5. 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. *Задано масив цілих чисел* $a_1, a_2, \ldots, a_n$. *Переставити його елементи так, щоб виконувалась умова* $a_1 \le a_2 \le \ldots \le a_n$.

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.

Для випробування алгоритмів необхідна певна інфраструктура: потрібно звідкись брати вхідний масив, друкувати його перед впорядкуванням та після, викликати алгоритм впорядкування. Масив будемо створювати динамічно і наповнимо його значеннями за допомогою генератора випадкових чисел.

```
int* CreateRandomVector(unsigned n)
{
	int* a = new int[n];
	std::srand(std::time(0));  // ініціалізуємо генератор випадкових чисел
	unsigned m = n * 2;        // найбільше можливе значення в масиві
	for (unsigned i = 0; i < n; ++i)
		a[i] = std::rand() % m;
	return a;
}
```

Якщо ми захочемо впорядкувати один і той же масив декілька разів різними алгоритмами, то в пригоді стане створення копії масиву:

```
int* CreateCopyVector(int* a, unsigned n)
{
	int* c = new int[n];
	for (unsigned i = 0; i < n; ++i) c[i] = a[i];
	return c;
}
```

Виведення масиву на друк має цілком очікуваний вигляд:

```
void PrintVector(int* a, unsigned n)
{
	for (unsigned i = 0; i < n; ++i)
		cout << std::setw(5) << a[i];
	cout << std::endl;
}
```

Випробування алгоритму впорядкування організовує функція *Manage*. Її параметрами є вказівник *Sort* на функцію сортування і масив *a* цілих чисел.

```
void Manage(void (*Sort)(int* a, unsigned n), int* a, unsigned n)
{
	int* vector = createCopyVector(a, n);
	cout << "\nПеред впорядкуванням:\n"; PrintVector(vector, n);
	Sort(vector, n);
	cout << "\nПісля впорядкування:\n"; PrintVector(vector, n);
	delete[] vector;
}
```

Тепер залишилося тільки описати самі алгоритми сортування. Оголосимо їх як окремі функції.

5.1. Сортування вставками

У п. 3.4 ми вже розглядали задачу щодо відшукання місця вставляння 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 відшукання 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 отримаємо алгоритм *сортування простими вставками*.

```
// ВПОРЯДКУВАННЯ МАСИВУ a ПРОСТИМИ ВСТАВКАМИ
void SimpleInsertSort(int* a, unsigned n)
{
	// спочатку впорядкованим є лише перший елемент послідовності
	// переберемо всі інші і кожен з них вставимо на відповідне місце
	for (unsigned index_to_insert = 1; index_to_insert < n; ++index_to_insert)
	{
		// шукаємо місце для чергового елемента
		int b = a[index_to_insert];
		int comparison_place = index_to_insert - 1;
		while (comparison_place >= 0 && a[comparison_place] > b)
		{                                // посуваєм впорядковані елементи
			a[comparison_place + 1] = a[comparison_place];
			--comparison_place;
		}
		// вставляєм черговий елемент у впорядковану частину
		a[comparison_place + 1] = b;
	}
	return;
}
```

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 $1+2+...+(n-1)$ порівнянь. Отже, $n-1 \le A \le (n^2-n)/2$.

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, $2\times(n-1) \le B \le n^2/2+3/2n-2$.

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.

5.2. 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 *Exchange*. Використаємо її алгоритм для відшукання значення та індексу максимального елемента невпорядкованої частини масиву. А щоб впорядкувати весь масив, пошук найбільшого і обмін повторимо n–1 раз.

```
// ВПОРЯДКУВАННЯ МАСИВУ a ЗА ДОПОМОГОЮ ВИБОРУ НАЙБІЛЬШОГО
void FindMaxSort(int* a, unsigned n)
{
	// шукатимемо найбільший елемент невпорядкованої частини
	// і обмінюватимемо його з її останнім елементом.
	// спочатку невпорядкований весь вектор

	for (unsigned last_unsorted = n - 1; last_unsorted > 0; --last_unsorted)
	{
		int max_value = a[0]; // початкові значення і номер найбільшого
		unsigned index_of_max = 0;
		for (unsigned i = 1; i <= last_unsorted; ++i)
			if (a[i] > max_value)
			{
				max_value = a[i];
				index_of_max = i;
			}
		if (index_of_max != last_unsorted) // ставимо найбільшого на його місце
		{
			a[index_of_max] = a[last_unsorted];
			a[last_unsorted] = max_value;
		}
	}
	return;
}
```

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 не більше як $3\times(n-1)$ раз у зовнішньому циклі. Тому сортування вибором особливо ефективне тоді, коли переміщення є затратними, наприклад, якщо сортувати необхідно масив ключів, і разом з ними переміщувати великі масиви інформації. Приклади таких програм наведемо пізніше.

5.3. 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, невпорядковані 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 випливають на поверхню масиву, тому метод сортування обмінами називають ще *методом бульбашки*. Наведемо текст функції, що реалізує цей метод:

```
// ВПОРЯДКУВАННЯ МАСИВУ a МЕТОДОМ БУЛЬБАШКИ (ОБМІНІВ)
void BubbleSort(int* a, unsigned n)
{
	// порівняємо кожну пару сусідів і обміняємо місцями
	// невпорядковані пари. перегляд повторимо n-1 раз
	for (unsigned last_unsorted = n - 1; last_unsorted > 0; --last_unsorted)
		for (unsigned i = 0; i < last_unsorted; ++i)
			if (a[i] > a[i + 1]) // «невпорядкованих» сусідів
			{                    // потрібно поміняти місцями
				int to_swap = a[i];
				a[i] = a[i + 1];
				a[i + 1] = to_swap;
			}
}
```

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 *BubbleSort* 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 невпорядкованої частини масиву. З урахуванням цих міркувань отримаємо функцію:

```
// ВПОРЯДКУВАННЯ МАСИВУ a ПОКРАЩЕНИМ МЕТОДОМ ОБМІНІВ
void ReplaceSort(int* a, unsigned n)
{	// порівняємо кожну пару сусідів і обміняємо місцями
	// невпорядковані пари. перегляд повторятимемо, поки потрібно
	unsigned place_of_swap;         // місце останнього виконаного обміну
	unsigned last_unsorted = n - 1; // спочатку весь масив - невпорядкований
	while (last_unsorted > 0)
	{
		place_of_swap = 0; // припустили, що всі пари є у правильному порядку
		for (unsigned i = 0; i < last_unsorted; ++i)
			if (a[i] > a[i + 1]) // знайшли «невпорядкованих» сусідів
			{                                 // потрібно поміняти їх місцями
				int to_swap = a[i];
				a[i] = a[i + 1];
				a[i + 1] = to_swap;
				place_of_swap = i;   // і запам’ятати місце, де це сталося
			}
		last_unsorted = place_of_swap; // змінили межу невпорядкованої частини
	}
}
```

Кількості порівнянь і переміщень за цим алгоритмом оцінюють так: $n-1 \le A \le (n^2-n)/2$, $0 \le B \le 3n^2/2-3n/2$. 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. Одна з них вирішує проблему «черепах».

Під час дослідження алгоритму *ReplaceSort* можна помітити, що його роботу суттєво сповільнюють елементи з малими значеннями, розташовані наприкінці масиву. За один крок зовнішнього циклу *ReplaceSort* переміщує найбільший елемент у кінець масиву і не звертає увагу на його мінімальні елементи. Тому, якщо останнім у масиві є його найменший елемент, алгоритмові *ReplaceSort*, незважаючи на покращення, доведеться виконати всі $n-1$ кроків зовнішнього циклу. У такому випадку мінімальний елемент і є тією черепахою, яка дуже поволі рухається на своє місце. Вона стала б дуже прудкою, якби алгоритм сортування умів переміщувати за один прохід найменше значення на початок масиву. Ця думка приводить до наступного вдосконалення: ми могли б спочатку переглядати пари сусідів зліва направо і переміщувати найбільший елемент у кінець масиву, а потім – справа наліво, щоб перемістити найменший на початок масиву. Новий алгоритм назвали методом змішування. Він створює впорядковані частини масиву і в кінці, і на його початку.

```
// ВПОРЯДКУВАННЯ МАСИВУ a МЕТОДОМ ЗМІШУВАННЯ
void ShakerSort(int* a, unsigned n)
{
      int first_unsorted = 0;  // межі невпорядкованої частини
      int last_unsorted = n - 1;
      while (first_unsorted != last_unsorted)
      {
            // переглядаємо невпорядковану частину зліва направо
            int last_place_of_swap = first_unsorted;
            for (int i = first_unsorted; i < last_unsorted; ++i)
                  if (a[i] > a[i + 1])
                  {                    // та виконуємо необхідні обміни
                        int to_swap = a[i];
                        a[i] = a[i + 1];
                        a[i + 1] = to_swap;
                        last_place_of_swap = i;
                  }
            last_unsorted = last_place_of_swap; // посунули праву межу ліворуч
            // а тоді - справа наліво
            int first_place_of_swap = last_unsorted;
            for (int i = last_unsorted; i > first_unsorted; --i)
                  if (a[i - 1] > a[i])
                  {                   // також виконуємо обміни
                        int to_swap = a[i];
                        a[i] = a[i - 1];
                        a[i - 1] = to_swap;
                        first_place_of_swap = i;
                  }
            first_unsorted = first_place_of_swap; // посунули ліву межу праворуч
      }
}
```

Оцінки складності цього алгоритму такі ж, як у попереднього.