3. 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.

3.1. Скільки є «правильних» серед усіх заданих?

У задачах, описаних у цьому параграфі, необхідно обчислити кількість членів заданої послідовності, які задовольняють 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. *Задано натуральні числа n, a_1, ..., a_n. Обчислити кількість членів a_i послідовності $\{a_i\}$, що задовольняють 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 *power_of_2*, а факторіал – у змінній *factorial*. Кожне значення a_i розглядають тільки один раз, тому для його зберігання використаємо просту змінну *a*. Перед початком циклу необхідно ініціалізувати змінні *power_of_2*, *factorial* і лічильник «правильних» членів послідовності – змінну *counter*:

```
void CountProper()
{
	cout << "\n *Обчислення кількості \"правильних\" членів послідовності*\n\n";
	unsigned n;
	cout << "Введіть кількість чисел: "; cin >> n;
	long long power_of_2 = 1; // степінь двійки
	long long factorial = 1;  // факторіал
	unsigned counter = 0;     // лічильник
	for (unsigned i = 1; i <= n; ++i)
	{
		cout << "Введіть " << i << "-е число: ";
		double a; cin >> a;
		power_of_2 *= 2;
		factorial *= i;
		if (power_of_2 < a && a < factorial + 3) ++counter;
	}
	cout << "k = " << counter << '\n';
	return;
}
```

Задача 13. *Задано натуральне число n і дійсні числа r, a_1, a_2, ..., a_n. Скільки серед точок (a_1, a_n), (a_2, a_{n-1}), ..., (a_n, a_1) є таких, що належать кругові радіуса r з центром у початку координат?*

За яким правилом змінюються індекси заданих чисел у послідовності точок? Номер абсциси збігається з номером точки, а з ординатами – складніше. Перша точка має ординату з номером *n*, друга – з номером $n-1$, третя – $n-2$ і так далі. Легко бачити, що ординатою *i*-тої

точки є число a_{n+1-i}. 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 кругові, то й точка (a_{n+1-i}, a_i) теж йому належить. Отже, умову $d_i \le r$ 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 $d_i \le r$, а $d_i^2 \le r^2$. Тоді відпаде необхідність щоразу добувати корінь, а величину r^2 можна обчислити і запам'ятати один раз перед циклом.

Чи вдасться нам об'єднати в одному циклі введення та опрацювання даних, як у попередніх задачах? Для обчислення d_1 нам необхідні значення a_1 і a_n. Перше з них ми легко отримаємо за допомогою звичайного введення, але не друге, бо воно розташоване аж наприкінці заданої послідовності чисел. Тому, перш ніж розпочинати обчислення, нам доведеться прочитати всю послідовність $\{a_i\}$. Прочитати і запам'ятати, адже ніхто не буде декілька разів уводити одні і ті ж значення. Для зберігання послідовності чисел добре підходить масив. Щоб оголосити статичний масив, потрібно знати тип елементів і їхню кількість. Тип чисел задано в умові задачі, а кількість a_i стає відомою на етапі виконання програми. За таких умов необхідний масив створимо динамічно. Нагадаємо, що в мові C++ нумерування елементів масиву починається з нуля.

```
void InCircle()
{
	cout << "\n *Обчислення кількості точок в крузі*\n\n";
	cout << "Введіть кількість координат: ";
	unsigned n; cin >> n;
	cout << "Введіть радіус: ";
	double r; cin >> r;
	double * a = new double[n]; // створення масиву
	cout << "Введіть " << n << " чисел: ";
	for (unsigned i = 0; i < n; ++i) cin >> a[i];
	// введення даних закінчено
	// розпочинаємо перевірки
	unsigned k = 0; // лічильник
	double r2 = r * r;
	// перевіримо першу половину точок
	for (unsigned i = 0; i < n / 2; ++i)
	{
		if (a[i] * a[i] + a[n-1-i] * a[n-1-i] <= r2) ++k;
	}
	// врахуємо другу, симетричну половину точок
	k *= 2;
	if (n % 2 > 0) // непарна кількість точок
	{	// перевіримо середню
		if (2 * a[n/2] * a[n/2] <= r2) ++k;
	}
	cout << "Точок в крузі є " << k << '\n';
	delete[] a;
	return;
}
```

У цьому алгоритмі введення даних відокремлене від опрацювання. Про одну з причин такого підходу ми вже говорили, але є ще одна: цикл перевірок виконує вдвічі менше кроків ніж цикл уведення.

3.2. 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_0, a_1, ..., a_{n-1}. Для зберігання найбільшого значення використаємо додаткову змінну. Назвемо її, наприклад, *max_value*. Початковим значенням для *max_value* може бути значення будь-якого члена послідовності, проте найзручніше вибрати перший. Далі перебираємо усі інші a_i (з наступного по останній) і перевіряємо, чи не знайдеться елемент, значення якого більше за *max_value*. Якщо так, то про старе значення *max_value* можна забути, а замість нього запам'ятати знайдене і продовжити порівняння. Операторами мови C++ це можна записати так:

```
max_value = a[0]; // значення найбільшого
for (int i = 1; i < n; ++i)
      if (a[i] > max_value) max_value = a[i];
```

Чи обов'язково початковим значенням для *max_value* має бути a_0? Ні. Можна шукати найбільший елемент і так:

```
max_value = a[n-1]; // значення найбільшого
for (int i = n-2; i >= 0; --i)
      if (a[i] > max_value) max_value = a[i];
```

або взяти за початкове будь-яке інше значення a_i, проте тоді параметр циклу мав би змінюватись від 0 до n–1.

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 *max_value*, запам'ятовувати і його номер *i* (наприклад, у змінній *index_of_max*):

```
max_value = a[0];
index_of_max = 0;
for (int i = 1; i < n; ++i)
      if (a[i] > max_value)
      {
            max_value = a[i];
            index_of_max = i;
      }
```

Виявляється, знайти і значення, і номер найбільшого можна за допомогою лише однієї змінної. Справді, якщо ми знаємо номер k елемента масиву, то легко можемо отримати його значення $a[k]$. Тепер запишемо ще один варіант алгоритму:

```
index_of_max = 0; // номер найбільшого
for (int i = 1; i < n; ++i)
      if (a[i] > a[index_of_max]) index_of_max = i;
```

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 *max_value* і *index_of_max* відбудеться тільки тоді, коли $a_i > max\_value$, і не відбудеться, коли $a_i = max\_value$. Тому, очевидно, змінна *index_of_max* 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 «≥».

Задача 14. *Задано дійсні* $a_1, a_2, ..., a_{50}$. *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 *index_of_max*. Щоб замінити місцями a_1 і $a_{index\_of\_max}$, нам у пригоді стане змінна *max_value*: вона містить копію значення $a_{index\_of\_max}$. Остаточно отримаємо програму:

```
void Exchange()
{
	cout << "\n *Обмін першого з найбільшим*\n\n";
	const int n = 50;
	cout << "Введіть " << n << " чисел: ";
	int a[n];
	for (int i = 0; i < n; ++i) cin >> a[i];
	// пошук найбільшого:
	double max_value = a[0];  // значення найбільшого
	int index_of_max = 0;     // номер найбільшого
	for (int i = 1; i < n; ++i)
	{
		if (a[i] > max_value)
		{
			max_value = a[i];
			index_of_max = i;
		}
	}
	// переставляння елементів
	a[index_of_max] = a[0];
	a[0] = max_value;
	cout << "Змінена послідовність:\n";
	for (int i = 0; i < n; ++i) cout << a[i] << ' ';
	cout << '\n';
	return;
}
```

У попередній задачі ми не знали точно кількість членів заданої послідовності, тому пам'ять для неї у функції *InCircle* виділяли динамічно. У функції *Exchange* кількість членів послідовності (і відповідний розмір масиву) задано константою, бо в умові задачі 14 зазначено, що їхня кількість дорівнює 50. Для того, щоб налаштувати *Exchange* на інший розмір послідовності, достатньо змінити значення константи *n*.

3.3. Перевірка впорядкованості

Іноді початківця може поставити у скрутну ситуацію така проста задача:

Задача 15. *Задано дійсні a_1, a_2, ..., 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 перевірками можна завершувати, коли у чергової пари сусідів порушено умову впорядкованості, або коли перебрали всю послідовність. Тоді відповідь на запитання задачі легко отримати, перевіривши, чи досягнуто кінець послідовності:

```
void IsGrowth()
{
	cout << "\n *Перевірка, чи впорядкована послідовність*\n\n";
	const int n = 50;
	cout << "Введіть " << n << " чисел: ";
	int a[n];
	for (int i = 0; i < n; ++i) cin >> a[i];
	int i = 1; // номер першого елемента неперевіреної частини послідовності
	// перевірка: поки ми в межах послідовності та її члени у правильному
	// порядку, можемо просуватися вперед
	while (i < n && a[i - 1] < a[i]) ++i;
	// чому закінчився цикл?
	if (i == n) cout << "Послідовність зростаюча\n";
	else cout << "Умову впорядкованості порушено перед a[" << i << "]\n";
	return;
}
```

Важливо правильно записати умову ітераційного циклу. Порівнювати $a[i-1]$ з $a[i]$ можна, якщо кожен з них існує, тобто, якщо $i < n$, тому перевірку $i < n$ потрібно записати першою. Якщо її обчислення дає в результаті хибу, тобто, цикл досяг кінця послідовності, то до обчислення другого операнда кон'юнкції справа не дійде, і звертання за хибним індексом не відбудеться. У C++ для обчислення логічних виразів застосовуються «ліниві» обчислення. Якщо перший операнд логічного «і» набуває значення *false*, то нема потреби обчислювати другий, оскільки відомо значення цілого виразу: *false*.

Якщо послідовність не є зростаючою, то цикл завершиться ще до того, як значення i досягне значення n, і на друк буде виведено повідомлення про номер елемента послідовності, перед яким 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 $a_{i-1} > a_i$.

3.4. 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. *Задано впорядкований за незростанням масив цілих або дійсних чисел $a_0 \le a_1 \le \ldots \le a_{n-1}$ і деяке число b (відповідно ціле або дійсне), для якого необхідно знайти таке місце серед чисел a_0, a_1, ..., a_{n-1}, щоб після вставлення числа b на це місце впорядкованість не порушилася. Розглянути для прикладу випадок n=10.*

Цю задачу називають задачею пошуку місця елемента. Для b є n+1 можливість: $b \le a_0$, $a_0 < b \le a_1$, ..., $a_{n-2} < b \le a_{n-1}$, $a_{n-1} < b$, і розв'язком задачі пошуку місця елемента b буде відповідно одне з чисел 0, 1, ..., n–1, n, 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, ..., a_{n-1} 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:

```
void ForwardInsert()
{
	cout << "\n *Вставка числа у впорядковану послідовність*\n"
		<< "    пошук від початку\n\n";
	const int n = 10;
	int a[n + 1]; // потрібно на 1 більше місця
	cout << "Введіть впорядковану послідовність " << n << " чисел: ";
	for (int i = 0; i < n; ++i) cin >> a[i];
	cout << "Введіть нове число: ";
	double b; cin >> b;
	int insertion_place = 0;
	// шукаємо місце для b: поки ми перебуваємо в межах послідовності,
	//  і виконується умова впорядкованості, рухаємося вперед
	while (insertion_place < n && a[insertion_place] < b) ++insertion_place;
	//посуваємо «хвіст»
	for (int i = n; i > insertion_place; --i) a[i] = a[i - 1];
	a[insertion_place] = b; // вставляємо b на звільнене місце
	// друкуємо змінену послідовність
	for (int i = 0; i <= n; ++i) cout << ' ' << a[i];
	cout << '\n';
	return;
}
```

Як і в попередній програмі, перевірка індекса в умові циклу пошуку місця виконується першою. Якщо виявиться, що $b \le a_0$, то цикл завершиться одразу, значення *insertion_place* залишиться рівним нулю, і нове число вставимо на початок послідовності. Якщо ж $b > a_{n-1}$, то цикл виконає n перевірок і завершиться при *insertion_place* = n, нове число вставимо в кінець послідовності.

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

```
void SimpleInsert()
{
	cout << "\n *Вставка числа у впорядковану послідовність*\n"
		<< "    пошук від кінця\n\n";
	const int n = 10;
	int a[n + 1]; // потрібно на 1 більше місця
	cout << "Введіть " << n << " чисел: ";
	for (int i = 0; i < n; ++i) cin >> a[i];
	cout << "Введіть нове число: ";
	double b; cin >> b;
	// шукаємо місце для b і посуваємо «хвіст»
	// перевірки починаємо з кінця послідовності
	int comparison_place = n - 1;
```

```
	while (comparison_place >= 0 && a[comparison_place] > b)
	{
		a[comparison_place + 1] = a[comparison_place];
		--comparison_place;
	}
	a[comparison_place + 1] = b; // вставляємо b в масив
	// друкуємо змінену послідовність
	for (int i = 0; i <= n; ++i) cout << ' ' << a[i];
	cout << '\n';
	return;
}
```

Бінарний пошук. Для відшукання 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 0 і n–1 (межі індексів заданого масиву), далі ці межі крок за кроком стискують 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 $[\log_2(n+1)]+1$:

```
void BinaryInsert()
{
	cout << "\n *Вставка числа у впорядковану послідовність*\n"
		<< "    двійковий пошук\n\n";
	const int n = 10;
	int a[n + 1];
	cout << "Введіть " << n << " чисел: ";
	for (int i = 0; i < n; ++i) cin >> a[i];
	cout << "Введіть нове число: ";
	double b; cin >> b;
	// початкові межі пошуку
	int left = 0; int right = n;
	while (left != right)
	{
		int middle = (left + right) / 2; // шукаємо середній елемент
		if (a[middle] < b) left = middle + 1; //і порівнюємо з ним
		else right = middle;
	}
	//посуваємо «хвіст»
	for (int i = n - 1; i >= left; --i) a[i + 1] = a[i];
	a[left] = b; // вставляємо b в масив
	// друкуємо змінену послідовність
	for (int i = 0; i <= n; ++i) cout << ' ' << a[i];
	cout << '\n';
	return;
}
```