4. Вкладені цикли в матричних задачах

В алгоритмах попереднього параграфа для обробки послідовностей значень ми використовували прості цикли. Одного циклу було достатньо, бо послідовність, вектор чи масив $a_1, a_2, ..., a_n$ 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 відшукання її максимального елемента, для виконання дій над матрицями тощо.

Під час розв'язування матричних задач нам доведеться виконувати однотипні дії: створювати динамічну матрицю заданого розміру та звільняти пам'ять від неї, друкувати матрицю-результат. Вирішення цих завдань опишемо за допомогою окремих функцій, які згодом використаємо для розв'язування складніших задач.

4.1. Керування пам'яттю для матриці

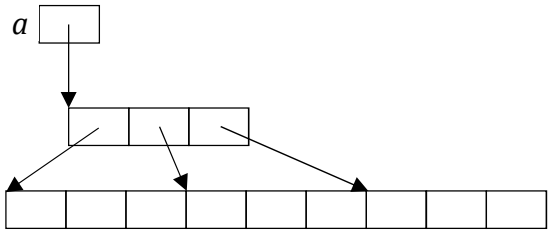

Рис. 1. Динамічна матриця порядку $n = 3$

Ми вже вміємо створювати динамічний одновимірний масив. Тепер навчимося створювати динамічну квадратну матрицю порядку n. Це завдання складніше, бо треба сворити двовимірну структуру. Звичайна матриця *int A* [3][3] займає неперервну ділянку пам'яті з 9-ти цілих і забезпечує двохрівневу індексацію. Спробуємо відтворити таку ж структуру і в динамічної матриці. На першому етапі створюється масив вказівників на рядки, далі виділяється пам'ять для n^2 елементів матриці, яку згодом ділимо вказівниками на ділянки по n елементів – на рядки матриці.

```
// виділення пам'яті для динамічної матриці
int** createMatrix(int n)
{
	int ** a = new int*[n]; // пам'ять для вказівників рядків
	a[0] = new int[n * n];  // пам'ять для елементів матриці
	for (int i = 1; i < n; ++i) // налаштування вказівників на рядки:
		a[i] = a[i -1] + n;   // адреса наступного == попереднього + довжина
	return a;
}
```

Функція *createMatrix* створює в динамічній пам'яті структуру, схематично зображену на рис. 1. Звільнити пам'ять від неї зовсім просто, як у *eraseMatrix*.

```
// звільнення пам'яті від матриці
void eraseMatrix(int ** a)
{
	delete [] a[0];
	delete [] a;
}
```

4.2. Виведення матриці на друк

Як надрукувати матрицю? Природньо було б так, як ми звикли її записувати: у вигляді прямокутної таблиці. Використаємо для цього два цикли. Зовнішній *for*(*i* ...) перебирає рядки матриці. Вкладений у нього *for*(*j* ...) виводить елементи *i*-го рядка матриці в один рядок екрана (по чотири символи для кожного елемента). Після завершення рядка матриці виводиться символ закінчення екранного рядка, що і створює потрібний вигляд.

```
#include <iomanip>

// виведення матриці на консоль
void printMatrix(int** a, int n)
{
	for (int i = 0; i < n; ++i)
	{
		for (int j = 0; j < n; ++j)
			cout << std::setw(4) << a[i][j];
		cout << '\n';
	}
	return;
}
```

4.3. 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. *Задано натуральне n. Побудувати і надрукувати одиничну матрицю n-го порядку.*

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:

```
void unitaryMatrix()
{
	cout << "\n *Побудова одиничної матриці заданого розміру (з перевірками)*"
		<<"\n\nВведіть розмір матриці: ";
	unsigned n; cin >> n;
	// виділення пам'яті для динамічної матриці
	int ** a = createMatrix(n);
	// заповнення матриці значеннями
	for (int i = 0; i < n; ++i)
		for (int j = 0; j < n; ++j)
			a[i][j] = (i == j) ? 1 : 0;
	// виведення результатів
	printMatrix(a, n);
	eraseMatrix(a);
	return;
}
```

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 $j = i+1, ..., n$ – переберемо їх за допомогою вкладеного циклу. Для правильної побудови циклів мусимо врахувати особливість C++ щодо номерування елементів масивів: перший має номер 0, другий – 1 і так далі. Це значить, що перший рядок матриці міститься у 0-му рядку двовимірного масиву програми, верхній трикутник – у рядках з 0-го по (n–2)-й масиву, у кожному з яких елементи прономеровано від (i+1) до (n–1). Ці значення і використаємо в циклах:

```
void symmetricBuildMatrix()
{
	cout << "\n *Побудова одиничної матриці заданого розміру (за структурою)*"
		<<"\n\nВведіть розмір матриці: ";
	unsigned n; cin >> n;
	// виділення пам'яті для динамічної матриці
	int ** a = createMatrix(n);
	// заповнення матриці значеннями:
	//   головна діагональ
	for (int i = 0; i < n; ++i) a[i][i] = 1;
	//   трикутники над і під діагоналлю
	for (int i = 0; i < n - 1; ++i)
		for (int j = i + 1; j < n; ++j)
			a[i][j] = a[j][i] = 0;
	// виведення результатів
	printMatrix(a, n);
	eraseMatrix(a);
	return;
}
```

Задача 18. *Задано натуральне n. Отримати квадратну матрицю порядку n 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. Врахуємо також, що для індексів матриці C++ рівняння побічної діагоналі має вигляд $j = n - 1 - i$.

```
void triangleMatrix()
{
	cout << "\n *Побудова трикутної матриці заданого розміру*\n\n"
		<<"Введіть розмір матриці: ";
	unsigned n; cin >> n;
	// виділення пам'яті для динамічної матриці
	int ** a = createMatrix(n);
	// заповнення матриці значеннями:
	int n1 = n - 1;
	//   побічна діагональ
	for (int i = 0; i < n; ++i) a[i][n1 - i] = n;
```

```
      //   верхній трикутник
      for (int i = 0; i < n1; ++i)
            for (int j = 0; j < n1 - i; ++j)
                  a[i][j] = i + j + 1;
      //   нижній трикутник
      for (int i = 1; i < n; ++i)
            for (int j = n - i; j < n; ++j)
                  a[i][j] = 0;
      // виведення результатів
      printMatrix(a, n);
      eraseMatrix(a);
      return;
}
```

Задача 19. *Задано натуральне n, дійсні числа* $a_1, a_2, \ldots, a_{n^2}$. *Отримати дійсну квадратну матрицю порядку n, елементами якої є ці числа, розташовані «змійкою» за схемою, зображеною на рис. 2.*

Цю задачу можна розв'язати принаймні двома способами: прочитати послідовність $a_1, a_2, \ldots, a_{n^2}$, і перебрати елементи матриці, присвоївши їм відповідні значення цієї послідовності; або перебрати елементи послідовності і вставити їх у відповідні місця матриці. Опишемо обидва способи.

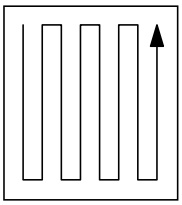

Рис. 2. Схема для $n = 8$

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 $k = j * n + i$ (в термінах індексів C++), а для парних – співвідношення $k = (j + 1) * n - i - 1$. Тепер можна сформувати матрицю, перебираючи її за стовпцями. Для наглядності як задану послідовність використаємо числа $1, 2, \ldots, n^2$.

```
void matrixFromSequence()
{
      cout << "\n *Заповнення матриці \"змійкою\" заданими значеннями*\n\n"
            <<"Введіть розмір матриці: ";
      unsigned n; cin >> n;
      // виділення пам'яті для динамічної матриці
      int ** a = createMatrix(n);
      // задану послідовність значень сконструюємо
      int * b = new int[n * n];
      for (int i = 0; i < n * n; ++i) b[i] = i + 1;
      // заповнення матриці значеннями:
      for (int j = 0; j < n; ++j)
      {
            if (j % 2 == 0) // непарний стовпець математичної матриці
            {
                  int l = j * n;
                  for (int i = 0; i < n; ++i)
                        a[i][j] = b[l + i];
            }
            else // парний стовпець математичної матриці
            {
                  int l = (j + 1) * n - 1;
                  for (int i = 0; i < n; ++i)
                        a[i][j] = b[l - i];
```

```
            }
      }
      // виведення результатів
      printMatrix(a, n);
      eraseMatrix(a);
      return;
}
```

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. 2, матрицею. Очевидно, що в непарних стовпцях вона опускається, у парних – підіймається. Тобто напрям руху змінюється тоді, коли заповнено черговий стовпець з n елементів і «змія» переходить до наступного. Заповнення розпочинається з елемента b_{11} ($b[0][0]$ у програмі), а закінчується, коли розташовано усі члени заданої послідовності. Для послідовного зчитування заданих чисел можна використати просту змінну, бо прочитане значення відразу розміститься в матриці. Проте, як і раніше, замість заданої послідовності ми використаємо 1, 2, ..., n^2.

```
void sequenceToMatrix()
{
      cout << "\n *Вкладення \"змійкою\" послідовності в матрицю*\n\n"
            <<"Введіть розмір матриці: ";
      unsigned n; cin >> n;
      // виділення пам'яті для динамічної матриці
      int ** a = createMatrix(n);
      // заповнення матриці значеннями:
      int i = 0; int j = 0;   // координати початкового елемента
      int step = 1;           // спочатку напрям руху - додатній
      for (int k = 1; k <= n * n; ++k)
      {                         // вкладаємо всі члени послідовності
            a[i][j] = k;        // розташували чергове число
            if (k % n == 0)     // стовпець заповнено
            {
                  ++j;          // перейшли до нового стовпця
                  step = -step; // і змінили напрям
            }
            else i += step;     // просуваємося стовпцем
      }
      // виведення результатів
      printMatrix(a, n);
      eraseMatrix(a);
      return;
}
```

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

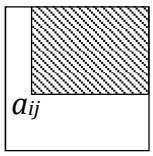

Рис. 3

Задача 20. *Задано квадратну матрицю A розміру n×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. 3 (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. 3. 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, записуємо програму. Як і раніше, враховуємо спосіб C++ номерування елементів масиву.

```
void regionsSumming()
{
	cout << "\n *Побудова матриці з сум прямокутних частин іншої матриці*\n\n";
	// Для спрощення початкову матрицю задамо в коді програми
	const unsigned n = 7;
	int a[n][n] = {{ 1, 2, 0,-1,-2, 1,}, {-1, 0, 1, 2, 3,-2,},
	    { 0, 1,-1, 1, 0, 2, 3}, { 2, 3,-2,-1, 1, 0, 2}, { 1, 0, 1,-1, 0,-1, 1},
	    { 2,-1, 1, 1, 1,-1,-2}, {-3, 1,-2, 1,-1, 1, 2}};
	// Ініціалізацію можна легко замінити введенням з файла
	/*
	ifstream f("matrix.txt");
	for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j) f >> a[i][j];
	*/
	// виділення пам'яті для динамічної матриці
	int ** b = createMatrix(n);
	for (int i = 0; i < n; ++i)
	{                                          // перебираємо елементи матриці B
	      for (int j = 0; j < n; ++j)
	      {
	            int s = 0;          // сумуємо елементи відповідної області
	            for (int k = 0; k <= i; ++k)
	                  for (int l = j; l < n; ++l) s += a[k][l];
	            b[i][j] = s;
	      }
	}
	// виведення результатів
	printMatrix(b, n);
	eraseMatrix(b);
	return;
}
```

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 $b_{1n} = a_{1n}$, $b_{in} = b_{i-1,n} + a_{in}$, $i = 2, \ldots, n$, а для усіх інших – співвідношення $b_{ij} = b_{i,j+1} + s_{ij}$, $j = n-1, \ldots, 1$, де $s_{1j} = a_{1j}$, $s_{ij} = s_{i-1,j} + a_{ij}$, $i = 1, \ldots, n$. Економічний щодо обчислень алгоритм реалізує така програма:

```
void economySumming()
{
	cout << "\n *Економна побудова матриці з сум прямокутних частин іншої
матриці*\n\n";
	// Для спрощення початкову матрицю задамо в коді програми
```

```
    const unsigned n = 7;
    int a[n][n] = {{ 1, 2, 0,-1,-2, 1,}, {-1, 0, 1, 2, 3,-2,},
        { 0, 1,-1, 1, 0, 2, 3}, { 2, 3,-2,-1, 1, 0, 2}, { 1, 0, 1,-1, 0,-1, 1},
        { 2,-1, 1, 1, 1,-1,-2}, {-3, 1,-2, 1,-1, 1, 2}};
    // виділення пам'яті для динамічної матриці
    int ** b = createMatrix(n);
    // формуємо останній стовпець
    int n1 = n - 1;
    b[0][n1] = a [0][n1];
    for (int i = 1; i < n; ++i)
          b[i][n1] = b[i - 1][n1] + a[i][n1];
    // рекурентно обчислюємо решту стовпців
    for (int j = n1 - 1; j >= 0; --j)
    {
          int s = 0;   // сума & нові елементи одночасно
          for (int i = 0; i < n; ++i)
          {
                s += a[i][j];
                b[i][j] = b[i][j + 1] + s;
          }
    }
    // виведення результатів
    printMatrix(b, n);
    eraseMatrix(b);
    return;
}
```

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

4.4. 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 загромаджувати тексти програм практично стандартними діями для створення-знищення та введення-виведення матриць, обмежимося лише інформативною частиною алгоритмів. Використовуватимемо в них такі оголошення:

```
const int n = 10; const int m = 12; const int t = 9;
double A[n][m]; double B[n][m]; double C[n][m];
double D[n][n]; double P[m][t]; double Q[n][t];
```

Задача 21. *Задано дійсні матриці A, B: n×m. Обчислити матрицю C, якщо C = A + B.*
Відомо, що матриці додаються поелементно, тому цю задачу легко розв’язати так:

```
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
        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. *Задано дійсні матриці A: n×m, P: m×t. Обчислити матрицю Q, якщо Q = A × P.*

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

```
double s;
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < t; ++j)
    {
        s = 0.0;
        for (int k = 0; k < m; ++k)
            s += A[i][k] * P[k][j];
        Q[i][j] = s;
    }
}
```

Задача 23. *Транспонувати задану 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 (int i = 0; i < n; ++i)
{
    for (int j = 0; j < n; ++j)
    {
        double s = D[i][j]; D[i][j] = D[j][i]; D[j][i] = s;
    }
}
```

Спробуйте, і ви пересвідчитеся, 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 (зверніть увагу на межі індексів циклів):

```
for (int i = 0; i < n-1; ++i)
{
    for (int j = i+1; j < n; ++j)
    {
        double s = D[i][j]; D[i][j] = D[j][i]; D[j][i] = s;
    }
}
```

4.5. Порівняння та переміщення елементів матриці

Про відшукання 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*×*m* матриця):

```
double b = A[0][0];
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        if (A[i][j] > b) 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 1?

Задача 24. *Задано матрицю A розміру n×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 відшукання 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. Такий обмін виконаємо поелементно:

```
void moveMax()
{
      cout << "\n *Переміщення max-елемента переставлянням "
           << "рядків і стовпців матриці*\n\n";
      // Для спрощення розміри матриці задамо в коді програми
      const unsigned n = 3;
      const unsigned m = 4;
      int a[n][m];
      // Введення заданої матриці
      cout << "Введіть елементи матриці " << n << 'x' << m << '\n';
      for (unsigned i = 0; i < n; ++i)
            for (unsigned j = 0; j < m; ++j) cin >> a[i][j];
      int b = abs(a[0][0]); // початкове значення максимального
      unsigned k = 0;           // та його координати
      unsigned l = 0;
      // тепер - пошук в матриці
      for (unsigned i = 0; i < n; ++i)
      {
            for (unsigned j = 0; j < m; ++j)
                  if (abs(a[i][j]) > b)  // знайшли більше
                  {
                        b = abs(a[i][j]); // і запам'ятали
                        k = i; l = j;
                  }
      }
      if (k != 0) // міняємо місцями рядки
            for (unsigned j = 0; j < m; ++j)
```

```
            {
                  int toSwap = a[0][j];
                  a[0][j] = a[k][j];
                  a[k][j] = toSwap;
            }
      if (l != 0) // міняємо місцями стовпці
            for (unsigned i = 0; i < n; ++i)
            {
                  int toSwap = a[i][0];
                  a[i][0] = a[i][l];
                  a[i][l] = toSwap;
            }
      // Друкуємо результати
      for (unsigned i = 0; i < n; ++i)
      {
            for (unsigned j = 0; j < m; ++j) cout << '\t' << a[i][j];
            cout << '\n';
      }
      return;
}
```

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 відшукання 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 відшукання найбільшого. Це ми також вміємо робити, тому задачу можна розв'язати!

```
void saddleElement()
{
      cout << "\n *Відшукання сідлового елемента матриці*\n\n";
      // Для спрощення розміри матриці задамо в коді програми
      const unsigned n = 3;
      const unsigned m = 4;
      int a[n][m];
      // Введення заданої матриці
      cout << "Введіть елементи матриці " << n << 'x' << m << '\n';
      for (unsigned i = 0; i < n; ++i)
            for (unsigned j = 0; j < m; ++j) cin >> a[i][j];
      // початковим значенням сідлового елемента є максимальний елемент
      // першого рядка матриці - знайдемо його!
      int saddle = a[0][0]; // початкове значення максимального
      unsigned k = 0;           // та його координати
      unsigned l = 0;
      // тепер - пошук в рядку
      for (unsigned j = 1; j < m; ++j)
            if (a[k][j] > saddle)  // знайшли більше
            {
                  saddle = a[k][j]; // і запам'ятали
                  l = j;
```

```
            }
      // тепер переглянемо всі інші рядки матриці
      for (unsigned i = 1; i < n; ++i)
      {
            // і знайдемо їхні максимальні елементи
            int currMax = a[i][0];
            unsigned p = 0;
            for (unsigned j = 1; j < m; ++j)
                  if (a[i][j] > currMax)
                  {
                        currMax = a[i][j];
                        p = j;
                  }
            // та порівняємо їх з поточним значенням сідлового елемента
            if (currMax < saddle)
            {
                  saddle = currMax;
                  k = i; l = p;
            }
      }
      cout << "Сідловим є елемент a[" << k <<"]["<< l <<"] = "<< saddle << '\n';
      return;
}
```

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.