**ПОШУК, СОРТУВАННЯ ТА ПОНЯТТЯ СКЛАДНОСТІ У ПРОГРАМУВАННІ**

**1. Пошук за ключем у масиві**

**1.1. Лінійний пошук**

Читачеві колись доводилося шукати своє прізвище в списках, надрукованих не в алфавітному порядку? Або уявіть собі словник на 100 тисяч слів, розташованих там без упорядкування за алфавітом. Потрібне слово шукати там доведеться довго. Дуже довго. Звичайно, якщо воно випадково не потрапило в самісінький початок. А якщо в кінець чи середину? А пошук слова в "нормальному" словнику займає чомусь кілька секунд незалежно від його розташування там.

У цьому параграфі ми наведемо два алгоритми. Перший описує пошук "підряд" у невпорядкованій послідовності, другий – той пошук, до якого ми звикли, шукаючи слова в словниках. Замість слів розглянемо цілі значення елементів масиву. Тип значень може бути й іншим – головне, щоб їх можна було порівнювати.

Нехай елементи масиву *A*[1], *A*[2], … , *A*[*n*] та змінна *key* ("ключ") мають той самий тип T. ***Пошук за ключем*** полягає в пошуку номера *i* такого, що *A*[*i*]= *key*. За відсутності такого номера результатом будемо вважати 0. Нехай діють означення

**const** maxn = 1000;

**type** T = integer;

Indx = 1 .. maxn; (17.1)

ArT = **array** [Indx] **of** T;

Подамо розв'язання задачі функцією

**function** srcseq ( **var** A : ArT; n : Indx; key : T) : integer;

**var** i : integer;

**begin**

i := 0;

**while** ( i <= n ) **do**

**if** A[i]<> key **then** i := i + 1 **else** **break**;

{ i = n + 1 або A[i] = key }

**if** i < n + 1 **then** srcseq := i

**else** srcseq := 0

**end**

З'ясуємо, яким чином час пошуку за цим алгоритмом залежить від кількості *n* елементів масиву. Узагальнимо присвоювання та операції над значеннями скалярних типів (порівняння, додавання, множення тощо) терміном ***елементарна* *дія***. Будемо вважати, що *на виконання кожної елементарної дії витрачається скінченний обмежений проміжок часу, незалежний від конкретних операндів*. За такого припущення час виконання програми (підпрограми) прямо пропорційний кількості елементарних дій у процесі виконання.

Очевидно, що кількість елементарних дій при виконанні функції srcseq прямо пропорційна кількості порівнянь A[i]<>key. В найгіршому випадку з ключем порівнюються всі елементи масиву. Звідси максимальна кількість дій та найбільший час виконання функції *прямо (лінійно) пропорційні* *n*, і найбільший час пошуку *t*1 є *лінійною функцією* від кількості елементів масиву. Описаний спосіб пошуку називається ***лінійним***. Лінійну залежність часу від кількості елементів, тобто довжини *n*, будемо позначати символом "O": *t*1 = O(*n*).

**1.2. Дихотомічний пошук**

Тепер опишемо так званий ***дихотомічний*** ***пошук*** у "нормальному словнику". Ми розкриваємо словник приблизно в його середині. Якщо слово повинно бути в словнику далі, то шукати треба лише в другій половині словника. Його середина стає для нас початком, і ми розкриваємо його на середині другої половини. Аналогічно, коли слово повинно бути в першій половині, ми залишаємо для пошуків лише її. Отже, кожне заглядання в словник поділяє "простір пошуку" на дві половини й зменшує його приблизно вдвічі. Звідси й назва, оскільки дихотомія – це поділ на дві половини. Такий пошук ще називають ***двійковим***, або ***бінарним***.

Нехай значення елементів масиву упорядковані за зростанням, тобто *A*[1] *A*[2] …  *A*[*n*] (кажуть, ***масив*** ***відсортований***). Опишемо дихотомічний пошук такою функцією:

**function** srcbin ( **var** A : ArT; n : Indx; key : T) : integer;

**var** i, hb, lb : integer;

**begin**

lb := 1; hb := n;

i := (lb + hb) **div** 2;

**while** (lb < hb) **do**

**begin**

**if** A[i] = key **then** **break** **else**

**if** A[i] > key

**then** hb := i - 1 { key може бути лише ліворуч від A[i] }

**else** lb := i + 1; { key може бути лише праворуч від A[i] }

i := (lb + hb) **div** 2

**end**;

{ lb >= hb або A[i] = key }

**if** (i=0) **or** (i=n+1) **then** srcbin := 0 **else**

**if** A[i] = key **then** srcbin := i **else** srcbin := 0

**end**

Виконання тіла циклу **while** вимагає сталого числа елементарних дій незалежно від значень змінних A, key, i, lb, hb. Звідси загальна кількість дій прямо пропорційна кількості повторень тіла циклу **while**. Але за кожного повторення різниця hb-lb зменшується принаймні у два рази. Спочатку hb-lb = *n* - 1, тому виконань тіла циклу не більше ніж log2*n* , звідки час виконання функції *t*2 = O(log2*n*). Ця пропорційність зумовлює ще одну назву описаного пошуку – ***логарифмічний***.

Чим більше *n*, тим більше відношення *n* до log2*n*. Наприклад, за *n*=10000 це більше 500. Коли треба відшукати значення один раз, можливо, комп'ютер упорається досить швидко і за лінійним алгоритмом. Але в реальних задачах доводиться шукати за ключем багаторазово, і різниця між лінійним і двійковим пошуком стає дуже відчутною.

Для двійкового пошуку необхідний відсортований масив, тому в наступному підрозділі почнемо розглядати способи сортування масивів.

Існує кілька інших способів швидкого пошуку в масивах. Їх докладне описання є в книзі [Кнут, т.3].

**Задача**

**1.**\* Вам треба дізнатися про день народження, тобто місяць і число, свого співбесідника (рік неважливий). Ви можете задавати питання типу: "День народження такого-то числа такого-то місяця?". Відповідь може бути "так", "раніше" або "пізніше". Скільки питань достатньо, щоб дізнатися про будь-який день народження?

**2. Бульбашкове сортування**

Розглянемо найпростіший (і найгірший з точки зору витрат часу) спосіб сортування масиву. Нехай *A*[1], *A*[2], ... , *A*[*n*] – масив із довільними значеннями елементів. Порівняємо *A*[1] і *A*[2]: якщо *A*[1]>*A*[2], то поміняємо їхні значення місцями. Потім порівняємо *A*[2] і *A*[3] та за необхідності обміняємо їхні значення. В результаті *A*[3] має найбільше значення серед *A*[1], *A*[2], *A*[3]. Продовжимо такі порівняння та обміни до кінця масиву:

*для всіх i від 1 до n-1 виконати*

*якщо A[i]>A[i+1], то*

*значення A[i] та A[i+1] обмінюються.*

Якщо значення елементів розглядати як розміри бульбашок, то ці порівняння та обміни схожі на те, як більші бульбашки відтісняють менших униз і спливають нагору. Тому цей метод називається ***бульбашковим*** ***сортуванням***. У результаті найбільша бульбашка стає верхньою, тобто останній елемент *A*[*n*] має найбільше значення. Наприклад, послідовність значень <3, 4, 2, 1> перетвориться на <3, 2, 1, 4>.

Далі почнемо все спочатку і перемістимо друге за величиною значення до передостаннього елемента *A*[*n*-1], перетворивши, наприклад, <3, 2, 1, 4> на <2, 1, 3, 4>. Потім третє за величиною значення перемістимо до *A*[*n*-2] тощо. Останній крок складається лише з порівняння *A*[1]<*A*[2] (та обміну значень за потреби).

За дії означень (17.1) опишемо бульбашкове сортування такою процедурою (bubble – це англійське "бульбашка"):

**procedure** Bubbles1 (**var** A: ArT; n: Indx);

**var** i, k: Indx;

**begin**

**for** k := n **downto** 2 **do**

{ k – права границя в черговому проході }

**for** i := 1 **to** k - 1 **do**

**if** A[i] > A[i+1]

**then** swap (A[i], A[i+1])

**end**

Тут і далі процедура swap задає обмін значень своїх параметрів. Як бачимо, разом виконується (*n*-1)+(*n*-2)+…+1= *n* (*n*-1)/2 порівнянь. Очевидно, що найбільше можливе число елементарних дій за цим способом прямо пропорційне кількості порівнянь. Тому час сортування масиву з *n* елементів у такий спосіб *прямо пропорційний n* (*n*-1).

**Задачі**

**2.**\* Якщо при черговому виконанні циклу із заголовком **for** i:=1 **to** k-1 **do** процедури Bubbles1 не було жодного обміну, то масив уже відсортовано. Тому подальші проходи масивом зайві. Переробити процедуру сортування так, щоб її виконання закінчувалося за відсортованого масиву.

**3.** ***Сортування* *вибором*** здійснюється так. Проглянемо елементи масиву від першого до останнього, визначимо елемент із найменшим значенням, та обміняємо це значення з першим. Потім так само виберемо найменше значення серед *A*[2]...*A*[*n*] і обміняємо його зі значенням *A*[2], потім виберемо наступне найменше та обміняємо з *A*[3] тощо. Написати процедуру сортування вибором.

**4. *Сортування простими вставками*** дозволяє створити відсортований масив із значень, що читаються, наприклад, із клавіатури. Перше значення записується на перше місце, тобто присвоюється першому елементу масиву. Друге значення порівнюється з першим і, якщо перше менше, то воно "витісняється" на друге місце. Інакше нове значення йде на друге місце. Потім третє порівнюється з другим та записується або на третє місце, або витісняє значення з другого місця на третє та порівнюється з тим, що на першому місці. Наприклад, за читання послідовності значень 3, 1, 2 створюються послідовності значень у масиві <3>, <1, 3>, <1, 2, 3>.

Узагалі, після читання *k*-1 елемента маємо відсортовану частину масиву *A*[1]*A*[2]...*A*[*k*-1]. Нове значення *v* порівнюємо зі значенням *A*[*k*-1]. Якщо *A*[*k*-1]>*v*, то значення *A*[*k*-1] зсуваємо на *k*-е місце. Після цього порівнюємо *v* зі значенням *A*[*k*-1]: якщо *A*[*k*-2]>*v*, то *A*[*k*-2] зсуваємо на (*k*-1)-е місце тощо. Коли за чергового порівняння *A*[*i*]<=*v*, то *v* записується на (*i*+1)-е місце. Якщо всі значення в масиві більші *v*, то вони зсуваються, а *v* записується на перше місце. Уточнити наведений алгоритм процедурою.

**3. Поняття складності алгоритму та задачі**

У цьому параграфі ми познайомимося з двома поняттями, які відіграють у програмуванні одну з ключових ролей. Цими поняттями є складність алгоритму та складність задачі. Почнемо з алгоритмів.

Нагадаємо, що алгоритм розв'язання масової задачі описує розв'язання будь-якого її екземпляра. Екземпляри багатьох задач можна охарактеризувати значенням деякого числового параметра. Наприклад, довжиною масиву чи кількістю чисел, які треба прочитати з клавіатури. Або натуральним числом, про яке ми хочемо дізнатися, чи просте воно. Найчастіше цим параметром є кількість скалярних значень, обробка яких задається алгоритмом. Кажуть, що ***екземпляр задачі має розмір N***, якщо задається даними, складеними з *N* скалярних значень (наприклад, масивом із *N* елементів).

Нехай *A* позначає алгоритм розв'язання деякої масової задачі. Позначимо через *F*(*A*, екземпляр) кількість елементарних дій у процесі розв'язання цього екземпляра задачі за алгоритмом *A*, а через *F*(*A*, *n*) – максимум кількості елементарних дій серед усіх екземплярів розміру *n*.

Наприклад, якщо при бульбашковому сортуванні масив спочатку відсортований навпаки, то слідом за кожним порівнянням відбувається обмін. А це ще три присвоювання. Якщо нехтувати допоміжними операціями із змінами індексів, то

*F*(*A*, *n*)=4 *n* (*n*-1)/2.

Кожному *n* = 1, 2, 3, … відповідає певне значення *F*(*A*, *n*), тобто ми маємо функціональну залежність між розмірами *n* та максимальними кількостями елементарних дій, виконуваних за алгоритмом *A*. Ця функція називається ***складністю******алгоритму A***. Алгоритми практично всіх реальних задач мають складність, монотонно неспадну за *n*.

Аналітичне задання функції *F*(*A*, *n*) для реальних алгоритмів, як правило, неможливе й не потрібне. Велике практичне значення має так званий ***порядок зростання F(A,n) за n***. Він задається за допомогою іншої функції з простим аналітичним виразом, яка є оцінкою для *F*(*A*, *n*).

Функція *G*(*n*) є ***оцінкою для функції F(n)***, або ***F(n) є функцією порядку G(n)***, коли існують такі додатні скінченні числа *C*1 і *C*2, що за деякого *m* при всіх *n* > *m*

*C*1 *G*(*n*)  *F*(*n*)  *C*2 *G*(*n*).

Така залежність між функціями позначається за допомогою знака "О":

*F*(*n*) = O(*G*(*n*)).

Для задання порядку зростання інколи користуються іншим означенням: функція *F*(*n*) називається ***функцією порядку G(n) за великих n***, якщо , де *C*>0, *C*< .

Функція *F*(*n*) називається ***функцією порядку, меншого від G(n)* *за великих n***, і це позначається *F*(*n*)=o(*G*(*n*)), якщо .

Для оцінки складності переважної більшості реальних алгоритмів достатньо логарифмічної, степеневої та показникової функцій, а також їх сум, добутків та підстановок. Усі вони монотонно зростають і задаються простими аналітичними виразами.

**Приклад** **1.** *n* (*n*-1) = O(*n*2), оскільки за *n* > 2 маємо

0.5 *n*2 < *n*\*(*n*-1) < *n*2.

Аналогічно неважко довести, що *n*3 + 100 *n*2 = O(*n*3) = o(*n*3.1) = o(2*n*), 100 log*n* + 10000 = O(log*n*) = O(lg*n*) = o(*n*).

**Приклад** **2.** Складність алгоритму бульбашкового сортування *tb*(*n*)=O(*n*2), алгоритму лінійного пошуку – *t*1(*n*)=O(*n*), бінарного пошуку – *t*2(*n*)=O(log*n*)=o(*n*).

Тепер означимо поняття складності задачі. Алгоритми пошуку в упорядкованому масиві свідчать, що одна й та сама *задача може мати алгоритми розв'язання з різною складністю*. Неформально, під складністю задачі розуміють найменшу зі складностей алгоритмів її розв'язання. Іншими словами, з***адача* *має складність порядку G(n)***, якщо існує алгоритм її розв'язання зі складністю O(*G*(*n*)) і не існує алгоритмів зі складністю o(*G*(*n*)).

Наприклад, складність задачі пошуку в упорядкованому масиві визначається складністю алгоритму двійкового пошуку, тому й оцінюється функцією log*n*. Задача сортування масиву має складність порядку *n* log*n*. Доводити ці факти ми не будемо – можна подивитися, наприклад, у книгу [АХУ]. Але в наступному параграфі ми наведемо алгоритми сортування з цією оцінкою складності.

**Задачі**

**5.** Оцінити складність задачі

а) \* про Ханойські вежі (підр.9.3); б) пошуку підмножини (підр.9.2).

**6.**\*Оцінити складність алгоритмів сортування вибором та простими вставками (задачі 17.3, 17.4).

**7.**\* **Задача про неспадну підпослідовність**. Задано *n*-елементний масив цілих, *n*<10000. Знайти:

а) максимальну довжину неспадних підпослідовностей значень масиву;

б) неспадну підпослідовність значень масиву максимальної довжини. Якщо таких кілька, то з них вибиpається та, що має найменшу суму елементів. Напpиклад, за масиву зі значеннями <2, 1, 5, 3> це буде <1, 3>.

Складність алгоритму повинна бути якомога меншою.

**4. Ефективні алгоритми сортування**

**4.1. Сортування злиттям**

В основі цього способу сортування лежить ***злиття*** ***двох*** ***упорядкованих*** ***ділянок*** ***масиву*** в одну впорядковану ділянку іншого масиву.

Злиття двох упорядкованих послідовностей можна порівняти з перебудовою двох колон солдатів, вишикуваних за зростом, в одну, де вони також розташовуються за зростом. Якщо цим процесом керує офіцер, то він порівнює зріст солдатів, перших у своїх колонах і вказує, якому з них треба ставати останнім у нову колону, а кому залишатися першим у своїй. Так він вчиняє, поки одна з колон не вичерпається – тоді решта іншої колони додається до нової.

Нехай у масиві *Y* з елемента *Y*[*m*] починається впорядкована ділянка довжиною *s*, а з елемента *Y*[*m*+*s*] – впорядкована ділянка довжини *r*. Наприклад,

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Y | … | 1 | 3 | … | 13 | 2 | 4 | … | 88 |
|   | … | m | m+1 |   | m+s-1 | m+s | m+s+1 | … | m+s+r |

Результатом злиття повинна бути ділянка довжини *r*+*s* у масиві *X*:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | … | 1 | 2 | 3 | 4 | … | 13 | … | 88 |
|   | … | m | m+1 | m+2 | m+3 |   | … |   | m+s+r |

За дії означень (17.1) таке злиття двох ділянок у масиві *Y* у ділянку масиву *X* задає процедура

**procedure** mrg( **var** Y : ArT; m, s, r : Indx; **var** X : ArT);

**var** mr, k : Indx; i, j : Extind;

**begin**

mr := m + s; { mr – початок правої частини }

i := m; j := mr; { i та j пробігають ліву й праву ділянки масиву Y}

**for** k := m **to** m + s + r - 1 **do**{заповнення X[m], … , X[m+s+r-1]}

**if** i > m + s - 1 **then**

**begin** X[k] := Y[j]; j := j + 1 **end else**

**if** j > mr + r - 1 **then**

**begin** X[k] := Y[i]; i := i + 1 **end else**

**if** Y[i] < Y[j] **then**

**begin** X[k] := Y[i]; i := i + 1 **end else**

**begin** X[k] := Y[j]; j := j + 1 **end**

**end**

Тепер розглянемо ***сортування*** ***масиву*** *A* ***злиттям***. На першому кроці елементи *A*[1], … , *A*[*n*] копіюються в допоміжний масив *B*[1], … , *B*[*n*]. Його елементи розглядаються парами *B*[1] і *B*[2], *B*[3] і *B*[4] тощо як упорядковані послідовності довжиною lp = 1 і зливаються за допомогою процедури mrg в масив *A*. Тепер там є впорядковані ділянки довжиною 2. За непарного *n* останній елемент *A*[*n*] залишається без змін як послідовність довжиною 1.

На наступному кроці після копіювання в масив *B* зливаються пари упорядкованих ділянок *B*[1]*B*[2] і *B*[3]*B*[4], *B*[5]*B*[6] і *B*[7]*B*[8] тощо. З'являються впорядковані ділянки довжиною 4. Нехай *t*=*n***mod**4 – довжина залишку масиву після останньої повної четвірки елементів. При *t*=1 або *t*=2 останні *t* елементів утворюють упорядковану ділянку після попереднього кроку. При *t*=3 зливаються упорядкована пара *B*[*n*-1]*B*[*n*-2] та ділянка *B*[*n*] у ділянку довжиною *t*.

Кроки повторюються з подвоєнням довжин упорядкованих ділянок lp, поки lp < *n*.

Розглянемо сортування злиттям масиву <11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1> довжини *n*=11. Упорядковані послідовності в ньому вказуються в дужках <>, а пари таких, що зливаються, відокремлені ";":

< <11><10> ; <9><8> ; <7><6> ; <5><4> ; <3><2> ; <1> >, lp=1

< <10, 11><8, 9> ; <6, 7><4, 5> ; <2, 3><1> >, lp=2

< <8, 9, 10, 11><4, 5, 6, 7>;<1, 2, 3> >, lp=4

< <4, 5, 6, 7, 8, 9, 10, 11><1, 2, 3> >, lp=8

<1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11>, lp=16, lp n.

Як бачимо, нам знадобилося 4 кроки злиття для того, щоб одержати впорядований масив.

За дії означень (17.1) такий спосіб сортування описується процедурою Mrgsort:

**procedure** Mrgsort (**var** A : ArT; n : Indx);

**var** B : ArT; lp, npp, cpp, bpp, tl : integer;

**begin**

lp := 1;

**while** lp < n **do**

**begin**

B := A; { копіювання }

npp := n **div** (2\*lp); { кількість пар ділянок }

tl := n **mod** (2\*lp); { довжина залишку }

**for** cpp := 1 **to** npp **do** { cpp – номер пари }

**begin**

bpp := (cpp - 1)\*2\*lp + 1;

{ індекс першого елемента лівої ділянки пари}

mrg ( B, bpp, lp, lp, A );

**end**;

{ обробка залишку }

**if** tl > lp **then**

mrg ( B, n - tl + 1, lp, tl - lp, A );

{ за tl <= lp залишок був упорядкований }

{ на попередньому кроці }

lp := lp\*2

**end**

{ lp >= *n* : масив упорядковано на останньому кроці }

**end**

Очевидно, що після *k*-го кроку впорядковані ділянки мають довжину lp=2*k*. Якщо 2*k*=*n*, то масив упорядковано. Якщо 2*k*>*n*, але 2*k*-1<*n*, то при виконанні останнього кроку мають місце співвідношення

lp = 2*k*-1 = 2*k* / 2 > *n*/2, npp = 0, tl = *n* > lp,

і злиття ділянки довжиною lp та залишку довжиною *n*-lp дає впорядкований масив.

Оцінимо складність наведеного алгоритму. При кожному виконанні тіла циклу **while** значення всіх елементів масиву копіюються в допоміжний масив і назад по одному разу, тобто виконується O(*n*) елементарних дій. Після останнього *k*-го кроку 2*k*<2*n*, тобто *k*<1+log*n*, і це означає, що тіло циклу **while** виконується 1+ log2*n* =O(log*n*) разів. Отже, складність алгоритму оцінюється як O(*n*log*n*).

Запишемо в таблицю значення виразів *n*, *n*(*n*-1)/2, *n*(1+ log2*n* ) та округлене відношення *r* двох останніх:

|  |  |  |  |
| --- | --- | --- | --- |
| ***n*** | ***n*(*n*-1)/2** | ***n*(1+ log2*n* )** | **r** |
| 10 | 45 | 40 | 1 |
| 100 | 4950 | 700 | 7 |
| 1000 | 499500 | 10000 | 50 |
| 10000 | 49995000 | 140000 | 350 |

Як бачимо, кожне зростання *n* у 10 разів веде до зростання *n*(*n*-1)/2 та *n*(1+ log2*n* ) приблизно в 100 й 14 разів відповідно, і за *n*=10000 сортування злиттям виконується в сотні разів скоріше, ніж бульбашкове!

Зауважимо, що в наведеному алгоритмі сортування злиттям копіювання масиву в допоміжний указано лише для того, щоб ідею алгоритму було простіше сприйняти. Цей алгоритм нескладно переробити так, щоб замість копіювання в додатковий масив відбувалося злиття в нього упорядкованих ділянок. Отже, на кроках з номерами 1, 3, 5, … має відбуватися злиття в допоміжний масив, а на кроках 2, 4, 6, … – злиття в протилежному напрямку. Переробка алгоритму залишається вправою.

Сортування злиттям можна задати рекурсивно: масив поділяється на дві приблизно рівні частини, які після сортування (тим самим способом – ось рекурсія!) зливаються. Коли ж довжина частини масиву зменшується до 1, відбувається просто повернення з рекурсії. Цей алгоритм уточнюється наступною процедурою Mrgrec. На відміну від процедури Merges, вона має два параметри-масиви (той, що сортується, та допоміжний), а також два числові параметри (початок і кінець частини масиву, яка сортується). Крім того, спочатку відбувається злиття ділянок основного масиву в допоміжний, а потім копіювання в основний:

**procedure** Mrgrec(**var** A, B : ArT; l, r : integer);

**var** m, k : integer;

**begin**

**if** l>=r **then** exit;

m:=(l+r) **div** 2;

Mrgrec(A, B, l, m); Mrgrec(A, B, m+1,r);

mrg(A, l, m-l+1, r-m, B);

**for** k:=l **to** r **do** A[k]:=B[k];

**end**;

Ця процедура набагато коротше нерекурсивної процедури Merges, але виконання її довше. Власне сортування починається лише після повернення з викликів, у яких l=r, а це практично "середина дистанції".

Завершуючи описання сортування злиттям, скажемо, що цей алгоритм є першим із ефективних алгоритмів сортування. У 1945 році його винайшов Джон фон Нейман, один із піонерів програмування.

Серйозним недоліком цього алгоритму є необхідність додаткового масиву такої ж довжини, як і в основного. За великих довжин можлива ситуація, коли на один масив пам'яті вистачає, а на два – ні. Розглянемо два алгоритми, позбавлені цього недоліку.

**4.2. Піраміда, вона ж дерево**

Уявіть собі, що ми розташували елементи масиву рядками, щоразу подвоюючи їх кількість: у першому рядку – перший елемент, у другому – елементи з індексами 2, 3, у наступному – 4, 5, 6, 7, далі 8, 9, 10, 11, 12, 13, 14, 15 тощо. Останній рядок може виявитися неповним. Наприклад, за кількості елементів n=12 маємо таку піраміду індексів:

1

2 3

4 5 6 7

8 9 10 11 12

Елементи цієї піраміди будемо називати ***вузлами***.

Між вузлами проведемо стрілки: від 1 – до 2 та 3, від 2 – до 4 та 5, від 3 – до 6 та 7 тощо, тобто від кожного вузла *k* до 2*k* та 2*k*+1, де k<n **div** 2 (рис.17.1). За парного *n* від вузла (*n* **div** 2) проводиться стрілка лише до вузла *n*. Вузли 2*k* та 2*k*+1 називаються ***синами*** вузла *k*, який називається їхнім ***батьком***. Вузол 1 називається ***вершиною*** ***піраміди***. Кожний вузол також можна вважати вершиною піраміди, складеної ним, його синами, їх синами тощо. Наприклад, вузол 2 є вершиною піраміди, складеної з 2, 4, 5, 8, 9, 10, 11, вузол 3 – піраміди з 3, 6, 7, 12.

Піраміду можна розглядати як дерево, гілки якого – стрілки від батьків до синів. Вершина піраміди називається ***коренем*** ***дерева***.

Припустимо тепер, що значення елементів масиву розташовано так, що *значення кожного елемента-батька не менше значень його синів*:

*A*[1]  *A*[2] та *A*[1]  *A*[3], *A*[2]  *A*[4] та *A*[2]  *A*[5] тощо.

Отже, за кожного *k*=1, 2, … , *n* **div** 2

*A*[*k*]  *A*[2\**k*] та *A*[*k*]  *A*[2\**k*+1] (17.2)

(за парного *n* елемент *A*[*n* **div** 2] має лише одного сина *A*[*n*]). Наприклад,

30

12 30

12 5 29 2

11 10 3 2 28 27

Цій піраміді відповідає послідовне розташування <30, 12, 30, 12, 5, 29, 2, 11, 10, 3, 2, 28, 27>.

Очевидно, що кожний елемент має значення, найбільше в тій піраміді, де він є вершиною. Наприклад, *A*[2] має значення, найбільше серед елементів із індексами 2, 4, 5, 8, 9, 10, 11. Зокрема, *A*[1] має значення, найбільше серед усіх елементів.

Якщо поміняти місцями значення *A*[1] і *A*[*n*], то елемент *A*[*n*] матиме найбільше значення. Про нього "можна забути" та зосередитися на сортуванні *A*[1], *A*[2], ... , *A*[*n*-1]. Зрозуміло, що умова *A*[1] *A*[2], *A*[1] *A*[3] після обміну може виявитися порушеною. Для її відновлення треба обміняти місцями значення *A*[1] та того з *A*[2], *A*[3], чиє значення максимальне. Нехай це буде *A*[3]. В останньому прикладі після обміну значень *A*[1] і *A*[12] на вершині піраміди значення 27, і 27<30, тобто *A*[1]<*A*[3]. Після обміну їх значень умова (17.2) може порушитися в піраміді з вершиною *A*[3]. Відновимо цю умову так само, як і для вершини *A*[1], опустившися при цьому до вузла *A*[6] чи *A*[7]. І так далі.

Після відновлення умови (17.2) можна буде обміняти значення першого елемента з передостаннім, "забути" про нього, знову відновити умову, знову загнати перше значення в кінець тощо.

Нехай процедура bld(A, n) задає початкову перестановку значень масиву таким чином, що виконується умова (17.2), а процедура reorg(A, i, k) – її відновлення у частині масиву *A*[*i*], … , *A*[*k*]. Тоді за дії означень (17.1) сортування задається процедурою Treesort:

**procedure** Treesort ( **var** a : ArT; n : Indx );

**var** j : Indx;

**begin**

bld (A, n);

**for** j := n **downto** 3 **do**

**begin**

swap (A[1], A[j]); reorg (A, 1, j-1)

**end**

**end**

Властивість (17.2) відновлюється в частині масиву *A*[1], … , *A*[*n*-1] таким чином. Обмінюються значення *A*[1] та того з *A*[2] або *A*[3] (позначимо його *A*[*k*]), чиє значення максимальне. У результаті властивість (17.2) може порушитися в частині масиву *A*[*k*], … , *A*[*n*-1]. У ній відновлення відбувається так само, як і серед *A*[1], … , *A*[*n*-1].

Опишемо відновлення частини масиву *A*[*i*], … , *A*[*j*] у загальному випадку значень *i*, *j*. Нехай у частині масиву *A*[*i*], … , *A*[*j*], де 2\**i*+1 *j*, треба відновити властивість (17.2), можливо, порушену з початку: *A*[*i*]<max{*A*[2\**i*], *A*[2\**i*+1]}. За умови *A*[2\**i*]>*A*[2\**i*+1] покладемо *k*=2\**i*, у противному разі – *k*=2\**i*+1. Обміняємо значення *A*[*i*] та *A*[*k*]; після цього, якщо необхідно, властивість (17.2) відновлюється в частині масиву *A*[*k*], … , *A*[*j*].

Коли 2\**i*=*j*, то лише порівнюються й, можливо, обмінюються значеннями *A*[*i*] та *A*[*j*], причому *k*=*j*.

Коли 2\**i*>*j*, то властивість (17.2) не може бути порушеною в частині масиву *A*[*i*], … , *A*[*j*].

За дії означень (17.1) відновлення задається рекурсивною процедурою reorg:

**procedure** reorg ( **var** A : ArT; i, j : Indx );

**var** k : Indx;

**begin**

**if** 2\*i = j **then**

**if** A[i] < A[j] **then** swap ( A[i], A[j] );

**if** 2\*i < j **then**

**begin**

**if** A[2\*i] > A[2\*i+1] **then** k := 2\*i

**else** k := 2\*i + 1;

**if** A[i] < A[k] **then**

**begin**

swap ( A[i], A[k] ); reorg ( A, k, j )

**end**

**end**

**end**

Після виконання виклику reorg( A, i, j ) за будь-якого *i*<*j***div**2 елемент *A*[*i*] має максимальне значення серед *A*[*i*], *A*[2\**i*], *A*[2\**i*+1]. Цим можна скористатися для побудови початкового масиву з властивістю (17.2): спочатку "відновлюється" частина масиву *A*[*n***div**2], … , *A*[*n*], потім частина *A*[*n***div**2-1], … , *A*[*n*] тощо. Звідси процедура bld:

**procedure** bld (**var** A : ArT; n : Indx );

**var** i : Indx;

**begin**

**for** i := n **div** 2 **downto** 1 **do** reorg ( A, i, n )

**end**

Оцінимо складність алгоритму. За наведеними процедурами очевидно, що складність алгоритму прямо пропорційна загальній кількості викликів процедури reorg. При виконанні процедури bld процедура reorg викликається *n***div**2 разів, а при виконанні циклу **for** процедури Treesort – ще *n*-2 рази. Отже, загальна кількість викликів процедури reorg з інших процедур є O(*n*). Кількість елементарних дій при виконанні операторів її тіла оцінюється сталою, тобто O(1). У кожному рекурсивному виклику процедури reorg значення її другого параметра не менше ніж подвоюється, тому кожний виклик reorg з інших процедур породжує не більше O(log*n*) рекурсивних викликів. Звідси випливає, що загальна кількість елементарних дій є O(*n*log*n*).

**4.3.** **Швидке сортування**

Ідея швидкого сортування така. Певним чином вибирається значення *v*. Значення елементів масиву *A* обмінюються так, що елементи на його початку мають менші від *v* значення, а від деякого місця *k* значення елементів більші або рівні *v*. Це значення називається ***відокремлюючим***; воно знаходиться на своєму місці, якщо *A*[*k*]=*v*. Після розміщення відокремлюючого значення на потрібному місці достатньо окремо відсортувати частини масиву від *A*[1] до *A*[*k*-1] та від *A*[*k*+1] до кінця.

Нехай діють означення (17.1). Сортування частини масиву *A*[lo] … *A*[hi] з відокремлюючим значенням *A*[lo] задається такою процедурою:

**procedure** quicksort ( A : ArT; lo, hi : Indx ) ;

**var** k, j : Indx; v : T;

**label** 1;

**begin**

**if** lo >= hi **then** **goto** 1;

**if** (lo = hi - 1) and (A[lo] > A[hi]) **then**

**begin**

swap ( A[lo], A[hi] );

**goto** 1

**end**;

k := lo + 1; j := hi;

v := A[lo]; {?!}

**while** ( k < j ) **do**

**if** A[k] < v **then** k := k + 1

**else**

**begin**

**if** A[j] < v **then** swap ( A[k], A[j] );

j := j - 1

**end**;

{ k = j }

**if** A[k] >= v **then** k := k - 1;

{ A[k] є останнім від початку елементом, }

{ значення якого менше v }

swap ( A[lo], A[k] ); { A[k] = v }

quicksort ( A , lo, k - 1 );

quicksort ( A , k + 1, hi );

1: **end**

Якщо за виконання кожного виклику після циклу **while** значення змінної *k* приблизно рівне (lo+hi)/2, то складність швидкого сортування масиву з *n* елементів можна оцінити як O(*n*log*n*). Середня кількість порівнянь елементів усіх можливих числових послідовностей довжини *n* також має оцінку O(*n*log*n*); доведення є, наприклад, у книзі [АХУ]. Емпіричні дослідження свідчать, що швидке сортування вимагає в середньому елементарних дій приблизно вдвічі менше, ніж сортування деревом, і вчетверо менше, ніж сортування злиттям. Але якщо початковий масив близький до відсортованого, то його сортування за наведеним алгоритмом вимагає вже O(*n*2) порівнянь. У такому випадку відокремлюючим елементом не можна вибирати значення *A*[lo].

Існує багато способів вибору іншого відокремлюючого значення. Наприклад, можна вибрати значення елемента з випадковим номером серед номерів lo, … , hi, або середнє із значень *A*[lo], *A*[hi], та *A*[(lo+hi)**div**2]. У такому разі перед оператором процедури

v := A[lo]; {?!}

треба задати обмін значень *A*[lo] та відповідного елемента, чиє значення вибирається відокремлюючим.

**Задачі**

**8.** Переробити процедуру Merges так, щоб на кроці з непарним номером перший масив без копіювання зливався в другий, а на кроці з парним номером, навпаки, другий без копіювання зливався в перший.

**9.**\* Написати нерекурсивний варіант сортування деревом (підр.17.4.2).

**10.** За масивом із *N* рядків визначається додатковий ***індексовий масив***: попаpно pізні значення його елементів належать діапазону 1..*N* і є індексами в масиві рядків. Написати процедуру

а) сортування індексового масиву таким чином, щоб рядки, на які посилаються його послідовні елементи, були лексикогpафічно впоpядковані;

б) друкування рядків масиву за зpостанням у лексикографічному порядку (кожний рядок друкується один pаз незалежно від кількості повтоpень у масиві).

**11.** Написати процедуру обчислення *N* (*N*  1000) найменших значень числової послідовності довільної довжини за умови, що

а)\* незалежно від кількості повтоpень усі числа враховуються один pаз кожне;

б) число враховується стільки pазів, скільки воно зустрілося в послідовності (але не більше ніж *N*);

в) числа враховуються один pаз кожне, але ведеться додатковий облік кількостей їх повтоpень.

**12.** Запрограмувати злиття

а) трьох б) п'яти в) тисячі

упорядкованих послідовностей в одну.

**13.**\* Запрограмувати сортування елементів зв'язаного списку. Складність алгоритму повинна бути O(*n*log*n*).

**14.** Запрограмувати сортування послідовностей із

а) чотирьох елементів так, щоб виконувалося не більше п'яти порівнянь;

б) п'яти елементів так, щоб виконувалося не більше восьми порівнянь;

в) п'яти елементів так, щоб виконувалося не більше семи порівнянь.

**5. Відсортуй, і багато чого побачиш**

Вам треба огородити сад із деревами, витративши на огорожу якомога менше матеріалу. Вважатимемо, що дерев не менше трьох, і вони задаються координатами на площині. Розв'язання полягає в побудові так званої ***опуклої оболонки*** навколо множини точок із заданими координатами. На площині вона являє собою опуклий багатокутник, що містить усі задані точки і не містить інших опуклих багатокутників, які також "накривають" усі точки.

Якщо намалювати план саду, то побудова оболонки стане очевидною. Треба відшукати якусь "крайню" початкову точку і рухатися від неї, малюючи опуклий багатокутник. Як результат, ми одержимо послідовність вершин – заданих точок, на які "натягнуто" багатокутник. Точки, що потрапили на сторону багатокутника, але не є вершинами, можна відкинути.

Наприклад, за множини точок із координатами (1,1), (2,2), (3,3), (5,-1), (2,-1), (3,0) ми одержимо багатокутник із вершинами (1,1), (3,3), (5,-1), (2,-1). Точка (3,0) опинилася всередині його, точка (2,2) – на стороні.

Комп'ютер не бачить плану саду, тому доведеться пояснити йому докладніше, як одержати потрібну оболонку. Один із способів оснований на тім, що спочатку точки сортуються за зростанням, наприклад, координати *Y*. Тепер це не просто множина, а певним чином упорядкована послідовність. Врахування цього дозволяє створити простий та прозорий алгоритм побудови оболонки.

За початкову точку можна взяти точку із найменшою координатою *Y*. Таких точок може бути кілька – виберемо з них ту, координата *X* якої найменша. Далі ми будуємо дві послідовності точок, починаючи обхід множини як зліва, так і справа. Назвемо їх лівою та правою. Друга точка додається до обох. Далі після обробки кожної точки будується оболонка тієї множини точок, які вже розглянуто.

Нехай *A* та *B* – остання й передостання точки послідовності, побудованої при обході зліва, *C* – нова точка. Якщо при переході з відрізка *BA* на відрізок *AC* ми робимо лівий поворот або рухаємося по прямій, то точку *A* можна вилучити з послідовності. Так само треба вилучити й точку *B*, якщо перед нею в послідовності є точка *Z*, така що, *ZBC* – лівий поворот або відрізок прямої. І так далі. Нарешті, або дві останні точки послідовності разом із новою утворюють правий поворот, або всі, крім першої, вилучено.

Визначення, чи утворюють точки *BAC* лівий поворіт, здійснюється за знаком векторного добутку , як у прикладі 7.5 з підрозділу 7.4. Є особливий випадок обробки точки – коли дві останні точки послідовності розташовані горизонтально. Якщо нова точка належить тій же горизонталі, то векторний добуток рівний 0. Нова точка "витісняє" останню точку *A*, коли відрізок *BA* продовжується нею або вона розташована ліворуч від *A* .

Після обробки послідовності нова точка додається до неї. Виключенням є випадок, коли нова точка лежить на відрізку між останніми точками лівої та правої послідовностей обходу. Перевірити це дуже просто, оскільки з упорядкованості точок за координатою *Y* випливає, що такий відрізок може бути лише горизонтальним.

Симетричним чином точка додається (або не додається) до правої послідовності.

Упорядкованість точок по вертикалі забезпечує, що при обробці нової точки не виникає одночасно лівого повороту до неї з кінця лівої послідовності та правого – з правої. Крім того, якщо точка була кінцевою в обох послідовностях, то після обробки наступної точки вона буде вилучена принаймні з однієї з них. Упорядкованість також гарантує, що необроблені точки лежать не нижче від тих, за якими побудовано поточну оболонку. Врахування усіх цих властивостей суттєво спрощує алгоритм.

Отже, після обробки останньої точки ми маємо ліву та праву послідовності. Результат ми одержимо, друкуючи, наприклад, ліву послідовність з її початку, а потім праву з її кінця. Це відповідає обходу нашого саду ззовні за годинниковою стрілкою.

Перейдемо до реалізації алгоритму. Координати точок подамо масивом структур з полями X і Y. При його сортуванні ми не будемо переставляти місцями значення його елементів, тобто структури. Створимо допоміжний масив індексів і відсортуємо його так, щоб перше значення в ньому вказувало на структуру з найменшим значенням Y, друге – на структуру з другим за порядком значенням тощо. Наприклад, якщо послідовні структури мають координати Y

10 -1 33 15 0,

то значеннями послідовних елементів допоміжного масиву є номери структур у їх упорядкуванні:

2 5 1 4 3.

У сортуванні масиву структур A за їх координатою Y з допоміжним індексовим масивом замість порівнянь полів та обмінів значень елементів A[i] та A[k] відбуваються порівняння елементів A[B[i]].Y і A[B[k]].Y та обміни значень B[i] і B[k].

Після сортування відшукаємо початкову точку – з найменшою координатою *X* серед тих, що мають найменшу координату *Y*.

Ліву та праву послідовності зберігатимемо в додатковому масиві індексів. Ліва заповнює його з початку, права – з кінця. Нехай всього точок *n*. Посилання на початкову точку запам'ятаємо двічі в крайніх елементах масиву. З точок, що залишилися, лише остання може з'явитися в обох послідовностях одночасно. Тому для них достатньо *n* елементівмасиву. Отже, всього нам потрібно *n*+2 елементи.

Наведемо лише необхідні означення. Доповнити їх до програми залишається вправою.

**const** mx=5000; {максимальна кількість точок}

**type** int=integer; pnt=**record** x, y : int **end**;

**var** a : array[1 .. mx] **of** pnt; {масив точок}

b, {індексовий масив}

c : array[0..mx+1]**of** int; {масив для лівої та правої послід-тей}

n : int; {кількість точок}

ll, rr : int;{індекси останніх елементів посл-тей у масиві c}

…

**procedure** init;

**var** k : int; km, xm, ym : int; …

**begin**

*Читання n точок у масив A;*

*Сортування за неспаданням координати Y – результатом є масив B, такий, що A[B[1]].Y<= … <= A[B[n]].Y*;

*Вибір початкової точки*:

k:=2; km:=1; ym:=a[b[1]].y; xm:=a[b[1]].x;

**while** (k<=n) **and** (a[b[k]].y=ym)**do**

**begin**

**if** a[b[k]].x < xm **then**

**begin** km:=k; xm:=a[b[k]].x **end**;

k:=k+1

**end**;

swap(b[1], b[km]);

**end**;

**function** left ( tp : pnt ) : boolean;

**var** x1, y1, x2, y2, x3, y3 : real;

**begin**

y1:=a[c[ll-1]].y; y2:=a[c[ll]].y; y3:=tp.y;

x1:=a[c[ll-1]].x; x2:=a[c[ll]].x; x3:=tp.x;

**if** (y3=y2) **and** (x3=x2) **then**

left:=**false else**

**if** (y2=y1) **and** (y3=y2) **then**

left:=( (x2-x1)\*(x3-x2)>0 ) **or** ( x3-x2<0 )

**else**

left:=( (x2-x1)\*(y3-y1)-(x3-x1)\*(y2-y1)>=0 )

**end**;

**function** right ( tp : pnt ) : boolean;

**var** x1, y1, x2, y2, x3, y3 : real;

**begin**

y1:=a[c[rr+1]].y; y2:=a[c[rr]].y; y3:=tp.y;

x1:=a[c[rr+1]].x; x2:=a[c[rr]].x; x3:=tp.x;

**if** (y3=y2) **and** (x3=x2) **then**

right:=**false else**

**if** (y2=y1) **and** (y3=y2) **then**

right:=( (x2-x1)\*(x3-x2)>0 ) or ( x3-x2>0 )

**else** right:=( (x2-x1)\*(y3-y1)-(x3-x1)\*(y2-y1)<=0 )

**end**;

**procedure** run;

**var** k : int; tp : pnt;

**begin**

c[0]:=b[1]; c[n+1]:=b[1]; c[1]:=b[2]; c[n]:=b[2];

ll:=1; rr:=n;

**if** n=3 **then begin** c[n]:=b[3]; **exit** **end**;

**for** k:=3 **to** n **do**

**begin**

tp := a[b[k]];

**while** (ll>=1) **and** left(tp) **do** ll:=ll-1;

**if** (tp.y > a[c[ll]].y) **or** (tp.y > a[c[rr]].y)**or**

**not**( (tp.x>a[c[ll]].x)**and**(tp.x<a[c[rr]].x) )

**then** **begin** ll:=ll+1; c[ll]:=b[k] **end**;

**while** (rr<=n) **and** right(tp) **do** rr:=rr+1;

**if** (tp.y > a[c[ll]].y) **or** (tp.y > a[c[rr]].y)**or**

**not**( (tp.x>a[c[ll]].x)**and**(tp.x<a[c[rr]].x) )

**then** **begin** rr:=rr-1; c[rr]:=b[k] **end**;

**end**;

**end**;

**procedure** **done**;

**var** k : int;

**begin**

**for** k:=0 **to** ll **do** writeln(a[c[k]].x, ' ', a[c[k]].y);

**if** **not**((a[c[ll]].x=a[c[rr]].x)**and**(a[c[ll]].y=a[c[rr]].y))

**then** writeln(a[c[rr]].x, ' ', a[c[rr]].y);

**for** k:=rr+1 **to** n **do**

writeln(a[c[k]].x, ' ', a[c[k]].y);

**end**;

**BEGIN**

init; run; done

**END**.

Проаналізуємо складність наведеного алгоритму. Для сортування масиву потрібно O(*n*log*n*) операцій. В процесі побудови оболонки кожна точка додається до послідовностей та вилучається з них не більше, ніж по одному разу. Тому складність власне побудови оболонки лінійна, і весь алгоритм має складність O(*n*log*n*).

**Задачі**

**15.** Написати підпрограму підрахунку кількості різних значень елементів числового масиву. Складність підпрограми повинна бути O(*n*log*n*), де *n* – кількість елементів масиву.

**16.** Написати підпрограму складності O(*n*log*n*), що задає обчислення

а)перетину; б) симетричної різниці

двох *n*-елементних числових множин, поданих масивами.

**17.** Обчислити кількість трикутників, які можна скласти з *N* відрізків попарно різних додатних довжин, *N* < 2000. Складність програми не повинна перевищувати O(*N*2).

**18.** За *N-*елементним масивом *A* та перестановкою *p*(1), *p*(2),… , *p*(*N*) чисел 1, 2, … , *N* побудувати масив *A*[*p*(1)], *A*[*p*(2)],… , *A*[*p*(*N*)]. Допоміжний масив не використовувати.

**19.** Розглянемо прямокутний масив. Розташуємо елементи кожного рядка за зростанням. Потім розташуємо за зростанням елементи кожного стовпця. Довести, що кожний рядок залишиться впорядкованим.

**20.** (**Задача про два станки**) Є *n* деталей, кожна з яких проходить обробку спочатку на першому станку, потім на другому. Кожний станок здатний обробляти одночасно одну деталь і готовий до обробки наступної деталі одразу після обробки попередньої. Відомий час обробки кожної деталі на кожному із станків. Упорядкувати деталі таким чином, щоб час від початку обробки першої деталі до закінчення обробки останьої був якнайменшим. Наприклад, якщо перша деталь обробляється на станках 20 і 10 одиниць часу, а друга – 10 і 20, то за порядку деталей (перша, друга) одержимо час 50, а за порядку (друга, перша) – 40.