsert*.

Відомо, що перша ланка лінійного списку є «особлива»: вказівники на всі інші ланки містяться у полях зв'язку попередніх ланок, а вказівник на першу – в окремій змінній, що вказує на початок всього списку. Тому спосіб опрацювання першої ланки, зокрема, вставки першої ланки, відрізняється від способу опрацювання усіх інших. Щоб уникнути потреби реалізувати ці два способи опрацювання на практиці вставляють у список перед його першою ланкою ще одну, додаткову – *заголовну*. Єдине її завдання – містити вказівник на першу ланку списку. Витрату зайвої пам'яті у цьому випадку виправдано спрощенням алгоритму обробки списку. Саме такий підхід ми і використовуємо.

Конкретна структура запису файлу несуттєва для алгоритму впорядкування за ключами. Домовимось, що записи файлу для написання програми матимуть тип *fileRec* – деякий комбінований тип, що містить цілочислове поле *key* для значення ключа запису. Склад і типи інших полів завжди можна конкретизувати відповідно до потреб задачі:

```

program sortByChain;
type fileRec=record key:integer;
    {... всі інші поля ...}
end; {fileRec}
dataBase=file of fileRec;
chain=^section; {список}
section=record elem:fileRec; {ланка містить один запис}
    link:chain {поле зв'язку}
end; {chain}
var s:string; f:dataBase; x:fileRec; k:integer;
    p,q,r:chain; {робочі змінні для формування списку}
begin write('Введіть ім'я файла: '); readln(s);
    assign(f,s); reset(f); {відкрили потрібний файл}
    p:=new(chain); p^.link:=nil; {створили початкову ланку}
    while not EoF(f) do
    begin read(f,x); k:=x.key; {прочитали черговий запис файла}
        q:=p; {тепер знайдемо для нього місце в списку}
        while (q^.link<>nil) and (q^.link^.elem.key<k)
        do q:=q^.link;
        r:=q^.link; q^.link:=new(chain); {і вставимо в нього}
        with q^.link^ do
        begin elem:=x; link:=r end {with}
    end; {while}
    q:=p; p:=p^.link; dispose(q); {вилучили початкову ланку}
    seek(f,0); q:=p; {запишемо в файл впорядкований список}
    while q<>nil do
    begin with q^ do
        begin write(f,elem); p:=link end; {with}
        dispose(q); q:=p
    end; {while} close(f) {закрили впорядкований файл}
end.

```

Під час сортування розмір файлу не змінюється, тому впорядковані дані ми записали у той самий файл «поверх» старих даних. Разом із зберіганням даних у програмі виконано також звільнення динамічної пам'яті від списку.

7.3. Впорядкування файлу бінарним деревом

Алгоритм впорядкування файлу лінійним списком використовує послідовний пошук місця елемента. Як зазначено у п. 3.4, ефективнішим є пошук місця за допомогою методу поділу відрізка навпіл (програма *binaryInsert*). Виграш особливо відчутний для послідовностей великих розмірів, а файли, зазвичай, є досить великими.

З метою реалізації бінарного пошуку використовують двійкове дерево (визначення дерева та опис алгоритмів їхньої обробки див. у п. 10.3), кожна вершина якого містить унікальний ключ, причому для кожної вершини правильним є твердження про те, що ліве її піддерево містить лише вершини з меншими ключами, а праве – з більшими. Таке дерево називають *деревом пошуку*. Необхідний ключ можна знайти, скориставшись алгоритмом бінарного пошуку: якщо шуканий ключ менший за ключ у корені, то пошук продовжують у лівому піддереві; якщо більший, – то у правому; якщо рівний, то елемент знайдено; в іншому випадку знайдено місце для нового елемента, його необхідно долучити до дерева. Час виконання цього алгоритму є величиною порядку $O(\log_2 n)$, де n – кількість ключів у дереві.

З метою впорядкування файлу за допомогою дерева необхідно спочатку за даними, записаними у файлі, побудувати двійкове дерево пошуку, а тоді виконати *зворотній обхід* (див. п. 10.3) дерева і записати всі його елементи у файл. Пошук місця для нового елемента дерева та збереження дерева зручно реалізувати за допомогою рекурсивних процедур. За умовою задачі вихідний файл містить записи з попарно різними ключами, тому в процедурі пошуку не потрібно перевіряти рівність ключів:

```

program sortByTree;
type fileRec=record key:integer;
    {... всі інші поля ...}
end; {fileRec}
    dataBase=file of fileRec;
    tree=^node; {дерево}
    node=record elem:fileRec; {вершина дерева містить запис,}
        left,right:tree end; {поля зв'язку з піддеревими.}

procedure searchAndInclude(x:fileRec; var t:tree);
{ процедура searchAndInclude виконує пошук з }
{ включенням запису x у дереві t }
begin if t=nil then {запис не знайшли, включаємо його в дерево}
    begin t:=new(tree); with t^ do
        begin elem:=x; left:=nil; right:=nil
        end {witn} end {then}
    else with t^ do {шукаємо в непорожньому дереві:}
        if x.key<elem.key then {менший ключ - ліворуч,}
            searchAndInclude(x,left) else {більший - праворуч.}

```

```

    searchAndInclude(x, right)
end; {searchAndInclude}

procedure writeTree(var f:dataBase; t:tree);
{ процедура writeTree записує непорожнє дерево t }
{ у файл f, виконуючи лівосторонній обхід.          }
begin with t^ do
    begin if left<>nil then writeTree(f, left); {друкуємо ліве}
        write(f, elem); {піддерево, потім - корінь,}
        if right<>nil then writeTree(f, right) {а тоді - праве.}
    end{with}
end; {writeTree}

var f:dataBase; fileName:string; {змінні для приєднання до}
                                     {заданого зовнішнього файла}
    x:fileRec; t:tree; {будуватимемо дерево T}
Begin write('Введіть ім'я файла: '); readln(fileName);
    assign(f, filename); reset(f); {відкрили файл}
    t:=nil; {дерево спочатку порожнє}
    while not EoF(f) do {читаємо і вставляємо всі слова}
        begin read(f, x); searchAndInclude(x, t)
        end; {while {дерево готове!}
    seek(f, 0); writeTree(f, t); {тепер перезапишемо файл}
    close(f)
End.

```

Обидва алгоритми впорядкування файла, описані в двох останніх параграфах, легко можна пристосувати для впорядкування лінійних однонапрямлених списків. З цією метою необхідно просто замінити читання даних з файла на отримання ланки списку, що підлягає сортуванню. У результаті виконання першого алгоритму (після його модифікації) за заданим списком одразу отримаємо відсортований список. За другим алгоритмом буде побудовано дерево, і замість його збереження у файлі необхідно виконати побудову списку за деревом.

7.4. Впорядкування файла злиттям

Як впорядкувати файл, занадто великий, щоб повністю помістити його у динамічній пам'яті? Перше, що спадає на гадку, це впорядкувати файл хоча б частково. Наприклад, завантажувати в пам'ять і сортувати стільки записів файла, скільки там поміститься. У результаті отримаємо файл, який містить впорядковані відрізки. Тепер, щоб завершити сортування, залишиться об'єднати ці відрізки в одну впорядковану послідовність. Об'єднання впорядкованих двох чи більше підпослідовностей в одну називають *злиттям*. Алгоритм сортування числового масиву за допомогою злиття вперше запропонував Джон фон Нейман, тому алгоритм впорядкування злиттям часто називають його іменем.

Якщо послідовність містить більше ніж два впорядковані відрізки, то об'єднувати їх можна по-різному: послідовно, попарно, по три відрізки в один чи ще якимось. На практиці використовують різні способи злиття, залежно від

конкретних умов та оцінок ефективності цих способів. Ми використовуємо один з найпростіших способів, об'єднуючи відрізки попарно. Проміжні результати, отримані на окремих кроках процесу сортування, зберігатимемо у допоміжних файлах.

Нехай a і b – файли, k – натуральне число. Говорять, що файли a і b узгоджено k -впорядковані, якщо:

- 1) у кожному з файлів a і b перші k записів, наступні за ними k записів і так далі утворюють упорядковані відрізки; останній відрізок файла (також упорядкований) може містити менше ніж k записів, однак у цьому випадку тільки один з файлів може містити неповний останній відрізок;
- 2) кількість впорядкованих відрізків файла a відрізняється від кількості впорядкованих відрізків файла b не більше ніж на одиницю;
- 3) якщо файли містять різну кількість відрізків, то неповним може бути тільки останній відрізок довшого файла.

Записи двох узгоджено k -впорядкованих файлів a і b можна розташувати у файлах f і g так, що f і g будуть узгоджено $2k$ -впорядкованими. З цією метою пару впорядкованих відрізків з файлів a і b об'єднують у вдвічі довший впорядкований відрізок. Таке об'єднання називають *злиттям*. Відрізки записів, отримані внаслідок злиття, по чергово розташовують то у файлі f , то у файлі g . На рис. 3 показано, що відбувається під час перших двох злиттів.

Впорядкування файла f методом *збалансованого двошляхового злиття* описує такий алгоритм (використовуються допоміжні файли g , a і b). Спочатку якимось способом розподіляють записи файла f до файлів a і b , щоб утворити початкові впорядковані відрізки, наприклад, записи з парними номерами – до одного файла, а з непарними – до іншого. Або розглядають пари записів файла f і по чергово записують їх у впорядкованому вигляді до файлів a та b , або утворюють початкові впорядковані відрізки за допомогою сортування бінарним деревом (на скільки це можливо за обсягом доступної динамічної пам'яті), чи іншим способом.

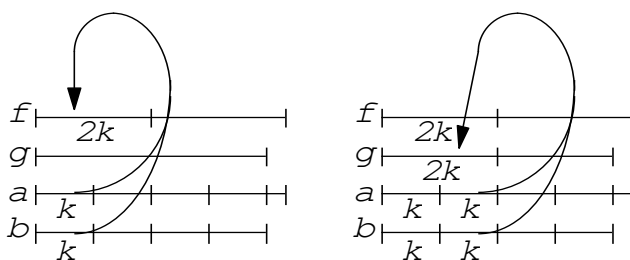


Рис. 3. Початковий етап сортування компонент файла злиттям

Далі файли a і b розглядають як k -впорядковані (де k – розмір початкового відрізка), і утворюють з їхніх компонент $2k$ -впорядковані файли f і g . Далі з файлів f і g утворюють $4k$ -впорядковані a і b і т. д. Оскільки кількість впорядкованих відрізків у файлах зменшується після кожного злиття, то настане момент, коли усі записи буде зібрано в одному файлі у вигляді одного впорядкованого відрізка. На цьому сортування записів файла буде завершено.

З метою опису алгоритму впорядкування файла злиттям, необхідно дати відповіді на такі запитання: як утворити початковий розподіл відрізків; як виконувати злиття відрізків; як контролювати повторення та завершення процесу злиття?

Найпростіше отримати *початковий розподіл* записів файла методом копіювання однієї половини з них до файла a , а другої – до файла b , проте це ніяк не наблизить нас до впорядкованості. Тому використаємо інший простий спосіб утворення файлів a і b : зчитуватимемо з файлу f по два записи і порівнюватимемо їхні ключі. Для пари записів дуже легко знайти більший ключ і записати їх у файл a чи b у правильному порядку й отримати таким чином впорядковані відрізки, довжина яких дорівнює двом. Якщо файл f містить непарну кількість записів, то останній відрізок буде неповним. Легко переконатися: якщо збереження впорядкованих відрізків почати з файлу a , то він завжди буде не меншим за файл b . Можливі такі варіанти: файли a та b містять однакову кількість початкових відрізків, у файлу b останній відрізок повний або ні; файл a містить на один відрізок (повний або неповний) більше. Алгоритм формування початкового розподілу опишемо у вигляді процедури *distribute*.

Злиття. З метою об'єднання двох впорядкованих відрізків в один довгий впорядкований використаємо такий алгоритм: прочитаємо по одному запису з цих відрізків, порівняємо між собою їхні ключі і запишемо менший, а замість нього прочитаємо з відповідного відрізка наступний запис. Далі повторюємо порівняння, запис та зчитування аж до завершення одного з відрізків. Після цього копіюємо у результуючий непрочитаний залишок другого відрізка. Алгоритм злиття опишемо за допомогою процедури *merge*. Наш алгоритм повинен працювати з відрізками різних довжин, тому задаватимемо їх як параметри цієї процедури.

Впорядкування файлу розпочнемо одразу ж після побудови початкового розподілу записів вихідного файлу. Впорядкування полягає у багатократному злитті впорядкованих відрізків, що містяться в одній парі файлів, та збереженні результуючих відрізків у парі інших файлів. Початковий розподіл ми отримаємо у файлах a і b , тому вони є джерелами відрізків для першого злиття, а файли f і g – приймачами. Проте після першого об'єднання джерелом мусять стати f і g . Як це зробити, не дуже загромождаючи алгоритм? Для приєднання до файлів у програмі можемо використати динамічні файлові змінні і після кожного об'єднання міняти місцями значення вказівників на файли a та f , і на b та g . Отже файли a і b завжди будуть джерелами, а файли f і g – приймачами.

Щоб визначити, скільки відрізків необхідно об'єднати, порахуємо їхню кількість у кожному з файлів під час побудови початкового розподілу та змінюватимемо належним чином під час виконання об'єднань. Процес впорядкування закінчиться, коли одна з кількостей відрізків дорівнюватиме нулю.

Особливої уваги потребує злиття останніх відрізків файлів a і b . Якщо файл a – довший, то його останній відрізок (незалежно від того, повний він чи ні) не має пари і його необхідно просто переписати у файл f чи g . Якщо ж файли a і b містять однакову кількість відрізків, то необхідно врахувати, що останній відрізок файлу b може бути неповним.

Тепер, здається, усі попередні пояснення наведені і можна записати саму програму. Окремі її кроки пояснені також у коментарях.

```

program sortByMerging;
type fileRec=record key:integer;
        {усі інші потрібні поля} end;
    dataBase=file of fileRec;
    dataPtr=^dataBase;

procedure distribute(var f,a,b:dataBase; var ka,kb:longint);
    {розподіляє записи файлу f до файлів a і b у відрізки по 2 }
    {записи; ka - кількість відрізків у файлі a, kb - у файлі b}
var x,y:fileRec;
begin ka:=0; kb:=0;          {цикл закінчимо процедурою Break,}
        while true do          {коли досягнемо кінець файлу}
        {*** Спочатку записуємо до файлу a ***}
        begin if eof(f) then Break else          {прочитаємо}
            begin read(f,x); inc(ka) end; {черговий запис}
            if eof(f) then          {він не має пари}
                begin write(a,x); Break end else
                begin read(f,y); {читаємо наступний і порівнюємо}
                    if x.key<y.key then write(a,x,y)
                        else write(a,y,x)
                    end;
                end;
        {*** Тепер повторимо все для файлу b ***}
            if eof(f) then Break else
                begin read(f,x); inc(kb) end;
            if eof(f) then
                begin write(b,x); Break end
            else begin read(f,y);
                if x.key<y.key then write(b,x,y)
                    else write(b,y,x)
                end
            end
        end {while}
end; {distribute}

procedure merge(var a,b,c:dataBase; ka,kb:longint);
    {об'єднує відрізок довжини ka з файлу a з відрізком довжини}
    {kb з файлу b і записує їх у файл c}
var x,y:fileRec; i,j:longint;
begin i:=1; read(a,x);
        j:=1; read(b,y); {прочитали перші елементи відрізків}
        while true do IF x.key<y.key THEN
            begin write(c,x); if i<ka then {записали значення з a}
                begin          {читаем наступне значення з a}
                    inc(i); read(a,x)
                end else          {дописуем залишок файлу b}
                begin write(c,y);
                    while j<kb do
                        begin read(b,y); inc(j); write(c,y)
                        end; {while} Break
                    end {else & if i<ka}
            end {THEN} ELSE
            begin write(c,y); if j<kb then {записали значення з b}
                begin          {читаем наступне значення з b}
                    inc(j); read(b,y)
                end
            end

```

```

        end else           {дописуем залишок файла a}
        begin write(c,x);
            while i<ka do
                begin read(a,x); inc(i); write(c,x)
                end; {while} Break
            end {else & if j<kb}
        end {ELSE & while}
    end; {merge}

var f:dataPtr; s:string; a,b,g,c:dataPtr; x,y:fileRec;
    k,k1,k2,i:longint;
begin           {ГОЛОВНА ПРОГРАМА}
    write('Введіть ім'я файла: '); readln (s);
    {Виконуємо приєднання до заданого файла}
    f:=new(dataPtr); assign(f^,s);
    {$I-} reset(f^); {$I+}           {наявність файла перевіримо самі}
    if IOResult<>0 then
        begin writeln('File ',s,' doesn't exist. '); exit end;
    {Будуємо початковий розподіл записів у тимчасових файлах}
    a:=new(dataPtr); b:=new(dataPtr);
    assign(a^,'_1.tmp'); rewrite(a^);
    assign(b^,'_2.tmp'); rewrite(b^);
    distribute(f^,a^,b^,k1,k2);
    close(f^); close(a^); close(b^);
    k:=2;           {початковий розмір впорядкованих відрізків}
    g:=new(dataPtr); assign(g^,'_3.tmp'); {ще один тимчасовий}
    {Виконаємо впорядкування файла, поки відрізків є понад 1}
    WHILE (k1>0) and (k2>0) DO
        begin reset(a^); reset(b^); {джерела відкрили для читання}
            rewrite(f^); rewrite(g^);           {приймачі - для запису}
            IF k1>k2 THEN                       {ФАЙЛ a - ДОВШИЙ}
                begin for i:=1 to k2 div 2 do   {об'єднуем дві пари}
                    begin merge(a^,b^,f^,k,k); merge(a^,b^,g^,k,k)
                    end; {for}
                    if odd(k2) then             {i останню пару}
                        begin merge(a^,b^,f^,k,k); k1:=k1 div 2; k2:=k1;
                            while not eof(a^) do {копіюємо залишок файла a}
                                begin read(a^,x); write(g^,x) end {while}
                            end {odd then} else   {усі пари вже об'єднано!}
                                begin k2:=k2 div 2; k1:=k2+1;
                                    while not eof(a^) do {копіюємо залишок файла a}
                                        begin read(a^,x); write(f^,x) end {while}
                                    end {else}
                                end {THEN} ELSE {ФАЙЛИ МІСТЯТЬ ОДНАКОВУ К-СТЬ ВІДРІЗКІВ}
                                    begin for i:=1 to (k2-1) div 2 do {об'єднуем дві пари}
                                        begin merge(a^,b^,f^,k,k); merge(a^,b^,g^,k,k)
                                        end; {for}
                                        if odd(k2-1) then
                                            begin merge(a^,b^,f^,k,k);           {останню повну пару i}
                                                {останній відрізок файла a з останнім (неповним?) файла b}
                                                merge(a^,b^,g^,k,fileSize(b^)-filePos(b^));
                                                k1:=k1 div 2; k2:=k1
                                            end {odd then} else {всі повні вже пари об'єднано!}
                                                begin merge(a^,b^,f^,k,fileSize(b^)-filePos(b^));

```

```

        k2:=k2 div 2; k1:=k2+1
    end {else}
end; {ELSE} k:=k*2;
close(a^); close(b^); close(f^); close(g^);
c:=f; f:=a; a:=c;           {мінємо місцями значення}
c:=g; g:=b; b:=c           {вказівників на джерела і приймачі}
end; {WHILE}
erase(b^); erase(f^); erase(g^); {знищимо тимчасові файли}
rename(a^,s);           {і повернемо ім'я впорядкованому файлові}
dispose(a); dispose(b); dispose(f); dispose(g)
end.

```

Зауважимо, що алгоритм побудовано так, що перший з пари файлів-приймачів завжди є не меншим від другого, тому результат сортування міститиметься у файлі, на який вказує змінна a . Усі інші файли більше непотрібні, тому їх можна просто вилучити з диска. Оскільки в програмі багаторазово виконуються переприсвоєння значень вказівників на файлові змінні, то змінна a може вказувати на заданий файл або на тимчасовий файл `'_1.tmp'`. Щоб не виникало плутанини, перед закінченням програми виконано переіменування цього файла.

8. Опрацювання текстової інформації

Обробка текстової інформації складає основу більшості програм. Завдяки таким алгоритмам використання комп'ютерів стало можливим у різних сферах життя.

Довільний алгоритм роботи з символами по суті включає лише дві основні дії: пошук і заміну. Згідно з тезою Маркова, довільний алгоритм можна зобразити як послідовність таких дій.

Зрозуміло, що пошук можна використати для виявлення певних властивостей вхідного тексту. Заміна дає змогу цей текст модифікувати певним чином.

Розглянемо кілька алгоритмів роботи з літерною інформацією для ілюстрації основних прийомів її опрацювання.

8.1. Розпізнавання чисел

Задача 30. Дано послідовність символів, яку закінчено крапкою. Визначити кількість цифр у цій послідовності.

Щоб розв'язати поставлену задачу, треба врахувати, що символи всіх цифр розташовано у кодовій таблиці підряд, починаючи від '0'. Тому цифрою є довільний символ c , для якого істинною є умова $(c \geq '0')$ and $(c \leq '9')$.

```

program searchDigit;
var c:char; k:byte;
begin writeln('Введіть послідовність літер, ',
    'яка закінчується крапкою. ');
    k:=0;           {лічильник кількості цифр}
    repeat read(c);

```

```

    if (c>='0')and(c<='9') then k:=k+1;
  until c='.';
  writeln('Послідовність містить ',k,' цифр.')
end.

```

Цю задачу можна використати для розв'язування складнішої задачі, яку розв'язує кожен компілятор.

Задача 31. Дано послідовність символів. Перевірити, чи задана послідовність є записом цілого числа у десятковій системі числення, і якщо так, то обчислити значення цього числа.

Ціле число – це послідовність цифр, якій може передувати знак '+' чи '-'. Цю послідовність можна зобразити рядком – структурою даних типу *string*, і скористатись стандартними засобами обробки цієї структури. У цьому алгоритмі ми застосуємо інший підхід з використанням частини алгоритму *isPolyndrome* (п. 1.2), яка будує значення числа за його цифрами.

```

program defineNumber;
var s:string;           {задана послідовність літер}
    zn:integer; k:longint; {знак числа і його значення}
    t:boolean;         {ознака правильності запису числа}
    i,cyf:byte;       {робочі змінні}
const o=ord('0');
begin writeln('Введіть послідовність літер');
  readln(s);
  k:=0;                {число, яке треба сформувати}
  i:=1; zn:=1;        {припускаємо, що число додатне}
  if s[i]='+' then i:=2 {цифри починаються з другої літери}
    else if s[i]='-' then {число - від'ємне, цифри - з}
      begin zn:=-1; i:=2 end; {другої літери}
  t:=true;            {спочатку вважаємо, що запис правильний}
  while(i<= length(s))and t do
  begin if (s[i]>='0')and(s[i]<='9') then
    begin cyf:=s[i]-o; {обчислили значення цифри}
      k:=k*10+cyf {обчислили відповідне значення числа}
    end {then} else t:=false; {помилка у записі}
    i:=i+1
  end; {while}
  if t then
    writeln('Послідовність є числом зі значенням ',zn*k)
  else writeln('Послідовність не є числом.')
end.

```

Послідовність, яку аналізує програма, повинна починатись із символів '+', '-' чи цифр. Усі інші символи сприймаються як помилкові. Проте цю програму легко модифікувати так, щоб вона ігнорувала пропуски перед числом.

8.2. Форматування виведення числової інформації

Для ілюстрації пошуку і заміни розглянемо приклад задачі, яка реалізується при форматному виведенні.

Задача 32. Дано текст з двох частин: перша – керуючий рядок; друга (можливо порожня) – аргументи. У керуючому рядку розташовано

спеціальні символи, які задають специфікації перетворення аргументів при виведенні. Аргументи відділені один від одного і від керуючого рядка комами. Перетворити текст, підготувавши його до друку. Тобто, використовуючи специфікації, помістити дані у потрібне місце у відповідному вигляді.

Вважатимемо, що: керуючий рядок – це послідовність довільних символів у лапках; кожна специфікація починається символом %; кількість специфікацій дорівнює кількості елементів у списку аргументів; рядок закінчується специфікацією.

Нехай аргументами будуть лише цілочислові значення, а специфікації задають такі перетворення:

d – число у десятковій системі;

h – число у шістнадцятковій системі.

Наприклад, якщо вхідний рядок має вигляд

```
"Значення змінних %d та %h",-123,428
```

то результуючий повинен бути таким:

```
Значення змінних -123 та 1A3
```

Зміст алгоритму полягає у пошуку аргументів та відповідних до них специфікацій і заміні специфікацій відповідними підрядками:

```
program formatting;
var vr:string;           {вхідний рядок}
    res:string;         {рядок - результат}
    hexs,des:string; b,t,err:boolean;
    zn:integer; n:longint; l,i,j,p,pm,k:byte;
    mvst:array[1..255]of byte;
const o=ord('0'); a=ord('A')-10;

procedure decHex(d:longint; var h:string);
{утворення шістнадцяткового запису h числа d}
var cyf:byte; c:char; z:Boolean;
begin h:='';           {запис спочатку порожній}
    if d>=0 then z:=false else
        begin d:=-d; z:=true end;
    while d>0 do
        begin cyf:=d mod 16;           {отримання чергової цифри числа}
            case cyf of
            0..9: c:=chr(cyf+o);
            10..15: c:=chr(cyf+a)
            end; {case} h:=c+h;
            d:=d div 16
        end; {while} if z then h:='-'+h
    end; {decHex}

procedure strDec(var s:string; var k:longint; var t:boolean);
{перетворення з символного представлення у числове}
var zn:integer; i,cyf:byte;
```

```

begin k:=0; i:=1; zn:=1;                                {число зі знаком}
  if s[i]='+' then i:=2 else
    if s[i]='-' then begin zn:=-1; i:=2 end;
  t:=true;                                             {припускаємо, що запис числа правильний}
  while (i<=length(s)) and t do
  begin if (s[i]>='0') and (s[i]<='9') then
    begin cyf:=ord(s[i])-o;                            {обчислили значення цифри}
      k:=k*10+cyf                                       {доповнили число}
    end else t:= false;
      i:=i+1
    end; {while} if t then k:=zn*k
end; {strDec}

begin {основна програма}
  writeln('Введіть рядок для форматування');
  readln(vr); res:=''; err:=false;
  {пошук і виділення керуючого рядка}
  i:=1; l:=length(vr); b:=true;
  if vr[i]<>'"' then
  begin writeln ('Неправильний початок керуючого рядка. ');
    err:=true
  end else
  begin i:=i+1; j:=0;
    while (i<=l) and b do {пошук кінця керуючого рядка}
    begin if vr[i]='"' then b:=false else
      if vr[i]='%' then
        begin j:=j+1; mvst[j]:=i {пошук місця вставки}
        end; {if & if} i:=i+1
      end; {while}
    if b then
    begin writeln('У рядку немає закриваючих лапок. ');
      err:=true
    end else
    begin {кінець керуючого рядка}
      if j=0 then
      begin writeln('Немає специфікацій. ');
        res:=res+copy(vr,2,i-3)
      end else
      begin k:=1; p:=2; {формування рядків аргументів}
        while (k<=j) and (not err) do
        begin des:=''; i:=i+1; b:=true;
          if vr[i] in ['+', '-'] then
            begin des:=des+vr[i]; i:=i+1 end;
          while b and not err do
          begin if vr[i] in ['0'..'9'] then
            begin des:=des+vr[i]; i:=i+1 end else
              if vr[i]=',' then b:=false else
                begin writeln('Неправильні аргументи');
                  err:=true
                end; {if & if} b:=b and (i<=l)
            end; {while i}
          if not b then {опрацювання аргумента}
          begin pm:=mvst[k]; k:=k+1;
            res:=res+copy(vr,p,pm-p);
          end;
        end;
      end;
    end;
  end;
end;

```

```

    p:=pm+2; {місце для продовження тексту
              початкового рядка}
    if vr[pm+1]='d' then res:=res+des
    {занесення в результуючий рядок десяткового значення}
    else if vr[pm+1]='h' then
    {занесення в результуючий рядок шістнадцяткового значення}
    begin STRDEC(des,n,t);
    if t then begin DECHEX(n,hexs);
    res:=res+hexs end
    else err:=true
    end else
    begin writeln('Неправильна специфікація');
    err:=true
    end;{if & if}
    end else
    begin writeln('невідповідність між',
    ' специфікаціями та аргументами');
    err:=true
    end;{if not b}
    end {while по специфікаціях}
    end {if j=0 - формування рядка з аргументами}
end {if b}
end;{vr[i]}
if err then writeln('Помилка у вхідному рядку')
else writeln('Результуючий рядок: ',res)
end.

```

Наведений приклад демонструє, що навіть не дуже складна обробка тексту вимагає значних зусиль. Більшу частину з них затрачено на перевірку правильності вхідного рядка.

9. Обчислення з заданою точністю

На практиці не менше значення ніж методи сортування даних мають числові методи розв'язування математичних задач. Сама назва «комп'ютер» означає *обчислювач*, а ще кілька років назад широко використовувався термін ЕОМ – електронно-обчислювальна машина. Такий наголос на здатності виконувати обчислення не випадковий: комп'ютер і було винайдено саме для автоматизації обчислень, а перші комп'ютери використовували виключно для розв'язування складних математичних задач, що вимагало виконання величезної кількості обчислень. Винайдення комп'ютера зумовило бурхливий розвиток цілої галузі математичної науки – обчислювальної математики, яка вивчає і розробляє спеціальні (числові) методи відшукування наближених розв'язків різноманітних задач: сумування рядів, числового інтегрування та диференціювання, обчислення значень коренів, розв'язків систем рівнянь тощо. Ми не маємо змоги в межах цього посібника докладно викласти теорію методів обчислень (ми і не ставили перед собою такої мети), проте познайомити читача з алгоритмами хоча б кількох з них – просто зобов'язані. Кожен освічений програміст знає, що означає для його програми обчислення синуса, кореня чи

іншої стандартної функції. До того ж числові методи дають прекрасну нагоду продемонструвати декілька важливих прийомів програмування.

9.1. Сумування рядів

Практично всі сучасні компілятори мов програмування високого рівня надають програмістові широкий вибір засобів для обчислення математичних функцій, проте ні один із сучасних процесорів не вміє безпосередньо обчислювати, наприклад, експоненту чи синус кута. Кожне звертання до математичної функції компілятор замінює на виклик відповідної процедури, яка обчислює значення цієї функції. Такі процедури здебільшого використовують розклад функції в ряд. Відомо, наприклад, що ряд Тейлора функції $\sin(x)$ має вигляд:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^{n+1} \frac{x^{2n-1}}{(2n-1)!} + \dots$$

і для невеликих за модулем значень x значення синуса можна обчислювати як суму цього ряду.

Задача 33. Дано дійсне значення x ($|x| \leq \pi/4$). Обчислити $\sin(x)$ з точністю 10^{-6} , використовуючи розклад синуса до ряду Тейлора.

Одразу виникає декілька запитань: як обчислювати члени цього ряду; коли закінчувати обчислення; що гарантує досягнення заданої точності?

Ряд Тейлора функції синус є знакозмінним, а для значень x з вказаного діапазону послідовність абсолютних величин членів цього ряду є монотонно спадною і збіжною до нуля. Тому, як відомо з курсу математичного аналізу, за ознакою Лейбніца цей ряд збігається, причому модуль його залишку не перевищує першого відкинутого члена. Це означає, що для обчислень умовою досягнення заданої точності є виконання нерівності $|d_n| \leq 10^{-6}$, де d_n – черговий член ряду. Як тільки на якійсь ітерації виконається ця умова, обчислення можна припиняти.

Кожен член ряду містить відповідні факторіал і степінь x . Зрозуміло, що обчислювати їх необхідно за допомогою множення, тим більше, що мова Pascal не має засобів піднесення до степеня. Чи можна використовувати дві різні змінні з метою накопичення степеня x і факторіала? Наприклад, так:

```
d:=x; s:=0;           {перший член ряду і початкове значення суми}
p:=x; f:=1;          {початкові значення степеня x і факторіала}
n:=1;                {номер члена ряду}
while abs(d)>1e-6 do  {У знакозмінного ряду}
begin if odd(n) then s:=s+d {непарні члени додаємо,}
      else s:=s-d;         {а парні - віднімаємо.}
      inc(n);              {Починаємо рахувати наступний член ряду:}
      p:=p*sqr(x);         {степінь зріс на x^2,}
      f:=f*(2*n-2)*(2*n-1); {факторіал - на два множники,}
      d:=p/f               {маємо член ряду.}
end; {while}
```

У цьому фрагменті і логічно, і синтаксично все написано добре, а все ж так виконувати обчислення суми зазначеного ряду не варто. Вся справа в тому, що черговий член ряду є часткою двох потенційно великих чисел: для великих за модулем x його степені зростає дуже швидко (добре, якщо його величина обмежена, як в нашій задачі, а якщо ні?!), а факторіал, взагалі, є найшвидше зростаючою математичною функцією. Що ж станеться, якщо «дуже велике» поділити на «дуже велике»? Сучасні комп'ютери відображають дійсні числа і виконують обчислення зі скінченною точністю. Наприклад, значення типу *real* у мові Free Pascal містять близько 11-ти точних десяткових знаків мантиси. Припустимо, що у виразі $d:=p/f$ чисельник є величиною порядку 10^{15} («дуже велике»), а знаменник – порядку 10^{20} (ще більше), і кожен з них містить до десяти точних знаків. Тоді порядок точних цифр мантиси знаменника є від 10^{11} до 10^{20} . За правилами виконання обчислень з плаваючою крапкою точні знаки результату теж будуть у цьому діапазоні, але d є величиною порядку 10^{-5} і тому не містить ні одного (!) точного знака.

З метою ефективного обчислення суми заданого ряду доцільно використати інший підхід і вивести рекурентну формулу для обчислення членів ряду. Легко бачити, що $d_n = (-1)^{n+1} x^{2n-1} / (2n-1)!$, а $d_{n+1} = (-1)^n x^{2n+1} / (2n+1)!$. Тоді $d_{n+1}/d_n = -x^2/(2n(2n+1))$, і шукані рекурентні формули мають вигляд $d_1 = x$, $d_{n+1} = -x^2/(2n(2n+1))d_n$, $n=1,2,\dots$. Зауважимо також, що величину $-x^2$ можна обчислити ще перед циклом, а значення n краще збільшувати не на 1, а на 2: не потрібно буде виконувати множення на 2 в знаменнику рекурентної формули:

```

program sinBySeries;
var x,d,s,y:real; n:word;
begin write('Введіть x: '); readln(x);
      y:=-sqr(x);
      d:=x; s:=x; n:=2;
      repeat d:=d*y/(n*(n+1));
            s:=s+d; inc(n,2)
      until abs(d)<=1e-6;
      writeln('s=',s:10:7,' sin=',sin(x):10:7)
end.

```

Пропонуємо читачеві самостійно простежити, як змінюватимуться значення d і s на кількох перших кроках циклу.

Продемонструємо використання описаного підходу на ще одному прикладі.

Задача 34. Дано дійсні числа x, ε ($x \neq 0, \varepsilon > 0$). Обчислити з точністю

$$\varepsilon \text{ значення } s = \sum_{k=0}^{\infty} \frac{(-1)^k x^{4k+3}}{(2k+1)!(4k+3)}.$$

Ряд в умові задачі є знакзмінним, тому для перевірки досягнення заданої точності, як і раніше, використаємо умову $|d_n| \leq \varepsilon$. Для обчислення членів ряду

введемо рекурентні формули: $d_0 = x^3/3$, $d_k = d_{k-1} \cdot \frac{-x^4}{2k(2k+1)} \cdot \frac{4k-1}{4k+3}$, $k = 1, 2, \dots$. Видно, що остання формула містить «зайвий» громіздкий співмножник $(4k-1)/(4k+3)$. Звідки він взявся? Кожен з членів ряду містить у знаменнику множник вигляду $(4k+3)$. У різних d_k ці множники різні, тому їх не доцільно зачислювати до рекурентної формули. Залишимо в ній тільки множники «відповідальні» за обчислення степеня і факторіала, а ділення на $(4k+3)$ виконуватимемо перед додаванням d_k до суми:

```

program seriesB;
var s,x,eps,d,y:real; k:integer;
begin write('Введіть x, eps: '); readln(x,eps);
      y:=sqr(x); d:=x*y; s:=d/3; y:=-sqr(y);
      k:=0;
      while abs(d)>eps do
      begin inc(k,2); d:=d*y/(k*(k+1));
          s:=s+d/(2*k+3)
      end; {while}
      writeln('s(',x,')=',s)
end.

```

9.2. Обчислення кореня алгебричного рівняння

Одним з найпростіших методів обчислення кореня рівняння $f(x)=0$ на проміжку $[a; b]$ є метод поділу відрізка навпіл. Його можна використовувати тоді, коли відомо, що $f(x)$ – неперервна функція, яка один раз змінює свій знак на заданому проміжку. Алгоритм методу дуже простий: обчислюють середину проміжку і значення функції на ній, визначають, на лівій чи правій половині відрізка функція змінює свій знак, і для цієї половини повторюють описані дії. Обчислення припиняють тоді, коли довжина чергового проміжку стає меншою за задану величину точності. За наближене значення кореня рівняння приймають середину останнього проміжку.

Задача 35. Обчислити методом поділу відрізка навпіл корені рівняння $x^2+x-30=0$ на проміжках $[-6,4; -4,7]$ та $[4,7; 5,2]$ з точністю 10^{-4} і корінь рівняння $e^{2x}=3$ на проміжку $[0,5; 1]$ з точністю 10^{-8} .

Метод реалізуємо у вигляді функції. Її вхідними параметрами будуть: a, b – межі відрізка; ε – точність обчислень; f – функція, що описує ліву частину заданого рівняння (параметр процедурного типу). Використаємо також локальні змінні: c – середина відрізка $[a; b]$; fa, fb, fc – значення функції $f(x)$ у точках a, b, c (їх необхідно зберігати, щоб не виконувати повторних обчислень $f(x)$):

```

program equationRoot;
type func=function(x:real):real;           {процедурний тип}

function binaryDivision(f:func; a,b,eps:real):real;
{функція обчислення простого кореня алгебраїчного рівняння}

```

```

{f(x)=0 на проміжку [a; b] з точністю eps методом поділу }
{відрізку навпіл }
var c,fa,fb,fc:real;
begin fa:=f(a); fb:=f(b);
  if fa*fb>0 then {неправильно задано проміжок}
  begin binaryDivision:=0;
    writeln('binaryDivision error: wrong region');
    exit
  end; {if}
  while b-a>eps do
  begin c:=(a+b)/2; fc:=f(c); {обчислили середину}
    {тепер знайдемо новий проміжок}
    if fa*fc<0 then begin b:=c; fb:=fc end {корінь - зліва}
      else begin a:=c; fa:=fc end {корінь - справа}
    end; {while}
    binaryDivision:=(a+b)/2
  end; {binaryDivision}

function parabola(x:real):real; far;
begin parabola:=(x+1)*x-30
end; {parabola}
function exponenta(x:real):real; far;
begin exponenta:=exp(2*x)-3
end; {exponenta}

const eps1=0.0001; eps2=1e-8;
var x1,x2:real; r:real;
begin x1:=binaryDivision(parabola,-6.5,-4.7,eps1);
  x2:=binaryDivision(parabola, 4.7, 5.2,eps1);
  writeln('Корені параболи: ',x1:8:5,x2:8:5); readln;
  r:=binaryDivision(exponenta,0.5,1,eps2);
  writeln('Корінь рівняння exp(2x)=3 ',r:12:9,
    ' точно',ln(3)/2:12:9)
end.

```

Ліві частини заданих рівнянь описано функціями *parabola* і *exponenta*. Директива *far* вказує, що їх необхідно відкомпілювати в режимі побудови далекого адресування, що є необхідним для використання цих функцій у ролі фактичних параметрів при звертанні до *binaryDivision*. Очевидно, що друге рівняння потрібно переписати у вигляді $e^{2x} - 3 = 0$.

У результаті виконання цієї програми отримано такі результати:

Корені параболи: -5.99998 5.00002

Корінь рівняння $\exp(2x)=3$ 0.549306143 точно 0.549306144

Очевидно, що обчислені наближені корені збігаються з точними у межах заданої точності.

9.3. Числове інтегрування

Існує багато аналітичних методів обчислення визначених інтегралів, складено багато довідкових таблиць зі значеннями часто вживаних інтегралів, однак на практиці завжди зустрічаються такі, яких нема в таблицях, і для яких аналітичні методи не працюють. Тоді на допомогу приходять методи числового інтегрування, які дають змогу обчислити наближене значення визначеного інтеграла. Ми розглянемо один з найпростіших методів числового інтегрування

$$\int_a^b f(x) dx \approx h \sum_{i=0}^{n-1} f(x_i) = I_n$$

– метод лівих прямокутників. За цим методом $h = \frac{b-a}{n}$, $x_i = a + ih$. З метою досягнення заданої точності задають деяке значення n , обчислюють і порівнюють I_n та I_{2n} . За правилом Рунге точності ϵ досягнуто, якщо виконується $|I_n - I_{2n}|/3 \leq \epsilon$. Якщо ж ні, то обчислюють I_{4n} і порівнюють його з I_{2n} і т. д.

Обчислення такої суми, як у формулі лівих прямокутників, легко запрограмувати з використанням одного простого циклу (використаємо тип *func* з попередньої програми):

```
function leftR(a,b:real; f:func; n:integer):real;
var i:integer; h,s:real;
begin h:=(b-a)/n; s:=0;
      for i:=0 to n-1 do s:=s+f(a+i*h);
      leftR:=s*h
end;
```

Здавалося б, тепер достатньо викликати цю функцію потрібну кількість разів, щоб виконати обчислення з заданою точністю:

```
var I1,I2:real;
begin ... n:=10; {задали початкове значення n}
      I1:=leftR(a,b,f,n); {обчислили значення I(n)}
      I2:=leftR(a,b,f,2*n); {та I(2n)}
      while abs(I1-I2)>eps do {запам'ятали старе значення,}
        begin I1:=I2; n:=n*2; {подвоїли кількість доданків}
              I2:=leftR(a,b,f,2*n) {i рахуем нове значення}
        end; {while}
      writeln(I2); ... {надрукували результат}
```

Справді, тут нема ні синтаксичних, ні логічних помилок, однак такий фрагмент програми дуже нераціональний. Він примушує функцію *leftR* виконувати багато зайвих обчислень. Яких? Функція *leftR* щоразу обчислює значення $f(x_i)$. Нехай $a=0$, $b=1$, тоді під час першого виклику *leftR* буде обчислено значення f у точках 0, 0.1, 0.2, ..., 0.9, а під час другого – в точках 0, 0.05, 0.1, 0.15, ..., 0.9, 0.95. Бачимо, що друга множина точок містить усі точки першої, і саме в них обчислення $f(x_i)$ будуть виконані удруге. Така ж ситуація буде повторюватися під час кожного наступного виклику *leftR*: якщо кількість точок x_i щоразу подвоювати (як ми це робимо), то тільки половина з них буде

новою, і тільки для них потрібно обчислювати $f(x_i)$, а для «старої» половини можна використати значення, обчислені на попередній ітерації. У наведеному ж вище фрагменті функція $leftR$ для кожного нового значення n обчислює всі значення $f(x_i)$.

Щоб уникнути зайвих обчислень, удосконалимо функцію $leftR$ і «заховаємо» перевірку точності в її тілі. Обмежимо також кількість подвоєнь n : якщо цього не зробити, існуватиме великий ризик зациклення програми для тих підінтегральних функцій, для яких важко досягти високої точності обчислення інтеграла за допомогою методу лівих прямокутників:

```

function leftRect(f:func; a,b,eps:real):real;
  {інтеграл від f на проміжку [a;b] методом лівих прямокутників}
  {p - значення I(2n); q - значення I(n); s - використовується }
  {для постійного накопичення суми значень f(i) }
  var s,h,h2,a2,p,q:real; n,i,k:word;
begin n:=15; h:=(b-a)/n; s:=0;
  for i:=0 to n-1 do s:=s+f(a+i*h); {перше значення інтеграла}
  p:=s*h; q:=0; k:=0;
  while (abs(p-q)/3>eps) and (k<13) do
  begin inc(k); q:=p; {запам'ятали попереднє I(n)}
    h2:=h/2; a2:=a+h2;
    for i:=0 to n-1 do s:=s+f(a2+h*i); {доповнили суму}
    p:=s*h2; {отримали нове значення I(2n)}
    h:=h2; n:=n+n {подвоїли кількість доданків}
  end; {while} leftRect:=p
end; {leftRect}

```

Таку підпрограму вже можна використовувати на практиці. Покажемо, як з її допомогою обчислюють означені інтеграли: одновимірні, одновимірні залежні від параметра, а також подвійні.

Задача 9.4. Обчислити методом лівих прямокутників $\int_0^{\pi} \sin x \, dx$ з

точністю 10^{-6} , серію інтегралів $S(t) = \int_{-1}^1 (t^2 x^2 + 1) \, dx$ для $t=0(0,5)3$ з

точністю 10^{-4} і значення $R = \int_0^1 \int_0^x (x+y)^2 \, dy \, dx$ з точністю 10^{-3} .

Підінтегральну функцію у $leftRect$ передають за допомогою параметра $f:func$, тобто, за допомогою функції з одним аргументом. Як обчислити інтеграл $S(t)$, коли підінтегральна функція залежить від числового параметра t ? Його не можна включити до формальних параметрів, тому використаємо для цього глобальну змінну (у програмі це змінна $t:real$).

Як пристосувати $leftRect$ для обчислення подвійних інтегралів? Можна використати зведення подвійного інтеграла до повторного. Тоді зовнішній інтеграл (по аргументу x) можна обчислити за допомогою $leftRect$ зі «спеціальною» підінтегральною функцією, яка в свою чергу містить виклик $leftRect$ для обчислення внутрішнього інтеграла (по аргументу y). У цьому

випадку в програмі буде неявна рекурсія, проте Pascal не накладає обмежень на такі виклики підпрограм:

```

program numericalIntegration;
type func=function(x:real):real;
function leftRect(f:func; a,b,eps:real):real;
var s,h,h2,a2,p,q:real; n,i,k:word;
begin n:=15; h:=(b-a)/n; s:=0;
    for i:=0 to n-1 do s:=s+f(a+i*h);
    p:=s*h; q:=0; k:=0;
    while (abs(p-q)/3>eps) and (k<13) do
    begin inc(k); q:=p; h2:=h/2; a2:=a+h2;
        for i:=0 to n-1 do s:=s+f(a2+h*i);
        p:=s*h2; h:=h2; n:=n+n
    end; {while} leftRect:=p
end; {leftRect}

var t:real; y:real; {глобальні змінні}
const eps=0.001;

function test(x:real):real; far;           {для простого інтеграла}
begin test:=sin(x)
end; {test}
function funParam(x:real):real; far; {для інтеграла, залежного}
begin funParam:=sqr(x*t)+1                {від параметра t}
end; {funParam}
function twoArg(x:real):real; far;       {для подвійного інтеграла}
begin twoArg:=sqr(x+y)                    {тут y - теж глобальна змінна}
end; {twoArg}
function firstInt(x:real):real; far;
{функція, що обчислює внутрішній інтеграл у повторному}
begin y:=x;
    firstInt:=leftRect(twoArg,0,x,eps)
end; {firstInt}

var i:integer;
begin {обчислимо одновимірний інтеграл}
    writeln('test=',leftRect(test,0,pi,1e-6):10:7); readln;
    writeln('  t      S(t)');           {a тепер - серію S(t)}
    writeln('-----');
    for i:=0 to 6 do
    begin t:=i/2; writeln(t:5:1,
        leftRect(funParam,-1,1,0.0001):10:6)
    end; readln;
        {і нарешті - подвійний}
    writeln('two',leftRect(firstInt,0,1,eps):9:4)
end.

```

Після виконання програми отримаємо такі результати.

```
test= 1.9999996
```

t	S (t)
0.0	2.000000
0.5	2.166759
1.0	2.666759
1.5	3.500052
2.0	4.666759
2.5	6.166703
3.0	8.000052

two 0.5830

Легко перевірити, що вони збігаються з точними із заданою точністю.

10. Покрокова розробка програм

Метод покрокової розробки програм використовують з метою спрощення роботи з великими програмами: для їхнього створення, перевірки правильності, модифікації.

Запровадження цього методу зумовлене обмеженими розумовими здібностями людини: неможливо, наприклад, охопити програму, що складається з кількох тисяч рядків. Дуже складно, або і неможливо одразу почати писати програму для нової незнайомої задачі. Завжди доводиться виконувати певну підготовчу роботу, аналізуючи умову задачі та можливості комп'ютера, обмірковувати можливі варіанти майбутнього алгоритму.

Ідея методу покрокової розробки полягає у побудові послідовності програм, розрахованих на виконавців різної кваліфікації: від уявних, «надзвичайно розумних» спочатку до звичайного Pascal-компілятора наприкінці. Початкова програма складається з кількох «абстрактних» операторів, виконати які в змозі лише наш уявний виконавець. Перехід до нової програми уточнює, деталізує ці оператори, «розшифровує» їх за допомогою дрібніших, ближчих до мови програмування. Деталізація закінчується, коли чергова програма цілковито записана операторами мови. Зазвичай, метод покрокової розробки алгоритмів застосовують разом з дотриманням принципів структурного програмування і називають його *структурним програмуванням «зверху-донизу»*.

Нагадаємо, що структурована програма використовує лише ті структури керування, які мають один вхід і один вихід: послідовні дії, галуження, цикли. Якщо на кожному етапі розробки програми для деталізації абстрактних операторів використовувати тільки такі структури, то вона буде наділена вбудованою крок за кроком правильністю та надійністю.

Деталізацію передусім виконують для тих частин програми, які не залежать від інших частин і не зазнаватимуть ніяких змін надалі. Проілюструємо тепер все сказане на прикладі обчислення великого степеня числа 2 – такого, що результат неможливо зобразити за допомогою числа типу *integer* чи *longint* (наприклад, 2^{1000}).

Задача 37. Дано натуральне число n ($n \leq 1000$). Обчислити і надрукувати 2^n .

На найвищому рівні абстракції маємо таку програму:

```

program power0;
begin надрукувати_заданий_ступінь_2
end.

```

Програма *power0* настільки абстрактна, що може хіба ствердити наше бажання обчислити ступінь двійки. Необхідно деталізувати її. Поміркуємо: завжди потрібно вводити вхідні дані задачі і завжди потрібно використовувати певні структури даних для відображення результатів. Тому отримаємо конкретнішу програму:

```

program power1;
  var n:integer; x:велике_число;
begin write('Введіть n: '); readln(n);
      write('2 в степені ',n,'=');
      обчислити_2^n_в_змінній_x;
      надрукувати_велике_число(x)
end.

```

Введення вхідних даних є очевидним, тому ми записали його вже операторами мови Pascal. Деталізовано спосіб отримання результату: обчислення і виведення виконується різними операторами програми *power1*. Ці дії відокремлені одна від одної і далі можуть уточнюватись незалежно. Під час деталізації ми дотримувались принципів структурного програмування: абстрактний оператор з *power0* замінено *послідовністю* операторів у *power1*.

Звернемо увагу на одну важливу обставину: деталізація, з одного боку, уточнює і пояснює великі оператори, а з іншого – звужує клас можливих розв'язків. Наприклад, *power1* вже відхилила всі програми, які обчислюють і друкують результат по одній цифрі.

Що деталізувати далі? Рішення необхідно приймати для тих операторів, які не залежать від ще не прийнятих рішень щодо інших операторів. Спосіб виведення *x* залежить від вибору структури даних, тому спочатку деталізуємо обчислення степеня:

```

program power2;
  var n:integer; x:велике_число;
      k:integer;
begin write('Введіть n: '); readln(n);
      write('2 в степені ',n,'=');
      {обчислити_2^n_в_змінній_x}
      велике_число_x:=1;
      for k:=1 to n do подвоїти_велике_число(x);
      надрукувати_велике_число(x)
end.

```

У цьому випадку для обчислення 2^n вибрано послідовне множення, а не, наприклад, швидке піднесення до степеня. Для деталізації використано дві структури керування: *послідовність* і *цикл*.

Подальша деталізація можлива лише після вибору структури даних для реалізації типу *велике_число*. Такий тип можна було б змоделювати за допомогою масиву або лінійного односпрямованого списку. Залежно від розміру

елемента масиву (чи списку) його використовують з метою зберігання різної кількості цифр великого числа: якщо розмір байт (тип *byte*), то один елемент масиву містить одну або дві цифри числа; якщо два байти (тип *word*), то один елемент може містити до чотирьох цифр числа; якщо чотири байти (тип *longint*), – до дев'яти цифр. Зупинимось на найпростішому варіанті і використаємо масив байтів, кожен з яких використовується для зберігання однієї цифри великого числа. У цьому випадку визначають максимальну довжину масиву і використовують додаткову змінну для зберігання номера останнього зайнятого елемента масиву. Легко обчислити кількість цифр у числі 2^{1000} : $[\lg 2^{1000}] + 1 = 302$. Тепер тип *велике_число* можна конкретизувати:

```
const long=302;
type largeNumber=record
    digits:array[1..long]of byte;
    last:1..long end;
var x:largeNumber;
```

Для зручності обчислень масив *x.digits* міститиме цифри числа *x* у зворотньому порядку: наймолодша цифра записана в перший елемент масиву. Тепер можна конкретизувати інші абстрактні оператори з програми *power2*.

Замість *велике_число_x:=1*

```
with x do
begin last:=1; digits[last]:=1 end;
```

Щоб уточнити *подвоїти_велике_число(x)*, пригадаємо алгоритм множення числа на 2 у стовпчик: необхідно помножити кожен цифру на 2, додати до неї перенесення з попереднього розряду, записати число одиниць отриманої суми і перенести число її десятків до наступного розряду. Для запам'ятовування проміжних результатів (суми і перенесення) використаємо додаткові змінні *temp* і *transfer*. Зауважимо також, що перенесення не додається до наймолодшої цифри числа, але цю особливість легко подолати, поклавши перед початком множення *transfer=0*. Остаточо, замість *подвоїти_велике_число(x)*

```
var i:1..long;           {номер цифри, яку опрацьовуємо}
    transfer:byte;       {перенесення до старшого розряду}
    temp:byte; {робоча змінна для результату множення на 2}
transfer:=0;
for i:=1 to x.last do
begin temp:=2*x.digits[i]+transfer;
    x.digits[i]:=temp mod 10;
    transfer:=temp div 10
end; if transfer>0 then with x do {ВИДОВЖИМО ЗАПИС ЧИСЛА x}
begin inc(last); digits[last]:=transfer end;
```

Оператор *надрукувати_велике_число(x)* є звичайною послідовною обробкою елементів масиву. Єдиною особливістю є те, що першою в масиві стоїть остання цифра шуканого числа, тому друк розпочинаємо з останнього елемента масиву. Таким чином, замість *надрукувати_велике_число(x)*:

```

var i:1..long;
for i:=x.last downto 1 do write(x.digits[i]);
writeln;

```

Ми записали мовою Free Pascal усі абстрактні оператори і оголошення. Отже, деталізацію завершено. Тепер можна просто зібрати все в одну програму. Проте краще оформити «мудрі» оператори у вигляді процедур. Адже процедура – це і є великий («мудрий») оператор:

```

program largePower;
const long=302;
type largeNumber=record
    digits:array[1..long]of byte;
    last:1..long end;
var x:largeNumber;
    n:integer;
procedure calculate2power(n:integer; var x:largeNumber);
var k,i:1..long; transfer,temp:byte;
begin with x do
    begin last:=1; digits[last]:=1 end;           {x=2^0}
    for k:=1 to n do
    begin transfer:=0;
        for i:=1 to x.last do
        begin temp:=2*x.digits[i]+transfer;
            x.digits[i]:=temp mod 10;
            transfer:=temp div 10
        end;{for i} if transfer>0 then with x do
            begin inc(last); digits[last]:=transfer
                end                                     {x=2^k}
            end;{for k}                                 {x=2^n}
        end;{calculate2power}
procedure writeLargeNumber(var x:largeNumber);
var i:1..long;
begin for i:=x.last downto 1 do write(x.digits[i]);
    writeln
end;{writeLargeNumber}
Begin write('Введіть n: '); readln(n);
    write('2 в степені ',n,'=');
    calculate2power(n,x);
    writeLargeNumber(x)
End.

```

Обчислення великого степеня числа 2 можна виконати і по-іншому, якщо вибрати швидкий алгоритм піднесення до степеня, або якщо обрати іншу структуру даних для зберігання великого числа. З метою порівняння наведемо ще одну програму обчислення 2^n , у якій використано лінійний односпрямований список. У цьому випадку немає практично жодних обмежень на величину показника n . Таке обмеження ми раніше використовували для обчислення найбільшого розміру масиву. Проте розмір списку може змінюватися динамічно: за необхідністю до нього долучають нові ланки для запису нових цифр числа.

Для елементів списку використаємо тип `word`, що дасть змогу записувати в них від одної до чотирьох цифр великого числа і зменшити удвічі порівнянно з попереднім розв'язком кількість елементів структури даних:

```

program powerByList;
  type largeNumber=^section;
    section=record digit:word; next:largeNumber end;
  var x:largeNumber; n:integer;
procedure calculate(n:integer; var x:largeNumber);
  var k:integer; temp,transfer:word; p:largeNumber;
begin x:=new(largeNumber);           {додаткова ланка для зручної}
  x^.next:=new(largeNumber);         {роботи зі списком}
  with x^.next^ do
  begin digit:=1; next:=nil end;
  for k:=1 to n do
  begin transfer:=0; p:=x;
    while p^.next<>nil do with p^.next^ do
    begin temp:=digit shl 1+transfer;   {shl 1 - це *2}
    if temp<10000
    then begin transfer:=0; digit:=temp end
    else begin transfer:=1; digit:=temp mod 10000 end;
    p:=p^.next
    end;{while & with}
    if transfer>0 then
    begin p^.next:=new(largeNumber); with p^.next^ do
      begin digit:=1; next:=nil end
    end {if}
    end;{for}
  p:=x; x:=p^.next; dispose(p)       {вилучили зайву ланку}
end;{calculate}
procedure writeListReverse(var x:largeNumber);
var p,q:largeNumber;
begin q:=nil;
  while x<>nil do {обернемо порядок цифр у записі числа}
  begin p:=x; x:=p^.next; p^.next:=q; q:=p end;
  while q<>nil do {друкуємо число і звільняємо пам'ять}
  begin write(q^.digit); p:=q; q:=p^.next; dispose(p)
  end; writeln
end;{writeListReverse}
Begin write('Введіть n: '); readln(n);
  write('2^',n,'=');
  calculate(n,x);
  writeListReverse(x)
End.

```

Ми свідомо не деталізуємо опис процесу створення програми *powerByList*. Структуровану програму легко модифікувати, пристосовуючи до нових умов (до використання іншої структури даних у нашому випадку). Читач має змогу самостійно порівняти ці дві програми, зазначити їхні особливості, зумовлені відмінністю використаних структур даних і різними типами елементів цих структур.

11. Рекурсія

Рекурсія – це спосіб визначення деякого поняття самого через себе. Класичним є приклад означення ідентифікатора, записаного формулами Наура-Бекуса:

$$\langle \text{ідентифікатор} \rangle ::= \langle \text{буква} \rangle \mid \langle \text{ідентифікатор} \rangle \langle \text{буква} \rangle \mid \langle \text{ідентифікатор} \rangle \langle \text{цифра} \rangle.$$

Це означення складається з трьох альтернатив. Перша з них дає означення найпростішого ідентифікатора: ідентифікатора, що складається лише з однієї букви (літери латинського алфавіту). Дві інші альтернативи вказують, що «довгий» ідентифікатор – це «коротший» ідентифікатор, до якого дописано літеру або цифру.

Рекурсію в програмуванні використовують як потужний засіб побудови алгоритмів. Рекурсивний розв'язок задачі завжди містить дві частини. Перша задає «елементарний» розв'язок – розв'язок задачі з найпростішими вхідними даними чи задачі найменшого розміру. Друга частина рекурсивного розв'язку описує, як отримати розв'язок задачі більшого розміру за допомогою розв'язку задачі меншого розміру (чи кількох задач менших розмірів). Дуже часто постановки задач відразу містять у собі і рекурсивний розв'язок (наприклад, задача обчислення чисел Фібоначчі).

Задача 38. Числа Фібоначчі задано рекурентними співвідношеннями $f_0 = f_1 = 0$; $f_n = f_{n-1} + f_{n-2}$, $n=2, 3, \dots$. Дано натуральне число n . Обчислити f_n .

Тут розміром задачі є порядковий номер n числа. Найпростішим обчислення f_n є для $n=0$ або для $n=1$. Для більших значень n задача зводиться до таких самих задач розміру $n-1$ і $n-2$.

У мові програмування Free Pascal будь-яка процедура може бути рекурсивною (може викликати сама себе), тому рекурсивні розв'язки задач легко програмувати. Обчислення потрібного числа Фібоначчі можна виконати за допомогою такої функції:

```
function Fibo(n:integer):longint;
begin if n<2 then Fibo:=1 {елементарний розв'язок}
      else Fibo:=Fibo(n-1)+Fibo(n-2) {зведення}
end; {Fibo}
```

Рекурсивні розв'язки можна будувати і в інших, не таких очевидних випадках.

Задача 39. Дано натуральне число n . Обчислити $n!$.

Відомо, що $n!=1 \times 2 \times \dots \times n$. Проте можна вказати і рекурсивний розв'язок цієї задачі: $n!=1$, якщо $n=0$, або $n=1$; $n!=n \times (n-1)!$, якщо $n>1$. Тепер легко отримати:

```
function factorial(n:integer):longint;
begin if n<2 then factorial:=1 {елементарний розв'язок}
      else factorial:=n*factorial(n-1) {зведення}
end; {factorial}
```

Приваблива простота! Однак наведені приклади демонструють, як **не варто** застосовувати рекурсію. Обчислення факторіала можна швидше виконати за допомогою такого оператора циклу:

```
factorial:=1;
for i:=1 to n do factorial:=factorial*i;
```

Зазначимо, що одна ітерація циклу виконується набагато швидше і потребує менше пам'яті, ніж один виклик підпрограми, як у функції *factorial*.

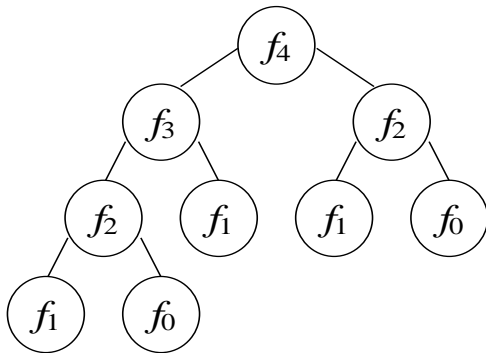


Рис. 4. Схема рекурсивних викликів функції *Fibo* під час обчислення f_4

З функцією *Fibo* ситуація ще гірша. Проаналізуємо, як відбуватиметься з її допомогою обчислення числа f_4 . Чотири не менше, ніж два, тому *Fibo*(4) викличе *Fibo*(3) і *Fibo*(2). Три також не менше, ніж два, тому *Fibo*(3) викличе *Fibo*(2) і *Fibo*(1) і т. д. Увесь процес зображено на рис. 4. Отже, що існує багато повторних викликів функції з однаковими аргументами: *Fibo*(0) та *Fibo*(2) викликаються двічі, а *Fibo*(1) – навіть тричі. Зі збільшенням номера n кількість зайвих обчислень зростає експотенційно. Водночас у п. 2.3 було наведено приклад простого циклічного алгоритму, що використовує три змінні для обчислення чисел Фібоначчі.

Безумовно, у цій задачі необхідно віддати перевагу саме циклічному алгоритмові.

Цей приклад наглядно демонструє підступність рекурсії. Не можна застосовувати її бездумно тільки тому, що так буде зручно написати програму. Для кожного рекурсивного алгоритму існує нерекурсивний, який реалізує таку ж задачу, бо рекурсія – це лише спосіб запису алгоритму. Проте існує цілий клас задач, для яких рекурсія є необхідним способом отримання розв'язку. З кількома такими задачами ми вас зараз познайомимо.

11.1. Задача про Ханойські вежі

Пригадайте, як виглядає вежа буддійського храму: безліч трикутних куполів з припіднятими краями, побудованих один над одним, причому верхній купол завжди менший за нижній. Схоже на карпатську смереку. Задача завдячує своєю назвою саме виглядові цих споруд, а мова в ній йтиме про трішки інші «вежі».

Задача 40. На горизонтальній дошці закріплено три вертикальні стержні. На лівий стержень нанизано n дисків різного розміру, на більшому диску лежить менший (як зображено на рис. 5). Необхідно перемістити всі диски з лівого стержня на правий, не порушуючи їхнього впорядкування. Диски не можна класти збоку від стержнів, не можна класти диск більшого розміру на менший. Переносити з одного стержня на інший можна тільки по одному диску, середній стержень можна використовувати для тимчасового зберігання дисків.

Схоже, ми зібралися погратися в дитячі пірамідки. Проте розв'язок цієї задачі дитячим не назвеш (хіба що для $n=1$ чи $n=2$). Умову задачі можна переформулювати і в термінах інформатики: дано три стеки; перший з них містить послідовність різних цілих чисел, впорядкованих за спаданням; необхідно перемістити послідовність з першого стека в третій, використовуючи другий як робочий і ніколи не порушуючи впорядкованості чисел у кожному зі стеків.

Що є розв'язком цієї задачі? Очевидно, послідовність команд про переміщення дисків з одного стержня на інший. Наприклад, для $n=2$ вона матиме вигляд «З лівого – на середній. З лівого – на правий. З середнього – на правий». Як побудувати такий розв'язок? Помилково намагатися записувати розв'язки вручну для різних значень n . Такий підхід не наблизить нас до побудови рекурсивного розв'язку. Тут корисно застосувати дещо специфічний спосіб мислення.

Як велику задачу звести до меншої? Як отримати розв'язок великої задачі, коли відомо розв'язок меншої? Якби $n-1$ диск уже був на середньому стержні (див. рис. 6), то нам залишилось би перемістити останній диск з лівого стержня на правий і знову $n-1$ диск з середнього на правий. А переміщення $n-1$ диска і є тією задачею меншого розміру, яка нам так необхідна! Остаточний рекурсивний розв'язок задачі про Ханойські вежі можна сформулювати так: *Якщо $n=1$, то перемістити диск зліва направо. Інакше перемістити $n-1$ диск з лівого стержня на середній, використовуючи правий стержень як робочий; перемістити n -ий диск зліва направо; перемістити $n-1$ диск з середнього стержня на правий, використовуючи лівий стержень як робочий.* Позначимо лівий, середній, правий стержні літерами L , M , R відповідно, і для зручності програмування запишемо цей розв'язок за такою схемою:

$$n : L \xrightarrow{M} R = \begin{cases} L \rightarrow R, & n = 1, \\ (n-1) : L \xrightarrow{R} M, L \rightarrow R, (n-1) : M \xrightarrow{L} R, & n \geq 2. \end{cases}$$

Тепер необхідно вирішити, як саме програма друкуватиме команди щодо переміщення дисків. Для позначення стержнів ми могли б використати значення деякого перелічуваного типу. Наприклад:

```
type position=(left,middle,right);
```

Для друкування назви стержнів доцільно використати таку типізовану константу:

```
const pName:array[position,1..6] of
char =
```

```
(' Left ', 'Middle', ' Right');
```

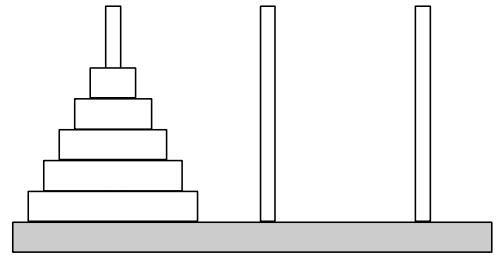


Рис. 5. Початкове розташування дисків у задачі про Ханойські вежі для $n=5$

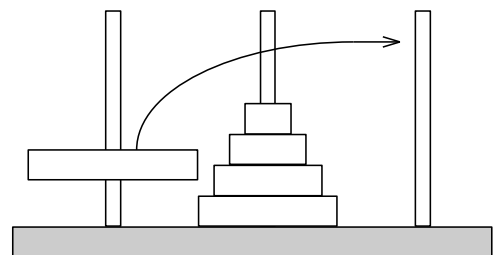


Рис. 6. Щоб перемістити останній диск, необхідно прибрати всі менші

та оголосити окрему процедуру виведення команди щодо переміщення одного диска. Остаточною програмою розв'язування задачі про Ханойські вежі матиме вигляд:

```

program Towers;
type position=(left,middle,right);
procedure moveDisk(from,_to:position);
  {друкує команду щодо переміщення одного диска}
  const pName:array[position,1..6]of char =
    (' Left ', 'Middle', ' Right');
  begin writeln(pName[from], ' --> ',pName[_to])
  end; {moveDisk}
procedure moveTower(h:byte; from,_to,work:position);
  {будує послідовність команд про переміщення вежі з
  h дисків зі стержня from на стержень _to}
begin if h=1 then moveDisk(from,_to)  {найменша задача}
  else {зведення до двох задач розміру h-1}
  begin moveTower(h-1,from,work,_to);
    moveDisk(from,_to);
    moveTower(h-1,work,_to,from)
  end {if}
end; {moveTower}
var n:byte;
Begin write('Введіть висоту вежі: '); readln(n);
  moveTower(n,left,right,middle)
End.

```

Для $n = 4$ ця програма друкує такий розв'язок:

```

Введіть висоту вежі: 4
Left  --> Middle
Left  --> Right
Middle --> Right
Left  --> Middle
Right --> Left
Right --> Middle
Left  --> Middle
Left  --> Right
Middle --> Right
Middle --> Left
Right --> Left
Middle --> Right
Left  --> Middle
Left  --> Right
Middle --> Right

```

Без використання рекурсії отримати цей розв'язок було б значно складніше.

11.2. Алгоритм швидкого сортування

У р. 5 ми розглянули кілька поширених алгоритмів упорядкування послідовності чисел. Опишемо ще один такий алгоритм. Його побудовано з використанням рекурсії.

Розглянемо таке перетворення масиву: нехай дано масив a_1, a_2, \dots, a_n . Необхідно переставити його елементи так, щоб спочатку в масиві розташувати групу елементів, менших за початкове значення a_1 , потім – власне це значення, а потім – групу більших елементів. Для цього необхідно виконати не більше ніж $n-1$ порівнянь і $n-1$ переміщень.

Використовуючи описане перетворення, можна побудувати рекурсивний алгоритм упорядкування масиву – *швидке впорядкування*. Якщо масив містить один елемент, то він впорядкований. У протилежному випадку застосовуємо до нього описане перетворення і визначаємо результат упорядкування так: спочатку в масиві розташовано першу групу елементів, упорядкованих за допомогою алгоритму швидкого сортування; потім без змін той елемент, який відокремлював першу і другу групи елементів; далі – другу групу елементів, упорядкованих за допомогою алгоритму швидкого сортування. Цей алгоритм не використовує додаткового масиву і потребує в середньому приблизно $n \log_2 n$ порівнянь і стільки ж переміщень елементів. Проте це лише середнє значення: у найгіршому випадку кількість порівнянь досягатиме $n(n-1)/2$.

З метою складення процедури швидкого сортування нам доведеться спочатку відповісти на кілька запитань. Перше з них: *як виконати однократне перетворення масиву?* Щоб відокремити дві групи елементів масиву – менших і більших за значення першого елемента – необхідно один раз перебрати елементи масиву, порівняти їх з цим значенням і виконати необхідні переміщення. Зауважимо, що у цьому випадку індекс елемента, в якому зберігатиметься перше значення, завжди буде меншим за індекс елемента, зі значенням якого його порівнюють. Для зберігання цих індексів використаємо змінні відповідно i та j . Отже, якщо черговий елемент більший за перше значення, то переходимо до наступного. У протилежному випадку групу елементів від i -го до $j-1$ -го необхідно перемістити на одну позицію вперед, а j -й елемент – на місце, що звільнилося. Схожі зміни відбуваються з масивом під час сортування методом бульбашки. Тільки тут «підіймається» не один (перший) елемент, а ціла група елементів, значення яких не менші за значення першого.

Сортування частин масиву можна виконати за допомогою рекурсивних викликів процедури сортування, однак у цьому випадку виникає друге запитання: *як передавати цій процедурі частини масиву?* Адже в мові Pascal діють правила строгої типізації, і відповідні формальні та фактичні параметри мусять мати **еквівалентні** типи. Щоб не виникало суперечностей під час звертання до процедури швидкого сортування для передавання їй масиву (чи довільної його частини), використаємо безтиповий параметр, який в тілі процедури наділятимемо типом «масив дійсних чисел». Відповідним йому фактичним параметром може бути довільний елемент масиву, що є першим у тій частині, яка підлягає впорядкуванню. Другий параметр процедури повинен задавати кількість елементів цієї частини.

Нижче наведено текст програми з оголошеною процедурою quickSort і двома викликами її для впорядкування масивів x та y . Ці масиви оголошено як типізовані константи:

```
program quick;
type v1=array[1..10]of real;
```

```

v2=array[1..17]of real;
const x:v1=(5,3,9,2,1,8,5,6,2,1);
      y:v2=(10,9,8,7,6,5,4,3,2,1,0,-1,-2,-3,-4,-5,-6);
procedure writeRealMas(var a; n:integer);
{процедура виведення масиву a дійсних чисел розміру n<=2000}
type mas=array[1..2000]of real;
var i:integer;
begin for i:=1 to n do write(mas(a)[i]:8:3);
      writeln
end;{writeRealMas}
procedure quickSort(var a; n:integer);
{процедура швидкого сортування масиву a дійсних чисел
розміру n<=2000}
type mas=array[1..2000]of real;
var c:real; i,j,k:integer;
begin i:=1; j:=2;                               {перетворення масиву}
      while j<=n do
        if mas(a)[i]<=mas(a)[j] then inc(j) else
          begin c:=mas(a)[j];                     {переміщуємо групу більших}
                for k:=j-1 downto i do mas(a)[k+1]:=mas(a)[k];
                mas(a)[i]:=c; inc(i); inc(j)
          end;{if & while}
        if i>2 then quickSort(a,i-1);             {впорядковуємо менші}
        if n-i>2 then quickSort(mas(a)[i+1],n-i) {та більші}
      end;{quickSort}
Begin writeln('Масив x'); writeRealMas(x,10);
      quickSort(x,10); writeln('Після впорядкування');
      writeRealMas(x,10); readln;
      writeln('Масив y'); writeRealMas(y,17);
      quickSort(y,17); writeln('Після впорядкування');
      writeRealMas(y,17)
End.

```

Результати роботи програми:

Масив x

5.000 3.000 9.000 2.000 1.000 8.000 5.000 6.000 2.000 1.000

Після впорядкування

1.000 1.000 2.000 2.000 3.000 5.000 5.000 6.000 8.000 9.000

Масив y

10.000 9.000 8.000 7.000 6.000 5.000 4.000 3.000 2.000 1.000
0.000 -1.000 -2.000 -3.000 -4.000 -5.000 -6.000

Після впорядкування

-6.000 -5.000 -4.000 -3.000 -2.000 -1.000 0.000 1.000 2.000 3.000
4.000 5.000 6.000 7.000 8.000 9.000 10.000

Упорядкування частини масиву за алгоритмом швидкого сортування подібне до сортування цілого масиву. Цю особливість алгоритму природньо визначати у термінах рекурсії, що ми і продемонстрували на прикладі останньої програми.

11.3. Обхід двійкового дерева

Рекурсивність алгоритму часто буває зумовлена особливостями структури даних, для опрацювання якої його розроблено. Наприклад, алгоритми обходу двійкових дерев.

Деревовидною структурою (деревом) називають множину взаємозв'язаних об'єктів, розташованих за рівнями за таким правилом:

- на першому рівні – один вузол – *корінь* дерева;
- будь-який вузол X наступного, i -го ($i \neq 0$) рівня пов'язаний лише з одним вузлом Y попереднього, $(i-1)$ -го рівня.

У такому випадку Y називають безпосереднім предком вузла X , а X – безпосереднім нащадком Y . Якщо вузол немає нащадків, то його називають *листочком*. Усі вузли, крім кореневого, які мають нащадків, називають внутрішніми вузлами, або *вершинами*.

Максимальну кількість безпосередніх нащадків того самого предка називають *степенем дерева*. Дерева степеня два називають *двійковими (бінарними) деревами*. Щоб реалізувати у Pascal-програмі бінарне дерево, використовують такі оголошення:

```

type treeElementType=... {тип інформаційної частини
    вершини може бути довільним залежно від потреби}
    tree=^node;           {дерево}
    node=record elem: treeElementType; {елемент}
        left, right: tree {зв'язок з нащадками}
    end; {node}

```

Дерево за своєю природою є рекурсивною структурою даних. Адже його означення можна сформулювати так: дерево з базовим типом Node – це або порожнє дерево, або деяка вершина типу Node зі скінченим числом зв'язаних з нею окремих дерев з базовим типом Node, які називають піддеревами. Тому для обходу дерева використовують рекурсивні алгоритми. Під час обходу алгоритм повинен опрацювати всі вершини дерева, кожна – один раз. Дерево обходять, щоб відшукати певний елемент, роздрукувати всі елементи тощо. Існують різні типи обходу (наприклад, лівосторонній, правосторонній, за рівнями).

Під час лівостороннього обходу першими опрацьовують елементи лівого піддерева, а потім – правого. Існує три способи лівостороннього обходу, що відрізняються порядком опрацювання кореня: *прямий* (preorder), за яким спочатку опрацьовують корінь, далі – рекурсивно ліве піддерево, а наприкінці – праве; *обернений* (inorder), за яким спочатку рекурсивно опрацьовують ліве піддерево, потім – корінь, а наприкінці – праве; *кінцевий* (postorder), за яким спочатку опрацьовують ліве та праве піддерева а наприкінці – корінь. Ми використовуватимемо *зворотній лівосторонній* обхід. Його алгоритм можна сформулювати так: якщо дерево непорожнє, то необхідно обійти ліве піддерево, переглянути корінь, обійти праве піддерево.

Задача 41. Дано двійкове дерево, елементами якого є цілі числа.

Написати функцію обчислення суми елементів цього дерева.

Розв'язання:

```

type tree=^node;           {дерево}
      node=record elem:integer; {treeElementType=integer}
          left, right: tree  {зв'язок з нащадками}
      end; {node}
function sum(t:tree):integer;
begin if t= nil then sum:=0
      else with t^ do sum:=sum(left)+elem+sum(right)
end; {sum}

```

Обхід дерева ця функція виконує за допомогою рекурсивних викликів, а опрацювання елемента полягає у додаванні його до суми. Кількість рекурсивних викликів у тілі функції можна зменшити, якщо виконувати додаткові перевірки:

```

function sum1(t:tree):integer;
var s:integer;
begin if t= nil then sum1:=0
      else with t^ do
          begin s:=elem;
              if left<>nil then s:=s+sum1(left);
              if right<>nil then s:=s+sum1(right);
              sum1:=s
          end; {sum1}

```

Тут рекурсивні виклики виконують лише для непорожніх піддерев, однак є певна надлишковість у перевірках, бо кожен екземпляр функції *sum1* перевіряє, чи непорожнє передане йому дерево. Така перевірка доцільна тільки тоді, коли *sum1* викликають вперше. Припустимо, що перший виклик функції завжди виконують тільки для непорожніх дерев. Запишемо:

```

function sum2(t:tree):integer;
var s:integer;
begin with t^ do
      begin s:=elem;
          if left<>nil then s:=s+sum2(left);
          if right<>nil then s:=s+sum2(right);
          sum2:=s
      end; {sum2}

```

Для обчислення суми елементів дерева ми використали в *sum* лівосторонній обхід, а в *sum1* і *sum2* – префіксний. Зрозуміло, що в даному випадку це ніяк не впливає на результат.

Наведемо ще одну програму, що виконує лівосторонній обхід дерева.

Задача 42. *Описати процедуру, що друкує елементи даного двійкового дерева, відображаючи його структуру за допомогою відступів, у такому порядку: спочатку друкується ліве піддерево з відступом на одну позицію, потім – корінь з початку рядка, потім – праве піддерево також з відступом на одну позицію.*

Структура процедури виведення дерева може бути схожою до структури функції *sum*: для обходу використаємо оператори виклику процедури

(рекурсивного), а опрацювання елемента полягатиме у виведенні його на друк після відповідної кількості пропусків. Для задання кількості пропусків використовуємо додатковий параметр:

```

procedure printTree(t:tree; shift:byte);
{ процедура printTree роздруковує дерево t, виконуючи }
{ лівосторонній обхід; вершини нижчих рівнів друкуються }
{ з відступом shift стосовно вершин вищих рівнів }
var i:byte;
begin if t<>nil then {опрацьовуємо непорожнє дерево}
  with t^ do
    begin printTree(left,shift+1); {спочатку друкуємо ліве}
      {піддерево, потім - елемент, записаний у корені,}
      for i:=1 to shift do write(' '); writeln(elem);
      printTree(right,shift+1) {а тепер - праве.}
    end {with}
  end; {printTree}

```

Таку структуру підпрограми можна використати і для інших задач, що потребують виконання обходу. Зміниться лише та частина, яка виконує власне обробку елемента.

12. Програмування з поверненням назад

Алгоритми з поверненням назад використовують для розв'язування задач, які потребують перевірки потенційно великого, проте скінченного числа розв'язків. Такий алгоритм шукає розв'язок задачі не за заданими правилами обчислень, а шляхом спроб і помилок. Він послідовно конструює розв'язок задачі, щоразу додаючи його чергову складову частину і перевіряючи можливість продовжити побудову. Якщо такої можливості нема, алгоритм відкидає останню чи кілька останніх доданих частин, повертається на кілька етапів побудови назад і пробує знаходити інші варіанти продовження. Здебільшого такі алгоритми визначають у термінах рекурсії. Загалом увесь процес можна трактувати як побудову та обрізання дерева підзадач. Підзадача має менший розмір, ніж вихідна задача, проте на кожному етапі побудови розв'язку таких підзадач є декілька. Вибір однієї з них визначає спосіб продовження побудови розв'язку. У багатьох випадках таке дерево росте дуже швидко. Швидкість зростання дерева підзадач залежить від параметрів задачі і здебільшого буває експотенційною (2^n , 3^n тощо). Навіть у випадку обрізання 99% можливих підзадач швидкість зростання залишається чималою: $M^n/100$ для великих n зростає експотенційно.

Продемонструємо процес побудови алгоритму з поверненнями на прикладі розв'язування конкретних задач.

12.1. Тур коня

Задача 43. Дано натуральне n . На квадратній дошці розміру $n \times n$ на полі з координатами (x_0, y_0) стоїть шахова фігура – кінь. Його можна пересувати за звичайними шаховими правилами. Знайти послідовність n^2-1 ходів коня, за які він обійде всі клітки дошки, побувавши у кожній лише один раз.

Одразу запропонувати спосіб отримання розв'язку задачі нелегко, тому розв'яжемо спочатку простішу задачу: *або виконати черговий хід, або довести, що будь-який хід неможливий*. З будь-якої клітки кінь може виконати не більше восьми ходів, тому існує лише вісім можливих варіантів продовження туру. Один з цих ходів можна виконати, якщо кінь не вийде за межі дошки і стане на клітку, у якій він ще не був. Перебирати кандидатів на продовження можна послідовно, пам'ятаючи при цьому номер останнього випробуваного кандидата. Якщо всі можливі продовження ведуть на зайняті клітки або за межі дошки, то хід зробити неможливо. Алгоритм перебору кандидатів схематично записують так:

```
repeat вибір_чергового_кандидата_зі_списку_ходів;
  if підходить then записати_хід;
until (підходить) or (кандидатів_більше_нема)
```

Продовжимо міркування. Щоб розпочати перебір кандидатів, необхідно ще перед циклом якимось ініціалізувати їхній вибір. Далі, якщо черговий хід вдалий, то потрібно продовжувати побудову туру (перейти до розв'язування підзадачі – побудови коротшого на одиницю туру). У протилежному випадку, якщо з даного поля хід зробити неможливо, коня необхідно повернути на попереднє поле і спробувати зробити інший хід. Отже, процедура відшукування туру коня матиме такий вигляд:

```
procedure TryNextMove;
begin ініціалізувати_вибір_ходу;
  repeat вибір_чергового_кандидата_зі_списку_ходів;
    if підходить then
      begin записати_хід;
        if дошка_не_заповнена then
          begin TryNextMove;
            if невдача then знищити_попередній_хід
          end
        end
      until (хід_був_вдалим) or (кандидатів_більше_нема)
  end; {TryNextMove}
```

Щоб конкретизувати «оператори» цієї процедури, спочатку визначають спосіб зображення шахівниці. Найзручніше з цією метою використати цілочислову матрицю розмірів 8×8 :

```
var board:array[1..n,1..n]of byte;
```

Її значеннями можуть бути порядкові номери ходів коня. Перед початком побудови туру матриця заповнена нулями: $board[x,y]=0$, якщо на поле (x, y) ще не ходили; $board[x,y]=i$, якщо на полі (x, y) кінь був на i -му ході. Нехай змінні u та v містять координати можливого наступного ходу.

Тепер легко конкретизувати такі оператори:

дошка_не_заповнена: перевірити умову $i < n^2$;

підходить: перевірити $(1 \leq u \leq n) \wedge (1 \leq v \leq n) \wedge (board[u, v] = 0)$;

невдача, хід_був_вдалим – для відображення цих даних використаємо додаткову глобальну змінну *success* логічного типу;

записати_хід – задати відповідне значення елемента масиву: $board[v, u] = i$ (першим індексом вказали вертикальну координату клітки для відповідності між матрицею і шахівницею);

знищити_хід – аналогічно до попереднього: $board[v, u] = 0$.

Усі ці рішення дають змогу виконати наступний крок деталізації:

```

var success:Boolean;
procedure TryNextMove(i:integer; x,y:byte);
  var u,v:integer;
begin ініціалізувати_вибір_ходу;
  repeat обчислити_(u,v)_координати_наступного_ходу;
    if (u in [1..n]) and (v in [1..n]) and (board[v,u]=0) then
      begin board[v,u]:=i;
        if i<sqr(n) then
          begin TryNextMove(i+1,u,v);
            if not success then board[v,u]:=0
          end else success:=true
        end
      until success or (кандидатів_більше_нема)
end; {TryNextMove}

```

Отже, нашу програму майже завершено. Залишилось тільки уточнити правила ініціалізації ходу та вибору наступного ходу. Усе написане дотепер ніяк від них не залежить, тому програма працюватиме за будь-яких правил переміщення фігури.

	3		2	
4				1
		Kn		
5				8
	6		7	

Рис. 7. Можливі ходи коня

Відомо, що кінь ходить «буквою Г». На рис. 7 цифрами позначено клітки, куди може походити кінь з клітки Kn. Як зміняться його координати після виконання ходу? Наприклад, якщо він походить на клітку «1», то значення x зросте на 2, а значення y – на 1. Такі зміни координат можна задати для кожного можливого ходу і записати у масиви, а для їхнього перебору використати змінну-лічильник ходів. Тепер легко завершити деталізацію програми: ініціалізація вибору ходу полягатиме у присвоєнні нуля лічильнику ходів, вибір наступного кандидата – збільшення лічильника і зміна поточних координат коня на відповідні зміщення. Усіх кандидатів на наступний хід буде вичерпано, коли лічильник досягне максимального значення 8. Остаточо отримаємо:

```

program knightsTourByRecursion;
type shifts=array[1..8]of shortint;
const dx:shifts=(2,1,-1,-2,-2,-1,1,2); {правила переміщення}
      dy:shifts=(1,2,2,1,-1,-2,-2,-1);
      nMax=10; {максимальний розмір дошки}
var n,nSqr:integer; success:Boolean; {ознака побудови туру}
    board:array[1..nMax,1..nMax]of integer; {дошка}

procedure TryNextMove(i:integer; x,y:byte);

```

```

var k,u,v:integer;
begin k:=0;                                     {лічильник можливих ходів}
  repeat inc(k);
    u:=x+dx[k]; v:=y+dy[k];                   {можливе продовження туру}
    if (u in[1..n])and(v in[1..n])and(board[v,u]=0) then
      begin board[v,u]:=i;                     {робимо хід}
        if i<nSqr then
          begin TryNextMove(i+1,u,v);          {шукаємо продовження}
            if not success then board[v,u]:=0 {повернення !}
          end else success:=true
        end
      until success or(k=8)
    end; {TryNextMove}
var i,j:integer;
Begin write('Введіть розмір дошки: '); readln(n);
  nSqr:=sqr(n); {довжину туру досить обчислити один раз}
  for i:=1 to n do
    for j:=1 to n do board[i,j]:=0;           {дошка - порожня}
    write('Введіть координати першого поля: '); readln(i,j);
    board[j,i]:=1;                             {поставили коня i}
    success:=false; Try(2,i,j); {спробували побудувати тур}
    if success then
      begin writeln(' Знайдено тур');
        for i:=n downto 1 do
          begin for j:=1 to n do write(board[i,j]:4);
                writeln          {надрукували знайдений розв'язок}
          end end{for & then}
        else writeln('No path')
      end
End.

```

Для $n = 5$ отримано такий розв'язок задачі щодо туру коня:

```

Введіть розмір дошки: 5
Введіть координати першого поля: 1 1
Знайдено тур
21 16 11  4 23
10  5 22 17 12
15 20  7 24  3
 6  9  2 13 18
 1 14 19  8 25

```

Зауважимо, що степінь дерева розв'язків цієї задачі є досить великим – він дорівнює 8. Тому час, необхідний для відшукування розв'язку зростає зі збільшенням розмірів дошки дуже швидко. Наприклад, для $n = 8$ кількість можливих варіантів дорівнює $2^{192} \approx 6,277 \times 10^{57}$.

Характерною особливістю записаного алгоритму є те, що він робить кроки у напрямі відшукування загального розв'язку. Всі кроки записують так, щоб можна було повернутись назад і відкинути ті з них, які заводять у тупик. Такий процес називають *поверненням назад* (англійською – *backtracking*).

За допомогою алгоритму процедури TryNextMove виводять універсальну схему розв'язування задач зі скінченною кількістю можливих розв'язків:

```

procedure Try;
begin ініціалізація_вибору_кандидатів;
    repeat вибір_чергового_кандидата;
        if підходить then
            begin запис_кандидата;
                if розв'язок_неповний then
                    begin Try;
                        if невдача then стерти_запис
                    end
                end
            end
        until (удача) or (кандидатів_більше_нема)
end; {Try}

```

Абстрактні «оператори» цієї схеми уточнюють для кожної конкретної задачі.

12.2. Тур коня без рекурсії

Ми уже говорили, що складність алгоритму з поверненнями є експотенційною. Деяко покращити ситуацію зі швидкістю могла б відмова від рекурсивних викликів: адже звертання до підпрограми вимагає додаткових затрат на розташування локальних змінних і налаштування адресів. А з якою метою використано рекурсію у процедурі TryNextMove? Зрозуміло, що ми звели задачу «знайди тур довжини n^2 » до двох задач: «зроби хід» і «знайди тур довжини $n^2 - 1$ », однак що ще? За рахунок чого відбуваються повернення і відкидання тупикових варіантів?

Очевидно, що в сегменті стеку програми зберігаються копії локальних змінних усіх викликаних екземплярів процедури TryNextMove. Наприклад, змінна i «пам'ятає» номер шуканого ходу, змінні x та y – координати фігури, k – кількість випробуваних кандидатів. Саме завдяки використанню цієї інформації TryNextMove здатна виконувати повернення. Проте для її зберігання зовсім не обов'язково використовувати системний стек. Ми могли б запровадити власний стек, наприклад, за допомогою використання масиву відповідного типу та розміру і зберігати у ньому всі необхідні дані.

Номер шуканого ходу та ознака успіху є загальними даними, їх можна зберігати в змінних програми, а до стеку необхідно буде занести координати виконаного ходу і кількість кандидатів, випробуваних на цьому ході. Отже, ми могли б використати такі оголошення:

```

type step=record x,y:index; k:byte end;
var steps:array[1..nSqr]of step;

```

Тут $nSqr$ – як і раніше, довжина туру, масив $steps$ буде використано як стек для зберігання інформації щодо виконаних ходів.

З метою побудови розв'язку ми використаємо такі ж підходи, як і в попередній програмі, тому без зайвих пояснень просто наведемо її текст:

```

program KnightTour;
const nMax=10;                {максимальні розміри дошки}
        nMSqr=nMax*nMax;      {i туру}
type index=1..nMax;
        shifts=array[1..8]of integer;

```

```

    step=record x,y:index; k:byte end;
const dx:shifts=(2,1,-1,-2,-2,-1,1,2); {правила виконання}
      dy:shifts=(1,2,2,1,-1,-2,-2,-1); {ходу конем}
var board:array[index,index]of word;   {дошка}
    steps:array[1..nMSqr]of step;      {розв'язок}
    x0,y0:index;                       {координати початкової клітки}
    kD:0..8;                            {номер кандидата}
    u,v:integer;                       {нові координати}
    success:Boolean; {ознака успішного вибору продовження}
    n:index;                            {розмір дошки}
    nSqr,l:word;                        {бажана і поточна довжина туру}
    i,j:index;
begin write('Введіть розмір дошки: '); readln(n); nSqr:=sqr(n);
      write('Введіть стартову клітку: '); readln(x0,y0);
FillChar(board, sizeof(board), 0);     {дошка порожня}
FillChar(steps, n*SizeOf(step), 0);    {розв'язок - теж}
board[y0,x0]:=1;                       {поставили коня на старт}
steps[1].x:=x0; steps[1].y:=y0;
l:=1;
repeat kD:=steps[1].k; success:=false;
  while (kD<8)and(not success) do {шукаємо продовження}
  begin inc(kD);
    u:=steps[1].x+dx[kD]; v:=steps[1].y+dy[kD];
    success:=(u in [1..n])and(v in [1..n])and(board[v,u]=0)
  end;{while}
  if success then {заносимо хід до стека}
  begin steps[1].k:=kD; inc(l); board[v,u]:=1;
    steps[1].x:=u; steps[1].y:=v;
  end{then}
  else with steps[1] do{вилучаємо зі стека попередній хід}
  begin k:=0; board[y,x]:=0; dec(l)
  end{with & else & if}
until (l=nSqr)or(l=0); {цикл закінчиться, коли побудуємо тур,
                       або коли відкинемо усі ходи як тупикові}

if l>0 then
begin writeln(' Тур коня');
  for i:=n downto 1 do
  begin for j:=1 to n do write(board[i,j]:4);
    writeln
  end end
else writeln(' Туру нема.')
end.

```

Продемонструємо окремі результати, отримані за цією програмою:

```

Введіть розмір дошки: 8
Введіть координати першого поля: 1 1
Тур коня
64 17 8 29 20 15 6 13
9 30 19 16 7 12 25 22
18 63 28 11 24 21 14 5
31 10 33 62 41 26 23 54
34 61 40 27 50 55 4 45
39 32 37 42 3 46 53 56
60 35 2 49 58 51 44 47
1 38 59 36 43 48 57 52

```

У цій програмі для контролю за тим, чи вже отримано розв'язок, використано змінну l – вона містить поточну довжину туру. Якщо в умові циклу залишити тільки умову $l = 0$, а в циклі друкувати матрицю *board* кожного разу, коли $l = n\text{Sqrt}$, то програма надрукує **всі** тури коня, що починаються на клітці з координатами (x_0, y_0) . Наприклад, для задачі з розміром дошки $n=5$ і стартовою кліткою $(1, 1)$ усіх турів є 304 (без урахування їхньої симетрії).

13. Метод часткових цілей

Метод часткових цілей застосовують для розв'язування складних нових задач, якщо нема відомих алгоритмів для схожих задач, і якщо врахувати одразу всі умови дуже важко. Цей метод передбачає зведення великої задачі до послідовності простіших. Звичайно ми сподіваємося, що простіші задачі легше розв'язати, ніж початкову, і що розв'язок початкової задачі можна отримати за допомогою розв'язування побудованих простіших задач.

Застосування методу часткових цілей на практиці є досить складним. Адже виокремлення простішої задачі справа скоріше інтуїції, ніж науки. Такому виокремленню можуть допомогти відповіді на такі запитання.

1. Чи можна розв'язати тільки частину задачі? Чи можна розв'язати всю задачу, ігноруючи деякі умови?
2. Чи можна розв'язати задачу для часткових випадків: створити алгоритм, що виконує усі вимоги до задачі тільки для деякої підмножини вхідних даних?
3. Чи добре ми зрозуміли постановку задачі? Чи не допоможе нам глибше розуміння деяких її особливостей?

Продемонструємо застосування методу часткових цілей на прикладі побудови *польського запису* заданого арифметичного виразу. Ми звикли записувати арифметичні вирази в *інфіксній* формі, коли знак операції записується *між* операндами (для бінарних операцій). Обчислення значення виразу, записаного в такій формі, передбачає його багаторазовий перегляд: щоб обчислити всі частини виразу, записані в дужках, тоді виконати множення та ділення, і наприкінці – додавання та віднімання.

Багатократного перегляду виразу можна уникнути, якщо записати його у *суфіксній* або *префіксній* формах. Інверсним польським записом називають суфіксну (постфіксну) форму, за якою спочатку записують обидва операнди, а потім – знак операції. Такий спосіб дає змогу обходитись без дужок і обчислювати значення виразу за один перегляд його зліва-направо, а тому широко використовується різноманітними компіляторами.

Задача 44. У вхідному текстовому файлі задано правильний запис деякого арифметичного виразу в інфіксній формі. (В арифметичних виразах використовують знаки чотирьох операцій: $+$, $-$, \times , $/$; круглі дужки для зміни порядку виконання операцій; операндами можуть бути цілі або дійсні числа.) Перетворити його до постфіксної форми.

Як передбачити усі можливі варіанти розташування дужок та знаків операцій? Як розпізнати операнди, що можуть бути цілими чи дійсними

числами? Задача не з простих. Розв'язати її одразу, мабуть, не вдасться, тому застосуємо метод часткових цілей і пошукаємо відповіді на сформульовані раніше запитання. Перше з них у пригоді нам не стане: важко собі уявити, як перетворити лише частину виразу, і як це може допомогти отримати розв'язок задачі. Третє запитання програмістові необхідно пам'ятати постійно. Воно буде корисним під час розв'язування будь-якої задачі, і не лише за допомогою методу часткових цілей.

Давайте скористаємось другим з перелічених запитань і полегшимо своє завдання, вилучивши з розгляду частину можливих варіантів вхідних даних. Спочатку побудуємо алгоритм переведення, що працюватиме для деяких простих виразів, а потім, поступово вдосконалюючи його, спробуємо отримати алгоритм переведення довільного арифметичного виразу.

Що можна спростити у вхідних даних? Розпізнавання операндів – багатоцифрових цілих чи дійсних – є дещо самостійною задачею. Її можна розглядати незалежно від побудови основного алгоритму. Тому обмежимося випадком, коли операнди є цілими одноцифровими числами. Наявність у виразі дужок змінює пріоритети операцій, а це ускладнює алгоритм побудови польського запису. Тому наступне спрощення полягатиме в тому, щоб розглядати вирази без дужок. І, нарешті, домовимося, що у виразі є лише бінарні операції. Унарні «+» чи «-» розглянемо пізніше.

Отже, запис нашого «простого» арифметичного виразу може містити цифри і знаки «+», «-», «*», «/» (у програмах замість знаку «x» для позначення множення використовують «*»), причому знаки завжди стоять між цифрами, наприклад: '3+7-2*4/5'. Польський запис цього виразу має вигляд '37+24*5/-'. Бачимо, що після перетворення знаки операцій пропустили поперед себе цифри-операнди. Знак «+» пропустив цифру «7», знак «*» – цифру «4», а знак «-» – взагалі усіх: і операнди, і знаки старших за пріоритетом операцій.

Припустимо, що інфіксийний запис заданого виразу зберігається у рядку *source*, а польський запис виразу необхідно побудувати у рядку *dest*. Щоб знак операції міг пропустити поперед себе інші літери, його потрібно десь тимчасово зберігати. Найкраще для цього використати стек. Якщо до виразу входять операції тільки однакового пріоритету, то до стека заносять знак операції тільки для того, щоб поміняти його місцями з другим операндом. Якщо ж у виразі є операції різних пріоритетів, то знак молодшої операції потрібно зберігати у стекові доти, доки до *dest* не буде перенесено операнди та знаки старших операцій.

Тепер сформулюємо алгоритм побудови рядка *dest*. Він полягає у послідовному перегляді літер рядка *source*. Залежно від значення чергової літери, її обробляють по-різному:

- якщо значенням є цифра, то її заносять у *dest*;
- якщо значенням є '+' чи '-', то аналізують стек: якщо він порожній, то просто заносять до нього літеру; а якщо непорожній, то вміст стека переносять до *dest* і заносять до стека літеру;
- якщо значенням є '*' чи '/', то аналізують стек: якщо він порожній, то заносять до нього літеру; якщо ні, то переносять знак операції з вершини стека до *dest*, якщо він є '*' чи '/', і записують літеру до стека.

Після аналізу усіх літери рядка *source* вміст стека переносять до *dest*.

Програмні засоби, що реалізують операції зі стеком, безпосередньо не стосуються алгоритму перетворення виразу, тому їхнє оголошення розташуємо в окремому модулі. Ось його повний текст:

```

unit stackTol;
interface
  function stackIsEmpty:Boolean;           {чи порожній стек?}
  function popStack:char;                 {вилучає елемент з вершини стека}
  procedure pushStack(c:char);           {заносить c у вершину стека}
  function checkStack:char;              {повертає значення вершини стека}
implementation
  type stack=^section; {стек реалізовано за допомогою списку}
      section=record elem:char; link:stack end;
  var s:stack; {модуль містить тільки один стек}
  function stackIsEmpty:Boolean;
      begin stackIsEmpty:=s=nil
      end;
  function popStack:char;
      var p:stack;
      begin p:=s; with p^ do
          begin popStack:=elem; s:=link end;
          dispose(p)
      end;
  procedure pushStack(c:char);
      var p:stack;
      begin p:=new(stack); with p^ do
          begin elem:=c; link:=s end;
          s:=p
      end;
  function checkStack:char;
      begin checkStack:=s^.elem
      end;
begin s:=nil end.

```

Функція *checkStack* реалізує дещо «нетрадиційну» операцію зі стеком: вона повертає значення елемента у вершині стека, не вилучаючи зі стека самого елемента. Використання такої функції зумовлене потребами алгоритму.

Тепер описаний алгоритм перетворення виразу можна реалізувати такою програмою:

```

program postfix1; uses stackTol;
var source,dest:string;           {вхідний та результуючий рядки}
    c:char; i:byte;                {робочі змінні}
begin write('Задайте арифметичний вираз: '); readln(source);
    write('Польський запис: '); dest:='';
    for i:=1 to length(source) do           {аналізуємо кожен}
        begin c:=source[i];                 {літеру заданого рядка}
            case c of
                '0'..'9': dest:=dest+c;     {операнд -> результуючий рядок}
                '+', '-': begin             {знаки операцій зі стека -> в dest}
                    while not stackIsEmpty do dest:=dest+popStack;
                    pushStack(c) end; {+, -}
                '*', '/': begin             {зі стека в dest операції рівного пріоритету}
                    while not stackIsEmpty and

```

```

        (checkStack in ['*', '/']) do dest:=dest+popStack;
        pushStack(c) end {*, /}
    end {case}
end; {for}                                {очищуємо стек}
while not stackIsEmpty do dest:=dest+popStack;
writeln(dest)                             {друкуємо результат}
end.

```

Початок покладено! Тепер справа за розвитком і поступовим удосконаленням програми. Навчимо її опрацьовувати вирази з дужками.

Під час обчислення арифметичного виразу передусім необхідно виконати операції в дужках, тому поява у записі виразу літери «(» означає початок частини виразу з вищим пріоритетом. Усі знаки операцій, які на цей момент перебувають у стекові, мусять там залишатися доти, доки не закінчиться пріоритетна частина виразу. А закінчується вона літерою «)». Щоб досягти бажаного ефекту, можна заносити до стека дужку, яка відкривається, і тримати її там (а під нею і всі знаки операцій, що були в стеку на момент початку пріоритетної частини) аж доти, доки не буде опрацьовано дужку, яка закривається.

Це хороше рішення, проте воно впливає на спосіб виконання алгоритму, описаного раніше. Значення чергової літери '+' чи '-' означало у ньому перенесення вмісту непорожнього стека у *dest*. Те, що було правильним для виразу без дужок, не завжди спрацьовує для виразу з дужками. Літера '(' у стеку розмежовує знаки операцій звичайної та пріоритетної частин виразу. Тому, опрацьовуючи '+', необхідно вилучати літери зі стека до появи у його вершині '(' або доти, доки він не спорожніє. Так само удосконалюють опрацьовання знаків інших операцій.

Бачимо, що кількість перевірок катастрофічно зростає, внаслідок чого алгоритм поступово заплутується. Необхідно його спростити. З цією метою припишемо знакам операцій і дужкам числові значення їхніх пріоритетів. Використаємо також ще одну хитрість: щоб однаково опрацьовувати порожній та непорожній стек, занесемо у вершину порожнього стека деяку спеціальну літеру (наприклад, '#') і припишемо їй такий самий пріоритет, як і дужкам:

літера	пріоритет
'*', '/', 'p'	2
'+', '-', '#'	1
'(', ')'	0

Тепер у перевітках можна використовувати числові значення пріоритетів і сформулювати удосконалений алгоритм перетворення виразу. Щоб побудувати рядок *dest*, послідовно перебирають і аналізують літери рядка *source*. Залежно від значення чергової літери обробляють по-різному:

- якщо значенням є цифра, то її заносять у *dest*;
- якщо значенням є знак операції, то вилучають зі стека і заносять до *dest* літери доти, доки їхній пріоритет більший чи дорівнює пріоритетові чергової літери; знак операції заносять до стека;
- якщо значенням є '(', то її заносять до стека;
- якщо значенням є ')', то вилучають зі стека і заносять до *dest* усі літери, які є у стекові над '('; вилучають зі стека '('.

Після завершення аналізу всіх літер рядка *source*, вміст стека (окрім спеціальної літери '#') переносять до *dest*. (Легко перекоонатися, що за цим алгоритмом ні знак операції, ні закриваюча дужка не «виштовхують» літеру '#' зі стека.)

Удосконалений алгоритм реалізує програма *postfix*.

```

program postfix2; uses stackTol;
var source,dest:string;           {вхідний та результуючий рядки}
    c:char; i,p:byte;              {робочі змінні}
function priority(c:char):byte;
  begin case c of
    '*','/': priority:=2;
    '+','-': priority:=1;
    '#','(',')': priority:=0
    else priority:=10 end
  end; {priority}
begin write('Задайте арифметичний вираз: '); readln(source);
  write('Польський запис: '); dest:=''; pushStack('#');
  for i:=1 to length(source) do           {аналізуємо кожну}
    begin c:=source[i]; p:=priority(p);      {літеру заданого рядка}
      case c of
        '0'..'9': dest:=dest+c;             {операнд -> результуючий рядок}
        '+','-','*','/': begin while priority(checkStack)>=p
          do dest:=dest+popStack; {знаки операцій}
          pushStack(c) end; {+,-,*,/}
        '(': pushStack(c);                  {початок пріоритетної частини}
        ')': begin while checkStack<>'(' do dest:=dest+popStack;
          popStack      {вилучили дужку, яка відкривається}
        end {}
      end {case}
    end; {for}                               {очищуємо стек}
  while checkStack<>'#' do dest:=dest+popStack; popStack;
  writeln(dest)                               {друкуємо результат}
end.

```

Це вже досить «розумний» алгоритм. Він правильно перетворює вирази з довільними послідовностями арифметичних операцій і з довільним вкладенням дужок. Навчимо його тепер опрацьовувати знаки унарних операцій. З унарним знаком '+' все просто: його необхідно вилучити з рядка *source*, бо він нічого не змінює у виразі. Унарний мінус можна опрацьовувати двома різними способами: використати для його позначення спеціальний знак і приписати йому найвищий пріоритет (числове значення 3), або перетворити його в бінарний, вставивши перед ним в рядок *source* літеру '0'. Другий спосіб дещо простіший, його і реалізуємо.

Для повного розв'язання задачі 44 необхідно ще навчити алгоритм опрацьовувати багатоцифрові операнди. Це нескладно зробити, бо тепер просто доведеться пересилати з *source* до *dest* не одну цифру, а групу цифр. Деяке ускладнення пов'язане тільки з тим, що тепер треба буде якось розділяти операнди в рядку *dest*. Використаємо для цього літеру ' ', яку додаватимемо після кожного операнда заданого арифметичного виразу.

Отримаємо програму *postfix3*:

```

program postfix3; uses stackTol;
var source,dest:string;           {вхідний та результуючий рядки}
    c:char; i,p:byte;              {робочі змінні}
function priority(c:char):byte;
  begin case c of
    '*','/': priority:=2;
    '+','-': priority:=1;
    '#','(',')': priority:=0
    else priority:=10 end
  end;{priority}
begin write('Задайте арифметичний вираз: '); readln(source);
  {перетворимо знаки унарних операцій у рядку source}
  case source[1] of                {спочатку - на початку рядка:}
    '-': source:='0'+source;       {мінус став бінарним}
    '+': delete(source,1,1)        {вилучили унарний плюс}
  end;{case} i:=1;
  while i<length(source) do      {потім - після кожної дужки ( }
  begin if source[i]='(' then
    case source[i+1] of
      '-': begin insert('0',source,i+1); inc(i) end;
      '+': delete(source,i+1,1)
    end;{case} inc(i)
  end;{while}
  source:=source+' ';
  write('Польський запис: '); dest:=''; pushStack('#');
  for i:=1 to length(source) do  {аналізуємо кожен}
  begin c:=source[i]; p:=priority(p); {літеру заданого рядка}
    case c of
      '0'..'9','.',',','e','E': begin dest:=dest+c;
        if not(source[i+1]in['0'..'9','.',',','e','E'])
        then dest:=dest+' '        {завершили операнд}
      end;{0..9}
      '+','-','*','/': begin while priority(checkStack)>=p
        do dest:=dest+popStack; {знаки операцій}
        pushStack(c) end;{+,-,*,/}
      '(': pushStack(c);           {початок пріоритетної частини}
      ')': begin while checkStack<>'(' do dest:=dest+popStack;
        popStack {вилучили дужку, яка відкривається}
      end {} }
    end {case}
  end;{for}                        {очищуємо стек}
  while checkStack<>'#' do dest:=dest+popStack; popStack;
  writeln(dest)                      {друкуємо результат}
end.

```

Задачу розв'язано: програма *postfix3* виконує всі сформульовані в умові вимоги. Проте межі досконалості немає! Програма *postfix4* могла б виявляти помилки у записі виразу, опрацьовувати не тільки арифметичні, але й алгебричні вирази, до запису яких належать піднесення до степеня, виклики стандартних математичних функцій тощо. Побудову такої програми, сподіваємося, виконають уже наші читачі.

14. Метод підйому. Евристики

За методом підйому будують алгоритм, який приймає деяке початкове припущення або обчислює початковий розв'язок задачі, а потім якомога швидше покращує його, намагаючись отримати оптимальний розв'язок задачі. Алгоритми підйому є доволі грубими, бо намагання покращити розв'язок у будь-який спосіб робить ці алгоритми дещо недалекоглядними. Такі алгоритми використовують для швидкого обчислення наближеного розв'язку задачі.

Евристикою називають алгоритм з такими властивостями:

- він переважно знаходить хороші, проте не завжди оптимальні розв'язки;
- його можна швидше і легше реалізувати, ніж будь-який точний алгоритм.

Безліч евристичних алгоритмів базується або на методі часткових цілей, або на методі підйому. Один загальний підхід до побудови евристичних алгоритмів полягає у переліченні всіх вимог до точного розв'язку і поділі їх на дві групи:

- 1) ті, які легко задовільнити;
- 2) усі інші.

Або:

- 1) ті, які необхідно задовольнити обов'язково;
- 2) ті, які можна виконати не повною мірою.

Тоді будують алгоритм, який виконує вимоги з першої групи і намагається виконати вимоги другої групи.

Продемонструємо процес побудови евристики методом підйому на прикладі задачі комівояжера. Це класична задача, умову якої сформульовано так:

Задача 45. Припустимо, що комівояжер мусить відвідати клієнтів, які живуть у n різних містах. Кожне місто він відвідує один раз і повертається додому (такий маршрут називається туром). Яким буде найкоротший тур (найдешевший, найшвидший)?

Цю задачу легше сформулювати, ніж розв'язати. Вона належить до недетерміновано поліноміальних за складністю (*NP*-складних): для таких проблем невідомі алгоритми, які здатні відшукати оптимальний розв'язок за час, що зростає як поліноміальна функція вхідних даних задачі. Справді, кількість можливих турів у задачі комівояжера зростає як $n!$. Наприклад, для ста міст існує понад 10^{155} шляхів (в історії Всесвіту пройшло лише 10^{18} секунд). Перебрати усі тури і знайти найкращий неможливо. У цьому випадку можна ставити задачу тільки про відшукування деякого наближеного розв'язку задачі комівояжера, наприклад, евристичного розв'язку.

Оптимальний розв'язок задачі комівояжера має дві основні властивості:

- 1) він складається з ребер, що утворюють простий цикл (тур);
- 2) вартість будь-якого іншого туру не буде меншою за його вартість.

Евристичний алгоритм задовольняє першу вимогу і не обов'язково виконує другу.

Щоб розпочати проектування алгоритму, необхідно передусім визначити, як задано інформацію щодо відстаней між містами, можливостей переїзду з міста до міста, вартості і тривалості. Пронумеруємо всі міста, які повинен

відвідати комівояжер. Якщо припустити, що він користується власним автомобілем (що доволі імовірно), то відстань між містами, вартість переїзду і час будуть практично пропорційними. Їхню числову величину можна задавати, наприклад, за допомогою матриці вартостей C . Елемент c_{ij} задає вартість переїзду з i -го міста до j -го. Очевидно, що $c_{ij} = c_{ji}$. Якщо між окремими містами немає сполучення, то значення відповідних елементів матриці вартостей можна задати максимально великим. Такими ж великими задамо і діагональні елементи, щоб усунути можливість «переїзду» комівояжера з будь-якого міста у те ж саме місто.

Тур комівояжера розпочинається з його рідного міста. Черговий переїзд вибиратимемо з умови мінімальності: наступним містом туру буде те, потрапити в яке з попереднього найдешевше. Щоб не зачисляти до туру місто удруге, запам'ятовуватимемо множину міст, які комівояжер уже відвідав. Програма, що реалізує такий алгоритм, матиме вигляд:

```

program tourVoyager;
const N=5;                                {розглянемо задачу з п'ятьма містами}
type town=1..N;
      matrix=array[town,town]of word;
const c:matrix=((65535, 1, 2, 7, 5),      {матриця вартостей}
                (1, 65535, 4, 4, 3),
                (2, 4, 65535, 1, 2),
                (7, 4, 1, 65535, 3),
                (5, 3, 2, 3, 65535) );
var visited:set of town;                  {множина відвіданих міст}
      tour:array[0..n]of town;            {шуканий тур}
      cost:word;                            {вартість туру}
      v:town;                                {поточне місто}
      min:word; u:town; k,i,j:town;
begin write('Введіть початкову вершину: '); readln(u);
      tour[0]:=u; visited:={u}; {тур починається в рідному місті}
      v:=u; cost:=0;
      for k:=1 to N-1 do                   {шукаємо продовження туру}
      begin min:=c[v,v];                     {з найменшою вартістю}
        for i:=1 to n do
        if not(i in visited) then         {серед невідвіданих міст}
        begin if c[v,i]<min then
          begin min:=c[v,i]; j:=i end
        end; tour[k]:=j; cost:=cost+c[v,u];
        visited:=visited+[j]; v:=j         {переїхали в нове місто}
      end; {for}                             {тур завершується}
      tour[n]:=u; cost:=cost+c[v,u];        {в рідному місті}
      for k:=1 to n do                     {друкуємо результат}
        writeln(tour[k-1], '-->', tour[k]);
        writeln('cost=', cost)
      end.

```

Матрицю вартостей у цій програмі задано за допомогою типізованої константи. Відповідний їй граф зображено на рис. 8, а.

Простежимо, як відбуватиметься побудова туру, якщо комівояжер вирушає з міста, зазначеного за номером 1. У початковий момент відвіданим є

тільки перше місто (на схемі позначено квадратиком), вартість туру поки що дорівнює нулю. Серед можливих продовжень туру з вартостями 1, 3, 7, 5 алгоритм вибере найдешевше, і комівояжер вирушить у місто за номером 2 (рис. 8, б). Тепер відвідано два міста. Продовження алгоритм шукатиме серед решти. На останньому кроці вибір у алгоритму невеликий (рис. 8, г): невідвіданим залишилось тільки четверте місто. Тому саме його наш алгоритм додасть до туру і завершить тур поверненням у перше місто, додавши тим самим до туру найдорощчу ланку.

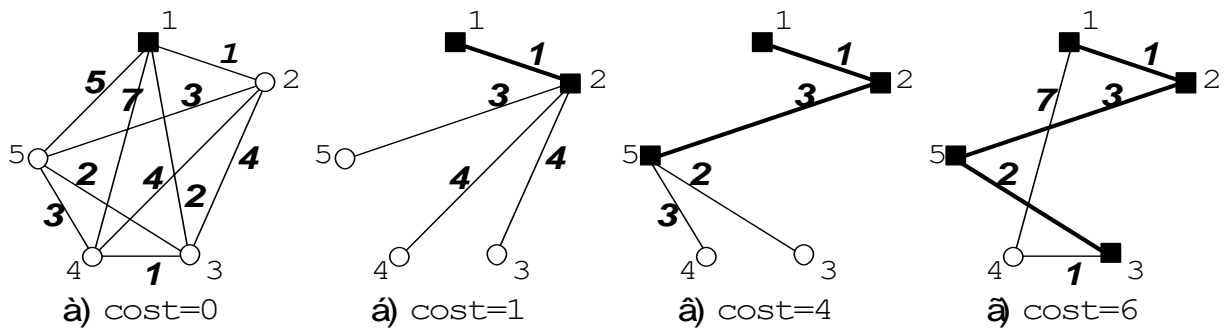


Рис. 8. Евристичний розв'язок задачі комівояжера з п'ятьма містами. Для кожного етапу розв'язування задачі вказано вартість побудованого туру (значення $cost$)

Результати виконання програми *tourVoyager*:

```

Введіть початкову вершину: 1
1-->2
2-->5
5-->3
3-->4
4-->1
cost=14

```

Бачимо, що остаточна вартість евристики дорівнює 14. Це не найгірший результат. А оптимальним для цієї задачі є тур $\{1, 2, 5, 4, 3, 1\}$ з вартістю 10.

Недоліком цього алгоритму є те, що наприкінці може залишитись місто, вартість переїзду до якого є великою. Такий розв'язок буде неоптимальним.

15. Генетичні алгоритми

Генетичні алгоритми розв'язування задач з'явилися порівняно недавно. Часто їх розглядають як альтернативу до традиційних методів, оскільки генетичний алгоритм використовує принципово новий підхід до відшукування розв'язку задачі. Традиційні методи використовують складний і «мудрий» математичний апарат для того, щоб за мінімальну кількість кроків отримати якнайкращий результат. На противагу їм генетичний алгоритм мало турбується про оптимальні характеристики кандидатів на розв'язок задачі, проте генерує їх чимало і швидко для того, щоб відбракувати гірші і продовжити генерувати нові, враховуючи позитивний досвід, здобутий кращими кандидатами. Протягом процедури виконання такого алгоритму кандидати на розв'язок удосконалюються, покращують свої характеристики – еволюціонують від початкового наближення до оптимального розв'язку. Тому розв'язування задач за допомогою генетичних алгоритмів часто називають *еволюційним численням*.

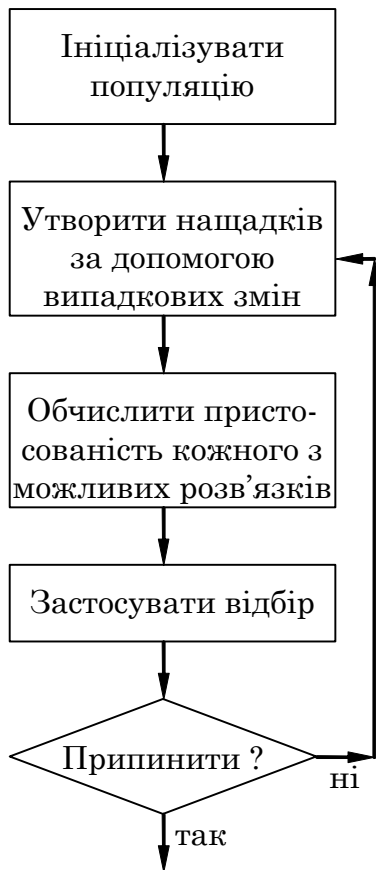


Рис. 9. Загальна схема роботи генетичного алгоритму

Природна еволюція розпочинається з первинної популяції організмів. Протягом багатьох поколінь випадкова мінливість і природний відбір формують поведінку та здібності особин популяції так, щоб максимально пристосуватись до вимог оточення. У найзагальніших термінах еволюцію можна описати як двокроковий ітераційний процес, що складається з випадкової зміни та подальшого відбору. Зв'язок між таким описом еволюції та оптимізаційним алгоритмом концептуально простий.

Як і природна еволюція, алгоритмічний підхід започатковують з вибору первинної множини альтернативних розв'язків певної проблеми. Далі ці «батьківські» розв'язки генерують «нащадків» за допомогою вибраних засобів випадкової варіації. Усі розв'язки – і батьків, і нащадків – оцінюють з огляду на те, наскільки добре вони виконують поставлене завдання. Остаточню застосовують критерій вибору, щоб відкинути ті розв'язки, цінність яких є під ризикою. Так само, як у природі «виживають найсильніші». Процес повторюють з відібраною множиною розв'язків з метою отримання наступних поколінь, аж доки не буде задоволено умову щодо отримання прийнятного розв'язку (рис. 9).

15.1. Сфера застосування

Щоб розв'язати певну проблему за допомогою традиційних алгоритмів оптимізації, дослідник повинен зробити низку припущень щодо способу оцінки

придатності розв'язку. Наприклад, використати цільову функцію, індекс відповідності тощо. Такий вибір накладає свої обмеження на задачу. Наприклад, алгоритми лінійного програмування передбачають, що цільова функція теж є лінійною: є сумою зважених цільових доданків. За іншого традиційного підходу – градієнтного – шукають точку нуля градієнта, можливо, мінімум чи максимум цільової функції, а це вимагає, щоб вона була гладкою, диференційовною. Градієнтні методи неможливо застосувати, якщо цільова функція має стрибки чи злами.

Еволюційні алгоритми таких припущень не потребують. Важливо лише, щоб індекс відповідності дав змогу впорядкувати два конкурентні розв'язки, визначивши, який з них з певних причин є кращим від іншого. Така невибагливість методу еволюційного числення дає змогу до його широкого застосування щодо тих задач, які не можна розв'язати традиційними числовими методами.

У реальному світі еволюційний підхід надає значні переваги. Адже традиційні методи використовують певні математичні моделі реальних задач. Як тільки окремі умови задачі змінюються (наприклад, вийшло з ладу обладнання на фабриці, змінилась вартість сировини чи змінився маршрут перевезень), традиційні розв'язки стають непридатними і всі обчислення необхідно виконувати з самого початку. На противагу цьому, в генетичному алгоритмі поточна популяція розв'язків зберігається як резервуар накопиченого знання, яке можна використати з метою пристосування до зміненого зовнішнього середовища.

Іншою перевагою еволюційного підходу є здатність за невеликий час генерувати хороші розв'язки – досить швидко для того, хто використовує цей підхід. Цю здатність, проілюстровано на прикладі класичної задачі щодо туру комівояжера для великої кількості міст, яку ми розглянули у попередньому параграфі. Мова йтиме не про обчислення оптимального розв'язку, а тільки про дуже якісну евристику.

Щоб реалізувати еволюційний підхід, необхідно виконати такі підготовчі кроки:

- вибрати зображення розв'язку;
- винайти оператор випадкових змін;
- задати правило виживання розв'язків;
- ініціалізувати популяцію.

Внаслідок їхнього виконання, еволюційний алгоритм генеруватиме розв'язки і покращуватиме їх від покоління до покоління. Розглянемо ці кроки детальніше.

15.2. Підготовчі кроки

Щоб зобразити можливий розв'язок задачі засобами програми, потрібно вибрати придатну для цього структуру даних. Вона мусить бути придатною для зображення *будь-якого* розв'язку і забезпечувати можливість легко його *оцінити*. Нема одного найкращого вибору зображення розв'язку, тому тут потрібна певна винахідливість.

Одним з можливих і досить зручних способів *зображення туру* комівояжера є перестановка номерів міст. Така перестановка вказує на порядок

відвідування міст, і будь-яка з них є більш чи менш придатним розв'язком. (Розв'язки задачі комівояжера для $n = 5$ у попередньому параграфі записано саме за допомогою перестановок). Самі перестановки можна зберігати у масиві або в лінійному однонапрямленому списку.

Отже, зображатимемо можливі тури за допомогою перестановок, записаних у масиві відповідного розміру. Згенерувати новий розв'язок означатиме створити нову перестановку. Далі потрібно визначити цільову функцію – *засіб оцінки* потенційних розв'язків. Подорож має бути найкоротшою, тому «ціною» розв'язку є довжина туру. Перевагу віддаватимемо коротшому турові у порівнянні з довшим.

Звичайно, завдання генерування та оцінювання турів може бути складнішим, якщо враховувати додаткові вимоги реального життя такі, як потреба зменшити час подорожі в години пік, або відвідувати певного покупця тільки після обіду або після інших покупців. Однак складність задачі робить еволюційний алгоритм придатнішим, бо швидко вилучає задачу зі сфери застосування традиційних оптимізаційних алгоритмів. Нехай для нашого прикладу мірою якості буде просто сумарна довжина туру.

Наступним кроком є конструювання *оператора* (або операторів) *випадкових змін*, який з батьківськими генеруватиме породжені розв'язки. Для цього можна використати багато різних способів. У природі існує два основні способи відтворення: статевий і нестатевий. У статевому відтворенні двоє батьків одного виду обмінюються генетичним матеріалом, який рекомбінується у потомстві. Безстатеве відтворення є за змістом клонуванням, однак під час передачі генетичного матеріалу від батька до нащадків цей матеріал може зазнавати різноманітних змін. Такі оператори відтворення можна змоделювати в еволюційному алгоритмі. Хоча можна міркувати ширше і запропонувати оператори, для яких не існує аналогів у природі. Наприклад, рекомбінувати генетичний матеріал від трьох батьків, чи ще якісь.

Остаточний успіх еволюційного алгоритму залежить від того, на скільки добре узгоджені варіаційний оператор, зображення розв'язку та цільова функція. Різні оператори можуть виявитися кращими у різних умовах. Так само як і з зображенням розв'язку, не існує одного оператора, найкращого для всіх задач.

Розглянемо два можливі варіанти генерування потомства для задачі комівояжера, що використовує перестановки. Перший спосіб – *однобатьківський* (рис. 10, а). Він схожий на безстатеве розмноження і полягає у випадковому виборі двох міст з батьківського туру та обміні їх місцями у породженому турі. Цей спосіб завжди генеруватиме допустимі тури: у них буде присутнє кожне з міст, і кожне – один раз.



Рис. 11. Оператори генерування нових поколінь для задачі комівояжера ($n=6$):
а) однобатьківський; б) двобатьківський

Другий спосіб – двобатьківський. Він з'єднує частини двох батьківських турів (виконує конкатенацію), розітнутих у випадково вибраній точці. Такий спосіб може генерувати недопустимі тури, які містять деякі міста двічі і не містять інших окремих міст. Це не означає, що в цій задачі не можна використовувати статевого розмноження – просто потрібно зачислити до оператора додаткові дії з відбракування чи виправлення генерованих турів.

У нашому прикладі використано описаний однобатьківський оператор.

Третім кроком є визначення *правила вибору* тих розв'язків, які будуть виживати, щоб стати батьками для наступних поколінь. Так само як з варіаціями, можна розглянути багато форм вибору. Наприклад, одне просте правило: виживають найпристосованіші! За ним тільки жменька найкращих розв'язків популяції залишається, тоді як всі інші вилучаються. Альтернативою може бути застосування змагального підходу, під час якого випадково попаровані розв'язки змагаються за виживання. Так само, як у професійному спорті, де слабший гравець чи команда виграють, бо їм пощастило у змаганні, слабші розв'язки в популяції часом виживають кілька поколінь, якщо використано такий підхід. Це може бути перевагою у складних проблемах, де буває легше покращити розв'язок, змінюючи гірший, ніж змінюючи тільки найкращі. Можливостей безліч, і будь-яке правило, яке загалом віддає перевагу у виживанні кращому розв'язку над гіршим, має сенс.

Останнім кроком є *вибір початкової популяції*. Якщо невідомо, як розв'язувати задачу, розв'язки вибирають випадково з простору можливих розв'язків. Для задачі комівояжера це означало б випадкове генерування низки перестановок n цілих чисел, кожна з яких зображає можливий розв'язок.

Альтернативою може бути залучення до початкової популяції розв'язків, близьких до добрих, отриманих за допомогою деякого іншого алгоритму чи з урахуванням апріорної інформації щодо задачі. Якщо ці розв'язки виявляться вартими уваги, то вони виживатимуть і продукувати нові покоління. Якщо ж ні, то разом з іншими – гіршими розв'язками – вони будуть вимирати. Ми використали випадкове генерування початкової популяції.

15.3. Програмна реалізація

Реалізуючи генетичний алгоритм відшукання туру комівояжера програмно, ми не ставили собі за мету знайти розв'язки якоїсь конкретної задачі, тому її постановка є досить умовною.

Задача 46. Знайти тур комівояжера, який проходить через 40 міст, розташованих на карті розміром 100×100 , і між кожною парою міст є пряма дорога.

Очевидно, що за такої постановки тур комівояжера можна зобразити за допомогою замкнутої ламаної, вершинами якої є міста. А цільовою функцією потрібно вибрати довжину цієї ламаної. Координати міст можна отримати за допомогою давача випадкових чисел і зберігати їх у двох одновимірних масивах:

```
const nT=40;
type coords=array[1..nT]of byte;
var x,y:coords;
```

У цьому випадку номер міста збігається з номерами елементів масивів x та y .

Як ми зазначали, будь-який тур можна зображати за допомогою перестановки номерів міст, яка зберігається у масиві відповідного типу:

```
type tour=array[1..nT]of byte;
```

Довжину туру легко обчислити за допомогою функції *cost*:

```
function cost(var t:tour):real;
var s:real; i,j,k:byte;
begin j:=t[nT]; k:=t[1];           {врахували довжину останнього}
      s:=sqrt(sqr(x[j]-x[k])+sqr(y[j]-y[k]));           {переїзду}
  for i:=2 to nT do
  begin j:=k; k:=t[i];           {додали всі інші переїзди}
        s:=s+sqrt(sqr(x[j]-x[k])+sqr(y[j]-y[k]))
  end; {for} cost:=s           {отримали довжину туру}
end; {cost}
```

Важливим питанням є спосіб зберігання популяції турів. Кожен з них має свою «цінність», яка, в нашому випадку, визначається його довжиною. Нам потрібно вилучати гірші тури і залишати кращі. Як ми будемо це робити? Змагальний підхід до відбору вимагає додаткових затрат на розбиття всієї популяції на конкурсні групи, тому зупинимося на простішому варіанті відбору: ранжуватимемо всі тури популяції за зростанням їхньої довжини і залишатимемо в ній тільки певну кількість кращих «особин», відбраковуючи усі гірші. За такого способу відбору зручно разом з кожним туром пам'ятати і його довжину (щоб не перераховувати її щоразу) та зберігати популяцію у впорядкованому вигляді. У цьому випадку кожен новий член популяції треба додавати до неї, не порушуючи впорядкованості. Яка структура даних підходить для цього найкраще? Очевидно, дерево пошуку або впорядкований лінійний список, однак, зважаючи на незначну кількість міст i , отже, незначну популяцію, зупинимо свій вибір на простішому щодо реалізації спискові.

Оголошення списку та процедури вставки в нього нової ланки матимуть такий вигляд. Для зручності список-популяція містить додаткову першу ланку:

```
type list=^section;           {оголошення списку}
      section=record c:real;   {довжина туру - ключ ланки}
                    t:Tour;    {перестановка номерів міст}
```

```

                                next:list end;                                {поле зв'язку}
var l:list;                        {загальна популяція - в глобальній змінній}
procedure insertTour(q:list);
{ вставляє ланку q в список l методом простої вставки }
var p:list;                        {робоча змінна}
begin p:=l;                        {переглядаємо список від початку}
    while (p^.next<>nil) and (p^.next^.c<q^.c)
        do p:=p^.next;                {і шукаємо місце}
    q^.next:=p^.next; p^.next:=q      {вставляємо ланку}
end; {insertTour}

```

Щоб створити початкову популяцію, потрібно задати її розмір і згенерувати кожну особину. Вибір розміру популяції важко обґрунтувати строго. Більша популяція містить більше розмаїтого генетичного матеріалу, проте потребує більшого часу на опрацювання її особин, тому на практиці доводиться вибирати якусь «золоту середину», наприклад, враховуючи експериментальний досвід.

Генерування деякого туру – це генерування випадкової перестановки n чисел. Таке завдання тільки на перший погляд може видатися простим. Далеко не достатньо отримати за допомогою давача випадкових чисел n різних значень: у перестановці мають бути *всі* числа, і кожне – *один* раз. Давач справді спочатку дає різні значення, проте що більше членів перестановки отримано, то частіше він видаватиме уже використані значення. На перевірки і генерування все нових і нових значень затрататиметься недопустимо багато часу.

Відомо кілька алгоритмів створення перестановок. Ми використали підхід, що нагадує витягання з колоди випадкової карти. З цією метою перед формуванням перестановки створюють рядок, що містить усі члени перестановки, записані підряд. Потім з рядка поступово вилучають випадково вибрані члени і записують у масив-тур. Формування випадкового туру реалізує функція *formTour*:

```

procedure formTour(var t:tour);
var s:string[nT]; i,j:byte;
begin s:='';                        {рядок s містить номери всіх міст,}
    for i:=1 to nT do s:=s+chr(i);    {записані підряд}
    for i:=1 to nT-1 do                {вибираючи їх випадково по-одному,}
    begin j:=random(nT-i)+1; t[i]:=ord(s[j]); {запишемо їх в}
        delete(s,j,1) end;           {масив і вилучимо з рядка}
    t[nT]:=ord(s[1])
end; {formTour}

```

А формування всієї популяції легко виконати за допомогою такого циклу:

```

const population=150;                {розмір популяції}
begin .....
    l:=new(list);                      {створили додаткову ланку, вона - порожня}
    l^.next:=new(list); with l^.next^ do {додали до списку}
    begin formTour(t); c:=cost(t); next:=nil {першу ланку}
    end; {with}
    for i:=2 to population do {формуємо решту особин популяції,}
    begin q:=New(list); with q^ do      {обчислюємо їхні довжини}

```

```

begin formTour(t); c:=cost(t); next:=nil
end; {with} insertTour(q);           {i вставляємо в список}
end; {for}

```

У результаті отримуємо початкову популяцію, особини якої впорядковано за спаданням пристосованості до умов задачі. Поки що ця властивість не дуже суттєва. Використаємо її для здійснення відбору, а зараз опишемо генерування нових поколінь розв'язків.

Раніше ми вирішили використати однобатьківський оператор породження потомства. Проведені нами експерименти з програмою засвідчили, що від вдалого вибору цього оператора суттєво залежить загальна ефективність алгоритму. Виявилось, що найкраще він знаходить розв'язки тоді, коли оператор продукує потомство як з незначними мутаціями, коли переставляють тільки два міста, так і з серйознішими, коли в обміні бере участь кілька міст. Його реалізовано за допомогою процедури *transform*. Кількість мутацій батьківського туру задає параметр *mutation*:

```

procedure transform(var p,q:list; mutation:byte);
{ створює тур q - мутацію батьківського туру p }
var i,j,k,l:byte;
begin q:=new(list); with q^ do
  for l:=1 to mutation do           {виконуємо mutation раз}
  begin i:=random(nT-1)+1; j:=random(nT-1)+1;
    t:=p^.t; k:=t[i]; t[i]:=t[j]; t[j]:=k;   {обміни міст i}
    c:=cost(t); next:=nil   {обчислюємо вартість нового туру}
  end{for & with}
end; {transform}

```

Щоб утворити нове покоління, потрібно вирішити, як зберігати потомство. Нові особини не можна долучати до популяції, поки старше покоління не закінчить відтворення, щоб різні покоління не переплуталися, бо тоді важко буде простежити, хто зі старших уже дав потомство, а хто – ні. Використаємо з метою тимчасового зберігання нового покоління стек. Після завершення відтворення нове покоління зі стека перенесемо до загальної популяції.

З метою запису туру до стека і вилучення його звідти використовують такі процедури:

```

var stack:list;
procedure Push(p:list);
  begin p^.next:=stack; stack:=p
  end; {Push}
procedure Pop(var p:list);
  begin p:=stack; stack:=stack^.next
  end; {Pop}

```

Нагадаємо, що популяція зберігається у впорядкованому списку, тому після об'єднання поколінь ми отримуємо єдину, впорядковану за пристосованістю, популяцію.

Увесь процес формування нового покоління відбувається так:

```

const kG=5;           {кількість потомків однієї особини}
var stack:list;      {стек для зберігання нового покоління}
begin                .....
  stack:=nil;         {спочатку стек порожній}
  p:=l^.next;        {переглядаємо популяцію від першої ланки}
  while p<>nil do
    begin for j:=1 to kG do           {створили kG потомків i}
      begin transform(p,q,j); Push(q)   {занесли їх у стек}
      end; {for} p:=p^.next
    end; {while}
  while stack<>nil do {долучаємо нове покоління до популяції}
    begin Pop(q); insertTour(q)
    end; {while}

```

Тепер відбір кращих можна виконати за допомогою процедури, яка залишає перші *population* ланок списку (саме вони містять кращі особини популяції) *l* і вилучає всі решту.

```

procedure truncList;
var p,q:list; i:byte;
begin p:=l;           {знаходимо адресу останньої ланки}
  for i:=1 to population do p:=p^.next;       {популяції}
  q:=p^.next; p^.next:=nil;                       {відрізаємо «хвіст» списку}
  while q<>nil do                                  {i ліквідуємо всіх недосконалих}
    begin p:=q; q:=q^.next; dispose(p)
    end; {while}
end; {truncList}

```

Ми описали всі найважливіші частини генетичного алгоритму розв'язування задачі комівояжера. Зібрати їх воедино, задати карту та організувати виведення результатів є «справою техніки». Очевидно, що для цієї задачі нагляднішим було б виведення числової та графічної інформації щодо знайденого туру, тому остаточна реалізація програми залежить від графічного середовища, в якому вона працюватиме. Тому ми вирішили не наводити повного тексту програми, обтяженого технічними деталями. Читач і сам зможе легко довести програму до завершеного вигляду.

Розглянемо результати, отримані для задачі 46 (рис. 12).

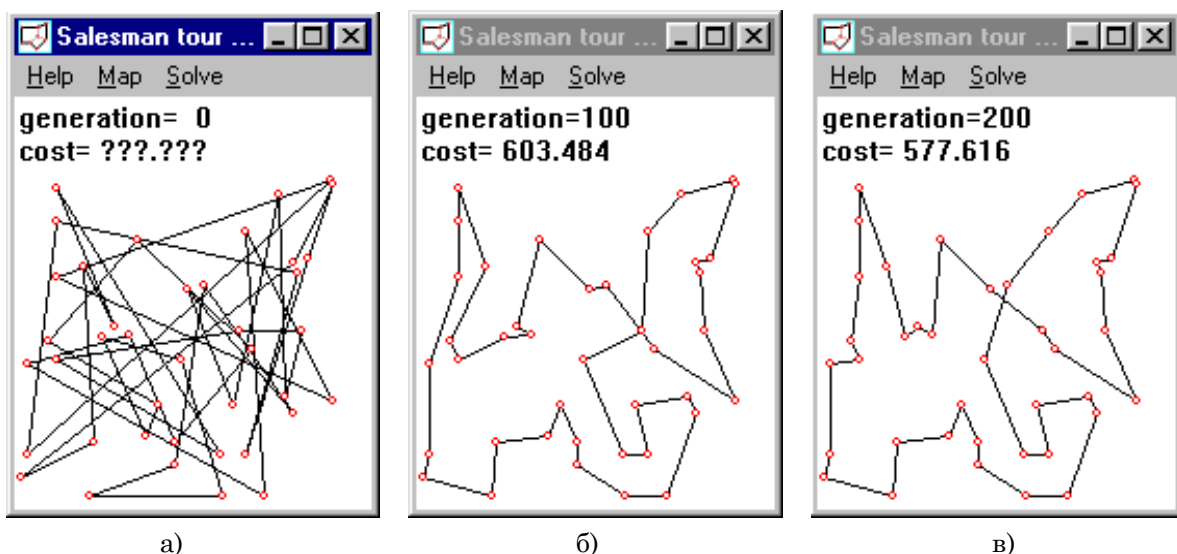


Рис. 12. Процес відшукування розв'язку задачі комівояжера (40 міст)

На рис. 12, а зображено міста, які потрібно відвідати комівояжерові. Їх сполучено відрізками в порядку нумерування. Ламана виявилася досить заплутаною. А вже соте покоління (рис. 12, б) містить непогане наближення до розв'язку. Наступні сто поколінь покращили його (рис. 12, в) і дали дуже якісну евристику. Залишається тільки виправити маршрут, щоб не було перехрещень, і ми отримаємо, мабуть, оптимальний тур.

16. Список літератури

1. *Абрамов С. А., Гнездилова Г. Г., Капустина Е. Н., Селюн М. И.* Задачи по программированию. – М.: Наука, 1988.
2. *Вирт Н.* Систематическое программирование. Введение. – М.: Мир, 1977.
3. *Вирт Н.* Алгоритмы + структуры данных = программы. – М.: Мир, 1985.
4. *Вьюнова Н. И., Галатенко В. А., Ходулев А. Б.* Систематический подход к программированию. – М.: Наука, 1988.
5. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. – М.: Мир, 1981.
6. *Кнут Д.* Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. – М.: Мир, 1978.
7. *David B. Fogel* Evolutionary computing.// IEEE Spectrum, February 2000. – p. 26-32.