NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 3
SẮP XẾP DỮ LIỆU VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 3
SẮP XẾP DỮ LIỆU VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

LỜI NÓI ĐẦU

Những năm trước khi lập trình VieTeX tôi toàn dùng C/C++ thu thập tài liệu nhiều nhưng không có thời gian để viết lại. Nay muốn viết lại thì sức khỏe không ổn định. Tôi đã cố gắng gom lại thành các tập lập trình theo chủ đề. Nội dung mỗi thuật toán bắt đầu từ lý thuyết đến lập trình bằng C/C++.

Cuốn sách viết ra không dành riêng cho các bạn học tin học, mà các bạ học toán, thầy cô giáo, các bạn thích tìm hiểu về thuật toán. Cũng như tôi bắt đầu có biết gì về lập trình đâu, tự học và chăm chỉ là thành công thôi. Tôi dùng trình biên dịch Dev-C++: https://www.bloodshed.net/
Hiện nay Dev-C++ cải tiến rất nhiều và chạy tốt với môi trường unicode. Những ví dụ trong tài liệu các bạn chép thẳng vào soạn thảo và biên dịch không cần cấu hình trình biên dịch.

Tôi đã làm các quyển sách:

1. Thuật toán và số học.
2. Thuật toán và dữ liệu.
3. Thuật toán xắp xếp
4. Thuật toán tìm kiếm
5. Thuật toán đồ thị,
6. Thuật toán quay lui
7. Thuật toán chia để trị
8. Thuật toán động
9. Thuật toán tham
10. Thuật toán nén
11. Một số đề thi Olympic Tin học.

Cuốn sách dành cho học sinh phổ thông yêu toán, học sinh khá giỏi môn toán, các thầy cô giáo, sinh viên đại học ngành toán, ngành tin học và những người yêu thích Toán - Tin. Trong biên soạn không thể tránh khỏi sai sót và nhầm lẫn mong bạn đọc cho ý kiến.

Hà Nội, ngày 25 tháng 2 năm 2022

Nguyễn Hữu Điển

NHỮNG KÝ HIỆU

Trong cuốn sách này ta dùng những kí hiệu với các ý nghĩa xác định trong bảng dưới đây:

$\mathbb{N}$	tập hợp số tự nhiên
$\mathbb{N}^*$	tập hợp số tự nhiên khác 0
$\mathbb{Z}$	tập hợp số nguyên
$\mathbb{Q}$	tập hợp số hữu tỉ
$\mathbb{R}$	tập hợp số thực
$\mathbb{C}$	tập hợp số phức
$\equiv$	dấu đồng dư
∞	dương vô cùng (tương đương với $+\infty$)
$-\infty$	âm vô cùng
$\emptyset$	tập hợp rỗng
C_m^k	tổ hợp chập k của m phần tử
$\vdots$	phép chia hết
$\not\vdots$	không chia hết
$UCLN$	ước số chung lớn nhất
$BCNN$	bội số chung nhỏ nhất
deg	bậc của đa thức
IMO	International Mathematics Olympiad
APMO	Asian Pacific Mathematics Olympiad

NỘI DUNG

DANH SÁCH CÁC HÌNH

DANH SÁCH CÁC BẢNG

Danh sách chương trình

CHƯƠNG 3
SẮP XẾP DỮ LIỆU VÀ THUẬT TOÁN

"... có nhiều phương pháp tốt nhất, tùy thuộc vào cái gì sẽ được sắp xếp, trên máy gì, cho mục đích gì."

D.E.Knuth

3.1. Giới thiệu

Thông thường, khi làm việc với dữ liệu lớn cùng loại, cần phải đưa ra một sắc lệnh để tạo điều kiện thuận lợi cho việc xử lý chúng. Như chúng ta sẽ thấy trong Chương 4, việc sắp xếp các phần tử có thể cho chúng ta một thuật toán tìm kiếm hiệu quả hơn nhiều so với khi dữ liệu không được sắp xếp.

Người ta chấp nhận rằng quá trình sắp xếp lại (hoán vị theo một cách thích hợp) các phần tử của một tập hợp các đối tượng theo một thứ tự nhất định được gọi là *sắp xếp*. Sắp xếp là một hoạt động chính với lĩnh vực ứng dụng rộng rãi: từ điển, danh bạ điện thoại, mục lục tài liệu tham khảo và nói chung ở mọi nơi cần tìm kiếm nhanh chóng và tìm thấy nhiều đối tượng khác nhau. Sắp xếp là một phần không thể thiếu trong cuộc sống hàng ngày của chúng ta - đằng sau mọi sự sắp xếp, từ bàn làm việc chúng ta ngồi, đến tủ quần áo, túi xách, v.v. có một số kiểu sắp xếp.

Sắp xếp là một khái niệm cực kỳ rộng và, tùy thuộc vào loại dữ liệu được sắp xếp, có thể được thực hiện theo nhiều cách khác nhau. Sự đa dạng của các thuật toán đến nỗi Knuth dành toàn bộ tập thứ ba (hơn 800 trang) cho chuyên khảo nổi tiếng của mình *Nghệ thuật lập trình máy tính* (xem Knuth-3/1968). Nicklaus Wirth, người tạo ra ngôn ngữ Pascal, cũng rất chú ý đến chúng trong "*Thuật toán + Cấu trúc dữ liệu = Chương trình*" (xem [Wirth-1980]).

Có nhiều cách phân loại khác nhau của các thuật toán sắp xếp. Có lẽ phổ biến nhất là tùy thuộc vào vị trí của dữ liệu. Dựa trên tiêu chí này, chúng ta phân biệt giữa nội bộ (dữ liệu nằm trong RAM của máy tính và thường có thể truy cập trực tiếp vào bất kỳ phần tử nào của tập hợp) và bên ngoài (dữ liệu nằm trong bộ nhớ ngoài

của máy tính và quyền truy cập thường nhất quán nghiêm ngặt, bắt đầu từ phần tử đầu tiên). Tùy thuộc vào phép toán được thực hiện trên các phần tử, chúng ta phân biệt giữa sắp xếp bằng cách so sánh (thường xuyên nhất với sự trợ giúp của $<, >$ và $=$) và bằng phép biến đổi (với sự trợ giúp của các phép toán số học, không so sánh trực tiếp các cặp phần tử). Các phân loại quan trọng khác dựa trên các thuộc tính nhất định của các thuật toán sắp xếp. Ví dụ, ổn định và không ổn định. Một phương thức được gọi là ổn định nếu thứ tự tương đối của các phần tử có khóa bằng nhau không thay đổi trong quá trình sắp xếp. Phương pháp sắp xếp ổn định được ưu tiên khi các phần tử của tập hợp đã được sắp xếp theo một số tiêu chí khác. Ví dụ: chúng ta muốn sắp xếp hệ thống tệp của mình theo tên và phần mở rộng: các tệp được sắp xếp theo tên và những tệp có cùng tên được sắp xếp theo phần mở rộng. Điều này có thể được thực hiện theo hai bước: 1) sắp xếp theo phần mở rộng và 2) sắp xếp theo tên. Trong khi ở loại đầu tiên (theo phần mở rộng), tính ổn định của thuật toán là không liên quan, ở loại thứ hai (theo tên), việc sử dụng một phương pháp ổn định bây giờ sẽ là bắt buộc.

Quá trình sắp xếp các phần tử từ một tập hợp, như đã đề cập ở trên, thường được rút gọn để sắp xếp lại chúng. Chúng ta sẽ định nghĩa các khái niệm một cách chặt chẽ hơn. Cho tập hợp M với các phần tử đã cho

$$a_1, a_2, ..., a_n,$$

và hàm f xác định trên chúng.

Bằng cách sắp xếp các phần tử của M, chúng ta có nghĩa là hoán vị của chúng theo thứ tự thích hợp

$$a_{i_1}, a_{i_2}, ..., a_{i_n}$$

để nó được hoàn thành:

$$f(a_{i_1}) \leq f(a_{i_2}) \leq ... \leq f(a_{i_n}).$$

Hàm f được gọi là *hàm sắp xếp* của tập hợp. Chúng ta sẽ xem xét rằng nó được tính toán trước cho từng phần tử và được ghi nhớ rõ ràng như một phần (trường, *khóa*) của nó. Các phần tử của tập hợp được sắp xếp thường được biểu diễn dưới dạng bản ghi, một trong các trường là khóa:

```
#define MAX 100
struct CElem {
int key;
/ * .............
Một số dữ liệu
.............. * /
} m [MAX];
```

Đối với các phương pháp sắp xếp phổ biến, và nếu có thể, chúng ta sẽ thực hiện theo khai báo ở trên về loại phần tử của tập hợp. Chúng ta thường sẽ biểu diễn tập hợp dưới dạng một mảng, nhưng đôi khi chúng ta cũng sẽ sử dụng biểu diễn động dưới dạng danh sách được liên kết.

Yêu cầu chính đối với các thuật toán sắp xếp là chi phí tối thiểu của bộ nhớ bổ sung. Một yêu cầu quan trọng khác là số lượng tối thiểu so sánh và trao đổi các yếu tố. Việc sắp xếp thường được thực hiện bằng cách hoán đổi đơn giản hai phần tử của mảng. Trong các chương trình sau để trao đổi giá trị của hai biến, chúng ta sẽ sử dụng hàm sau:

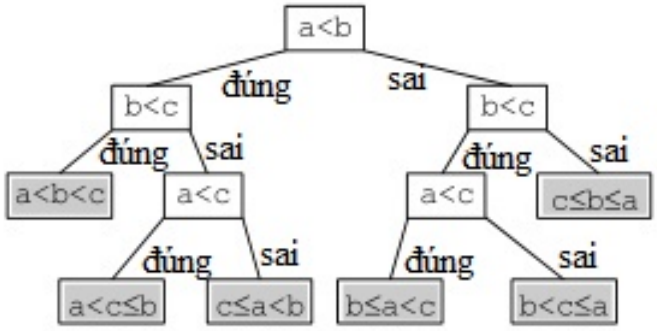
```
/ * Hoán đổi các giá trị của *x1 và *x2 * /
void swap(struct CElem *x1, struct CElem *x2)
{ struct CElem tmp = *x1; *x1 = *x2; *x2 = tmp; }
```

3.2. Sắp xếp theo so sánh

Tất cả các thuật toán trong đoạn này đều thuộc loại thuật toán sắp xếp so sánh. Đây là các thuật toán cổ điển trong đó phép toán duy nhất được phép là so sánh giữa các cặp phần tử sử dụng các phép toán $<(\leq), >(\geq)$ và $=(\neq)$.

3.2.1. Cây so sánh

Đây có lẽ là phương pháp cơ bản nhất để phân loại theo phương pháp so sánh. Nó dựa trên một loạt các so sánh, mỗi so sánh mang lại cho chúng ta thông tin bổ sung. Quá trình tiếp tục cho đến khi tập hợp được sắp xếp hoàn toàn. Tùy thuộc vào kết quả của phép so sánh, chúng ta nhận được các tỷ lệ khác nhau giữa các yếu tố và do



Hình 3.1. Cây so sánh cho tập $\{a, b, c\}$.

đó - sự tiếp tục khác nhau của quá trình. Chúng ta có thể giả định rằng bất kỳ so sánh nào của dạng $x < y$ đều có hai kết quả: có, nếu x nhỏ hơn y và không - nếu không. Quá trình so sánh có thể được biểu diễn bằng đồ thị dưới dạng cây nhị phân, trong danh sách chứa các trình tự được sắp xếp và trong danh sách không - so sánh giữa các cặp phần tử của tập hợp. Hình 3.1 minh hoạ phương thức trên tập ba phần tử $\{a, b, c\}$.

Cây cung cấp cho chúng ta thuật toán sau để sắp xếp tập hợp:

```
if (a < b)
  if (b < c) printf("a,b,c");
  else
    if (a < c) printf("a,c,b");
    else printf("c,a,b");
else
  if (b < c)
    if (a < c) printf("b,a,c");
    else printf("b,c,a");
  else printf("c,b,a");
```

Thuật toán trên rất hay và hữu ích trong điều kiện nó 1) đảm bảo cho chúng ta số lượng phép so sánh tối thiểu (Tại sao?), Và 2) đưa ra một cây so sánh rõ ràng (thuật toán thứ hai được sử dụng để chứng minh một kết quả quan trọng: độ phức tạp thời gian tối thiểu của bất kỳ thuật toán nào để phân loại bằng cách so sánh - xem 3.2.10). Ưu điểm của nó, thật không may, kết thúc ở đó. Không khó để nhận

thấy rằng số lượng đầu ra có thể có (lá cây) là chương trình $n!$. Cần tìm kiếm các phương pháp sắp xếp khác hiệu quả hơn.

Bài tập

▷ **3.1.** Chứng minh rằng cây so sánh đảm bảo số lượng so sánh tối thiểu trong trường hợp xấu nhất.

3.2.2. Sắp xếp theo cách chèn

Có ba phương pháp phổ thông cơ bản cổ điển để sắp xếp theo cách so sánh: theo chèn, theo lựa chọn và theo phương pháp bong bóng. Trọng tâm của mỗi chúng là một ý tưởng đơn giản cho phép thực hiện nhanh chóng và rõ ràng. Phương pháp sắp xếp cơ bản có hiệu quả với số lượng phần tử tương đối nhỏ (khoảng 20 phần tử) và thường được sử dụng trong thực tế. Thật không may, với số lượng phần tử lớn hơn, tốc độ của chúng giảm mạnh, điều này khiến bạn cần phải sử dụng các phương pháp khác. Thật vậy, cả ba phương pháp cơ bản đều được đặc trưng bởi độ phức tạp thuật toán $\Theta(n^2)$, chậm hơn nhiều so với độ phức tạp $\Theta(n.log_2 n)$, đây là đặc điểm của các phương pháp sắp xếp hiện đại như phương pháp hình chóp (xem 3.2.9) hoặc sắp xếp nhanh (xem 3.2.6). Tuy nhiên, các phương pháp cơ bản có vị trí của chúng, vì đối với các trình tự đủ nhỏ, chúng hiệu quả hơn và như chúng ta sẽ thấy bên dưới, thường được sử dụng trong các biến thể lai để tăng tốc độ. Ví dụ, một cách tiếp cận sắp xếp nhanh được sử dụng rộng rãi là sử dụng một phương pháp đơn giản hơn, chẳng hạn như sắp xếp chèn, khi đến một phân vùng có ít phần tử.

Sắp xếp chèn là một phương pháp sắp xếp thứ tự nổi tiếng, trong đó người chơi cầm quân bài ở tay trái, lấy từng quân bài ở tay phải và đặt chúng vào đúng vị trí. Để cắm thẻ vào đúng vị trí, cần phải so sánh nhất quán (bằng cách nhìn nhanh) với các thẻ đã được sắp xếp cho đến khi tìm được vị trí chính xác.

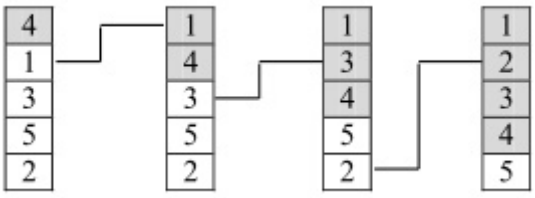
Hãy quay trở lại thuật ngữ của chúng ta. Chúng ta có một mảng các phần tử `struct` CElem với một khóa key mà chúng ta muốn sắp xếp. Mảng được chia thành khu vực được sắp xếp và không được sắp xếp. Vùng đã sắp xếp thường nằm ở đầu mảng, ban đầu chỉ bao gồm phần tử đầu tiên của nó. Việc sắp xếp diễn ra trong $n - 1$ bước.

Trong bước thứ i, khu vực đã sắp xếp được mở rộng bởi một phần tử ở bên phải và cho mục đích này $(i+1)$ phần tử đầu tiên (giả sử ký hiệu là x) được chèn vào một vị trí thích hợp trong chuỗi đã sắp xếp, tức là trong số các phần tử từ 1 đến i.

Việc chèn được thực hiện như thế nào? Một thuật toán hiển nhiên là so sánh tuần tự và có thể trao đổi x với phần tử bên trái của nó. Quá trình tiếp tục cho đến khi một trong các tình huống sau xảy ra (xem Hình 3.2):

1) đến phần tử có khóa nhỏ hơn hoặc bằng khóa của x;

2) đến phần tử đầu tiên của mảng.



Hình 3.2. Sắp xếp theo cách chèn.

Sau đây là một ví dụ thực hiện. Lưu ý rằng phần tử đầu tiên của mảng thực sự là m[0]

Chương trình 3.1. Sắp xếp theo cách chèn (301insert_s.c)

```
void straightInsertion (struct CElem m[], unsigned n)
{ struct CElem x;
  unsigned i;
  int j;
  for (i = 0; i < n; i++) {
    x = m[i];
    j = i-1;
    while (j >= 0 && x.key < m[j].key)
      m[j+1] = m[j--];
    m[j+1] = x;
  }
}
```

Tất cả các kiểm tra có thể được hợp nhất thành một bằng cách thêm một phần tử khóa $-\infty$ làm phần tử 0 của mảng. Phương pháp

này được gọi là *phương pháp giới hạn*. Vì giá trị của nó khó biểu diễn trong hệ thống máy tính, nên việc sử dụng một khóa từ tập hợp các khóa hợp lệ làm giới hạn sẽ dễ dàng hơn. Không khó để coi rằng giá trị của khóa x sẽ thực hiện một công việc tuyệt vời. (Tại sao?) Chúng ta nhận được phần mềm triển khai thuật toán chèn trực tiếp sau (Lần này các phần tử nằm ở vị trí $1, 2, ..., n$):

Chương trình 3.2. Sắp xếp theo cách chèn (302inserts2.c)

```
void straightInsertion (struct CElem m[], unsigned n)
{ unsigned i, j;
  struct CElem x;
  for (i = 1; i <= n; i++) {
    x = m[i]; m[0].key = x.key;
    for (j = i - 1; x.key < m[j].key; j--)
      m[j+1] = m[j];
    m[j+1] = x;
  }
}
```

Cách chèn	Số tối thiểu	Số trung bình	Số tối đa
So sánh	$n-1$	$(n^2+n-2)/4$	$(n^2+n)/2-1$
Trao đổi	$2(n-1)$	$(n^2+9n-10)/4$	$(n^2+3n-4)/2$

Bảng 3.1. So sánh và trao đổi trong việc sắp xếp theo cách chèn.

Không khó để xác định rằng số lượng so sánh trung bình được thực hiện, cũng như số lượng trao đổi được yêu cầu (lưu ý rằng trong chương trình trên hầu như không có các trao đổi trực tiếp, thay vào đó sử dụng các *phép gán một chiều*), là bậc n^2 . Điều này cũng có thể được nhìn thấy từ Bảng **??** đính kèm. Người đọc tò mò có thể tìm thấy lời giải thích chi tiết cho những đánh giá trên trong [Knuth-3/1968] [Wirt-1980].

Chúng ta có thể tăng hiệu quả của việc phân loại chèn không? Và có, và không. Thật vậy, bằng cách sử dụng tìm kiếm nhị phân (xem 4.3.), Chúng ta có thể giảm số lượng *phép so sánh* cần thiết để tìm vị trí chèn x từ bước thứ i, thành $\log_2 i$:

Chương trình 3.3. Sắp xếp theo cách chèn (303insert_b.c)

```
void binaryInsertion(struct CElem m[], unsigned n)
{ struct CElem x;
  unsigned i, med, r;
  int j, l;
  for (i = 1; i < n; i++) {
    x = m[i];
    l = 0;
    r = i - 1;
    /* Tìm kiếm nhị phân*/
    while (l <= (int)r) {
      med = (l + r) / 2;
      if (x.key < m[med].key)
        r = med - 1;
      else
        l = med + 1;
    }
    /* Nơi đã được tìm thấy. Sau đó chèn và di chuyển sang bên phải
    */
    for (j = i - 1; j >= l; j--)
      m[j + 1] = m[j];
    m[l] = x;
  }
}
```

Thật không may, điều sau sẽ không ảnh hưởng đến số lượng trao đổi, bởi vì, mặc dù chúng ta biết vị trí cụ thể của phần chèn, nhưng chúng ta phải xem qua các phần tử của mảng cho đến khi nó đạt đến. Sự bất tiện này có thể được khắc phục bằng cách sử dụng danh sách được liên kết động, nhưng sau đó chúng ta không thể áp dụng tìm kiếm nhị phân.

Bài tập

▷ **3.2.** Hãy chứng minh rằng khóa của phần tử được chèn là một giới hạn tốt khi sử dụng phương pháp của giới hạn để thực hiện sắp xếp bằng cách chèn.

▷ **3.3.** Hãy so sánh các biến thể khác nhau của sắp xếp theo cách chèn.

▷ **3.4.** Tính toán công thức từ Bảng 3.1.

3.2.3. Sắp xếp theo thứ tự chèn với bước giảm dần. Thuật toán của Shell

Năm 1959, Shell đề xuất một cải tiến tốt về phương pháp sắp xếp chèn. Ý tưởng là thực hiện nhiều sắp xếp bằng cách chèn trực tiếp một phần của các phần tử của mảng với bước δ, bước này giảm dần và cuối cùng đạt đến 1.

Quá trình sắp xếp bằng cách chỉ chèn trực tiếp các tập con có dạng $x_k, x_k + \delta, x_k + 2\delta, ...(1 \leq k \leq \delta)$ sẽ được gọi là *δ-sắp xếp*. Cho dãy các bước $\delta_s, \delta_{s-1}, ..., \delta_1$, thỏa mãn điều kiện

$$\delta_s > \delta_{s-1} > ... > \delta_1 = 1.$$

Rõ ràng, do ứng dụng tuần tự của δ_s-sắp xếp, δ_{s-1}-sắp xếp, ..., δ_1-sắp xếp tập hợp đầu ra sẽ được sắp xếp. Không khó để thấy rằng nếu một dãy đã được δ-sắp xếp, thì sau khi áp dụng δ'-sắp xếp ($\delta' < \delta$), nó tiếp tục được (-sắp xếp. (Tại sao?) Như vậy, mỗi lần sắp xếp tiếp theo sẽ tận dụng lợi thế của lần sắp xếp trước đó và cuối cùng việc sắp xếp theo bước 1 sẽ hoàn thành phần còn lại của công việc.

Mỗi lần sắp xếp được giảm xuống δ số lượng kiểu bằng cách chèn trực tiếp, mỗi kiểu cần có giới hạn riêng. Thêm một phần tử 0 ở đầu mảng là không đủ. Chúng ta cần tổng số δ_s trong số lượng dấu phân cách, điều này yêu cầu mở rộng mảng thêm δ_s về số lượng phần tử. Chúng ta sẽ làm việc với một mảng duy nhất và chúng ta sẽ sử dụng các phần tử đầu tiên của nó làm giới hạn. Để có thể tham chiếu đến các phần tử có chỉ số âm, chúng ta sẽ sử dụng một mẹo: chúng ta sẽ gọi hàm shellSort() không phải với shellSort(m, MAX), mà với shellSort(m + step0 + 1, MAX) .

Chương trình 3.4. Thuật toán shell (304shell.c)

```
#define MAX 100
#define STEPS_CNT 4
#define steps0  40
const unsigned steps[STEPS_CNT] = { steps0, 13, 4, 1 };
struct CElem {
  int key;
```

```
  /* ..............
     Một số dữ liệu
     .............. */
};
...................
void shellSort(struct CElem m[], unsigned n)
{ int i,j,k,s;
  unsigned stepInd;
  struct CElem x;

  for (stepInd = 0; stepInd < STEPS_CNT; stepInd++) {
    s = -(k = steps[stepInd]); /*Giới hạn*/
    for (i = k + 1; i <= (int)n; i++) {
      x = m[i];
      j = i - k;
      if (0 == s)
        s = -k;
      m[++s] = x;
      while (x.key < m[j].key) {
        m[j + k] = m[j];
        j -= k;
      }
      m[j + k] = x;
    }
  }
}
...................
int main() {
  struct CElem m[MAX + steps0 + 2];
  ...................
  shellSort(m + steps0 + 1, MAX);
  ...................
  return 0;
}
```

Ý tưởng về dấu phân cách là tốt cho việc sắp xếp chèn, nhưng ở đây cần có một số dấu phân cách, điều này không hiệu quả và dẫn đến sự bất tiện trong việc định địa chỉ mảng, mà chúng ta đã giải quyết ở trên bằng "chương trình nắm bắt". Nó có thể hiệu quả và mang tính hướng dẫn, nhưng nó có cần thiết không? Nếu chúng ta

suy nghĩ một chút, chúng ta dễ dàng đi đến kết luận rằng chúng ta thực sự có thể làm mà không có giới hạn!

Kết quả là, chúng ta nhận được một đoạn mã ngắn hơn và rõ ràng hơn nhiều. Tuy nhiên, chúng ta lưu ý rằng chúng ta "trả tiền" cho điều này một cách hiệu quả, vì thuật toán mới của chúng ta (xem bên dưới) thực hiện nhiều gấp đôi so sánh trong vòng lặp while nội bộ của nó. Ở đây chúng ta sử dụng một dãy 16 phần tử hiệu quả khác (chúng ta sắp xếp các phần tử từ $l+1$ đến r).

Chương trình 3.5. Thuật toán shell (305shell2.c)

```
void shellSort(struct CElem m[], unsigned l, unsigned r)
{ static unsigned incs[16] = { 1391376,463792,198768,86961,33936,
                               13776,4592,1968,861,336,112,48,
                               21,7,3,1 };
  unsigned i, j, k, h;
  struct CElem v;

  for (k = 0; k < 16; k++)
    for (h = incs[k], i = l+h; i <= r; i++) {
      v = m[i]; j = i;
      while (j > h && m[j-h].key > v.key) {
          m[j] = m[j-h];
          j -= h;
      }
    m[j] = v;
  }
}
```

Làm thế nào để chọn các bước chúng ta cần? Vấn đề này cực kỳ phức tạp và về mặt toán học vẫn chưa được giải quyết. Để có được lợi ích tối đa từ việc áp dụng nhất quán các phân loại, chúng cần phải tương tác ở mức tối đa. Điều này dẫn chúng ta đến ý tưởng rằng các bước nên là bội số của nhau, ví dụ như độ liên tiếp của cặp đôi.

Papernov và Stasevich chỉ ra rằng nếu trình tự được sử dụng

$$1, 3, 7, 15, 31, 63, 127, ..., 2^k - 1, ...$$

thuật toán yêu cầu $\Theta(n\sqrt{n})$ bước [Papernov] [Stasevic-1975].

Pratt cung cấp

$$1, 2, 3, 4, 6, 8, 9, 12, 16, ..., 2^p 3^q, ...,$$

điều này chứng tỏ rằng $\Theta(n(\log_2 n)^2)$ bước là cần thiết. Về mặt tiệm cận, đường của Pratt là tốt nhất, nhưng tiếc là nó phát triển chậm và quá dài (xem [Pratt-1979]). Knuth (xem [Knuth-3/1968]) cung cấp phạm vi

$$1, 4, 13, 40, 121, ...,$$

được cho bởi các phương trình $\delta_{k-1} = 3\delta_k + 1, \delta_s = 1, s = [\log_3 n] - 1$, được viết theo thứ tự ngược lại. Một bộ truyện hay khác, cũng do Knut đề xuất, là:

$$1, 3, 7, 15, 31, ...$$

được cho bởi các phương trình $\delta_{k-1} = 2\delta_k + 1, \delta_s = 1, s = [\log_2 n] - 1$, lại được viết theo thứ tự ngược lại. Trong trường hợp này, số phép so sánh có bậc là $n^{1,2}$. Kết quả tốt thu được đối với chuỗi đạt đến mức tối đa của việc giảm cấp số nhân của một tư nhân 1,7. Trong trường hợp này, độ phức tạp của thuật toán là $\Theta(n(\log_2 n)^2)$.

Bài tập

▷ **3.5.** Chứng minh rằng nếu một dãy đã được sắp xếp, thì sau khi áp dụng δ'-sắp xếp, $\delta' < \delta$ nó tiếp tục được sắp xếp.

3.2.4. Phương pháp bong bóng

Phương pháp bong bóng chắc chắn là phổ biến nhất trong số các lập trình viên, đặc biệt là với số lượng phần tử sắp xếp đủ nhỏ. Như Bảng 3.2 dưới đây. tuy nhiên, sự phổ biến của nó hầu như không do tính hiệu quả của nó (Đây là thuật toán tệ nhất trong tất cả các thuật toán được thảo luận trong chương này!). "Sự quyến rũ" của ông có thể được tìm kiếm thay vì dễ dàng nó được hiện thực hóa, trong sự đơn giản trong ý tưởng của ông và có lẽ ở một mức độ nào đó trong cái tên kỳ lạ của ông. Thông thường, thuật toán sắp xếp đầu tiên mà một lập trình viên mới học về là phương pháp bong bóng. Không thể bỏ qua thực tế là phương pháp này là cơ sở để phân loại nhanh Hoor - thuật toán sắp xếp phổ quát nhanh nhất.

Ý tưởng của thuật toán là xem một cách nhất quán các phần tử của mảng và, nếu đối với một cặp phần tử liền kề x_{i-1} và x_i thì kết quả là $x_{i-1} > x_i$, chúng sẽ hoán đổi vị trí cho nhau (giả sử sắp xếp theo thứ tự tăng dần) . Trong quá trình sắp xếp, các phần tử nhẹ nhất, như bong bóng, nổi lên "bề mặt", tức là ở cuối bên trái của mảng. Do đó tên của phương thức.

Chương trình 3.6. Phương pháp bong bóng (306bubsort1.c)

```
void bubbleSort1(struct CElem m[], unsigned n)
{ unsigned i, j;
  for (i = 1; i < n; i++)
    for (j = n-1; j >= i; j--)
      if (m[j-1].key > m[j].key)
        swap(m+j-1, m+j);
}
```

Cách nổi bóng	Số tối thiểu	Số trung bình	Số tối đa
So sánh	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
Trao đổi	0	$3(n^2-n)/4$	$3(n^2-n)/2$

Bảng 3.2. So sánh và trao đổi trong sắp xếp theo cách nổi bong bóng.

Bảng 3.2 cung cấp thông tin về các đặc điểm chính của phân loại bong bóng. Có thể thấy, số phép so sánh được thực hiện bởi thuật toán luôn giống nhau, cụ thể là $n(n-1)/2$. Điều này khiến chúng ta nghĩ rằng một số cải tiến là có thể. Ví dụ: chúng ta có thể thêm một cờ cho biết liệu một cuộc trao đổi đã diễn ra trên lần lặp hiện tại hay chưa. Quá trình phân loại sẽ tiếp tục miễn là các cuộc trao đổi diễn ra. Tuy nhiên, các thử nghiệm thực tế cho thấy rằng lợi ích của việc giới thiệu một cờ là gần như không thể nhận thấy và với một mảng được sắp xếp ngược lại, thuật toán thậm chí còn chậm lại.

Một cải tiến đáng kể khác có thể được thực hiện từ ý tưởng rằng việc xem xét lại các yếu tố chắc chắn đã biết là ở vị trí cuối cùng của chúng là vô nghĩa. Lần này chúng ta sẽ sắp xếp theo hướng ngược lại. Tất nhiên, phần sau là không đáng kể, nhưng nó mang lại một cái nhìn mới về thuật toán: chúng ta bắt đầu từ "dưới cùng" ở cuối mảng và các phần tử nặng nhất chìm dần về phía nó. Chúng ta sẽ

duy trì biến *i*, cho biết chỉ số tối đa mà lần lặp trước đó đã được trao đổi. Rõ ràng, các yếu tố bên phải của cuộc trao đổi cuối cùng nằm ở vị trí cuối cùng của chúng, điều này cho chúng ta lý do để không nhìn vào chúng nữa.

Chương trình 3.7. Phương pháp bong bóng (307bubsort2.c)

```
void bubbleSort2(struct CElem m[], unsigned n)
{ unsigned i, j, k;
for (i = n; i > 0; i = k)
  for (k = j = 0; j < i; j++)
    if (m[j].key > m[j+1].key) {
      swap(m+j, m+j+1);
      k = j;
    }
}
```

Bài tập

▷ **3.6.** Tìm công thức trong Bảng 3.2

3.2.5. Sắp xếp bằng cách lắc

Mặc dù hiệu quả thấp, phương pháp bong bóng cho phép một số cải tiến. Chúng ta đã đề cập đến hai trong số chúng ở trên - thêm một cờ và duy trì một biến cho biết chỉ số trao đổi tối đa từ bước trước. Có một tính năng khác: một phần tử sáng riêng biệt ở cuối mảng sẽ xuất hiện trong một lần lặp duy nhất, trong khi phần tử nặng "đơn độc" nằm gần bề mặt sẽ chìm ở mỗi bước của thuật toán chỉ một mức xuống dưới cùng. Nếu chúng ta đảo ngược hướng của các chu kỳ, mọi thứ sẽ hoàn toàn ngược lại, nhưng sự bất đối xứng rõ ràng sẽ vẫn còn. Điều này khiến chúng ta nghĩ đến việc đảo ngược hướng sắp xếp ở mỗi bước. Vì vậy, chúng ta nhận được cải tiến thứ ba của thuật toán. Áp dụng ba cải tiến cùng lúc, chúng ta nhận được một thuật toán mới - sắp xếp bằng cách lắc:

Chương trình 3.8. Sắp xếp bằng cách lắc (308shaker.c)

```
void shakerSort(struct CElem m[], unsigned n)
{ unsigned k = n, r = n-1;
  unsigned l = 1, j;
```

```
    do {
      for (j = r; j >= l; j--)
        if (m[j-1].key > m[j].key) {
          swap(m+j-1,m+j);
          k = j;
      }
      l = k + 1;
      for (j = l; j <= r; j++)
        if (m[j-1].key > m[j].key) {
          swap(m+j-1,m+j);
          k = j;
        }
      r = k - 1;
    } while (l <= r);
}
```

Trong trường hợp tốt nhất, thuật toán trên thực hiện $n-1$ so sánh và trong trường hợp giữa - lại $\Theta(n^2)$. Mặc dù có một số cải thiện về mặt này so với phương pháp bong bóng cổ điển, nhưng hiệu quả tổng thể của thuật toán không cải thiện nhiều, vì sắp xếp theo kiểu rung chuyển không làm giảm số lượng trao đổi được thực hiện. Và về nguyên tắc, cuộc trao đổi khó hơn nhiều lần so với việc so sánh.

Bài tập

▷ **3.7.** So sánh các tùy chọn sắp xếp bong bóng khác nhau:

a) bong bóng không có cờ;

b) bong bóng có cờ;

c) phân loại đá.

3.2.6. Nhanh chóng phân loại Hoor

Các thuật toán dựa trên trao đổi phần tử được thảo luận ở trên có hiệu quả khá khiêm tốn, vì phương pháp bong bóng cổ điển thậm chí có các tham số tồi tệ nhất so với tất cả các thuật toán sắp xếp được thảo luận cho đến nay. Mặc dù chúng ta đã nỗ lực rất nhiều (ba cải tiến đã được thực hiện trong quá trình phân loại!), Những đặc điểm này không cải thiện nhiều. Như thể phân loại thông qua

trao đổi không dẫn đến các thuật toán hiệu quả ... Có thực sự là như vậy không?

Trong quá trình rung chuyển, mặc dù đã có những cải tiến, nhưng chúng ta không đạt được kết quả khả quan, bởi vì, giảm số lượng so sánh, chúng ta không ảnh hưởng đến số lượng trao đổi của các phần tử. Bây giờ chúng ta sẽ quay lại: chúng ta sẽ cố gắng chủ yếu để giảm thiểu số lượng trao đổi. Vì việc trao đổi hai yếu tố đắt hơn nhiều lần so với việc so sánh, nên một sự cải thiện nghiêm trọng có thể được mong đợi. Nhưng làm thế nào để giảm số lượng trao đổi? Chúng ta hãy nhớ lại cải tiến thứ ba của phương pháp bong bóng (xem 3.2.4), Mà chúng ta đã thực hiện trong quá trình thực hiện lắc (xem 3.2.5): Đảo ngược hướng truyền của các phần tử ở mỗi bước của thuật toán. Lý do cho điều này là do chúng ta quan sát thấy rằng một nguyên tố nặng duy nhất nằm gần bề mặt sẽ chìm ở mỗi bậc chỉ với một vị trí ở phía dưới. Đồng thời, một phần tử ánh sáng riêng biệt nằm ở cuối mảng sẽ xuất hiện đến vị trí cuối cùng của nó trong một lần lặp lại duy nhất. Tuy nhiên, lưu ý rằng ngay cả trong trường hợp này, phần tử ánh sáng sẽ phải được trao đổi với tất cả các phần tử trên đường đến vị trí cuối cùng của nó, và thao tác này rất tốn kém! Làm thế nào để giảm số lượng trao đổi? Trong trường hợp mảng nghịch đảo, chúng ta có thể hoán đổi phần tử đầu tiên với phần tử cuối cùng, phần tử thứ hai với phần tử áp chót, v.v., sắp xếp toàn bộ mảng cho $[n/2]$ các hoán đổi. Kết luận là hiển nhiên: khoảng cách giữa các sàn giao dịch càng lớn thì hiệu quả của chúng càng cao.

Ý tưởng của thuật toán do Hoor đề xuất là chọn một phần tử x và chia mảng thành hai phân vùng: bên trái, trong đó các phần tử nhỏ hơn x và bên phải, trong đó chúng lớn hơn. Chúng ta áp dụng cùng một thuật toán cho các phần bên trái và bên phải, giảm dần ranh giới bên trái và bên phải của các mảng con được xem xét cho đến khi chúng ta đạt đến các khoảng chứa một phần tử duy nhất. Sau khi thuật toán hoàn thành, mảng sẽ được sắp xếp. (Tại sao?)

Gọi q là chỉ số của x trong mảng, tức là $x = m[q]$. Hãy sắp xếp mảng con $m[l, l+1, ..., r]$ và partition() chia nó thành hai phần: bên trái $(m[l, l+1, ..., q])$ và bên phải $(m[q+1, q+2, ..., r])$ và trả về kết quả là q. Lưu ý rằng sự tách biệt gần như chắc chắn liên quan đến

sự trao đổi của các phần tử, tức là partition() không tìm kiếm một phần tử x trong mảng, nhưng chọn nó và chia nó thành hai vùng tùy theo giá trị của nó. Dưới đây chúng ta sẽ xem cách này có thể được thực hiện.

Sắp xếp nhanh có thể được viết theo cách này (Mảng m[] không được truyền dưới dạng tham số, nhưng được coi là một biến toàn cục: điều này tiết kiệm không gian ngăn xếp và tăng tốc độ vì hàm là đệ quy.):

```
void quickSort(int l, int r)
{ int q;
  if (l < r) {
    q = partition(l,r);
    quickSort(l,q);
    quickSort(q+1,r);
  }
}
```

Giả sử rằng dựa trên một số công thức, chúng ta đã tính được q. (Đầu tiên chúng ta sẽ chọn phần tử ngoài cùng bên trái hoặc ngoài cùng bên phải của mảng trước phân vùng.) Lưu ý rằng $m[q]$ chứa giá trị của $x$trước phân vùng. Sau phép chia q sẽ là một chỉ số biên: ở bên trái của nó (bao gồm) chúng ta sẽ có các phần tử có khóa không vượt quá giá trị của x và ở bên phải - khóa lớn hơn x.

Câu hỏi quan trọng tiếp theo được đặt ra: Làm thế nào để thực hiện việc phân chia? Nhìn chung có hai cách tiếp cận khác nhau. Đầu tiên sau đây:

```
unsigned partition(int l, int r)
{ int q, j, x;
q = l - 1; x = m[r].key;
for (j = l; j <= r; j++)
if (m[j].key <= x) {
q++;
swap(m+q,m+j);
}
if (q == r) q--; /*Tất cả các phần tử đều <= x. Diện tích giảm đi 1.*
    /
return q;
}
```

Phương pháp đề xuất hoạt động như thế nào? Chọn phần tử x để thực hiện tách. Các phần tử nhỏ hơn hoặc bằng x phải nằm ở bên trái của mảng và lớn hơn x phải ở bên phải. Phần được coi là của mảng $m[l, l+1, ..., r]$ được xem từ trái sang phải, trong khi phần bên trái của nó là một vùng không ngừng mở rộng của các phần tử nhỏ hơn hoặc bằng x được xây dựng. Cuối bên phải của khu vực được xác định bởi q. Khi j đến cuối r của phân vùng, q sẽ chỉ ra ranh giới giữa hai vùng. (Lưu ý: Trong một chương trình thực, bạn nên thay thế hàm swap() bằng mã của nó để đạt hiệu quả cao hơn.)

Một phương pháp khả thi khác là sử dụng hai chỉ số i và j, cho thấy ranh giới của hai khu vực liên tục mở rộng từ hai đầu đến trung tâm. Ở mỗi bước, một nỗ lực được thực hiện để mở rộng vùng bên trái sang bên phải càng lâu càng tốt, nghĩa là miễn là có một phần tử nhỏ hơn hoặc bằng x ở bên phải của nó. Điều tương tự cũng được thực hiện đối với khu vực bên phải (cả hai chu kỳ trong khi). Sau đó, vị trí của hai phần tử kết giới được hoán đổi cho các khu vực bên trái và bên phải đã ngừng mở rộng, và quá trình này được lặp lại từ đầu. Kết thúc ở "cuộc gặp gỡ" của hai lĩnh vực. Chỉ số i cho biết điểm cuối bên phải của vùng bên trái và j- điểm cuối bên trái của vùng bên phải:

```
unsigned partition(int l, int r)
{ unsigned i, j, x;
  i = l; j = r; x = m[l].key;
  do {
    while (x > m[i].key) i++;
    while (x < m[j].key) j--;
    if (i <= j) {
      swap(m+i,m+j);
      i++; j--;
    }
  while (i <= j);
  return j;
}
```

Sau khi kết hợp partition() và quickSort(), chúng ta nhận được (cho tùy chọn đầu tiên):

```
void quickSort(int l, int r)
{ int i,j,x;
  i = l-1; x = m[r].key;
  for (j = l; j <= r; j++)
    if (m[j].key <= x) {
      i++;
      swap(m+i,m+j);
    }
  if (i == r) /* Tất cả các phần tử đều <= x. Diện tích giảm đi 1. */
    i--;
  if (l < i)
    quickSort(l,i);
  if ((i+1) < r) /***/
    quickSort(i+1,r); /***/
}
```

Đối với phương án thứ hai, chúng ta có:

Chương trình 3.9. Sắp xếp nhanh (309qsort.c)

```
void quickSort(int l, int r)
{ int i, j, x;
  i = l;
  j = r;
  x = m[r].key;
  do {
    while (x > m[i].key) i++;
    while (x < m[j].key) j--;
    if (i <= j) {
      swap(m+i, m+j);
      i++;
      j--;
    }
  } while (j >= i);
  if (j > l)
    quickSort(l, j);
  if (i < r) /***/
    quickSort(i, r); /***/
}
```

Hai phép đảo ngược đệ quy liên tiếp được thực hiện để sắp xếp

hai phần của mảng. Rõ ràng là chúng không thể được thực hiện cùng một lúc và việc sau này luôn bị hoãn lại. Điều này hóa ra là "không cần thiết" theo nghĩa là không cần đệ quy mới - bây giờ chúng ta có thể xử lý lặp đi lặp lại. Một số trình biên dịch có thể tự động nhận dạng và loại bỏ đệ quy thứ hai. Tất nhiên, với nỗ lực tối thiểu, chúng ta có thể tự mình làm điều đó, nhận được một phiên bản bán lặp lại. Vì mục đích này, chỉ cần thay thế các hàng được đánh dấu bằng `/ *** /` bằng phép gán `l = i + 1;` (đối với biến thể đầu tiên) và `l = i;` (đối với tùy chọn thứ hai), sau đó lặp lại các hành động của hàm một lần nữa. Với mục đích này, chúng ta sẽ phải "mặc quần áo" cho cô ấy trong một chu kỳ khác.

Nếu chúng ta muốn, chúng ta có thể loại bỏ hoàn toàn đệ quy. Với mục đích này, chúng ta sẽ sử dụng cách tiêu chuẩn để loại bỏ nó - sử dụng ngăn xếp. Chúng ta sẽ lưu ý rằng đệ quy và lặp lại trong quá trình phát triển các thuật toán là hoàn toàn có thể hoán đổi cho nhau. Thật vậy, mỗi chu trình có thể được mô phỏng bằng đệ quy. Ở chiều ngược lại, mọi thứ phức tạp hơn. Chúng ta sẽ chỉ lưu ý rằng máy tính là máy lặp, và do đó mọi đệ quy chúng ta viết cuối cùng đều chuyển sang lặp. Mặt khác, đệ quy được máy tính mô hình hóa bằng cách đặt một số dữ liệu nhất định vào ngăn xếp hệ thống. Vậy tại sao không sử dụng ngăn xếp của riêng chúng ta một cách rõ ràng? Với mục đích này, ở mỗi bước, phần bên trái của mảng sẽ được xử lý trực tiếp, trong khi phần bên phải sẽ được đặt trong ngăn xếp và quá trình xử lý của nó sẽ bị hoãn lại cho đến phần cuối cùng với phần bên trái và tất cả các bài toán con có liên quan (ở đây x chúng ta chọn làm phần tử ở giữa mảng con được xem xét):

```
void quickSort(void)
{ int i,j,l,r,s,x;
  struct { int l, r; } stack[MAX];
  stack[s = 0].l = 0;
  stack[0].r = n-1;
  for (;;) {
    l = stack[s].l;
    r = stack[s].r;
    if (0 == s--)
      break;
    do {
```

```
      i = l; j = r; x = m[(l+r)/2].key;
      do {
        while (m[i].key < x) i++;
        while (m[j].key > x) j--;
        if (i <= j) {
          swap(m+i,m+j);
          i++;
          j--;
        }
      } while (i<=j);
      if (i < r) { /***/
        stack[++s].l = i; /***/
        stack[s].r := r; /***/
      } /***/
      r = j; /***/
    } while (l < r);
  }
}
```

Chúng ta hãy thử đánh giá ngăn xếp phần mềm cần thiết trong hoạt động của giải pháp lặp lại được đề xuất. Rõ ràng là trong trường hợp tối ưu, chúng ta sẽ giảm một nửa phân vùng mỗi lần, dẫn đến nghịch lưu đệ quy $\Theta(log_2 n)$ ngăn xếp. Tuy nhiên, trong trường hợp xấu nhất, với sự lựa chọn sai phần tử, kích thước của phân vùng được đề cập sẽ chỉ giảm 1 lần mỗi lần, nơi chúng ta sẽ có $\Theta(n)$ ngăn xếp. Trong phiên bản đệ quy, mọi thứ thậm chí còn tồi tệ hơn, bởi vì có nhiều dữ liệu hơn trong ngăn xếp: địa chỉ trả về, các tham số và các biến cục bộ. Một cách có thể giải quyết là xếp chồng các phần dài hơn và xem xét các phần ngắn hơn ngay lập tức. Không khó để thấy rằng trong cách tiếp cận này, ngăn xếp được yêu cầu sẽ luôn là $\Theta(log_2 n)$. Vì mục đích này, các dòng được đánh dấu bằng `/ *** /` nên được thay thế bằng đoạn chương trình sau (Tại sao?):

```
/* Đặt phần lớn hơn vào ngăn xếp*/
if (j - l < r - i) {
  /* Chèn phần bên phải vào ngăn xếp*/
  if (i < r) {
    stack[++s].l = i;
    stack[s].r = r;
  }
```

```
    r = j;
  }
  else {
    /* Chèn phần bên trái vào ngăn xếp */
    if (l < j)
      stack[++s].l = l;
      stack[s].r = j;
    }
    l = i;
  }
```

Rõ ràng rằng các phần tử ở bên trái và bên phải của mảng càng được phân bố đều do kết quả của việc phân tách, thì càng cần ít bước hơn. Trong trường hợp tốt nhất, nếu chúng ta chọn phần tử cỡ vừa mỗi lần, $\log_2 n$ phân vùng sẽ đủ cho việc sắp xếp tổng thể của mảng. Với tổng số phép so sánh trong trường hợp tối ưu, chúng ta nhận được $n.\log_2 n$. Số lượng trao đổi dự kiến tại mỗi lần tách có thứ tự là $n/6$. Do đó số lần trao đổi trong trường hợp tối ưu là $(n/6).\log_2 n$. Mặc dù xác suất bắn trúng phần tử ở giữa là $1/n$, hiệu quả trung bình của việc sắp xếp nhanh không phụ thuộc vào n, và nó kém hơn mức trung bình chỉ $2.\ln 2$ [Knuth-3/1968].

Quay trở lại câu hỏi: Làm thế nào để chọn phần tử x, mà chúng ta chia mảng? Một cách tiếp cận khả thi là luôn lấy một phần tử cố định từ mảng, chẳng hạn như phần tử đầu tiên, cuối cùng hoặc giữa (như chúng ta đã làm ở trên). Việc lựa chọn một phần tử phân chia là cực kỳ quan trọng đối với tốc độ của thuật toán: khóa của phần tử được chọn càng gần khóa của phần tử cỡ vừa cho phân vùng, thì việc chia mảng thành hai phần càng đồng đều và do đó độ sâu của cây đệ quy càng nhỏ, tức là thuật toán càng hiệu quả. (Tại sao?)

Do đó, trung vị của hai hoặc nhiều phần tử đôi khi được chọn làm phần tử phân chia. Tuy nhiên, điều này có liên quan đến các hoạt động bổ sung, và do đó dẫn đến sự chậm trễ trong lựa chọn, điều này không phải lúc nào cũng được bù đắp bằng lợi ích thu được và không nhất thiết dẫn đến cải tiến. Chúng ta khuyên người đọc quay lại *sắp xếp nhanh* sau, sau khi đã làm quen với phiên bản đệ quy của *sắp xếp bit* từ 3.3.3, Và để so sánh cách chọn phần tử phân tách.

Phân loại nhanh ẩn chứa một số điều bất ngờ. Như đã đề cập ở trên, đây là thuật toán sắp xếp phổ quát nhanh nhất được biết đến (trong 3.2. Chúng ta sẽ thấy rằng trong một số trường hợp đặc biệt quan trọng với các hạn chế bổ sung thì có các thuật toán tuyến tính!). Độ phức tạp thuật toán trung bình của nó là $\Theta(n.\log_2 n)$, cũng như độ phức tạp của đối thủ cạnh tranh chính - sắp xếp theo hình chóp (xem 3.2.9), Sẽ được thảo luận sau. Tuy nhiên, thực tế cho thấy rằng phân loại nhanh sẽ "đánh bại" mức trung bình của hình chóp từ hai đến ba lần. Đồng thời, phân loại của Hoor thất thường hơn nhiều và nếu chọn sai x, hiệu quả của nó có thể giảm đáng kể. Trong trường hợp xấu nhất, khi mỗi lần kích thước của phần được đề cập chỉ giảm đi 1, thì độ phức tạp của nó tỷ lệ với n^2. [Knuth-3/1968] [Wirth-1980]

Điểm mạnh của phân loại nhanh là ở cờ bạc của nó. Mảng càng xáo trộn, hiệu quả của phương pháp càng lớn. Tác giả Hoor khuyến nghị rằng việc lựa chọn x được thực hiện một cách ngẫu nhiên hoặc thậm chí là giá trị trung bình cộng của ba hoặc nhiều phần tử được chọn ngẫu nhiên. Trong thực tế, một chiến lược như vậy hầu như không ảnh hưởng đến thuật toán nói chung, trong khi tăng đáng kể hiệu quả của nó trong trường hợp xấu nhất. Cần lưu ý rằng, giống như hầu hết các phương pháp nhanh hiện đại, sức mạnh của sắp xếp nhanh không được biểu hiện ở một số lượng nhỏ các phần tử. Một cách tiếp cận tốt để loại bỏ thiếu sót này là áp dụng sắp xếp nhanh cho các phân vùng có ít nhất k (ví dụ: $k = 20$) phần tử, sử dụng thuật toán bong bóng đơn giản hơn cho các phân vùng nhỏ hơn (xem 3.2.4), lựa chọn trực tiếp (xem 3.2.8) hoặc chèn (xem 3.2.2). Một chiến lược thậm chí còn phức tạp hơn là không làm gì khi đến một phân vùng có ít hơn k phần tử. Sau khi hoàn thành việc sắp xếp nhanh đã sửa đổi, mảng không được sắp xếp theo thứ tự mà được chia thành nhiều nhóm nhỏ gồm các phần tử có giá trị tương tự nhau và mỗi phần tử của nhóm bất kỳ đều lớn hơn các phần tử của tất cả các nhóm ở bên trái được xem xét. Bằng cách này, mảng gần như được sắp xếp và là một đầu vào tốt để sắp xếp bằng cách chèn: các phần chèn sẽ được thực hiện trong khu vực tương ứng, nghĩa là được đảm bảo ở khoảng cách ngắn.

Bài tập

▷ **3.8.** Chứng minh rằng phân loại nhanh Hoor sắp xếp từng trình tự đầu vào một cách chính xác.

▷ **3.9.** Chứng minh rằng sắp xếp Hoor nhanh yêu cầu một chồng có thứ tự $\Theta(\log_2 n)$.

▷ **3.10.** Để kiểm tra việc triển khai phần mềm được đề xuất của phân loại nhanh Hoor cho tất cả các trình tự đầu vào có thể có với tối đa 32 phần tử:

(a) cho một dãy số ngẫu nhiên;

b) với mọi hoán vị của các số từ 1 đến n;

(c) cho tất cả các hoán vị của các số trong một tập hợp nhiều (với các phần tử lặp lại);

(d) sử dụng nguyên tắc số không và số một.

▷ **3.11.** Làm các bài tập kiểm tra thực nghiệm cho từng tiểu mục của bài toán trên và so sánh kết quả với lý thuyết.

▷ **3.12.** Tìm ra một biến thể lặp đi lặp lại của sắp xếp nhanh Hoor.

▷ **3.13.** Số phần tử trong phân vùng đã sắp xếp nên sử dụng thuật toán đơn giản như sắp xếp chèn để sắp xếp nhanh?

▷ **3.14.** Chứng minh rằng phần tử càng gần trung vị là phần tử mà sự phân chia hai phần xảy ra trong sắp xếp nhanh Hoor, thuật toán càng hiệu quả.

▷ **3.15.** Để xác định về mặt lý thuyết, cách tốt nhất để chọn một phần tử phân tách trong phân loại nhanh Hoor (xem 3.2.6):

(a) đầu tiên;

b) cuối cùng;

c) ở giữa;

d) trung bình cộng của hai phần tử;

e) trung bình cộng của ba phần tử;

(f) trung vị: phần tử cỡ trung bình trên mỗi cổ phiếu (xem **??**);

(g) một phần tử nhân tạo, ví dụ như trong phân loại bit (xem 3.3.3).

▷ **3.16.** Thực hiện các bài kiểm tra thực nghiệm cho từng tiểu mục của bài toán trên và so sánh kết quả với lý thuyết.

▷ **3.17.** Đề xuất sửa đổi cách sắp xếp Hoor nhanh, điều này sẽ đảm bảo độ phức tạp về thời gian $\Theta(n.\log_2 n)$ trong trường hợp xấu nhất.

▷ **3.18.** Để thay đổi kiểu tham số chính thức của hàm quickSort() từ **int**sang **unsigned**. Sự cố nào xảy ra và tại sao? Bạn đưa ra giải pháp nào?

▷ **3.19.** Đối với mỗi n tự nhiên để tìm một chuỗi đầu vào trong đó sắp xếp nhanh có độ phức tạp:

a) $\Theta(n.\log_2 n)$

b) $\Theta(n^2)$

3.2.7. Phương pháp Thỏ và Rùa

Chúng ta sẽ cố gắng cải thiện hiệu quả khiêm tốn của phương pháp bong bóng (xem 3.2.4) Bằng cách đi theo hướng khác. Như chúng ta đã nhiều lần lưu ý ở trên (xem 3.2.5 và 3.2.6), nhược điểm chính của thuật toán là tính không đối xứng trong hành vi của nó - một phần tử nhẹ từ cuối mảng "xuất hiện" ngay lập tức, trong khi một phần tử nặng duy nhất ở mỗi bước "chìm" xuống một mức duy nhất. Các yếu tố ánh sáng, tức là những yếu tố di chuyển nhanh, sẽ được gọi là thỏ, và những yếu tố chậm - rùa. Các quan sát cho thấy hầu hết mọi chuỗi yếu tố lớn đều chứa một con rùa, và mỗi con rùa dẫn đến độ trễ tối đa.

Làm thế nào để giảm thiệt hại từ rùa? Sự thay đổi về hướng thu thập thông tin mà chúng ta áp dụng cho thuật toán sắp xếp bập bênh (xem 3.2.5) chỉ đơn giản là hoán đổi vai trò của thỏ và rùa. Năm 1991, trong hai bài báo liên tiếp trên tạp chí Byte, Lacey và Box đề xuất một cách tiếp cận triệt để khác - loại bỏ rùa bằng cách cho phép chúng "nhảy" đến vị trí cuối cùng thay vì "bò". Vì mục đích này, so sánh được thực hiện giữa các phần tử ở xa thay vì các phần tử liền kề, như trong bong bóng cổ điển. Khoảng cách giữa các phần tử được so sánh được xác định theo bước giảm dần.

Làm thế nào để chọn bước? Sau khoảng 200.000 lần thử, Lacey và Box đã đi đến kết luận thực nghiệm rằng bước giảm tối ưu (hệ số

co) là 1,3. Khi bạn đến bước 1, thuật toán đã loại bỏ các con rùa và hoạt động như một bong bóng bình thường. Ở các giá trị bước nhỏ hơn 1,3, thuật toán chậm lại do số lượng phép so sánh thừa tăng lên và ở các giá trị cao hơn do va phải một vài con rùa. Hai người cũng đã thực nghiệm phát hiện ra rằng việc phân chia mảng thành các khu vực, mỗi khu vực được xử lý bằng một bước khác nhau, cũng như việc sử dụng hệ số co ngót giảm dần hoặc thay đổi theo cấp số nhân, không dẫn đến cải thiện.

Tuy nhiên, họ đã cố gắng thực hiện một số cải tiến, mà họ xây dựng thành "Quy tắc 11". Ý tưởng là sử dụng 11 thay vì các bước của 9 và 10. Với hệ số co rút gần với mức tối ưu 1,3, bước này có thể được giảm xuống 1 trong ba cách sau:

9 6 4 3 2 1
10 7 5 3 2 1
11 8 6 4 3 2 1

Theo cách thứ ba, các con rùa nhỏ biến mất trước khi bước trở thành 1. Đồng thời, trong hai cách đầu tiên, với xác suất 8%, danh sách vẫn chứa các con rùa nhỏ, do đó thuật toán làm chậm đi 15 -20%.

Thuật toán được mô tả theo cách này rất giống với sắp xếp Shell (xem 3.2.3) - cả hai đều sử dụng hệ số thu nhỏ. Tuy nhiên, không khó để tin rằng sự giống nhau chỉ là bên ngoài. Thuật toán của Shell là một phép chèn (xem 3.2.2) Với bước giảm dần, trong khi phương pháp của thỏ và rùa dựa trên ý tưởng bong bóng (xem 3.2.4). Ngoài ra, Shell thực hiện sắp xếp hoàn chỉnh các danh sách con có liên quan ở mỗi bước.

Do đó một số khác biệt khác. Trong khi đối với Shell hệ số co rút tối ưu là 1,7, đối với "thỏ và rùa" là 1,3. Cuối cùng, độ phức tạp thuật toán của Shell là $\Theta(n.(\log_2 n)^2)$, trong khi ở thỏ và rùa là $\Theta(n.\log_2 n)$ trong cả trường hợp trung bình và xấu nhất. Điều này xếp hạng phương pháp của Lacey và Boxing trong số các phương pháp phân loại hiện đại nhanh nhất. Sau đây là một ví dụ triển khai:

Chương trình 3.10. Thuật toán thỏ và rùa (310combsort.c)

```
void combSort(struct CElem m[], unsigned n)
{ unsigned s, i, j, gap = n;
  do {
```

```
    s = 0;
    gap = (unsigned) (gap/1.3);
    if (gap < 1)
      gap = 1;
    for (i = 0; i < n-gap; i++) {
      j = i + gap;
      if (m[i].key > m[j].key) {
        swap(m+i, m+j);
        s++;
      }
    }
  } while (s != 0 || gap > 1);
}
```

Như một nhược điểm nhất định của việc phân loại theo phương pháp thỏ và rùa, chúng ta có thể chỉ ra công việc với số dấu phẩy động và sử dụng phép toán chia nặng. Thay vào đó, có thể dễ dàng tránh được phép chia bằng cách sử dụng phép nhân. Với mục đích này, đơn đặt hàng:

```
gap = (long) (gap / 1,3);
```

có thể được thay thế bởi:

```
gap = (long) (gap * 0,76923076923);
```

Quá trình chuyển đổi sang các phép toán số nguyên được thực hiện với:

```
gap = gap * 8/11;
```

Chúng ta có thể loại bỏ sự phân chia bằng cách di chuyển xa hơn ra khỏi phạm vi tối ưu, nhưng tăng tốc đáng kể các tính toán:

```
gap = gap * 6 >> 3.
```

Kể từ khi bài báo được xuất bản, một số nỗ lực đã được thực hiện để cải thiện nó, trong đó quan trọng nhất là của Jim Will. Nó gợi ý sử dụng hằng số 1,279604943109628 thay vì 1,3. Trên thực tế, anh ta không sử dụng nó trực tiếp, nhưng đề nghị rằng các bước được cố định trước. Sau đây là trình tự của nó (được sử dụng theo thứ tự giảm dần) cho các mảng có tối đa 4 tỷ phần tử:

```
11, 13, 17, 23, 29, 37, 47, 61, 79, 103, 131, 167, 216, 277, 353, 449
    , 577, 739, 947, 1213, 1553, 1987, 2543, 3259, 4166, 5333, 6829,
```

8741, 11177, 14310, 18313, 23431, 29989, 38371, 49103, 62827, 80407, 102881, 131648, 168463, 215573, 275840, 352973, 451669, 577957, 739560, 946346, 1210949, 1549547 253720 , 2147483647

Bài tập

▷ **3.20.** Chứng minh rằng phương pháp của thỏ và rùa có độ phức tạp $\Theta(n.\log_2 n)$ cả ở giữa và trong trường hợp xấu nhất.

▷ **3.21.** Để so sánh về mặt lý thuyết và thực nghiệm thuật toán Shell (xem 3.2.3) Và phương pháp của thỏ và rùa.

3.2.8. Sắp xếp theo lựa chọn trực tiếp

Một phương pháp sắp xếp cơ bản khác có độ phức tạp $\Theta(n^2)$ là phương pháp chọn trực tiếp hay còn gọi là phương pháp chọn trực tiếp. Mảng được chia thành phần được sắp xếp và phần chưa được sắp xếp, và ở mỗi bước của thuật toán, vùng được sắp xếp sẽ được mở rộng sang bên phải một phần tử. Bước đầu tiên là phần tử tối thiểu của mảng và được trao đổi với phần đầu tiên. Trong bước thứ hai, phần tử tối thiểu khác được trao đổi với phần tử thứ hai của mảng, v.v. Ở mỗi bước tiếp theo, phần tử nhỏ nhất của phần chưa được sắp xếp sẽ hoán đổi với phần tử đầu tiên của phần chưa được sắp xếp (lưu ý rằng nó lớn hơn hoặc bằng mỗi phần tử của phần chưa được sắp xếp, tại sao?), Do đó mở rộng phần được sắp xếp trong khi nó không bao gồm toàn bộ mảng.

Chương trình 3.11. Sắp xếp lựa chọn trực tiếp (311selsort.c)

```
void straightSelection(struct CElem m[], unsigned n)
{ unsigned i, j;
  for (i = 0; i < n-1; i++)
    for (j = i+1; j <= n; j++)
      if (m[i].key > m[j].key)
        swap(m+i, m+j);
}
```

Một phiên bản sửa đổi một chút của việc triển khai ở trên cũng phổ biến: Trong chương trình tiếp theo, sự khác biệt là trong vòng lặp bên trong, địa chỉ của phần tử thứ j là $m[j]$ được lưu và vì mục đích này, biến x là đã giới thiệu.

Chương trình 3.12. Sắp xếp lựa chọn trực tiếp (312selsort2.c)

```
void straightSelection(struct CElem m[], unsigned n)
{ unsigned i, j, ind;
  struct CElem x;
  for (i = 0; i < n - 1; i++)
    for (x = m[ind = i], j = i + 1; j < n; j++)
      if (m[j].key < x.key) {
        x = m[ind = j];
        m[ind] = m[i];
        m[i] = x;
      }
 }
```

Chọn trực tiếp	Số tối thiểu	Số trung bình	Số tối đa
So sánh	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
Trao đổi	$3(n-1)$	$n \ln n$	$\lceil n^2/4 \rceil + 3(n-1)$

Bảng 3.3. So sánh và trao đổi trong sắp xếp theo cách chọn trực tiếp.

Tương tự như sắp xếp bong bóng, thuật toán lựa chọn trực tiếp luôn thực hiện $n(n-1)/2$ phép so sánh, như trong Bảng 3.3. Nói chung, thuật toán lựa chọn trực tiếp thích hợp hơn phương pháp bong bóng, mặc dù trong trường hợp đặc biệt của một mảng được sắp xếp trước hoặc sắp xếp gần hết, bong bóng cho kết quả tốt hơn.

Bài tập

▷ **3.22.** Chứng minh rằng trong lựa chọn trực tiếp sắp xếp, mỗi phần tử của phần không được sắp xếp lớn hơn hoặc bằng mỗi phần tử của phần đã sắp xếp.

▷ **3.23.** Để so sánh về mặt lý thuyết và thực nghiệm hai phương án sắp xếp lựa chọn trực tiếp.

▷ **3.24.** Trích xuất các công thức từ Bảng 3.3.

3.2.9. Sắp xếp kim tự tháp Williams

Sắp xếp theo lựa chọn trực tiếp (xem 3.2.8) Xếp hạng trong số những cách kém hiệu quả nhất, chẳng hạn như sắp xếp theo bong

bóng (xem 3.2.4) Và chèn trực tiếp (xem 3.2.2). Hạn chế chính của nó là ở bước thứ i, luôn luôn thực hiện chính xác n-i so sánh, bất kể dữ liệu đầu vào là gì.

Trên thực tế, thuật toán tìm phần tử nhỏ nhất của mảng n phần tử yêu cầu chính xác $n - 1$ phép so sánh. Thật vậy, trong mỗi phép so sánh có đúng một ứng cử viên bị loại, và để xác định yếu tố tối thiểu, cần loại bỏ đúng $n - 1$ ứng viên.

Do đó, trong bước đầu tiên của một thuật toán tùy ý dựa trên sự lựa chọn, $n - 1$ phép so sánh sẽ được thực hiện và kết quả này, theo các cân nhắc ở trên, không thể cải thiện được. Thoạt nhìn, có vẻ như bước thứ hai sẽ yêu cầu thêm $n - 2$ phép so sánh. Suy nghĩ một cách linh hoạt, chúng ta nhận thấy rằng ở bước thứ i, chúng ta sẽ cần chính xác $n - i$ so sánh. Chúng ta không thể cải thiện đánh giá này? Người đọc chú ý chắc hẳn đã nhận thấy rằng bước đầu tiên và bước thứ hai của thuật toán không hoàn toàn bằng nhau. Thật vậy, kết quả của việc áp dụng bước đầu tiên, chúng ta không chỉ thu được phần tử tối thiểu của mảng mà còn có một số mối quan hệ giữa các cặp giữa các phần tử khác mà chúng ta có thể lưu trữ và sử dụng trong bước thứ hai. Rõ ràng là chúng ta càng nhận được nhiều tỷ lệ bổ sung, thì chúng ta sẽ cần ít so sánh hơn trong các bước tiếp theo, tức là cây so sánh nhị phân càng thấp và càng nhiều nhánh (xem 3.2.1) thì càng mang nhiều thông tin hơn.

Có thể thu được một cây so sánh tốt bằng cách sử dụng cơ chế giải đấu loại trừ (xem [Knuth-3/1968], [Reingold, Nivergelt, Deo-1980]), cơ chế này xây dựng cây so sánh từ lá đến gốc. Trong vòng đầu tiên của giải đấu, các trận đấu $x_1 : x_2, x_3 : x_4, ..., x_{n-1} : x_n$ được diễn ra. Ở vòng thứ hai, các cặp thắng ở vòng trước sẽ đấu, v.v. Trong trận chung kết, hai người tham gia gặp nhau và xác định nhà vô địch của giải đấu. Nếu trong vòng thứ i của giải đấu mà số người tham gia là số lẻ (và điều này sẽ xảy ra ít nhất một lần, nếu n không phải là bậc 2, tại sao?), Thì một trong những người chơi chết, tự động đủ điều kiện cho người tiếp theo vòng. Sử dụng chiến lược đã chỉ ra để xác định phần tử tối thiểu của một mảng, chúng ta nhận được: Trong vòng đầu tiên của giải đấu với $n/2$ phép so sánh, phần tử tối thiểu cho mỗi cặp được xác định và với $n/4$ phép so sánh, phần tử tối thiểu giữa hai cặp là xác định (ví dụ: e. 4 phần tử), v.v.

Rõ ràng, thuật toán được trình bày xây dựng cây của giải, tức là xác định phần tử nhỏ nhất của mảng với chính xác $n-1$ phép so sánh. Bây giờ, thay phần tử nhỏ nhất (nhà vô địch) bằng $-\infty$ trong trang tính tương ứng của nó và tính toán lại nội dung của các đỉnh trên đường lên đỉnh, chúng ta nhận được phần tử lớn nhất tiếp theo ở trên cùng của cây. Vì cây có chiều cao là $\lceil \log_2 n \rceil$, nên quy trình được mô tả để xác định "về nhì", tức là phần tử lớn thứ hai của mảng, yêu cầu $\lceil \log_2 n \rceil - 1$ so sánh thay vì $n-2$, như Do đó, quá trình sắp xếp mảng hoàn chỉnh yêu cầu không quá $n-1+(n-1)(\lceil \log_2 n \rceil - 1)$ phép so sánh.

Mặc dù thuật toán được trình bày đại diện cho một sự cải tiến thực sự nghiêm trọng liên quan đến thuật toán lựa chọn trực tiếp, nhưng vẫn còn nhiều điều mong muốn. Trước hết, một phương pháp hiệu quả nên được xác định để giảm thiểu số lượng trao đổi. Nhớ lại rằng trao đổi là một hoạt động chậm hơn nhiều so với so sánh và không nên đánh giá thấp vấn đề: chính số lượng trao đổi quá nhiều đã ngăn cản quá trình phân loại rung chuyển (xem 3.2.5) đạt được hiệu quả tiệm cận tốt hơn về phân loại bong bóng (xem **??**), mặc dù số lượng so sánh được thực hiện giảm đi rất nhiều. Một nguồn bất tiện khác là các giá trị của nó, dẫn đến việc so sánh không cần thiết, cuối cùng lấp đầy toàn bộ cây. Hơn nữa, thuật toán giải đấu loại trừ, mặc dù nó xây dựng một cây cân bằng hoàn hảo (xem 2.4), không tạo ra một cây mà về mặt lý thuyết có thể mong đợi kết quả tối ưu, vì nó cho phép các lá xuất hiện ở mọi cấp độ. Cuối cùng nhưng không kém phần quan trọng, những người tham gia giải đấu nằm trong danh sách, với những ngọn cây so sánh chỉ sao chép chúng. Điều tự nhiên là muốn giảm thiểu dung lượng bộ nhớ cần thiết, trong thuật toán của giải đấu với loại bỏ có bậc là $2n-1$. Với mục đích này, chúng ta sẽ cần một cơ chế hiệu quả để tuyến tính hóa cây, cho phép chúng ta chỉ giữ lại những chiếc lá mà không làm mất đi các mối quan hệ đã biết giữa chúng. J. Williams đưa ra giải pháp hiệu quả cho tất cả các vấn đề được nêu ra thông qua cấu trúc kim tự tháp

Định nghĩa 3.1. Ta sẽ gọi một hình chóp là một cây nhị phân cân bằng hoàn hảo với chiều cao h, đồng thời thỏa mãn các điều kiện sau:

1) tất cả các lá ở mức h và $h-1$;

2) tất cả những người thừa kế của một đỉnh nhỏ hơn nó;

3) tất cả những người thừa kế cấp h đều bị lệch cực đại sang trái.

Điều thú vị ở đây là kim tự tháp cho phép hiện thực hóa hiệu quả không chỉ qua gỗ mà còn thông qua mảng n phần tử, vì các kết nối di truyền giữa các phần tử được định nghĩa là một hàm tuyến tính đơn giản của vị trí của chúng. Phát biểu sau đây cung cấp cho chúng ta một cơ chế cụ thể để thực hiện hiệu suất được mô tả:

Mệnh đề 3.1. *Hình chóp tương đương với dãy khóa* $h_l, h_{l+1}, ..., h_r (1 \leq l \leq r \leq n)$, *cho mà* $h_i \geq h_{2i}$ *và* $h_i \geq h_{2i+1}, i = l, l+1..., r/2$.

Rõ ràng rằng đỉnh h_1 của hình chóp $h_1, h_2, ..., h_n$ chứa phần tử lớn nhất của nó. (Tại sao?) Trên thực tế, câu lệnh được cố ý đưa ra dưới dạng tổng quát hơn, dưới đây sẽ nói rõ tại sao. Đặc biệt, tại $l = 1$, sự tương đương là hiển nhiên. Chúng ta sẽ chỉ lưu ý rằng h_{2i} và h_{2i+1} lần lượt là người thừa kế bên trái và bên phải của h_i.

Giả sử rằng chúng ta có các thuật toán hiệu quả để xây dựng một kim tự tháp buildHeap(), cũng như để khôi phục một kim tự tháp từ phần tử k, restoreHeap(k), sau khi đỉnh của nó đã được thay thế bằng một phần tử tùy ý. Sau đó, sau khi xây dựng một hình chóp có n phần tử, chúng ta có thể hoán đổi đỉnh h_1 (phần tử lớn nhất của nó) với phần tử cuối cùng h_n, với đỉnh cũ lấy vị trí cuối cùng của nó: cuối cùng trong mảng đã sắp xếp. Vì hoạt động này phá hủy điều kiện của Assertion, nên cần phải sàng lọc đỉnh mới xuống kim tự tháp. Kết quả là ta thu được một hình chóp mới $h_1, h_2, ..., h_{n-1}$, chứa ít hơn một phần tử. Tiếp theo là một trao đổi mới của đỉnh của hình chóp với phần tử cuối cùng $(n-1)$ của nó, tiếp theo là một sàng lọc mới. Quá trình được mô tả được lặp lại $n-1$ lần (không cần đến bước thứ n, vì cuối cùng phần tử thứ n đã ở đúng vị trí, tại sao?). Chúng ta nhận được thuật toán sau để sắp xếp kim tự tháp của một mảng:

```
buildHeap();
for (i = n; i >= 2; i--) {
  swap(m+1, m+i);
  restoreHeap(i-1)
}
```

Làm thế nào để xây dựng kim tự tháp? Bây giờ, hãy giả sử rằng n là số chẵn. Sau đó, theo Mệnh đề cho các phần tử $h_n/2+1, h_n/2+2, ..., h_n$, không yêu cầu tỷ lệ bậc nhất vì không có hai phần tử i và j nào được thỏa mãn $j = 2i$ hoặc $j = 2i+1$. Thật vậy, các yếu tố được đề cập (và chỉ chúng!) Là lá của cây (nằm ở tầng cuối cùng hoặc có thể là áp chót) và không bắt buộc phải thỏa mãn bất kỳ tỷ lệ nào. Hãy mở rộng kim tự tháp bên trái một phần tử. Phần tử x mới được thêm vào sẽ ở mức áp chót và sẽ có ít nhất một phần tử kế nhiệm. Điều này đòi hỏi phải kiểm tra để xem liệu các mối quan hệ trong Yêu cầu bồi thường có bị vi phạm hay không, sau đó là trao đổi x với một trong hai người thừa kế của anh ta. Bây giờ các phần tử $h_n/2, h_n/2+1, ..., h_n$ lại tạo thành một hình chóp. Tiếp theo là sự mở rộng mới của kim tự tháp bên trái, và quá trình này tiếp tục cho đến khi tất cả các yếu tố được bao gồm. Đối với n lá lẻ là $h_n/2+1, h_n/2+2, ..., h_n$. Trong thực tế, trong việc triển khai C không cần phải phân biệt giữa hai trường hợp, vì phép toán / là một số nguyên.

Chúng ta sẽ lưu ý rằng việc sàng lọc phần tử mới được thêm vào x xuống kim tự tháp có thể diễn ra trong nhiều giai đoạn, tức là sau khi trao đổi x với một trong những phần tử kế nhiệm của nó, nó có thể trở lại rằng x không đáp ứng các điều kiện của Mệnh đề, v.v. Vì vậy, x được sàng xuống kim tự tháp, chìm xuống một bậc ở mỗi bậc cho đến khi vị trí của nó. Một cải tiến rõ ràng có thể được thực hiện ở đây, đẩy nhanh quá trình, cụ thể là trao đổi x với nhỏ hơn của hai người kế nhiệm, mặc dù điều này không phải lúc nào cũng có lợi. Ngoài ra, việc trao đổi x với những người kế nhiệm của nó có thể bị hoãn lại cho đến khi tìm được vị trí chính xác của nó và thay vào đó, các bài toán một chiều được thực hiện trong quá trình tìm kiếm vị trí cuối cùng của nó.

Lưu ý rằng cả việc xây dựng và trùng tu đều yêu cầu sàng lọc phần tử ở đỉnh kim tự tháp xuống kim tự tháp. Điều này dẫn chúng ta đến ý tưởng triển khai một hàm sàng lọc sift () chung:

Sàng lọc phần tử (trong 313heapsort.c)

```
/ * Sàng lọc phần tử từ trên xuống của kim tự tháp * /
void sift (struct CElem m[], unsigned l, unsigned r)
{ unsigned i = l, j = i + i;
```

```
  struct CElem x = m[i];
  while (j  <= r)  {
    if (j  < r  && m[j].key < m[j+1].key)
      j++;
    if (x.key >= m[j].key)
      break;
    m[i] = m[j];
      i  = j;
      j  <<= 1; /*tương đương với j*=2; */
  }
  m[i] = x;
}
```

Bây giờ việc xây dựng kim tự tháp (hàm buildHeap()) có thể được thực hiện bằng cách sử dụng đoạn chương trình sau:

```
for (k = n/2 + 1; k > 1; k--) {
   sift (m,k-1,n);
}
```

Để sắp xếp nó, liên tục hoán đổi đỉnh của kim tự tháp với phần tử cuối cùng của nó, tiếp theo là sàng lọc (chức năng restoreHeap()), chúng ta nhận được như sau:

```
for (k = n; k > 1; k--) {
  swap(m+1,m+k);
   sift (m,1,k-1);
}
```

Cuối cùng, để sắp xếp hình chóp, chúng ta nhận được:

Chương trình 3.13. Sắp xếp kiểu kim tự tháp (313heapsort.c)

```
void heapSort(struct CElem m[], unsigned n) /*Sắp xếp kim tự tháp
    */
{ unsigned k;
  /* 1.Xây dựng kim tự tháp*/
  for (k = n/2 + 1; k > 1; k--) {
     sift (m,k-1,n);
  }
  /* 2. Xây dựng một chuỗi được sắp xếp*/
  for (k = n; k > 1; k--) {
    swap(m+1,m+k);
```

```
        sift (m,1,k-1);
    }
}
```

Bài tập

▷ **3.25.** Chứng minh rằng trong một giải đấu quần vợt loại trực tiếp, nếu số người tham gia không phải là cấp 2 thì ít nhất trong một hiệp sẽ phải có một đấu thủ nghỉ.

▷ **3.26.** Chứng minh rằng đỉnh của hình chóp chứa phần tử lớn nhất của nó.

▷ **3.27.** Phát triển một phương pháp sắp xếp bằng cách sử dụng một kim tự tháp bậc ba. Kim tự tháp bậc ba là một bản tóm tắt của kim tự tháp cổ điển và dựa trên các cây bậc ba (ba ngôi) hoàn chỉnh. Chúng ta có nên mong đợi gia tốc so với kim tự tháp cổ điển?

▷ **3.28.** Để phát triển một thuật toán để nhanh chóng hợp nhất hai kim tự tháp.

3.2.10. Độ phức tạp về thời gian tối thiểu của việc sắp xếp theo cách so sánh

Tất cả các thuật toán được xem xét cho đến nay đều thuộc loại thuật toán sắp xếp so sánh, tức là những thuật toán trong đó phép toán duy nhất được phép là so sánh giữa các cặp phần tử sử dụng các phép toán $>$ $(\geq), <$ $(\leq)$ và $=$ $(\neq)$. Đơn giản nhất trong số chúng (ví dụ: chèn, bong bóng hoặc lựa chọn trực tiếp, xem 3.2.2, 3.2.4 và 3.2.8) thực hiện $\Theta(n^2)$ số phép so sánh, trong khi phép so sánh tốt nhất có độ phức tạp $\Theta(n.\log_2 n)$ trong ở giữa (phân loại nhanh, xem 3.2.6) hoặc thậm chí trong trường hợp xấu nhất (phân loại theo hình chóp: xem 3.2.9, phương pháp thỏ và rùa: xem 3.2.7 và phân loại theo hợp nhất, sẽ được xem xét trong 7.3.) . Đối với mỗi thuật toán như vậy, một trình tự đầu vào có thể được tìm thấy để đạt được các ước tính này, tức là các thuật toán sắp xếp tốt nhất mà chúng ta biết cho đến nay có độ phức tạp thuật toán $\Theta(n.\log_2 n)$ Có thể không cải thiện đánh giá này? câu trả lời cho câu hỏi này như sau:

Mệnh đề 3.2. *Mỗi thuật toán sắp xếp so sánh thực hiện một số phép so sánh bậc của* $\Theta(n.\log_2 n)$.

Bằng chứng. Đối với mỗi thuật toán sắp xếp so sánh, một cây so sánh nhị phân tương ứng có thể được so sánh duy nhất, tương tự như trong Hình 3.1. Những chiếc lá là $n!$ theo số lượng và mỗi trong số chúng chứa chính xác một trong các hoán vị của chuỗi đầu vào. Các tờ rơi, đến lượt nó, chứa các so sánh giữa một cặp yếu tố. Không giới hạn cộng đồng, chúng ta có thể giả định rằng tất cả các so sánh đều thuộc loại $<$. Thật vậy, với mỗi quan hệ $=, \neq, \leq, <, >$ và $\geq$ chúng ta có rằng nó hợp lệ hoặc không, nghĩa là mỗi phép so sánh như vậy có hai đầu ra, tương ứng với hai người thừa kế trong cây. Việc thực thi thuật toán sắp xếp phụ thuộc vào việc tìm đường dẫn trong cây nhị phân. Việc tìm kiếm bắt đầu từ gốc và kết thúc khi đạt đến lá. Hoán vị có trong trang tính xác định thứ tự cần thiết của chuỗi đầu vào.

Như đã đề cập ở trên, độ phức tạp của thuật toán sắp xếp ở mức tồi tệ nhất được xác định bởi số lượng phép so sánh tối đa mà nó thực hiện, tức là độ dài của đường dẫn tối đa từ gốc đến một trong các lá. Để sắp xếp đúng mỗi dãy đầu vào của n phần tử, thuật toán phải có kết quả là mỗi trong số n có thể! hoán vị, tức là mỗi hoán vị phải xuất hiện dưới dạng một lá trong cây so sánh của nó ít nhất một lần.

Gọi h là chiều cao của cây (tức là chiều dài của con đường tối đa từ gốc đến một trong các lá của nó). Bất đẳng thức là hợp lệ:

$$n! \leq 2^h.$$

Từ đây sau khi tính logarit ta được: $h \geq \log_2(n!)$

Để logarit $n!$, chúng ta sẽ sử dụng xấp xỉ Stirling $n! > (n/e)^n$, nơi chúng ta nhận được:

$$h \geq \log_2(n!) \geq \log_2(n/e)^n = n.\log_2 n - n.\log_2 e,$$

khi nó trực tiếp theo sau $h \in \Omega(n.\log_2 n)$. Nghĩa là, chiều cao của cây, và do đó số phép so sánh bắt buộc tối thiểu, là $\Omega(n.\log_2 n)$.

Ngay sau phát biểu trên rằng phân loại theo hình chóp, phương pháp của thỏ và rùa và sắp xếp theo tiệm cận là tối ưu về mặt tiệm

cận. Ở đây, chúng ta sẽ bảo lưu rằng trong một số điều kiện hạn chế mạnh hơn (chủ yếu liên quan đến việc phân phối các giá trị khóa và / hoặc đại diện bên trong của chúng) thì sẽ có các thuật toán hiệu quả hơn, một số thuật toán trong số đó chúng ta sẽ xem xét trong đoạn 3.2 tiếp theo.

3.2.11. Bài tập

▷ **3.29.** Chứng minh rằng mọi tập hợp con của một tập hợp có pháp lệnh tuyến tính cũng được sắp xếp theo thứ tự tuyến tính.

3.3. Sắp xếp theo sự biến đổi

Có một loại thuật toán rộng khác không dựa trên so sánh mà dựa trên việc thực hiện các phép toán (chủ yếu là số học) trên các phần tử của tập hợp - *sắp xếp theo phép biến đổi*. Sự khác biệt giữa hai cách tiếp cận có thể so sánh với sự khác biệt giữa tìm kiếm nhị phân (xem 2.3) và băm (xem 2.5). Thông thường, các thuật toán sắp xếp chuyển đổi chỉ có thể áp dụng trong một số trường hợp cụ thể dưới các điều kiện hạn chế bổ sung liên quan đến loại và / hoặc tập hợp các giá trị cho phép và / hoặc số lần xuất hiện chính của các phần tử được sắp xếp.

3.3.1. Sắp xếp theo tập hợp

Đại diện đầu tiên của loại thuật toán sắp xếp theo phép biến đổi, mà chúng ta sẽ xem xét, là thuật toán sắp xếp theo tập hợp. Nó có thể áp dụng cho một tập hợp có hàm thứ tự f thỏa mãn đồng thời các điều kiện sau (Ta ký hiệu tập hợp là M và số phần tử của nó là n):

1) các giá trị của f là các số tự nhiên trong khoảng $[a, b]$ đã cho, chứa $m = b - a + 1$ số nguyên. Không phải mọi số nguyên từ $[a, b]$ đều phải là giá trị của f, nhưng mọi giá trị của f phải từ $[a, b]$.

2) f là đơn ánh, nghĩa là với $x_1, x_2 \in M$ và $x_1 \neq x_2$, chúng ta có $f(x_1) \neq f(x_2)$ (tức là không lặp lại).

Để đơn giản hóa thuật toán, chúng ta sẽ không làm việc với các phần tử có kiểu `struct` CElem, nhưng chúng ta sẽ sử dụng kiểu `unsigned`, giả sử rằng giá trị của phần tử lớn nhất không vượt quá

một hằng số hợp lý MAX_VALUE. (Trên thực tế, MAX_VALUE là ký hiệu chương trình của m) Đây là cách chúng ta sẽ làm việc với khoảng $[0, MAX_VALUE)$. Việc sắp xếp được thực hiện một cách tuyến tính bằng cách chuyển một qua các phần tử của mảng, sau đó chuyển qua các số trong khoảng. Lúc đầu, chúng ta khởi tạo với 0 các phần tử của set[] (kiểu char, chúng ta sẽ coi là Boolean), sau đó chúng ta đi qua mảng và với mỗi phần tử của nó, chúng ta đặt 1 phần tử tương ứng của tập . Trong bước thứ hai, chúng ta kiểm tra tuần tự tất cả các số trong khoảng thời gian xem chúng có thuộc set[] hay không. Ở đầu lần vượt qua thứ hai, chúng ta coi rằng m[] không chứa bất kỳ phần tử nào và chúng ta xây dựng lại nó, lần này là một dãy đã được sắp xếp. Các chi tiết có thể được nhìn thấy từ đoạn chương trình đính kèm.

Chương trình 3.14. Sắp xếp theo tập hợp (314setsort.c)

```
void setSort(unsigned m[],unsigned n)
{ char set[MAX_VALUE];
unsigned i,j;

/* 0.Khởi tạo tập hợp*/
for (i = 0; i < MAX_VALUE;i++)
set[i] = 0;

/* 1. Sự hình thành của tập hợp*/
for (j = 0; j < n; j++) {
assert(m[j] >= 0 && m[j] < MAX_VALUE);
assert(0 == set[m[j]]);
set[m[j]] = 1;
}

/* 2.Tạo trình tự đã sắp xếp */
for (i = j = 0; i < MAX_VALUE;i++)
if (set[i])
m[j++] = i;
assert(j == n);
}
```

Độ phức tạp của thuật toán trên là $\Theta(m+n)$ và đối với các giá trị của n gần với m, nó có thể được coi là tuyến tính đối với n. Cần lưu

ý rằng điều này không phải là "miễn phí": Một mặt, chúng ta có một giả định bổ sung nặng nề về các giá trị và loại khóa của các phần tử được sắp xếp (ví dụ: chúng ta không thể sắp xếp các số thực), và mặt khác - chúng ta sử dụng bộ nhớ bổ sung của $\Theta(m)$. Chúng ta sẽ lưu ý rằng hầu hết các thuật toán được thảo luận ở trên đều sử dụng bộ nhớ bổ sung không đổi. Một số thuật toán đệ quy, chẳng hạn như sắp xếp nhanh (xem 3.2.6), yêu cầu bộ nhớ logarit và trong trường hợp xấu nhất là thậm chí bộ nhớ tuyến tính (điều này bị ẩn đằng sau cơ chế đệ quy). Tuy nhiên, có một yêu cầu lớn hơn ở đây: không phải $\Theta(n)$, còn $\Theta(m)$, vì $m > n$, và $m >> n$ là có thể (tức là m có thể lớn hơn n nhiều).

Chúng ta có thể tóm tắt thuật toán, mở rộng phạm vi của nó. Chúng ta sẽ "thoát khỏi" sự hạn chế khó chịu của việc chỉ làm việc với các số nguyên. Lần này chúng ta sẽ sử dụng các phần tử kiểu `struct` CElem mà khóa thỏa mãn hai điều kiện trên. Độ phức tạp của thuật toán này một lần nữa sẽ là $\Theta(m+n)$, trong đó m là số nguyên trong khoảng $[a,b]$.

Lần này khi tổ chức tập hợp, chúng ta sẽ phải giữ thông tin không chỉ về việc một số từ $[a,b]$ có xảy ra như một khóa của một phần tử của m[] hay không, mà còn cả giá trị của phần tử này. Nhìn chung, có hai cách tiếp cận về vấn đề này. Trong cách tiếp cận đầu tiên, chỉ số của phần tử tương ứng trong m[] được giữ trong tập hợp:

```
struct CSetEl {
  char found;
  unsigned index;
} m[MAX];
```

Trường tìm được có thể được lưu bằng cách giả sử rằng phần tử rỗng của m[] không được sử dụng. Số không có nghĩa là không có phần tử có khóa như vậy và bất kỳ giá trị nào khác không sẽ là chỉ số trong m[] của phần tử có khóa tương ứng:

```
unsigned m[MAX];
```

Nhân tiện, từ quan điểm của ngôn ngữ của chúng ta, tốt hơn là sử dụng cho giá trị chính thức không phải là 0, mà ngược lại - giá trị lớn nhất của kiểu không dấu. Đây là một giá trị khác nhau cho các nền tảng khác nhau (ví dụ: trong DOS là 65,535 và trong Windows

là 4,294,967,295). Chúng ta sẽ sử dụng cách tiếp cận được đề cập trong Chương 1 để lấy nó - dựa trên sự biểu diễn bên trong của các số trong mã bổ sung: `(unsigned)(-1)` .

Ưu điểm của hai khai báo trên là tính kinh tế, trong điều kiện bộ nhớ được sử dụng, không phải các phần tử của m[] được bảo toàn, có thể là các bản ghi kích thước lớn, nhưng chỉ các chỉ số của chúng. Lưu ý hai (các) xác nhận bảo vệ chúng ta khỏi các giá trị khóa trùng lặp và không hợp lệ.

Chương trình 3.15. Sắp xếp theo tập hợp (315setsort2.c)

```
#define NO_INDEX (unsigned)(-1)
/*Sắp xếp một mảng bằng cách sử dụng tập hợp*/
void setSort(struct CElem m[], unsigned n)
{ unsigned indSet[MAX_VALUE]; /*Bộ chỉ mục*/
  unsigned i,j;

  /* 0. Khởi tạo tập hợp*/
  for (i = 0; i < MAX_VALUE; i++)
    indSet[i] = NO_INDEX;

  /* 1.Sự hình thành của bộ*/
  for (j = 0; j < n; j++) {
      assert(m[j].key >= 0 && m[j].key < MAX_VALUE);
      assert(NO_INDEX == indSet[m[j].key]);
    indSet[m[j].key] = j;
  }

  /* 2.Tạo trình tự đã sắp xếp*/
  for (i = j = 0; i < MAX_VALUE; i++)
    if (NO_INDEX != indSet[i])
      do4Elem(m[indSet[i]]);
}
```

Chương trình đính kèm giải quyết một phần vấn đề. Nó nhận được chuỗi các phần tử đã được sắp xếp, nhưng không có trong mảng m[]. Thay vào đó, nó gọi hàm `do4Elem()` cho mỗi mục tiếp theo trong chuỗi đã sắp xếp. Điều này có thể chấp nhận được nếu chúng ta muốn thực hiện một thao tác trên các phần tử một lần. Ví dụ: để hiển thị chúng ở dạng đã sắp xếp.

Việc duy trì trình tự đã sắp xếp sẽ hủy bỏ m[], và do đó các chỉ số tương ứng. Một cách khả thi là lưu trước một bản sao của m[]. Chúng ta có thể tự giải quyết rắc rối khi duy trì một bản sao bằng cách giữ các phần tử tương ứng của m[] trong tập hợp. Tất nhiên, phần sau có liên quan đến sự lãng phí bộ nhớ bổ sung - một mục nhập kiểu `struct` CElem cho mỗi số trong khoảng [a, b]. Có tính đến kích thước của các bản ghi đã sắp xếp là cố định, chúng ta nhận được rằng kích thước của bộ nhớ bổ sung cần thiết một lần nữa là $\Theta(m)$, lần này đằng sau ký hiệu $\Theta(...)$ có một hằng số lớn hơn.

Bài tập

▷ **3.30.** Sửa đổi việc triển khai nhiều sắp xếp được đề xuất để thu được trình tự đã sắp xếp trong mảng đầu ra.

3.3.2. Sắp xếp theo số đếm

Chúng ta hãy cố gắng phát triển hơn nữa ý tưởng về đa phân loại. Nhược điểm chính của phương pháp này là yêu cầu về tính duy nhất của khóa, tức là không được phép lặp lại. Giả sử chúng ta muốn sắp xếp các phần tử của một tập hợp các số tự nhiên từ một khoảng cho trước, mỗi phần tử có thể xảy ra không quá k lần. Ở đây các phần tử của mảng không thuộc kiểu bản ghi và vai trò của khóa được đảm nhận bởi danh tính, tương tự như trường hợp sắp xếp các số nguyên theo một tập hợp. Chúng ta định nghĩa một mảng cnt[], trong đó cnt[i] chứa số lần xuất hiện của số i. Chúng ta bắt đầu bằng cách đặt lại các phần tử của cnt[], sau đó đi qua các phần tử của m[] và tìm số lần xuất hiện cho mỗi số nguyên của $[a, b]$. Trong bước thứ hai, chúng ta đi qua các phần tử của cnt[], bao gồm trong dãy đã sắp xếp cnt $[x]$ số x, với mỗi số nguyên $x \in [a, b]$.

Chương trình 3.16. Sắp xếp theo số đếm (316counts.c)

```
void countSort(unsigned m[], unsigned n) /* Sắp xếp theo số đếm */
{ unsigned char cnt[MAX_VALUE];
unsigned i,j;
/* 0.Khởi tạo tập hợp*/
for (i = 0; i < MAX_VALUE; i++)
cnt[i] = 0;
/* 1.Sự hình thành của bộ*/
```

```
for (j = 0; j < n; j++) {
assert(m[j] >= 0 && m[j] < MAX_VALUE);
cnt[m[j]]++;
}
/* 2. Tạo trình tự đã sắp xếp */
for (i = j = 0; i < MAX_VALUE; i++)
while (cnt[i]--)
m[j++] = i;
assert(j == n);
}
```

Thuật toán trên hoạt động tốt khi giả sử rằng mỗi số xảy ra không quá k lần, đối với một hằng số k xác định trước hợp lý. Về mặt lý thuyết, lĩnh vực ứng dụng của thuật toán trên vẫn bị giới hạn bởi các yêu cầu sau:

- các phần tử phải nằm trong khoảng $[a, b]$, và không phải mọi số từ $[a, b]$ đều phải là giá trị của f, mà tất cả các giá trị của f phải từ $[a, b]$.
- Mỗi số $[a, b]$ xảy ra không quá k lần;
- Chỉ sắp xếp các số, không phải các mục nhập ngẫu nhiên.

Hai hạn chế đầu tiên của thuật toán trên là tự nhiên. Nhân tiện, những hạn chế như vậy là đặc trưng của hầu hết các thuật toán sắp xếp theo phép biến đổi. Tuy nhiên, yêu cầu thứ hai không quá cơ bản và chúng ta có thể dễ dàng loại bỏ nó. Vấn đề chính là cần phải giữ thông tin không chỉ về việc một số nguyên x của $[a, b]$ có xảy ra như một khóa của một phần tử của M hay không, mà còn là giá trị của phần tử này. Chúng ta giải quyết vấn đề tương đối dễ dàng: chúng ta sẽ giữ thông tin về việc khóa x có xảy ra hay không và ngoài ra - phần tử tương ứng với khóa x.

Khi sắp xếp bằng cách đếm cho mỗi giá trị cho phép của khóa, có thể có một số yếu tố liên quan, điều này làm phức tạp vấn đề: cần phải tổ chức một danh sách. Vì số lần xuất hiện của một khóa không vượt quá k nên mỗi danh sách như vậy có thể chứa nhiều nhất k phần tử. Tuy nhiên, việc cấp phát bộ nhớ tĩnh có kích thước k cho mỗi khóa p dẫn đến sự lãng phí không cần thiết.

Trong chương trình dưới đây, mỗi khóa được liên kết với một danh sách động chứa các phần tử tương ứng của M. Việc duy trì

một con trỏ duy nhất ở đầu danh sách có thể là một nguồn tiềm ẩn của các vấn đề, vì nó làm cho việc sắp xếp không bền vững (danh sách không được liên kết). Do đó, các phần tử có cùng khóa rơi vào trình tự ban đầu theo thứ tự ngược lại với thứ tự mà chúng có trong M. Phần tử sau có thể dễ dàng được sửa chữa, ví dụ bằng cách thêm một con trỏ bổ sung vào cuối danh sách (hoặc để báo cáo sau), khi tập hợp dãy đã sắp xếp: lật tất cả các danh sách). Độ phức tạp của thuật toán lại là $\Theta(m+n)$.

Chương trình 3.17. Sắp xếp theo số đếm (317counts2.c)

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 100
struct CElem {
  int key;
  /* ..............Một số dữ liệu .............. */
};
struct CList {
  struct CElem data;
  struct CList *next;
};

struct CList* init(unsigned n) /* Điền vào mảng với các số ngẫu nhi
    ên*/
{ struct CList *head, *p;
  unsigned i;
  srand(time(NULL));
  for (head = NULL, i = 0; i  < n;  i++) {
    p = (struct CList *)  malloc(sizeof(struct CList));
    p->data.key = rand();
    assert(p->data.key);
    p->next = head;
    head = p;
  }
  return head;
}

struct CList *bitSort(struct CList *head)
```

```
{ struct CList *zeroEnd, *oneEnd, *zero, *one;
  unsigned maxBit, bitPow2;
  /* 0. Xác định mặt nạ bit tối đa */
  maxBit = 1 << (8*sizeof(head->data.key)-1);

  /* 1. phần tử giả ở đầu danh sách */
  zero = (struct CList *) malloc(sizeof(struct CList));
  one = (struct CList *) malloc(sizeof(struct CList));

  /* 2. Sắp xếp */
  for (bitPow2 = 1; bitPow2 < maxBit; bitPow2 <<= 1) {

  /* 2.1. Phân phối theo danh sách*/
  for (zeroEnd = zero, oneEnd = one; NULL != head; head = head->
    next)
    if (!(head->data.key & bitPow2)) {
      zeroEnd->next = head;
      zeroEnd = zeroEnd->next;
    }
    else {
      oneEnd->next = head;
      oneEnd = oneEnd->next;
    }
  /* 2.2. Hợp nhất danh sách */
  oneEnd->next = NULL;
  zeroEnd->next = one->next;
  head = zero->next;
 }

  /* 3. Giải phóng bộ nhớ */
  free(zero);
  free(one);
  return head;
}

void print(struct CList *head)
{ for (; NULL != head; head = head->next)
  printf("%8d", head->data.key);
  printf("\n");
}
```

```
void check(struct CList *head)
{ if (NULL == head) { return; }
  for (; NULL != head->next; head = head->next)
  assert(head->data.key <= head->next->data.key);
}
void clear(struct CList *head)
{ struct CList *p = head;
  while (NULL != head) {
    head = (p = head)->next;
    free(p);
  }
}

int main() {
  struct CList *head;
  head = init(MAX);
  printf("Mảng trước khi sắp xếp:\n"); print(head);
  head = bitSort(head);
  printf("Mảng sau khi sắp xếp :\n"); print(head);
  check(head);
  clear(head);
  return 0;
}
```

Bài tập

▷ **3.31.** Triển khai một phương án bền vững của sắp xếp bằng cách đếm.

3.3.3. Sắp xếp theo bit

Bất chấp những cải tiến đã được thực hiện, sắp xếp theo số đếm (xem) kế thừa hầu hết những thiếu sót của sắp xếp theo nhiều (xem) và khó có thể khẳng định là ít nhất một phần phổ quát. Vấn đề chính của thuật toán là cần phải duy trì một mảng bổ sung với kích thước tỷ lệ với số giá trị có thể có của các khóa m. Rõ ràng rằng m có thể là một số đủ lớn và không cho phép cấp phát lượng bộ nhớ cần thiết. Trong thực tế, hầu hết các khóa có thể có có thể không được tìm thấy, trong khi đối với các khóa khác, có thể có sự tích tụ đáng kể của các phần tử, dẫn đến việc sử dụng bộ nhớ không hiệu quả.

Sắp xếp bằng cách đếm chuyển một lần qua các phần tử của tập hợp, yêu cầu thời gian $\Theta(n)$ và một lần - qua tập các giá trị cho phép đối với các khóa, trong thời gian $\Theta(m)$. Tổng độ phức tạp của thuật toán trở thành $\Theta(m+n)$. Kết quả thu được cho thấy thuật toán là tuyến tính đối với m và n. Sau đó nó là tuyến tính như thế nào? Rõ ràng rằng đối với $m > n$, thuật toán sắp xếp theo phép đếm là tuyến tính với số giá trị có thể có của các khóa m. Tuy nhiên, đối với chúng ta, điều quan trọng hơn là nó phức tạp như thế nào về số phần tử n. Không khó để nhận thấy rằng đối với $m = n^3$, độ phức tạp của thuật toán trở thành $\Theta(n^3)$, tức là nó thậm chí còn tệ hơn việc sắp xếp theo phương pháp bong bóng! Và nếu $m = n^7$? Chúng ta nhận được sự thiếu hiệu quả tuyệt vọng ...

Mặc dù thoạt nhìn điều này có vẻ không ổn, nhưng ý tưởng sắp xếp bằng cách đếm rất hay và có thể đưa chúng ta đến một thuật toán sắp xếp tuyến tính thực sự. (Tất nhiên, một lần nữa với một số hạn chế ...) Đầu tiên chúng ta sẽ tập trung vào thuật toán sắp xếp bit, sau đó chúng ta sẽ xem xét tóm tắt tự nhiên của nó - phương pháp hệ thống số (xem 3.3.4).

Ý tưởng về sắp xếp theo bit chủ yếu dựa trên biểu diễn bên trong nhị phân của các số trong máy tính. Hãy đặt một tập hợp các số nguyên không dấu mà chúng ta sẽ sắp xếp các phần tử. Chúng ta chia các số thành hai danh sách tùy thuộc vào giá trị của chữ số nhị phân nhỏ nhất của chúng. Các số chẵn nằm trong danh sách đầu tiên, và các số lẻ - trong danh sách thứ hai. Sau đó, chúng ta thêm danh sách thứ hai vào cuối danh sách đầu tiên, nơi chúng ta nhận được một danh sách chung. Không giống như đếm, mục mới phải ở cuối danh sách. Ý nghĩa của việc làm sáng tỏ này sẽ trở nên rõ ràng ở phần sau. Lặp lại thao tác với bit áp chót, sau đó với bit áp chót, v.v. Quá trình kết thúc sau khi thực hiện thao tác với bit cao nhất. Không khó để thấy rằng quá trình được mô tả theo cách này thực sự dẫn đến một trình tự được sắp xếp (Tại sao?). Yêu cầu trên bây giờ đã rõ ràng. Nó đảm bảo với chúng ta rằng mỗi bước tiếp theo sẽ được hưởng lợi từ những bước trước đó và trong trường hợp các giá trị bằng nhau của bit tương ứng, nó sẽ giữ nguyên lệnh thu được trong các bước trước đó. Vì vậy, mỗi bước trở nên bền vững, và do đó phương pháp là bền vững.

Dựa trên biểu diễn bên trong của dữ liệu trong máy tính, không khó để giả định rằng phương pháp được mô tả có thể áp dụng, với những sửa đổi tối thiểu, để sắp xếp bất kỳ kiểu dữ liệu nào - sắp xếp các cấu trúc phức tạp hơn như mảng, bản ghi và chuỗi thực tế được giảm thành sắp xếp trong số các con số. Tuy nhiên, cần lưu ý rằng điều này không phải lúc nào cũng đơn giản như vậy. Ví dụ: có một số vấn đề với việc sắp xếp các số âm (được trình bày nội bộ trong mã bổ sung) cũng như các số dấu phẩy động.

Chương trình 3.18. Sắp xếp theo bit (318bitsort.c)

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 100
struct CElem {
  int key;
  /* ..... Dữ liệu khác ..... */
};
struct CList {
  struct CElem data;
  struct CList *next;
};

struct CList* init(unsigned n) /*Điền vào mảng với các số ngẫu nhiê
    n*/
{ struct CList *head, *p;
  unsigned i;
  srand(time(NULL));
  for (head = NULL, i = 0; i  < n;  i++) {
    p = (struct CList *)  malloc(sizeof(struct CList));
    p->data.key = rand();
    assert(p->data.key);
    p->next = head;
    head = p;
  }
  return head;
}

struct CList *bitSort(struct CList *head)
```

```
{ struct CList *zeroEnd, *oneEnd, *zero, *one;
  unsigned maxBit, bitPow2;
  /* 0.Xác định mặt nạ bit tối đa */
  maxBit = 1 << (8*sizeof(head->data.key)-1);
  /* 1. Phần tử giả ở đầu danh sách*/
  zero = (struct CList *) malloc(sizeof(struct CList));
  one = (struct CList *) malloc(sizeof(struct CList));
  /* 2. Sắp xếp*/
  for (bitPow2 = 1; bitPow2 < maxBit; bitPow2 <<= 1) {
  /* 2.1. Phân phối theo danh sách*/
  for (zeroEnd = zero, oneEnd = one; NULL != head; head = head->
    next)
    if (!(head->data.key & bitPow2)) {
      zeroEnd->next = head;
      zeroEnd = zeroEnd->next;
    }
else {
    oneEnd->next = head;
    oneEnd = oneEnd->next;
  }

  /* 2.2. Hợp nhất danh sách*/
  oneEnd->next = NULL;
  zeroEnd->next = one->next;
  head = zero->next;
  }
  /* 3. Giải phóng bộ nhớ */
  free(zero);
  free(one);
  return head;
}

void print(struct CList *head)
{ for (; NULL != head; head = head->next)
  printf("%8d", head->data.key);
  printf("\n");
}

void check(struct CList *head)
{ if (NULL == head) { return; }
```

```
  for (; NULL != head->next; head = head->next)
    assert(head->data.key <= head->next->data.key);
}

void clear(struct CList *head)
{ struct CList *p = head;
  while (NULL != head) {
    head = (p = head)->next;
    free(p);
  }
}

int main() {
  struct CList *head;
  head = init(MAX);
  printf("Mảng trước khi sắp xếp:\n"); print(head);
  head = bitSort(head);
  printf("Mảng sau khi sắp xếp:\n"); print(head);
  check(head);
  clear(head);
  return 0;
}
```

Trong cách triển khai trên, dữ liệu được tổ chức thành các danh sách động. Hiệu quả tốt hơn có thể đạt được khi triển khai tĩnh. Với mục đích này, một mảng n phần tử bổ sung duy nhất là đủ, trong đó hai danh sách phát triển so với nhau. Điều này yêu cầu sửa đổi quá trình duyệt qua các phần tử của mảng, bởi vì, mặc dù chúng hợp nhất, hai danh sách vẫn tách biệt. Trong bước tiếp theo, danh sách sẽ phát triển từ cả hai đầu của mảng ban đầu, v.v. Cuối cùng, một bước bổ sung sẽ được yêu cầu để đảo ngược danh sách thứ hai, sẽ yêu cầu không quá $n/2$ lần trao đổi (trong trường hợp xấu nhất, danh sách thứ hai chứa tất cả các phần tử). [Shishkov-1995] [ComputerNews-1994a] [TopTeam-1997]

Không khó để nhận thấy rằng sắp xếp theo bit là tuyến tính và có độ phức tạp của bậc $C.n$, trong đó C là số chữ số của các số được sắp xếp. Trong trường hợp chung, trong triển khai động, bộ nhớ được yêu cầu bổ sung là không đổi.

Thuật toán có thể được đảo ngược và buộc phải xem các bit từ

già nhất đến trẻ nhất. Ý tưởng là, giống như cách sắp xếp nhanh của Hoor, chia mảng thành hai phân vùng: bên trái - với bit 0 cao nhất và bên phải - với bit 1 cao nhất, rồi áp dụng thao tác tương tự cho từng phân vùng, như lần này sự phân chia diễn ra theo chút thâm niên tiếp theo. Phép chia đệ quy kết thúc sau khi xem xét bit nhỏ nhất. Không giống như cách sắp xếp nhanh Hoor cổ điển, số $2^b, b = 0, 1, 2, ...$ ở đây được sử dụng như một dấu phân cách. Vì 2^b không nhất thiết phải chứa trong tập nguồn, nên không có gì đảm bảo rằng trong lần lặp hiện tại, một phần tử sẽ đi đến vị trí cuối cùng của nó. Do đó, đối với $i = j$, có thể xảy ra trao đổi dư thừa các phần tử. (Tại sao?) Sau đây là một ví dụ thực hiện.

Chương trình 3.19. Sắp xếp theo bit (319bitsort2.c)

```
struct CElem m[MAX];
.................
void bitSort2(int l, int r, unsigned bitMask)
{ int i,j;
  if (r > l && bitMask > 0) {
    i = l; j = r;
    while (j != i) {
      while (!(m[i].key & bitMask) && i < j) i++;
      while ((m[j].key & bitMask) && j > i) j--;
      swap(&m[i], &m[j]);
    }
    if (!(m[r].key & bitMask)) j++;
    bitSort2(l,j-1,bitMask >> 1);
    bitSort2(j,r,bitMask >> 1);
  }
}
.................
int main() {
  .................
  bitSort2(0,MAX-1,1 << 8*sizeof(m[0].key) - 1);
  .................
  return 0;
}
```

Phiên bản đệ quy của sắp xếp bit có một hành vi thú vị: mảng càng xáo trộn, nó càng hoạt động tốt hơn và với một mảng được xáo trộn nhiều, nó thậm chí còn vượt trội theo tốc độ phân loại nhanh

của Hoor.

Các triển khai trên là ví dụ và giả định rằng các giá trị chính được phân phối tương đối đồng đều trên phạm vi. Trong thực tế, điều này không phải luôn luôn như vậy. Giả sử chúng ta giả định các giá trị chính trong phạm vi 0-65535, trong khi thực tế chúng nằm trong 0-1000. Do đó, khi áp dụng sắp xếp bit, 6 bước cuối cùng của thuật toán trở nên thừa, vì chúng không thay đổi thứ tự của các phần tử theo bất kỳ cách nào: tất cả các số đều có 0 trong bit tương ứng.

bài tập

▷ **3.32.** Chứng minh rằng việc sắp xếp bit sẽ sắp xếp từng chuỗi đầu vào một cách chính xác.

▷ **3.33.** Để triển khai một biến thể tĩnh của sắp xếp bit, trong đó một mảng bổ sung duy nhất được sử dụng, trong đó hai danh sách, được xây dựng trong quá trình triển khai lặp lại, tăng lên so với nhau.

▷ **3.34.** Để so sánh biến thể lặp lại và đệ quy của sắp xếp bit.

3.2.4.

3.3.4. Phương pháp hệ đếm số

Thuật toán trên cho kết quả xuất sắc. Lần đầu tiên, chúng ta triển khai một thuật toán sắp xếp tuyến tính thực sự, giảm các yêu cầu bổ sung nhiều nhất có thể. Chưa hết . . .. Điều gì sẽ xảy ra nếu chúng ta sử dụng không phải hai mà là ba danh sách? Và tại sao không phải là 10? Hay 16? Hoặc thậm chí 256? Rõ ràng, khi số lượng danh sách tăng lên, hằng số C sẽ giảm, tức là số lần đi qua các phần tử của tập hợp.

Ý tưởng của phương pháp hệ đếm số tương tự như phương pháp sắp xếp theo từng bit. Hãy có số danh sách. Trong bước đầu tiên, các phần tử được phân phối trong danh sách tùy thuộc vào phần còn lại mà chúng cho khi chia cho s. Tuy nhiên, yêu cầu chỉ thêm vào cuối danh sách vẫn có hiệu lực. Tiếp theo là nối các danh sách theo thứ tự tăng dần của các phần dư. Quá trình được lặp lại cho đến khi hết các chữ số của các số.

Trong khi sắp xếp theo bit, chúng ta xem xét các chữ số nhị phân của các số, trong phương pháp đã trình bày, chúng ta làm việc với các giá trị của các chữ số trong biểu diễn chữ số s của chúng, tức là chúng ta coi các số như được viết trong hệ thống chữ số s. Do đó tên của thuật toán - một phương pháp của hệ thống số. Đặc biệt, đối với $s = 2$, chúng ta nhận được thuật toán sắp xếp bit.

Về nguyên tắc, không có giới hạn về số lượng danh sách, nhưng phải cẩn thận, vì số lượng danh sách s lớn hơn sẽ làm giảm tốc độ sắp xếp bởi một n đủ nhỏ (ví dụ: $n < s$). Tốt hơn là con số này là lũy thừa của 2 để tìm lượng dư hiệu quả hơn.

Chương trình đính kèm hoạt động với 16 danh sách và số có không quá 8 chữ số thập lục phân. Phần đầu và phần cuối của danh sách được kết hợp thành một mảng 16 phần tử và việc nối của chúng cũng dễ dàng như sắp xếp theo từng bit. Và ở đây, như trong sắp xếp theo bit, chúng ta có thể bỏ việc sử dụng danh sách động, thay thế chúng bằng mảng. Kết nối một lần nữa có thể cực kỳ đơn giản. Với mục đích này, mỗi mảng được cung cấp một con trỏ tới mảng tiếp theo. Tuy nhiên, điều này phải trả giá - cần có thêm bộ nhớ tĩnh.

Chương trình 3.20. Sắp xếp the hệ số đếm (320radsort.c)

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 100

#define BASE 16 /*Cơ sở của hệ thống số*/
#define POW2 4 /* 16 = 1 << 4 */
#define DIG_CNT 8 /*Số chữ số */

truct  CElem {
  int key;
 /*  Các dữ liệu nào đó */
};

struct CList {
  struct CElem data;
  struct CList *next;
```

```
};

struct CList *init(unsigned n) /*Điền vào mảng với các số ngẫu nhiê
    n*/
{ struct CList *head, *p;
  unsigned i;
  srand(time(NULL));
  for (head = NULL, i = 0; i < n; i++) {
    p = (struct CList *) malloc(sizeof(struct CList));
    p->data.key = rand();
    assert(p->data.key);
    p->next = head;
    head = p;
  }
  return head;
}

struct CList *radixSort(struct CList *head)
{ struct { struct CList *st, *en; } mod[BASE];
  unsigned i, dig, mask, shrM;
  /* 1.Khởi tạo*/
  for (i = 0; i < BASE; i++)
    mod[i].st = (struct CList *) malloc(sizeof(struct CList));

  /* 2. Sắp xếp*/
  mask = BASE-1; shrM = 0;
  for (dig = 1; dig <= DIG_CNT; dig++) {

    /* 2.1. Khởi tạo */
    for (i = 0; i < BASE; i++)
      mod[i].en = mod[i].st;

    /* 2.2.Phân phối các phần tử theo danh sách*/
    while (NULL != head) {
      /* 2.2.1. Tìm chữ số thứ i trong bản ghi số BASE */
      i = (head->data.key & mask) >> shrM;
      /* 2.2.2.Bao gồm số trong danh sách tương ứng */
      mod[i].en->next = head;
      mod[i].en = mod[i].en->next;
      head = head->next;
```

```
    }

  /* 2.3.Hợp nhất danh sách*/
  mod[BASE-1].en->next = NULL;
  for (i = BASE - 1; i > 0; i--)
    mod[i-1].en->next = mod[i].st->next;
  head = mod[0].st->next;
  /* 2.4. Tính toán mặt nạ mới */
  shrM += POW2; mask <<= POW2;
}
  /* 3. Giải phóng bộ nhớ*/
  for (i = 0; i < BASE; i++)
    free(mod[i].st);
  return head;
}

void print(struct CList *head)
{ for (; NULL != head; head = head->next)
  printf("%8d", head->data.key);
  printf("\n");
}

void clear(struct CList *head)
{ struct CList *p = head;
  while (NULL != head) {
    head = (p = head) ->next;
    free(p);
  }
}

int main() {
  struct CList *head;
  head = init(MAX);
  printf("Mảng trước khi sắp xếp:\n");
  print(head);
  head = radixSort(head);
  printf("Mảng sau khi sắp xếp:\n");
  print(head);
  clear(head);
  return 0;
```

```
}
```

Phương pháp của các hệ thống số có thể được tổng quát hóa hơn nữa cho các số được viết trong bất kỳ hệ thống số nào. Ví dụ, bản ghi có thể được coi là yếu tố cấu thành, mỗi yếu tố được ghi trong hệ thống số riêng của nó. Toàn bộ hồ sơ hóa ra được ghi lại trong một hệ thống số hỗn hợp. Hãy xem cách chúng ta đo thời gian, ví dụ: giây, phút, giờ, ngày, tuần, năm. Đây là một hệ hỗn số với các cơ số $d_1 = 60, d_2 = 60, d_3 = 24, d_4 = 7, d_5 = 52$. Chúng ta có thể xem năm phần tử này như một khóa bản ghi tổng hợp, mỗi phần tử trong số đó được ghi lại trong hệ thống số riêng của nó. Sau đó, trong bước đầu tiên của thuật toán, chúng ta sẽ phân phối các phần tử trong 60 danh sách tùy thuộc vào số giây, trong danh sách thứ hai - một lần nữa trong 60 danh sách được xác định theo số phút, trong danh sách thứ ba - trong 24 danh sách tương ứng với giờ, vân vân.

Bài tập

▷ **3.35.** Hãy so sánh sắp xếp theo bit và phương pháp của hệ thống cơ số.

3.3.5. Sắp xếp theo hoán vị

Một đại diện thú vị khác của các phương pháp sắp xếp biến đổi là sắp xếp hoán vị. Sắp xếp hoán vị có thể áp dụng trong các điều kiện hạn chế mạnh hơn so với sắp xếp đếm (xem 3.3.3) hoặc sắp xếp nhiều (xem 3.3.3). Khi sắp xếp theo tập hợp cho hàm thứ tự, chúng ta yêu cầu phải đáp ứng đồng thời hai điều kiện sau (Biểu thị tập hợp của M và số phần tử của nó theo n):

1) các giá trị của f là các số tự nhiên từ một khoảng đóng $[a, b]$ chứa m số nguyên, $m = b - a + 1$. Không phải mọi số nguyên từ $[a, b]$ đều nhất thiết phải là giá trị của f, nhưng tất cả giá trị của f phải từ $[a, b]$.

2) f là vi phân, tức là với $x_1, x_2 \in M, x_1 \neq x_2$ ta có$f(x_1) \neq f(x_2)$ (tức là không lặp lại)

Ở đây chúng ta đặt ra một yêu cầu bổ sung khác:

3) $f : M \to M$ là xạ ảnh, tức là với mọi $s \in M$ thì tồn tại $x \in M$

sao cho $f(x) = s$.

Gọi số phần tử của M là n, tức là $|M| = n$. Ta ký hiệu S là tập $\{1, 2, ..., n\}$. Sau đó, ba yêu cầu trên có thể được xây dựng như sau:

1) $f : S \to S$

2) f là một phép song ánh (nghĩa là cùng một lúc là phép đơn ánh và phép tràn ánh)

Trong điều kiện này, hàm bậc f tác dụng lên các phần tử của tập M như một hoán vị. Điều này cung cấp cho chúng ta một thuật toán tuyến tính hiệu quả cho phép chúng ta sắp xếp các phần tử của M tại chỗ mà không cần sử dụng một mảng bổ sung, mà chúng ta cần khi sắp xếp theo nhiều hoặc bằng cách đếm.

Trong quá trình sắp xếp, khóa của các phần tử khác nhau không được so sánh trực tiếp. Kiểm tra duy nhất là liệu phần tử có khóa i có ở vị trí thứ i hay không. Trong trường hợp phần tử có khóa j,$j = m[i], i \neq j$ ở vị trí thứ i, chúng ta sẽ hoán đổi phần tử thứ i và thứ j, trong đó phần tử có khóa j sẽ chuyển sang vị trí thứ j . Nếu sau khi hoán đổi vị trí thứ i có một phần tử có khóa i thì ta chuyển sang xét phần tử thứ $(i + 1)$ tiếp theo của m[]. Nếu không, chúng ta thực hiện một trao đổi mới: với phần tử có chỉ số m[i]. Dễ thấy rằng quá trình này là cuối cùng và đến một lúc nào đó phần tử có khóa i sẽ đến vị trí thứ i. (Tại sao?)

Chúng ta sẽ minh họa phương pháp được mô tả trên hoán vị 4375612. Chúng ta bắt đầu với $i = 1$ và hoán đổi liên tiếp phần tử đầu tiên m [1] với m [m [1]] cho đến khi phần tử có khóa 1 chuyển sang vị trí 1 (Chúng ta sẽ chỉ viết các phím, nhấn mạnh các yếu tố mà chúng ta trao đổi.):

4375**6**12
5374**6**12
6374**5**1**2**
1374562

Bây giờ số 1 đã ở đúng vị trí. Chúng ta tiến hành 2:

1**3**74562
1**7**3456**2**
1234567

Số 2 cũng được đặt ra. Nó vẫn còn để đi qua các yếu tố khác và đảm bảo rằng chúng cũng ở đúng vị trí. Quá trình được mô tả rất giống với thuật toán tìm chu trình hoán vị [Nakov-1998]. Trên thực tế, thuật toán lần lượt duyệt qua các chu kỳ hoán vị, với một hoặc hai phần tử sẽ đi đến vị trí cuối cùng của chúng mỗi khi chúng được trao đổi. Rõ ràng là số lượng trao đổi không vượt quá n. Số lần so sánh không vượt quá $2n$. Thật vậy, mỗi phần tử của hoán vị được kiểm tra nhiều nhất hai lần: một lần khi đi qua chu kỳ tương ứng của nó và lần thứ hai khi đi qua vị trí cuối cùng của nó. Trong trường hợp một phần tử được đặt trong hoán vị ban đầu, nó sẽ chỉ tham gia vào một phép so sánh.

Chương trình 3.21. Sắp xếp theo hóa vị(321permsort.c)

```
void permSort(struct CElem m[], unsigned n)
{ unsigned i;
for (i = 0; i < n; i++)
  while (m[i].key != (int)i)
    swap(&m[i],&m[m[i].key]);
  }
```

Bài tập

▷ **3.36.** Chứng minh rằng với mọi $i(1 \leq i \leq n)$ và với mọi hoán vị của các phần tử của mảng tại trao đổi liên tiếp phần tử m[i] với m[m[i]] tại một điểm m[i] sẽ chứa i.

3.4. Sắp xếp song song

Một tính năng đặc trưng của các thuật toán sắp xếp nổi tiếng hơn là chúng không cho phép thực hiện đồng thời nhiều hơn một phép toán (so sánh). Thoạt nhìn, sự hạn chế như vậy dường như là sự phản ánh hoàn toàn tự nhiên của các đặc điểm kiến trúc của máy tính phổ biến ngày nay. Thật vậy, với một vài trường hợp ngoại lệ, chúng đều là máy nối tiếp trong đó các quá trình tính toán là tuần tự nghiêm ngặt. Tuy nhiên, gần đây, mọi thứ đã bắt đầu thay đổi. Việc cải tiến công nghệ sản xuất, giảm chi phí và liên quan đến việc giảm giá CPU đã tạo nên sự tiên tiến ngày càng đáng tin cậy của các hệ thống và cụm đa xử lý. Cho đến gần đây, chỉ kỳ lạ, ngày nay

các hệ thống máy tính với hai hoặc nhiều bộ xử lý bắt đầu được coi là một cái gì đó hoàn toàn bình thường. Sự gia tăng hàng loạt của chúng, kết hợp với sự gia tăng số lượng bộ vi xử lý có thể xảy ra, chắc chắn sẽ dẫn đến sự quan tâm ngày càng tăng đối với các thuật toán cho tính toán song song. Đồng thời, sự cải tiến của công nghệ mạng và giảm tương đối chi phí của hệ thống máy tính và máy chủ sẽ dẫn đến sự công nhận ngày càng tăng của cái gọi là điện toán phân tán, nơi công việc giải quyết một vấn đề được phân chia giữa một số hệ thống máy tính nối mạng. Hơn nữa, ngày nay song song hóa đã được giải quyết ở mức thậm chí còn thấp hơn - bên trong chính bộ xử lý. Thật vậy, có quá trình băm bên trong và thực hiện nhiều hoạt động cùng một lúc, với việc thực thi từng hoạt động tiếp theo bắt đầu ngay khi tài nguyên tương ứng trong bộ xử lý được giải phóng, mà không cần đợi quá trình hoàn thành cuối cùng trước đó. Nói chung, ngày nay song song hóa đang tấn công trên mọi mặt trận;))

Mặc dù sự xâm nhập hàng loạt của nó là một hiện tượng tương đối mới, nhưng sẽ là sai lầm nếu coi nó là thành quả của công nghệ hiện đại. Thật vậy, tính song song đã có mặt trong máy tính kể từ khi chúng ra đời. Ví dụ, khái niệm cơ bản của byte liên quan đến việc xử lý đồng thời tám chữ số nhị phân (bit, bit). Song song có thể được tìm thấy trong việc truyền dữ liệu qua bus hoặc mạng, trong hoạt động của các hệ thống nhiều người dùng (máy tính lớn), trong việc thực hiện các phép tính phức tạp trên các hệ thống đa xử lý và hơn thế nữa.

Các ví dụ trên có thể tiếp tục. Tuy nhiên, nó không phải là quá trình song song hóa là quan trọng đối với chúng ta, nhưng tác động của nó đến sự phát triển và sử dụng các thuật toán. Để phát triển khả năng của phần cứng song song, cần có các công cụ phần mềm để điều khiển quá trình phân rã (phân tích) bài toán và phân phối các bài toán con riêng lẻ giữa các thiết bị phần cứng song song (thường là bộ xử lý): cần có các thuật toán song song.

Phương pháp xử lý dữ liệu song song không phải là mới và đã xuất hiện từ lâu trước khi bùng nổ song song đang xuất hiện ngày nay. Trên thực tế, thậm chí rất lâu trước khi thiết bị điện toán đầu tiên xuất hiện. Từ thời xa xưa, mọi người đã nhận thấy rằng nhiều

hoạt động được thực hiện có thể được tăng tốc đáng kể nếu chúng song song với nhau. Ví dụ, trong xây dựng (ví dụ, trong việc xây dựng các kim tự tháp) và hoạt động nông nghiệp, song song hóa dẫn đến gia tốc đáng kể và được sử dụng rộng rãi. Một ví dụ khác về sự song song thành công là cấu trúc của bộ não con người cực kỳ song song: 10 tỷ tế bào thần kinh được kết nối trong một mạng lưới, mỗi tế bào có khoảng 10.000 tế bào khác. Và, mặc dù việc truyền tín hiệu giữa hai tế bào thần kinh được kết nối chậm hơn hàng triệu lần so với các máy tính hiện đại, nhưng có một số tác vụ mà bộ não con người có thể xử lý nhanh hơn nhiều so với máy tính mạnh nhất.

Tất nhiên, không phải mọi quy trình đều có thể được thực hiện song song, và nó thường hóa ra rằng quy trình này phải đi trước quy trình khác. Ví dụ, trước khi chúng ta xây dựng mái của một tòa nhà, chúng ta nên xây dựng mọi thứ khác, bắt đầu từ nền móng.

Có một số thuật toán được sử dụng song song một cách dễ dàng, trong khi những thuật toán khác lại khó hoặc hoàn toàn không. Hơn nữa, không có thuật toán phổ quát cho song song hóa. Mặc dù có một số chương trình chung tạo thuận lợi cho quá trình này, về nguyên tắc, mỗi trường hợp nên được tiếp cận theo cách khác nhau. Vấn đề của song song là cực kỳ phức tạp và nằm ngoài phạm vi của bài báo này.

Dưới đây, chúng ta sẽ tập trung nỗ lực vào việc song song hóa các thuật toán sắp xếp và đặc biệt là phát triển một biến thể sắp xếp song song bằng cách hợp nhất (biến thể không song song cổ điển sẽ được xem xét trong **??**). Khi phát triển thuật toán, chúng ta sẽ giới hạn bản thân trong một máy trừu tượng được tạo thành từ nhiều phần tử giống nhau với hai đầu vào và hai đầu ra được đánh số, chúng ta sẽ gọi là máy so sánh. Khi hai số x và y nhập vào đầu vào của một bộ so sánh, đầu ra đầu tiên là $\min(x, y)$ và thứ hai là $\max(x, y)$. Chúng ta sẽ xem xét rằng chúng ta có đủ số lượng bộ so sánh, cũng như bất kỳ số lượng nào trong số chúng có thể hoạt động đồng thời, tức là song song. Vì chúng ta có một thao tác duy nhất được thực hiện bởi trình so sánh, nên số lượng thuật toán sắp xếp mà chúng ta có thể xây dựng sẽ bị giới hạn. Ví dụ, chúng ta sẽ không thể xem xét các thuật toán dựa trên việc đếm hoặc sử dụng các tính năng của biểu diễn nhị phân bên trong của các số và các

thuật toán khác.

Chúng ta sẽ cố gắng tạo sơ đồ của bộ so sánh và đường kết nối (cáp). Chúng ta sẽ muốn tạo một mạch với n đầu vào và n đầu ra, với mỗi đầu vào $< a_1, a_2, \ldots, a_n >$ cho kết quả là n-bộ $< b_1, b_2, \ldots, b_n >$, là a hoán vị của các phần tử, nhận được ở đầu vào, tức là có thể nhận được từ đầu ra n-bộ chỉ bằng cách trao đổi các cặp phần tử. Chúng ta muốn $a_1, a_2, \ldots, a_n$ là các phần tử của một số tập hợp A trong đó giới thiệu pháp lệnh tuyến tính, tức là mọi phần tử x và y của A đều có thể so sánh được và chính xác một trong ba phần tử là tỷ lệ hợp lệ: $x < y, x = y$ hoặc $x > y$ (tính chất nổi tiếng của phép tam phân). Trực quan rõ ràng rằng các phần tử của A có thể được sắp xếp theo thứ tự tăng dần trong một hàng. Dưới đây, để đơn giản, chúng ta sẽ coi các phần tử của A là số. Chúng ta sẽ nhắc lại rằng các tập hợp có thứ tự tuyến tính tạo thành số tự nhiên, số nguyên, số thực, chuỗi ký hiệu, v.v. Hơn nữa, không khó để coi rằng mỗi tập con của một tập hợp có pháp lệnh tuyến tính cũng được sắp xếp theo thứ tự tuyến tính. (Tại sao?)

Chúng ta sẽ vẽ các sơ đồ dưới dạng n đường thẳng ngang song song, ở những vị trí được nối thành từng cặp bằng các đường thẳng đứng - các đường so sánh. Chúng ta sẽ giả sử rằng sau khi chuyển một cặp dòng qua bộ so sánh, dòng trên sẽ chứa số nhỏ hơn trong số các số được truyền trên hai dòng và dòng dưới - số lớn hơn. Chúng ta sẽ giả định rằng thời gian hoạt động của mỗi bộ so sánh là không đổi, ví dụ 1. Trong trường hợp này, mỗi bộ so sánh sẽ chỉ tạo ra một đầu ra sau khi đã nhận được các số trên cả hai dòng đầu vào của nó. Do đó, chỉ một số nhóm bộ so sánh trong mạch sẽ thực sự hoạt động song song, và những nhóm khác sẽ kết thúc hoặc sẽ chờ số ở cả hai đầu vào.

Hãy thử xác định thời gian hoạt động của một mạch so sánh. Giả sử rằng mỗi bộ so sánh thực hiện công việc của nó trong thời gian 1 và việc truyền dữ liệu trên các đường truyền mất thời gian không đáng kể, mà chúng ta có thể coi là 0, chúng ta có thể xác định thời gian hoạt động của mạch là thời gian mà mỗi đường đầu ra nhận được. giá trị của nó. Rõ ràng là thời gian hoạt động của mạch trùng với số bộ so sánh tối đa mà qua đó một phần tử của hoán vị đầu vào đi qua trước khi đến đầu ra.

Theo cách tương tự, chúng ta có thể đưa ra định nghĩa đệ quy về độ sâu đường thẳng tại một điểm nhất định. Tất cả các dòng đầu vào có độ sâu bằng 0. Nếu đầu vào của bộ so sánh nhận được các dòng có độ sâu lần lượt là $d(x)$ và $d(y)$, thì các dòng đầu ra của nó có độ sâu $\max(d(x), d(y)) + 1$. Độ sâu của bộ so sánh được định nghĩa là độ sâu của các dòng đi ra khỏi nó và độ sâu của mạch - là độ sâu tối đa của bộ so sánh của mạch. Chúng ta sẽ gọi một sơ đồ sắp xếp nếu với mỗi chuỗi đầu vào $< a_1, a_2, \ldots, a_n >$ các phần tử của đầu ra tăng đơn điệu: $b_1 \leq b_2 \leq \ldots \leq b_n$.

Trước khi tiếp tục, chúng ta sẽ xây dựng một nguyên tắc đặc biệt hữu ích tạo điều kiện thuận lợi cho việc xác minh các thuật toán và lược đồ sắp xếp.

3.4.1. Nguyên tắc về số không và số một

Định lý 3.1 (Nguyên tắc của số không và số một)**.** *Nếu một thuật toán (lược đồ) sắp xếp chính xác từng chuỗi đầu vào từ 0 và 1, nó cũng sẽ sắp xếp chính xác từng chuỗi đầu vào với các phần tử thuộc một tập hợp có thứ tự tuyến tính.*

Định lý trên còn được gọi là "Nguyên lý của Zeros và Một". Tính hữu ích của nó rõ ràng là rất lớn, vì nó tạo điều kiện thuận lợi đáng kể cho việc chứng minh chính thức (và kiểm tra thực nghiệm) về tính đúng đắn của một thuật toán sắp xếp. Nó đủ để chứng minh rằng nó hoạt động chính xác trên mỗi trong số 2^n dãy số đầu vào 0 và 1 độ dài n, với mỗi n tự nhiên, để chứng minh rằng nó hoạt động chính xác. Nói chung, nguyên tắc này có thể hữu ích khi kiểm tra một chương trình thực hiện thuật toán sắp xếp: Vì số 2^n phát triển tương đối chậm, nhà phát triển có thể dễ dàng kiểm tra tính đúng đắn của chương trình của mình đối với tất cả các chuỗi đầu vào có không quá 30 phần tử. Trước khi tiến hành chứng minh nguyên hàm, chúng ta sẽ hình thành và chứng minh hai bổ đề.

Bổ đề 3.1. *Cho ta có một hàm tăng đơn điệu f và một lược đồ với một bộ so sánh duy nhất có đầu vào là $f(x)$ và $f(y)$. Sau đó, tại các đầu ra trên và dưới của nó, lần lượt thu được $f(\min(x, y))$ và $f(\max(x, y))$.*

Chứng minh. Thật vậy, từ tính đơn điệu của f mà $\min(f(x), f(y)) =$

$f(\min(x,y))$ và $\max(f(x),f(y)) = f(\max)((x,y))$, trực tiếp theo sau phát biểu của bổ đề. □

Bổ đề 3.2. *Cho là một hàm đơn điệu f. Cho một lược đồ khác của các bộ so sánh được đưa ra, tại đầu vào $a =< a_1, a_2, \ldots, a_n >$ cho kết quả là $b =< b_1, b_2, \ldots, b_n >$. Khi đó nếu f (a) = <f (a 1), f (a 2),), f (an)> được cho ở đầu vào của mạch thì sẽ thu được f (b) = <f (b 1) ở đầu ra, f (b 2),. . . , f (bn)>.*

Chứng minh. Ta xét chắc chắn rằng hàm số f đang tăng đơn điệu, tức là với mọi phần tử $x < y$ ta có $f(x) \leq f(y)$. Bây giờ, sử dụng quy nạp dọc theo độ sâu của mỗi dòng, chúng ta có thể chứng minh một phát biểu thậm chí còn chặt chẽ hơn so với bổ đề, từ đó nó sẽ tuân theo trực tiếp, đó là: Nếu một dòng nhận giá trị của y tại đầu vào x, nó sẽ nhận giá trị của $f(y)$ tại đầu vào $f(x)$. Chúng ta sẽ chứng minh phát biểu này bằng cách quy nạp dọc theo độ sâu d của đoạn thẳng:

1) *Cơ sở*: $d = 0$. Với x ở đầu vào ta có y ở đầu ra của dòng. Khi đó, rõ ràng với $f(x)$ ở đầu vào, chúng ta sẽ có $f(y)$ ở đầu ra.

2) *Giả thiết quy nạp*: Giả sử với bất kỳ đường thẳng nào có độ sâu nhỏ hơn $d(d > 0)$, nếu ở đầu vào x ta có đầu ra là y, thì ở đầu vào $f(x)$ ta sẽ có đầu ra $f(y)$.

3) *Bước quy nạp*: Cho đầu vào của bộ so sánh có độ sâu d nhận các dòng có giá trị x_1 và x_2 tương ứng. Theo định nghĩa của một bộ so sánh tại hai đầu ra của nó, chúng ta sẽ nhận được $y_1 = \min(x_1, x_2)$ và $y_2 = max(x_1, x_2)$ tương ứng. Mặt khác, theo giả thiết quy nạp cho hai dòng đầu vào, tại đầu vào $f(x_1), f(x_2)$ trước khi vào bộ so sánh chúng sẽ nhận được các giá trị $f(y_1)$ và $f(y_2)$ tương ứng. Nhưng f là một hàm tăng trưởng đơn điệu. Khi đó, theo Bổ đề 1, tại đầu ra của bộ so sánh, chúng ta sẽ nhận được lần lượt là $\min(f(y_1), f(y_2))$ và $\max(f(y_1), f(y_2))$.

Trường hợp của một hàm giảm đơn điệu f được coi là tương tự. Bổ đề 3.2 được chứng minh. □

Bây giờ chúng ta đã sẵn sàng để tiến tới việc chứng minh nguyên lý của số không và số một.

Chứng minh (Nguyên tắc của số không và số một): Ở trên trong chứng minh của Bổ đề 3.2 chúng ta đã sử dụng phương pháp quy nạp toán học hoàn chỉnh. Bây giờ chúng ta sẽ sử dụng một phổ quát khác phương pháp chứng minh nội dung: giả sử ngược lại. Để một lược đồ các bộ so sánh được đưa ra, sắp xếp chính xác tất cả các chuỗi từ 0 đến 1 và giả sử rằng tồn tại một dãy $< a_1, a_2, \ldots, a_n >$ các phần tử của một số tuyến tính đã sắp xếp thứ tự tập A, mà nó cho một chuỗi không được sắp xếp. Sau đó, nó sẽ tồn tại ít nhất một cặp phần tử a_i và $a_j (a_i < a_j)$ sao cho ở đầu ra của mạch a_j nằm ở vị trí nào đó trước a_i. Cho phép bây giờ chúng ta định nghĩa hàm f như sau:

$$f(x) = \begin{cases} 0 & , x \leq a_i; \\ 1 & , x > a_j. \end{cases}$$

Không khó để thấy rằng hàm được định nghĩa như vậy là tăng trưởng đơn điệu. Khi đó từ thực tế rằng a_j đứng trước a_i bởi Bổ đề 3.2, chúng ta nhận được rằng $f(a_j)$ đứng trước $f(a_i)$. Mặt khác, theo định nghĩa của f, chúng ta có $f(a_j) = 1$ và $f(a_i) = 0$. Nhưng khi đó lược đồ đã xét không sắp xếp đúng dãy $< f(a_1), f(a_2), \ldots, f(a_n) >$, là một dãy 0 và 1. Chúng ta có một mâu thuẫn với giả thiết rằng lược đồ sắp xếp đúng tất cả các dãy 0 và 1.

3.4.2. Trình tự bitonic

Nguyên tắc vừa được chứng minh sẽ được sử dụng cụ thể đối với chúng ta trong phát biểu sau đây. Dưới đây chúng ta sẽ cố gắng phát triển một thuật toán sắp xếp song song hiệu quả dựa trên các mạch so sánh. Để làm điều này, trước tiên chúng ta sẽ xem xét một thuật toán để sắp xếp các chuỗi bitonic. Theo chuỗi bitonic, chúng ta có nghĩa là một chuỗi tăng trưởng đơn điệu, sau đó giảm đơn điệu, hoặc ngược lại: giảm đơn điệu, sau đó tăng đơn điệu. Ví dụ:

$$< 1, 7, 9, 11, 27, 5, 4, 3, 3, 1 >$$

$$< 37, 20, 17, 16, 10, 10, 10, 18, 25, 47 >$$

Đặc biệt, mọi thứ được sắp xếp theo thứ tự tăng dần hoặc giảm dần cũng là bitonic. Các chuỗi bitonic bao gồm 0 và 1 có dạng $0^i1^j0^k$ hoặc $1^i0^j1^k, i, j, k \geq 0$. Tận dụng nguyên tắc của số không và số một,

dưới đây chúng ta sẽ giới hạn trong việc phát triển - của lược đồ sắp xếp trình tự biton chỉ từ 0 và 1.

3.4.3. "Rõ ràng một nửa"

Để đơn giản hóa việc lập luận dưới đây, chúng ta sẽ làm việc với các dãy có một số phần tử chẵn. Theo các giả định này, chúng ta sẽ phát triển một lược đồ các bộ so sánh, hoạt động theo nghĩa tương tự như sắp xếp nhanh (xem 3.2.6), và chúng ta sẽ gọi là "rõ ràng một nửa". Khi nhận được một chuỗi bitonic có độ dài n ở đầu vào của đầu ra, sẽ thu được hai chuỗi bitonic, mỗi chuỗi có độ dài $n/2$. Trong trường hợp này, các giá trị lớn hơn sẽ nằm trong dãy thứ hai, tức là mỗi phần tử của dãy thứ nhất sẽ nhỏ hơn hoặc bằng mỗi phần tử của dãy thứ hai. Từ đó rõ ràng là có ít nhất một trong hai chuỗi chỉ bao gồm 0 (trên) hoặc chỉ 1 (dưới), do đó có tên "rõ ràng một nửa". Lược đồ được xem xét có độ sâu 1 và thực hiện song song tất cả các phép so sánh thuộc loại $(i, n/2 + i)$ cho $i = 1, 2, \ldots, n/2$.

3.4.4. Sắp xếp chuỗi bitonic

Giả sử rằng $n/2$ cũng là số chẵn. Sau đó, chúng ta có thể áp dụng "một nửa rõ ràng" cho mỗi trong hai trình tự mới thu được. Nếu $n/4$ lại chẵn, thì chúng ta có thể tiếp tục quá trình trên từng trong bốn trình tự mới, v.v. Nếu n là lũy thừa của 2, quá trình được mô tả có thể được tiếp tục một cách đệ quy cho đến khi đạt đến các chuỗi có độ dài 1. Không khó để thấy rằng bất kỳ chuỗi bitonic nào cũng có thể được sắp xếp theo cách này. (Tại sao?)

Chúng ta hãy thử đánh giá mức độ phức tạp của quy trình được mô tả. Vì mỗi bước yêu cầu thời gian 1 (thời gian để chạy "xóa một nửa"), bộ so sánh hoạt động song song (theo định nghĩa của lược đồ bộ so sánh) và độ dài của các hàng được xem xét giảm đi một nửa mỗi lần, tổng thời gian cần thiết là $\log_2 n$. Chúng ta đã thu được một thuật toán để *sắp xếp các chuỗi bitonic 0 và 1* với độ phức tạp $\Theta(\log_2 n)$. Bây giờ nó tuân theo nguyên tắc của các số không và các số không mà lược đồ đề xuất hoạt động cho một chuỗi bitonic tùy ý, tức là không chỉ từ 0 và 1.

3.4.5. Sắp xếp sơ đồ hợp nhất

Bây giờ chúng ta có một thuật toán song song hiệu quả để sắp xếp *các chuỗi bitonic*. Chúng ta sẽ cố gắng tóm tắt nó để chúng ta có được một thuật toán hiệu quả để sắp xếp các chuỗi ngẫu nhiên, mà không nhất thiết yêu cầu chúng phải bitonic. Để làm điều này, trước tiên chúng ta sẽ phát triển một thuật toán để hợp nhất hai chuỗi đã được sắp xếp thành một chuỗi được sắp xếp chung như được mô tả bởi Betcher.

Nếu chúng ta đảo ngược trình tự thứ hai, chúng ta sẽ nhận được một trình tự bitonic mà chúng ta đã biết cách sắp xếp. Rõ ràng là quá trình nghịch đảo có thể diễn ra trong thời gian 1 khi sử dụng đủ số lượng bộ so sánh. Sau đó, chúng ta có thể áp dụng sơ đồ sắp xếp theo trình tự bitonic đã phát triển ở trên. Tổng thời gian cần thiết để hợp nhất hai chuỗi sẽ là $1 + \log_2 n$. Nếu chúng ta xem xét kỹ hơn hai bước đầu tiên của thuật toán kết quả, chúng ta có thể thấy rằng chúng có thể được kết hợp tương đối dễ dàng thành một, tức là, dãy thứ hai có thể được đảo ngược đồng thời và "xóa một nửa". Với mục đích này, thay vì so sánh các cặp $(i, n/2 + i)$, chúng ta nên so sánh $(i, n - i + 1)$, với $i = 1, 2, \ldots, n/2$. Kết quả là, chúng ta nhận được hai chuỗi bitonic, với mỗi phần tử của phần tử thứ nhất nhỏ hơn hoặc bằng mỗi phần tử của phần thứ hai. Sau đó, chúng ta có thể tiếp tục áp dụng phân loại bitonic trực tiếp từ bước thứ hai. Kết quả là, chúng ta nhận được một thuật toán để hợp nhất hai chuỗi đã được sắp xếp cho nhật ký thời gian $\log_2 n$.

3.4.6. Sắp xếp sơ đồ phân loại

Khi chúng ta đã triển khai một lược đồ hiệu quả của các bộ so sánh để hợp nhất các chuỗi đã sắp xếp, giờ đây chúng ta đã sẵn sàng để tiến tới mục tiêu thực sự của mình: phát triển một thuật toán hiệu quả để sắp xếp bất kỳ chuỗi đầu vào nào. Chúng ta sẽ tập trung nỗ lực vào việc thực hiện phiên bản sắp xếp hợp nhất song song (xem **??**) Dựa trên đề án sáp nhập ở trên.

Để tạo điều kiện thuận lợi cho việc suy luận thêm, chúng ta hãy giả sử rằng n là một lũy thừa của 2. Chúng ta sẽ suy nghĩ theo cách quy nạp. Giả sử chúng ta có hai dãy đã được sắp xếp, mỗi dãy có độ

dài $n/2$. Sau đó, chúng ta có thể hợp nhất chúng thành một chuỗi được sắp xếp chung cho nhật ký thời gian $\log_2 n$ bằng cách sử dụng lược đồ hợp nhất. Nhưng làm thế nào để chúng ta có được hai chuỗi được sắp xếp? Giả sử chúng ta có 4 dãy đã sắp xếp l_1, l_2, l_3, l_4 với độ dài $n/4$ mỗi dãy. Sử dụng lược đồ hợp nhất, chúng ta có thể hợp nhất l_1 và l_2 trong một chuỗi được sắp xếp chung có độ dài $n/2$ cho nhật ký thời gian $2(n/2)$. Chúng ta có thể hợp nhất l_3 và l_4 cùng một lúc, sử dụng bản sao thứ hai của sơ đồ sáp nhập mà không cần thêm thời gian. Kết quả là, chúng ta sẽ nhận được hai dãy đã được sắp xếp theo yêu cầu có độ dài $n/2$. Và làm thế nào để có các dãy số l_1, l_2, l_3, l_4? Mọi thứ cũng tương tự như vậy. Một lần nữa, chúng ta giả sử rằng chúng ta có tám chuỗi đã được sắp xếp với độ dài $n/8$, chúng ta tập hợp và hợp nhất thành từng cặp cho nhật ký thời gian $2(n/8)$, v.v. Chúng ta tiếp tục giảm kích thước của bài toán một cách đệ quy để thu được các chuỗi đơn phần tử, mỗi chuỗi chúng ta có thể xem xét được sắp xếp. Bây giờ, ngược lại của đệ quy, chúng ta nhận được một chuỗi đã được sắp xếp.

Hãy thử ước tính độ sâu của sơ đồ phân loại được xây dựng theo cách này. Độ sâu $D(n)$ của mạch sắp xếp thứ tự n phần tử có thể được tính bằng độ sâu $D(n/2)$ của mạch sắp xếp thứ tự $(n/2)$ (trên thực tế, chúng ta có hai lược đồ như vậy, nhưng chúng hoạt động song song), cộng với độ sâu $\log_2 n$ của mạch hợp nhất. Chúng ta thu được sự phụ thuộc lặp lại sau:

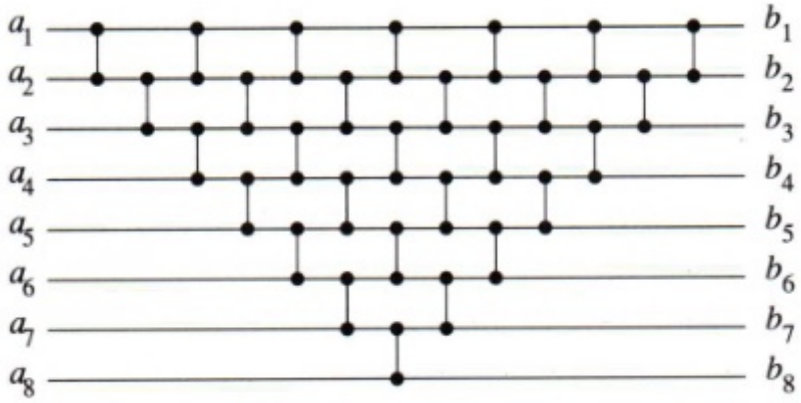
$$D(n) = \begin{cases} 0 & n = 1; \\ D\left(\frac{n}{2}\right) + \log_2 n & n = 2^k, k \geq 1. \end{cases}$$

Từ đây theo Định lý Cơ bản (xem **??**) Chúng ta nhận được:

$$D(n) \in \Theta\left(\log_2^2 n\right).$$

3.4.7. Sơ đồ phân loại chuyển vị

Tất nhiên, phương án đề xuất ở trên không phải là phương án phân loại khả thi duy nhất. Hầu như mọi thuật toán sắp xếp phổ quát chỉ trao đổi các phần tử tạo ra một lược đồ sắp xếp tương ứng bằng các bộ so sánh. Hình 3.3 hiển thị sơ đồ tương ứng với việc sắp xếp theo cách chèn (xem 3.2.2).



Hình 3.3. Lược đồ sắp xếp để phân loại chèn.

Bởi vì phân loại chèn chỉ so sánh các phần tử liền kề, các bộ so sánh trong sơ đồ trên chỉ kết nối các dòng liền kề. Tình hình cũng tương tự với phương pháp bong bóng. Các lược đồ trong đó các bộ so sánh chỉ kết nối các cặp đường liền nhau được gọi là các *lược đồ chuyển vị*. Tất nhiên, không phải tất cả các lược đồ chuyển vị đều là bộ sắp xếp.

3.4.8. Sơ đồ sáp nhập Betcher chẵn lẻ

Sơ đồ hợp nhất chẵn-lẻ được Betcher phát triển vào năm 1960. Ý tưởng như sau: Hãy cho hai dãy n phần tử được sắp xếp $< a_1, a_2, \ldots, a_n >$ và $< a_{n+1}, a_{n+2}, \ldots, a_{2n} >$, mà chúng ta muốn hợp nhất thành một dãy được sắp xếp chung (n là lũy thừa của 2). Chúng ta sẽ suy nghĩ một cách linh hoạt. Hãy để chúng ta hợp nhất số lẻ ($< a_1, a_3, \ldots, a_{n-1} >$ và $< a_{n+1}, a_{n+3}, \ldots, a_{2n-1} >$) và chẵn ($< a_2, a_4, \ldots, a_n >$ và $< a_{n+2}, a_{n+4}, \ldots, a_{2n} >$) phần tử của cả hai dãy. Bây giờ chúng ta có thể kết hợp các chuỗi chẵn và lẻ, sử dụng bộ so sánh $n/2$, được đặt giữa $2i-1$ và $2i$ cho $i = 1, 2, \ldots, n$. Kết quả là, chúng ta thu được một lược đồ có độ sâu $\Theta(\log_2 n)$.

3.4.9. Lược đồ sắp xếp chẵn-lẻ

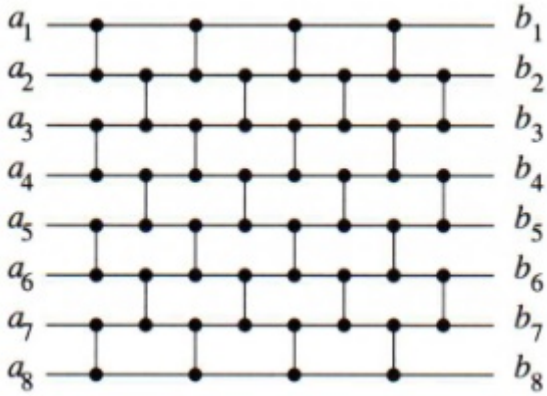
Phát triển thêm ý tưởng về sơ đồ hợp nhất chẵn-lẻ, Butcher đã quản lý để có được một sơ đồ sắp xếp với độ sâu $\Theta\left(\log_2^2 n\right)$. Ở đây

các bộ so sánh ở mức n và vị trí của chúng được tính bằng một công thức đơn giản. Chúng ta hãy biểu thị bằng i là số của dòng $(i = 2, 3, ..., n-1)$ và bằng d - độ sâu của bộ so sánh $(d = 1, 2, \ldots, n)$. Khi đó dòng i nối với dòng $j = i + (-1)^{i+d}, 1 \leq j \leq n$ ở độ sâu d. Đó là, lược đồ có thể được xây dựng theo thuật toán sau: (xem Hình 3.4)

Với $d = 1, 2, ..., n$ chúng ta lặp lại:

1) Nếu d chẵn, nối các dòng $2i - 1$ và $2i$ với một bộ so sánh, với $i = 1, 2, ..., [n/2]$.

2) Nếu d là số lẻ, nối các dòng $2i$ và $2i + 1$ bằng một bộ so sánh, với $i = 1, 2, ..., [(n-1)/2]$.



Hình 3.4. Lược đồ sắp xếp chẵn-lẻ.

3.4.10. Lược đồ hoán vị

Hãy xem xét một mạch trong đó công tắc hai trạng thái được sử dụng thay vì bộ so sánh, có thể bật và tắt. Giống như bộ so sánh, bộ chuyển mạch cũng có hai đầu vào và hai đầu ra. Khi đầu ra công tắc tắt, các giá trị nhận được ở đầu vào được cấp không thay đổi và khi bật, chúng được trao đổi (gạch chéo). Giả sử là một lược đồ có n dòng và đưa ra giá trị đầu vào của nó là $< 1, 2, \ldots, n >$. Rõ ràng rằng, bằng cách điều chỉnh các công tắc, chúng ta có thể thu được các hoán vị khác nhau của n số tự nhiên đầu tiên. Trong trường hợp chúng ta có thể nhận được tất cả n! hoán vị, chúng ta sẽ gọi là hoán

vị lược đồ. Có thể chỉ ra rằng việc thay thế tất cả các bộ so sánh bằng các công tắc trong bất kỳ sơ đồ sắp xếp nào sẽ dẫn đến một lược đồ hoán vị.

Hầu hết các nguồn đều chỉ ra Armstrong, Nelson và O'Connor là những nhà nghiên cứu đầu tiên về kế hoạch sắp xếp công bố nghiên cứu của họ vào năm 1954. Sau đó vào năm 1960, Butcher đã phát triển kế hoạch sắp xếp và hợp nhất chẵn lẻ của mình. Chúng ta cũng nợ phân loại dựa trên chuỗi bitonic. Trong một thời gian dài, đây là những thuật toán được biết đến nhiều nhất - với độ phức tạp của thuật toán $\Theta\left(\log_2^2 n\right)$.

Năm 1983, Aitai, Komlos và Zemeredi đã cải thiện đáng kể kết quả này bằng cách phát triển một sơ đồ sắp xếp với độ sâu $\Theta\left(\log_2 n\right)$ sử dụng bộ so sánh $\Theta\left(\log_2 n\right)$. Mặc dù có độ phức tạp theo lôgarit, nhưng thuật toán này rất khó áp dụng trong thực tế do hằng số rất lớn đằng sau $\Theta(...)$.

Bài tập

▷ **3.37.** Chứng minh rằng thuật toán của 3.4.4 sắp xếp chính xác từng chuỗi bitonic (xem 3.4.4).

▷ **3.38.** Chứng minh rằng mỗi lược đồ chuyển vị sắp xếp với n đầu vào chứa $\Omega(n^2)$ bộ so sánh (xem 3.4.7).

▷ **3.39.** Xác định độ sâu và tìm số bộ so sánh trong sơ đồ chuyển vị với n đầu vào (xem 3.4.7), Tương ứng với:

(a) phân loại chèn (xem 3.2.2);

(b) phân loại bong bóng (xem 3.2.4).

▷ **3.40.** Sử dụng quy nạp theo số dòng, chứng minh rằng một lược đồ chuyển vị có n dòng được sắp xếp nếu và chỉ khi nó sắp xếp theo dãy $(n, n-1, \ldots, 1)$.

▷ **3.41.** Chứng minh rằng với $n > 2$ đối với mỗi mạch hoán vị có n đầu vào sẽ có ít nhất một hoán vị, mà có ít nhất hai cách khác nhau (xem 3.4.10).

3.5. Câu hỏi và bài tập

▷ **3.42.** *So sánh các phương pháp sắp xếp sơ cấp.*
Để so sánh trong thực tế các phương pháp sắp xếp cơ bản: chèn (xem 3.2.2), Bong bóng (xem 3.2.4) Hoặc lựa chọn trực tiếp (xem 3.2.8). Để thực hiện lý luận lý thuyết và kiểm tra thực nghiệm cho các vấn đề khác nhau:

(a) số phần tử: 10; 20; 50; 100; 1000; 10000;

b) thứ tự của mảng: có thứ tự; sắp xếp lại; khuấy nhẹ; bị khuấy động mạnh.

▷ **3.43.** *Đủ số lần lặp.*
Chứng minh rằng đối với nội tiếp, phép chọn trực tiếp, bong bóng và hình chóp $n - 1$ lần lặp của trường hợp ngoại tiếp là đủ.

▷ **3.44.** *So sánh các phương pháp sắp xếp nhanh.*
Để so sánh giữa phân loại nhanh theo lý thuyết và thực nghiệm (xem 3.2.6), Phương pháp phân loại thỏ và rùa (xem 3.2.7) Và phân loại theo hình chóp (xem 3.2.9).

▷ **3.45.** *Kết hợp phân loại nhanh với một phương pháp sơ cấp.*
Để xác định phương pháp sắp xếp cơ bản nào (chèn, bong bóng hoặc lựa chọn trực tiếp: xem 3.2.2, 3.2.4 và 3.2.8) Về mặt lý thuyết, tốt nhất là kết hợp với sắp xếp nhanh (xem 3.2.6). Biên dịch việc triển khai chương trình thích hợp và thực hiện các bài kiểm tra thích hợp. Các giả thiết lý thuyết sơ bộ đã được xác nhận trong thực tế chưa?

▷ **3.46.** *Trường hợp xấu nhất và tốt nhất.*
Đối với mỗi thuật toán sắp xếp bằng cách so sánh để xác định trường hợp xấu nhất và tốt nhất của nó.

▷ **3.47.** *Sự bền vững.*
Thuật toán sắp xếp nào được thảo luận trong chương này là mạnh mẽ?

▷ **3.48.** *Truy cập nhất quán.*
Thuật toán nào được coi là có thể áp dụng để sắp xếp tệp và danh sách tuyến tính mà không có khả năng truy cập trực tiếp vào các phần tử?

▷ **3.49.** *Đảm bảo số lượng so sánh và trao đổi tối thiểu.*
Thuật toán sắp xếp nào được thảo luận trong chương này đảm bảo số lượng tối thiểu trong trường hợp xấu nhất:

a) so sánh;

b) trao đổi.

▷ **3.50.** *Số lần trao đổi tối thiểu.*
Một mảng các số nguyên không có các phần tử bằng nhau và quan hệ giữa một số phần tử của nó được đưa ra. Để biên dịch một thuật toán sắp xếp mảng, thực hiện một số lượng trao đổi tối thiểu.

▷ **3.51.** *Phân loại kép.*
Một ma trận $A = (a_{ij})$ được đưa ra. Mỗi hàng của nó được sắp xếp riêng lẻ theo thứ tự tăng dần, sau đó các trụ của nó được sắp xếp riêng biệt. Các hàng có được sắp xếp theo thứ tự tăng dần không?