NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 7
THUẬT TOÁN CHIA ĐỂ TRỊ

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 7
THUẬT TOÁN CHIA ĐỂ TRỊ

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 7
THUẬT TOÁN CHIA ĐỂ TRỊ

7.1. Giới thiệu

Nguồn gốc của ý tưởng chia bài toán phức tạp thành nhiều bài toán đơn giản hơn, dễ tấn công riêng lẻ hơn và giải pháp của nó cho phép dễ dàng xây dựng giải pháp cho vấn đề ban đầu, đã trở lại rất xa xưa. Nó đạt đến đỉnh cao trong thời kỳ Đế chế La Mã, nơi đã hình thành và thúc đẩy "Chia rẽ và Chinh phục" như một nguyên tắc cơ bản trong chính sách đối ngoại của mình đối với các bộ tộc láng giềng của đế chế. Trên thực tế, người La Mã hoàn toàn không phải là người khám phá ra nguyên tắc, và nó là trọng tâm của sự bành trướng của bất kỳ đế chế vĩ đại nào trước và sau thời La Mã.

Làm thế nào là mọi thứ trong lập trình? Việc áp dụng phương pháp diễn ra trong ba bước. Đầu tiên, bài toán ban đầu được chia thành nhiều bài toán con, thường là hai (chia). Nó sau để giải quyết từng người trong số họ riêng biệt. Trong bước thứ ba, dựa trên các giải pháp của các bài toán con, một giải pháp của bài toán ban đầu (chính) được xây dựng.

Tất nhiên, nguyên tắc này không phải là phổ biến và việc áp

dụng nó không phải là khả thi và thích hợp trong mọi công việc. Các lý do tại sao không thể áp dụng phương pháp này có thể khác nhau. Ví dụ: bài toán có thể không cho phép chia nhỏ các bài toán con thích hợp có thể được giải quyết riêng lẻ. Có thể giải pháp của các bài toán phụ riêng lẻ không tạo thuận lợi đáng kể cho giải pháp của bài toán ban đầu hoặc thậm chí không liên quan gì đến nó. Mặc dù ý tưởng của phương pháp này rất đơn giản, nhưng việc áp dụng nó, cũng như xác định khi nào nó có thể được áp dụng, là một môn nghệ thuật mà bạn dễ dàng nắm vững nhất dựa trên các ví dụ cụ thể.

7.2. Tìm phần tử lớn nhất thứ k

Một trong những thuật toán đầu tiên được giới thiệu cho lập trình viên mới là tìm phần tử tối đa của một mảng. Đây là một thuật toán khá đơn giản yêu cầu so sánh $n - 1$:

Chương trình 7.1. Tìm phần tử lớn nhất trong (701maximum.c)

```
int findMax(int m[], unsigned n) /* Tìm phần tử lớn nhất */
{ unsigned i;
  int max;
  for (max = m[0], i = 1; i < n; i++)
    if (m[i] > max) /**/
      max = m[i];
  return max;
}
```

Chúng ta có thể cải thiện kết quả này không? Không khó để thuyết phục rằng thuật toán được đề xuất về mặt lý thuyết thực hiện số phép so sánh phần tử tối thiểu. Điều sau rất dễ nhận thấy nếu chúng ta nghĩ về các yếu tố là những người tham gia vào một giải đấu loại trực tiếp. Thật vậy, để xác định người thắng cuộc, mỗi người tham gia, trừ chính xác một (người thắng cuộc), phải thua ít nhất một lần để bị loại.

Từ đó có thể dễ dàng thấy rằng số cuộc họp cần thiết là $n - 1$. Ngoài số lượng phép so sánh, một tính năng quan trọng của bất kỳ thuật toán nào như vậy là số lần gán giá trị. Từ mã của đoạn chương trình được đề xuất, có thể thấy ngay rằng số lượng bài tập không

vượt quá số lượng phép so sánh. Tuy nhiên, không quá rõ ràng con số trung bình của các vụ biển thủ là bao nhiêu. Chúng ta để người đọc chứng tỏ rằng nó có bậc là $\Theta(log_2 n)$.

Hãy tưởng tượng một chương trình để hình dung một tập hợp các dấu chấm trên màn hình máy tính. Vì số lượng và vị trí của các chấm có thể rất khác nhau, chương trình sẽ cần chia tỷ lệ tọa độ của chúng theo một cách nào đó để chúng trông đẹp trên màn hình. Màn hình có dạng hình chữ nhật nên sẽ rất tiện lợi khi "đóng" các điểm lại trong một hình chữ nhật phù hợp, từ đó tính ra tỉ lệ tương ứng. Rõ ràng là hình chữ nhật này được xác định bởi các tọa độ x và y lớn nhất và nhỏ nhất. Ở đây nảy sinh đồng thời bài toán tìm phần tử tối thiểu và tối đa.

Bài toán mới cũng có một phương pháp giải tuyến tính. Cần so sánh $2n - 1$ để tìm ra phần tử lớn nhất và sau đó/trước và nhỏ nhất một cách độc lập theo thuật toán trên (Khi tìm kiếm phần tử nhỏ nhất, chỉ cần xoay dấu so sánh theo thứ tự được đánh dấu bằng /* */.). Tuy nhiên, giải pháp này là không tối ưu, vì trong quá trình tìm kiếm giá trị tối thiểu, các tỷ lệ đã biết thu được trong tìm kiếm giá trị tối đa không được tính đến.

Dưới đây, chúng ta sẽ trình bày một thuật toán khác, trong trường hợp xấu nhất, chúng ta sẽ không cần nhiều hơn $\lceil 3n/2 \rceil$ phép so sánh. Chúng ta sẽ xem xét các phần tử theo từng cặp: đầu tiên chúng ta sẽ so sánh hai phần tử với nhau, sau đó phần tử lớn hơn với mức tối đa hiện tại và phần tử nhỏ hơn với mức tối thiểu hiện tại, thực hiện ba so sánh cho mỗi cặp phần tử.

Tìm đồng thời max và min trong 701maximum.c

```
void findMinMax(int *min, int *max, const int m[], const
    unsigned n)
/*Tìm đồng thời max và min*/
{ unsigned i, n2;
  for (*min = *max = m[n2 = n/2], i = 0; i < n2; i++)
    if (m[i] > m[n-i-1]) {
      if (m[i] > *max) *max = m[i];
      if (m[n-i-1] < *min) *min = m[n-i-1];
    }
    else {
      if (m[n-i-1] > *max) *max = m[n-i-1];
```

```
        if (m[i] < *min) *min = m[i];
    }
}
```

Mọi thứ như thế nào nếu chúng ta đang tìm kiếm phần tử lớn thứ hai? Rõ ràng, chúng ta có thể tận dụng cùng một chiến lược, sửa đổi chức năng một cách thích hợp để duy trì không chỉ một, mà cả hai giá trị tốt nhất. Ý tưởng rất đơn giản: chúng ta sẽ so sánh phần tử tiếp theo với phần tử lớn thứ hai hiện tại và chỉ khi phần tử lớn hơn được tìm thấy, chúng ta sẽ thực hiện kiểm tra bổ sung xem phần tử đó không lớn hơn phần tử lớn nhất hiện tại hay chưa:

Tìm phần tử max thứ hai trong 701maximum.c

```
void swap(int *el1, int *el2)  /* Hoán đổi giá trị  */
  { int tmp = *el1; *el1 = *el2; *el2 = tmp; }

int findSecondMax(int m[], unsigned n)
{ int x,y;
  unsigned i;
  x = m[0]; y = m[1];
  if (y > x) swap(&x,&y);
  for (i = 2; i < n; i++)
    if (m[i] > y)
      if ((y = m[i]) > x) swap(&x,&y);
  return y;
}
```

Hãy thử ước tính độ phức tạp của thuật toán. Không khó để thấy rằng trường hợp xấu nhất là mảng nghịch đảo, trong đó hai phép so sánh được thực hiện trên mỗi $(n-2)$ lần lặp của chu kỳ. Thêm phép so sánh trước chu kỳ, với tổng số phép so sánh cần thiết, chúng ta nhận được $2n-3$.

Bây giờ chúng ta hãy xem xét vấn đề tổng quát hơn là tìm *phần tử nhỏ nhất thứ k* của một tập hợp n phần tử đã cho. Bài toán được xem xét tương đương với việc tìm phần tử $(n-k+1)$ lớn nhất. Hơn nữa, nếu chúng ta có một thuật toán giải quyết vấn đề ban đầu, đảo ngược các so sánh $<$ (hoặc $\leq$) và $>$ (hoặc $\geq$), với những sửa đổi tối thiểu, chúng ta có thể tìm thấy phần tử lớn nhất thứ k.

Một giải pháp nhỏ cho vấn đề là sắp xếp hoàn chỉnh tập hợp,

kết quả là phần tử thứ k sẽ chiếm vị trí thứ k. Nhược điểm chính của phương pháp này là, bất kể giá trị của k là bao nhiêu, nó yêu cầu một số phép so sánh bậc $n.\log_2 n$. Tất nhiên, chúng ta có thể giảm con số này xuống $\Theta(n + k.\log_2 n)$, chỉ sắp xếp k phần tử đầu tiên bằng cách sắp xếp theo hình chóp (việc xây dựng kim tự tháp cần thời gian theo thứ tự của $\Theta(n)$ và sắp xếp theo hình chóp có độ phức tạp $\Theta(n.log2n)$ cả trong trường hợp trung bình và trong trường hợp xấu nhất, xem **??**). Phát triển thêm ý tưởng, chúng ta thu được độ phức tạp $\Theta(\min +(n + k.\log_2 n, n + (n - k + 1).\log_2 n))$. Thật vậy, nếu $k > [n/2]$, chúng ta có thể đảo ngược hướng sắp xếp theo thứ tự giảm dần. Tuy nhiên, nếu k đủ gần với $n/2$, thuật toán kết quả rõ ràng sẽ không hiệu quả, vì nó sẽ dẫn đến việc sắp xếp hoàn toàn k hoặc $(n - k + 1)$ phần tử của mảng, điều này sẽ dẫn đến một lượng lớn thông tin không cần thiết.

Chương trình 7.2. Tìm phần tử thứ k (702heap.c)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int m[MAX];
const unsigned n = 100; /* Số phần tử trong mảng */
const unsigned k = 10; /* Số thứ tự của phần tử được tìm kiếm */

void init(int m[], unsigned n) /* Khởi tạo với số ngẫu nhiên */
{ unsigned i;
  for (i = 0; i < n; i++)
    m[i] = rand() % (2*n + 1);
}

void siftMin(int l,int r) /* Sàng lọc phần tử từ đỉnh tự tháp*/
{ int i,j;
  int x;
  i = l;
  j = i + i + 1;
  x = m[i];
  while (j <= r) {
    if (j < r)
      if (m[j] > m[j+1])
          j++;
```

```
    if (x <= m[j])
      break;
    m[i] = m[j];
    i = j;
    j = j*2 + 1;
  }
  m[i] = x;
}

void siftMax(int l,int r) /* Sàng lọc phần tử từ đỉnh tự tháp */
{ int i,j;
  int x;
  i = l; j = i + i + 1; x = m[i];
  while (j <= r) {
    if (j < r)
      if (m[j] < m[j+1]) j++;
    if (x >= m[j])
      break;
    m[i] = m[j];
    i = j;
    j = j*2 + 1;
  }
  m[i] = x;
}

void heapFindK(unsigned k) /* Tìm phần tử thứ k từ tự tháp*/
{ int l,r;
  char useMax;
  if (useMax = (k > n/2))
    k = n - k - 1;
  l = n/2; r = n - 1;
  /* Xây dựng tự tháp */
  while (l-- > 0)
    if (useMax) siftMax(l,r); else siftMin(l,r);
  /* (k-1)-bội lần bỏ phần tử nhỏ nhất */
  for (r = (int)n-1; r >= (int)(n-k); r--) {
    m[0] = m[r];
    if (useMax) siftMax(0,r); else siftMin(0,r);
  }
}
```

```
void print(int m[], unsigned n) /* biểu diễn mảng lên màn hình */
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%8d", m[i]);
}

int main() {
   init (m,n);
  printf("Mảng trước khi tìm:"); print(m,n);
  printf("\nTìm phần tử thứ k: k=%u", k);
  heapFindK(k);
  printf("\nPhần tử thứ k là: %d", m[0]);
  return 0;
}
```

Trong một số điều kiện ràng buộc mạnh hơn, chúng ta có thể dễ dàng thu được các thuật toán tuyến tính thậm chí. Ví dụ, bằng cách sử dụng ý tưởng của thuật toán đếm (xem 3.2.2.), Chúng ta có thể tìm số lần xuất hiện c j cho mỗi khóa j trong số các giá trị cho phép của các khóa. Giả sử rằng các khóa là các số nguyên trong khoảng $[l, r]$. Sau đó ta có thể tính tổng liên tiếp $c_l, c_{l+1}, c_{l+2}, ...$ cho đến khi đạt được chỉ số j mà tổng của lần đầu tiên trở nên lớn hơn hoặc bằng k. Phần tử ở giữa bắt buộc sẽ có giá trị là j. Tuy nhiên, một lượng lớn thông tin dư thừa lại tích lũy và độ tuyến tính của thuật toán phụ thuộc rất nhiều vào sự phân bố của các giá trị khóa.

Mặc dù thoạt nhìn điều này không rõ ràng, nhưng ý tưởng sắp xếp nhanh (xem 3.1.6.) Để chia một mảng phân vùng cung cấp một thuật toán hiệu quả để giải quyết vấn đề. Đây là một thuật toán chia để trị được đề xuất bởi Hoor, tác giả của sắp xếp nhanh (cũng dựa trên phép chia để trị).

Mảng được chia thành hai phần với biên trái $l = 0$, biên phải$r = n - 1$ và $x = m[k]$ như một phần tử ngăn cách (chúng ta giả sử rằng mảng được lập chỉ mục từ 0 đến $n - 1$). Sau khi chia theo thuật toán sắp xếp nhanh cho các chỉ số i và j, các bất đẳng thức được thỏa mãn:

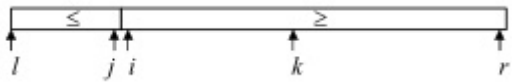
$m[h] \leq x$, cho $h < i$

$m[h] \geq x$, cho $h > j$

$i > j$

Ba trường hợp có thể xảy ra:

- $i < k$. Sau đó x chia mảng cho ít hơn yêu cầu. Việc tách nên được tiếp tục bằng cách đặt $l = i$. (xem Hình 7.2)

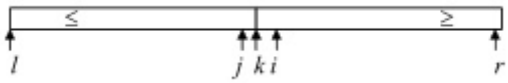


Hình 7.1. Mảng trong trường hợi $i < k$

- $j > k$. Sau đó x chia mảng theo tỷ lệ lớn hơn giá trị được tìm. Việc tách nên được tiếp tục bằng cách đặt $r = j$. (xem Hình **??**)

Hình 7.2. Mảng trong trường hợi $j > k$

- $j < k < i$. Sau đó x chia mảng thành các tỷ lệ bắt buộc và do đó là phần tử bắt buộc. Thật vậy, tất cả các phần tử bên trái của nó đều nhỏ hơn hoặc bằng nó, và những phần tử ở bên phải lớn hơn hoặc bằng nó. Quá trình phân tách được kết thúc. (xem Hình 7.3)



Hình 7.3. Mảng trong trường hợi $j < k < i$

Chúng ta để người đọc thấy rằng các trường hợp khác về vị trí tương hỗ của i, j và k là không thể. Khi đạt đến trường hợp 3, phần tử thứ k sẽ ở vị trí thứ k trong mảng.

Chương trình 7.3. Tìm phần tử thứ k (703midelem.c)

```
void find(int m[],unsigned n,unsigned k) /*Tìm phần tử thứ k */
{ int i,j,l,r;
  int x;
  l = 0; r = n - 1;
  while (l < r) {
    x = m[k]; i = l; j = r;
    for(;;) {
      while (x > m[i]) i++;
      while (x < m[j]) j--;
      if (i > j)
        break;
      swap(m + i, m + j);
      i++;
      j--;
    }
    if (j < (int)k)
      l = i;
    if ((int)k < i)
      r = j;
  }
}
```

Thuật toán được mô tả yêu cầu một số phép so sánh theo bậc của $2n$, trong trường hợp ở mỗi bước, phân hoạch chứa phần tử thứ k bị giảm đi một nửa. Chất lượng này xác định nó hiệu quả hơn những chất lượng được đề xuất ở trên, dựa trên cách sắp xếp hình chóp, yêu cầu một số so sánh về thứ tự của $\Theta(min(n + k.\log_2 n, n + (n - k + 1)).\log_2 n))$, cả trong trường hợp xấu nhất và trong trường hợp tốt nhất. Thật không may, trong trường hợp xấu nhất, khi tại mỗi bước diện tích chỉ giảm đi 1, độ phức tạp của nó sẽ có bậc là $\Theta(n^2)$. Do đó, kế thừa nhược điểm chính của sắp xếp Hoor nhanh, mặc dù nói chung là rất tốt, thuật toán được mô tả tỏ ra cực kỳ kém hiệu quả trong trường hợp xấu nhất.

Chúng ta sẽ đề xuất một thuật toán đệ quy khác, lần này dựa trên phân loại Hoor nhanh với các đặc điểm tương tự. Chúng ta sẽ chia đối với một số phần tử x, bất kể phần tử nào ($m[r]$ được chọn trong chương trình bên dưới) và để nó rơi xuống vị trí *mid* từ đầu mảng hoặc đến vị trí p từ đầu trong tổng số cổ phần đang được xem

xét. Nếu $k \leq p$, thì chúng ta nên tìm phần tử thứ k trong phân hoạch bên trái (l, mid). Nếu không, chúng ta nên tìm phần tử $(k - p)$ trong phân vùng bên phải $(mid + 1, r)$. Quá trình tiếp tục đệ quy cho đến khi đạt được một phân vùng chứa một phần tử (cụ thể là các tìm kiếm):

Chương trình 7.4. Tìm theo Hoor (704midel2.c)

```
unsigned partition(unsigned l,unsigned r) /* Phân chia theo Lomuto
    */
{ int i;
  unsigned j;
  int x;
  i = l - 1; x = m[r];
  for (j = l; j <= r; j++)
    if (m[j] <= x) {
      i++;
      swap(m+i,m+j);
    }
  if (i == (int)r) /* Tất cả <= x. Thu hẹp khu vực với 1. */
    i--;
  return i;
}

unsigned find(int l, int r, unsigned k) /* T theo Hoor */
{ unsigned mid, p;
  if (l == r) return l;
  mid = partition(l,r);
  p = mid - l + 1;
  return k < p ? find(l,mid,k) : find(mid+1,r,k-p);
}
```

Thuật toán được mô tả có cùng hành vi với thuật toán trước: rất tốt trong trường hợp chung và cực kỳ kém hiệu quả trong trường hợp xấu nhất. Hành vi xấu của cả hai thuật toán là do cách chúng ta chọn phần tử x. Sẽ là tốt nếu chọn x để chia diện tích thành hai phần gần như bằng nhau, tức là. càng gần phần tử giữa càng tốt.

Làm thế nào để chọn một x như vậy? Chúng ta sẽ xem xét thuật toán được đề xuất bởi Bloom, Floyd, Pratt, Rivest và Tarjan. Hãy suy nghĩ theo cảm tính. Giả sử chúng ta biết một thuật toán tuyến tính

để tìm phần tử thứ k với $28n$ phép so sánh. Với $n \leq 55$, không khó để xác định một thuật toán cụ thể như vậy. Ví dụ, phương pháp bong bóng (xem 3.1.4.) Yêu cầu $n(n-1)/2$ trong số phép so sánh để sắp xếp mảng hoàn chỉnh và đối với $n \leq 55$, áp dụng bất đẳng thức $28n \geq n(n-1)/2$. Giả sử rằng phép so sánh $28t$ là đủ cho $t < n$. Chúng ta chia mảng thành $\lceil n/7 \rceil$ mảng con, mỗi mảng chứa 7 phần tử và nếu cần chúng ta bổ sung mảng con cuối cùng bằng $-\infty$ (chia), sau đó chúng ta sắp xếp hoàn chỉnh từng mảng con. Nếu chúng ta sử dụng phương pháp bong bóng, chúng ta sẽ cần không quá $7(7-1)/2 = 21$ phép so sánh. Như vậy, tổng số phép so sánh cần thiết không vượt quá $21(n/7) = 3n$. Sau đây là một ứng dụng đệ quy của thuật toán lựa chọn cho $n/7$ bảng được sắp xếp để tìm giá trị trung bình của các phương tiện (quy tắc). Điều này yêu cầu $28(n/7) = 4n$ phép so sánh.

Thuật toán tìm giá trị trung bình của các trung bình

1. Chia các phần tử thành các nhóm 7. Hãy ký hiệu chúng bằng S_i, với $i = 1, 2, ..., \lceil n/7 \rceil$. Nhóm thứ hai có thể chứa ít hơn 7 phần tử.

2. Chúng ta sắp xếp hoàn toàn mỗi nhóm S_i và do đó chúng ta tìm thấy trung vị của nó m_i.

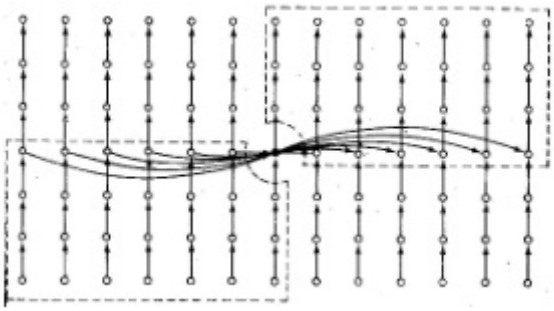
3. Chúng ta tìm trung vị M của các trung bình m_i bằng cách tham chiếu đệ quy đến cùng một thuật toán.

Chúng ta sẽ mô tả thuật toán trên kỹ hơn một chút với mã giả. Chúng ta sẽ xem xét rằng chúng ta có `partition()`, bởi một phần tử x cho trước chia mảng `L[]` thành ba vùng: bên trái `L1[]`, trong đó các phần tử nhỏ hơn x, trung bình `L2[]`, trong đó chúng bằng x và `L3[]`, trong đó chúng lớn hơn x

Chia thành phần

```
unsigned select(int L[], /* Mảng */
unsigned k) /* Đánh số phần tử tìm được*/
{ n = length(L); /* Số lượng phần tử trong L[] */
  if (n <= 7) {
    bubbleSort(L,n);
    return L[k];
}

  /* Chia L thành n/7 nhóm S[i] với đánh số 7 phần tử */
```

```
  split(L, S = {S[i] | i = 1,2,...,n/7});
  for (i = 0; i < n/7; i++)
    x[i] = select(S[i],3); /* Phần tử trung bình là thứ ba */
  M = select({x[i] | i=1,2,...,n/7}, n/14);

  partition(L,L1,L2,L3);

  if (k <= length(L1)) then
    return select(L1,k);
  else if (k > length(L1) + length(L2)) then
    return select(L3,k-length(L1)-length(L2));
  else return M;
}
```



Hình 7.4. Chọn điểm trung vị

Trung vị được chọn của các phương tiện x đủ gần với phần tử trung bình, vì có ít nhất $2n/7$ nhỏ hơn x và ít nhất $2n/7$ lớn hơn x phần tử (xem Hình 7.4). Cho đến nay, chúng ta đã sử dụng 3n so sánh để sắp xếp các bảng, 4n khác - để tìm giá trị trung bình, n so sánh - để chia bảng theo phương pháp Hoor và không quá $28(5n/7)$ - so sánh cho ứng dụng đệ quy của thuật toán cho mỗi vùng tuần hoàn đệ quy giảm ít nhất $2n/7$ phần tử). Cuối cùng, hóa ra là chúng ta cần không quá $28n$ phép so sánh. Đó là, thuật toán của chúng ta là tuyến tính. Tất nhiên, 28 là một hằng số đủ nghiêm trọng, khác xa với 2, trong trường hợp tối ưu của các thuật toán trước đó. Hơn nữa, đối với $n \leq 16384$, thuật toán được đề xuất thực hiện nhiều phép so sánh hơn là sắp xếp hình chóp hoàn chỉnh của mảng. Tuy nhiên, tại $n > 16384$, thuật toán hoạt động đủ tốt trong

cả trường hợp chung và trường hợp xấu nhất. Ngoài ra, điểm số $28n$ có thể dễ dàng được hạ thấp hơn nữa, ví dụ như xuống còn $15n$. Bloom, Floyd, Pratt, Rivest và Tarjan quản lý để giảm nó xuống $391/72 \approx 5,4306.n$. Schönhager, Patterson và Pipenger ghi được $3n$, và Dor và Zwick đã tìm cách cải thiện số điểm đó lên $2,95n$ so với các phép so sánh! [Dor, Zwick-1996]

Bài tập

▷ **7.1.** Hãy thực hiện một chương trình tìm phần tử lớn nhất thứ k, dựa trên việc sắp xếp bằng cách đếm.

▷ **7.2.** Hãy chứng minh rằng trong thuật toán tìm phần tử lớn nhất thứ k dựa trên phép tách, tương tự như trong sắp xếp nhanh, không có vị trí tương hỗ nào khác của các biến i, j và k, ngoại trừ các vị trí được chỉ ra trong Hình 7.2, **??** và 7.3.

▷ **7.3.** Hãy so sánh hai biến thể được đề xuất của thuật toán tìm phần tử lớn nhất thứ k, dựa trên phân tách, tương tự như sắp xếp nhanh.

▷ **7.4.** Có nên mong đợi một cải tiến trong việc chia các phần tử thành các nhóm 5 khi tìm giá trị trung bình của các trung vị không? Và 3? Số lần so sánh dự kiến trong mỗi trường hợp là bao nhiêu? Xác định số phần tử tối ưu trong một nhóm.

▷ **7.5.** Hãy thực hiện thuật toán tìm trung vị của các trung vị.

7.3. Phần tử trội

Định nghĩa 7.1. Giả sử là một đa tập hợp n phần tử (tức là một tập hợp trong đó sự lặp lại của các phần tử được phép). Chúng ta sẽ nói rằng một phần tử của tập hợp là *phần tử trội* nếu nó xuất hiện *nhiều hơn* $n/2$ lần.

Do đó, đa tập hợp $\{2, 3, 3, 1, 3\}$ có 3 là phần tử trội, còn đa tập hợp $\{1, 1, 2, 3\}$ không có phần tử trội.

Rõ ràng, theo định nghĩa trên, một tập hợp nhiều không thể có hai yếu tố phần tử trội. Dưới đây, chúng ta sẽ xem xét một số cách khác nhau để tìm ra nguyên nhân phần tử trội và chúng ta sẽ trình

bày tập hợp nhiều chiều với mảng một chiều. Trừ khi chúng ta đã tuyên bố rõ ràng khác, chúng ta sẽ không giả định bất kỳ loại phần tử cụ thể nào, cũng như sự tồn tại của một thứ tự (một phần hoặc toàn bộ) trong loại này, ít hơn nhiều là một thứ tự của các phần tử trong mảng. Trong các chương trình bên dưới, chúng ta sẽ sử dụng macro `CDataType`, mà chúng ta sẽ định nghĩa là `char`. Tính độc lập của thuật toán `char` có thể được kiểm tra bằng cách thay đổi macro, ví dụ thành `int` hoặc `float`.

Thuật toán đầu tiên có thể xuất hiện trong tâm trí người đọc là xem qua các phần tử của mảng và với mỗi phần tử để kiểm tra xem nó có xảy ra nhiều hơn $n/2$ lần hay không. Trong cách triển khai được đề xuất, hàm `findMajority()` được tìm thấy dưới dạng một tham số mảng, kích thước và địa chỉ của nó để lưu các majorant, trong trường hợp có một tham số trong mảng (hàm tác dụng phụ). Hàm trả về giá trị 1 nếu tìm thấy trường hợp phần tử trội và 0 - nếu không. Việc tìm số lần gặp của một phần tử trong một mảng được phân tách trong hàm `count()`, mà chúng ta sẽ sử dụng bên dưới cho các thuật toán khác. Trong trường hợp phát hiện sớm phần tử trội, công việc tiếp theo sẽ được chấm dứt ngay lập tức.

Chương trình 7.5. Tìm phần tử trội 1(705major1.c)

```
#include <stdio.h>
#define CDataType char

unsigned count(CDataType m[], unsigned size, CDataType candidate)
{ unsigned cnt, i;
  for (i = cnt = 0; i < size; i++)
    if (m[i] == candidate)
      cnt++;
  return cnt;
}

char findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, size2 = size / 2;
  for (i = 0; i < size; i++)
    if (count(m, size, m[i]) > size2) {
      *majority = m[i];
      return 1;
```

```
    }
  return 0;
}
int main() {
  CDataType majority;
  if (findMajority("AAACCBBCCCBCC", 13, &majority))
    printf("Phần tử trội là: %c\n", majority);
  else
    printf("Không có phần tử trội.\n");
  return 0;
}
```

Rõ ràng, độ phức tạp của thuật toán được đề xuất là $\Theta(n^2)$, vì mỗi phần tử trong số n phần tử được kiểm tra là phần tử trội, và kiểm tra yêu cầu một lần vượt qua khác. Chúng ta có thể "lừa" và giảm các lần kiểm tra mà không cần thay đổi đáng kể thuật toán. Vì mục đích này, cần lưu ý rằng tại mỗi lần gọi sau, hàm `count()` thực hiện ngày càng nhiều công việc thừa hơn, đi qua tất cả các mảng. Không khó để nhận thấy rằng việc đếm có thể dễ dàng bỏ sót các phần tử ở bên trái của phần tử hiện tại. Thật vậy, theo cách này, chúng ta có thể bỏ lỡ các lần gặp của phần tử được thử nghiệm. Tuy nhiên, trong trường hợp đó, nó sẽ không phải là phần tử trội, bởi vì trong một số bước trước đó, chúng ta lẽ ra đã xem xét nó và từ chối nó. Lập luận tương tự cho phép chúng ta hoàn thành việc xác minh $(n/2)$ phần tử của mảng. Thật vậy, nó là phần tử cuối cùng ở bên phải mà có đủ phần tử khác, có khả năng ngang hàng với nó, những phần tử có thể khiến chính nó trở thành phần tử trội.

Chương trình 7.6. Tìm phần tử trội cải tiến 2(706major2.c)

```
char findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, j, cnt, size2 = size / 2;
  for (i = 0; i <= size / 2; i++) {
    for (cnt = 0, j = i; j < size; j++)
      if (m[i] == m[j]) cnt++;
    if (cnt > size2) {
      *majority = m[i];
      return 1;
    }
```

```
  }
  return 0;
}
```

Thật không may, bất chấp những cải tiến được thực hiện, cả ở giữa và trong trường hợp xấu nhất, khi $n/2+1$ phần tử đầu tiên khác với nhau, độ phức tạp vẫn là $\Theta(n^2)$. Chúng ta có thể cố gắng hạn chế hơn nữa các yếu tố được xem xét trong quá trình đếm. Ý tưởng là ngừng đếm ngay khi thấy rõ rằng số lượng phần tử còn lại không đủ để làm cho phần tử được đề cập trở thành trường hợp phần tử trội, ngay cả khi tất cả chúng đều trùng khớp với nó.

Chương trình 7.7. Tìm phần tử trội cải tiến 3(707major3.c)

```
char findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, j, cnt, size2 = size / 2;
  for (i = 0; i <= size / 2; i++) {
    for (cnt = 0, j = i; j < size; j++)
      if (cnt + size - j <= size2)
        break;
      else if (m[i] == m[j])
        cnt++;
    if (cnt > size2) {
      *majority = m[i];
      return 1;
    }
  }
  return 0;
}
```

Tuy nhiên, lợi ích của "cải tiến" này trong trường hợp chung là khá đáng ngờ, vì nó yêu cầu bổ sung: một phép cộng, một phép trừ và một phép so sánh ở mỗi bước của số đếm. Có nghĩa là, trong trường hợp xấu nhất, quyết định này tồi tệ hơn quyết định trước. Mặc dù độ phức tạp tiệm cận của nó lại là $\Theta(n^2)$.

Chúng ta sẽ không nghiên cứu sâu hơn theo hướng này, vì có nhiều thuật toán hiệu quả hơn. Hãy xem điều gì sẽ xảy ra nếu chúng ta làm suy yếu các hạn chế nghiêm ngặt được áp đặt ngay từ đầu và cho phép loại yếu tố mà một quy định hoàn chỉnh được xác định.

Trong trường hợp này, chúng ta có thể sắp xếp các phần tử của mảng và kiểm tra xem phần tử ở giữa của nó có phải là trường hợp phần tử trội hay không. Chúng ta nhận được thuật toán sau (Bằng chứng rằng nếu chúng ta sắp xếp mảng, phần tử ở giữa của nó sẽ bị chiếm bởi phần tử phần tử trội, nếu có, chúng ta để nó cho người đọc như một bài tập nhẹ.)

Chương trình 7.8. Tìm phần tử trội cải tiến 4(708major4.c)

```
char findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ heapSort(m, size); /* hoặc mergeSort(m,size); */
  if (count(m, size, m[size / 2]) > size2) {
    *majority = m[size / 2];
    return 1;
  }
  return 0;
}
```

Bởi vì việc kiểm tra xem một phần tử có phải là phần tử trội hay không là tuyến tính, độ phức tạp của thuật toán bị chi phối bởi độ phức tạp của sắp xếp. Nếu chúng ta sử dụng một thuật toán như sắp xếp theo hình chóp (xem **??**) Hoặc sắp xếp hợp nhất (xem 7.5. Bên dưới), có độ phức tạp đảm bảo $\Theta(n.\log_2 n)$ trong trường hợp xấu nhất, chúng ta nhận được độ phức tạp tổng thể là thuật toán $\Theta(n.\log_2 n)$. Chúng ta sẽ lưu ý rằng việc phân loại nhanh Hoor sẽ không hiệu quả với chúng ta. Mặc dù trong trường hợp trung bình, nó có độ phức tạp là $\Theta(n.\log_2 n)$ và chia ra từ 2 đến 3 lần, trong trường hợp xấu nhất thì độ phức tạp của nó là $\Theta(n^2)$.

Tuy nhiên, để tìm phần tử ở giữa (và lớn nhất thứ k) của một mảng, không cần thiết phải sắp xếp nó. Nhớ lại rằng thuật toán của Bloom, Floyd, Pratt, Rivest và Tarjan tìm phần tử trung bình của độ phức tạp tuyến tính $\Theta(n)$, tạo ra $5,43n - 163$ phép so sánh, $n > 32$ (xem7.2)

Chương trình 7.9. Tìm phần tử trội cải tiến 5(709major5.c)

```
char findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ CDataType med;
  med = findMedian(m, size);
```

```
  if (count(m, size, med) > size2) {
    *majority = med;
    return 1;
  }
  return 0;
}
```

Do đó, chúng ta đã thu được một thuật toán tuyến tính để tìm một trường hợp phần tử trội, nhưng với giả thiết bổ sung rằng có một thứ tự tuyến tính giữa các phần tử. Hãy xem những gì chúng ta sẽ nhận được nếu chúng ta từ bỏ thứ tự, nhưng yêu cầu số lượng các giá trị khác nhau có thể có của các phần tử của mảng phải được biết trước và là một số đủ nhỏ.

Trong trường hợp này, chúng ta lại có thể nhận được một thuật toán tuyến tính. Về bản chất, ý tưởng của thuật toán đếm được sử dụng (xem **??**). Với một lần đi qua các phần tử của mảng là số cuộc họp của tất cả các ứng viên cho trường hợp phần tử trội. Tiếp theo là xem qua các giá trị có thể có của các ứng viên và kiểm tra từng giá trị xem nó có xảy ra đúng hơn $n/2$ lần hay không. Điều này xảy ra với thời gian $\Theta(k)$, không phụ thuộc vào số lượng phần tử trong mảng mà chỉ dựa trên số lượng các giá trị khác nhau của k mà chúng có thể chấp nhận. Do đó, với tổng độ phức tạp của thuật toán, chúng ta nhận được $\Theta(n+k)$. Với các giá trị đủ nhỏ của k (ví dụ $k < n$) ta có $\Theta(n)$.

Chương trình 7.10. Tìm phần tử trội cải tiến 6 (710major6.c)

```
#define MAX_NUM 127
CDataType cnt[MAX_NUM + 1];
char findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, j, size2 = size / 2;
  /* Khởi tạo */
  for (i = 0; i < MAX_NUM; i++)
  cnt[i] = 0;
  /* đếm */
  for (j = 0; j < size; j++)
    cnt[m[j]]++;
  /*Kiểm tra trội*/
  for (i = 0; i < MAX_NUM; i++)
```

```
    if (cnt[i] > size2) {
      *majority = i;
      return 1;
    }
  return 0;
}
```

Tuy nhiên, chúng ta hãy quay lại điều kiện ban đầu, trong đó chúng ta không thể dựa vào sắc lệnh cũng như một số lượng nhỏ các giá trị có thể có của các phần tử đã biết trước, và cố gắng thực hiện chiến lược chia để trị. Chia mảng thành hai phần gần như bằng nhau (trong trường hợp số phần tử lẻ thì một phần sẽ có thêm một phần tử). Ý tưởng cơ bản là nếu x là điều kiện phần tử trội của mảng đầu ra, thì nó sẽ là điều kiện phần tử trội của ít nhất một trong hai phần. Từ đây ta dễ dàng có được thuật toán đệ quy tương ứng. Chúng ta chia mảng thành hai phần gần như bằng nhau và tìm kinh giới trên mỗi phần. Có ba trường hợp:

1) Cả hai mảng con đều không có yếu tố phần tử trội. Khi đó sẽ không có mảng đầu ra.

2) Một mảng con có phần tử trội x và mảng còn lại thì không. Kiểm tra xem x có phải là phần tử trội của mảng đầu ra hay không.

3) Cả hai mảng con đều có các đại lượng x và y, có thể khác nhau. Cả x và y đều được kiểm tra. (Nếu x là một trường hợp phần tử trội, thì bài kiểm tra y có thể bị bỏ qua.)

Thuật toán tương tự được áp dụng một cách đệ quy để tìm điều phần tử trội của từng mảng con và quá trình kết thúc khi đến một mảng có một phần tử, trong trường hợp đó phần tử này là trường hợp phần tử trội. Quá trình có thể kết thúc sớm hơn (ví dụ: với hai yếu tố) vì lý do hiệu quả, nhưng chúng ta sẽ không xem xét kỹ lưỡng theo hướng này. Chúng ta sẽ lưu ý rằng việc duy trì các giới hạn bên trái và bên phải yêu cầu một sự thay đổi nhỏ trong các tham số của cả hai hàm; lệnh gọi ban đầu được thực hiện với biên bên trái của mảng được coi là 0 và bên phải - size - 1

Chương trình 7.11. Tìm phần tử trội cải tiến 7 (711major7.c)

```
unsigned count(CDataType m[], unsigned left,
               unsigned right, CDataType candidate)
```

```
{ unsigned cnt;
  for (cnt = 0; left <= right; left++)
    if (m[left] == candidate)
      cnt++;
  return cnt;
}

char findMajority(CDataType m[], unsigned left,
                  unsigned right, CDataType *majority)
{ unsigned mid;
  if (left == right) {
    *majority = m[left];
    return 1;
  }
  mid = (left + right) / 2;
  if (findMajority(m, left, mid, majority))
    if (count(m, left, right, *majority) > (right - left + 1)/2)
      return 1;
  if (findMajority(m, mid + 1, right, majority))
    if (count(m, left, right, *majority) > (right - left + 1)/2)
      return 1;
  return 0;
}
```

Chúng ta đã đạt được những gì? Trong trường hợp tổng quát, ở mỗi mức đệ quy, chúng ta có hai phép nghịch lưu đệ quy với mảng, với ít phần tử hơn một nửa. Đây là phần tách. Kết hợp các kết quả của hai phép nghịch lưu đệ quy (phần quy tắc) yêu cầu các phép so sánh 0, n hoặc $2n$, tương ứng cho các trường hợp 1), 2) và 3). Trong trường hợp giới hạn của một phần tử, chúng ta có độ phức tạp không đổi mà không cần so sánh. Nếu chúng ta biểu thị số lượng phép so sánh được thực hiện cho một mảng n phần tử bằng $T(n)$, trong trường hợp xấu nhất, chúng ta thu được các phụ thuộc lặp lại sau:

$$T(1) = 0$$
$$T(n) = T(n/2) + 2n$$

Sử dụng định lý cơ bản (xem 1.4.10) chúng ta thu được rằng $T(n) \in \Theta(n.\log_2 n)$. Chúng ta không thể đạt được kết quả tốt hơn

sao? Hóa ra là chúng ta có thể dễ dàng thu được độ phức tạp tuyến tính của thuật toán, lưu ý rằng nếu mảng có một trường hợp phần tử trội, thì các câu lệnh sau là hợp lệ:

Mệnh đề 7.1. *Nếu mảng có một phần tử phần tử trội và chúng ta loại bỏ hai phần tử khác nhau, phần tử đặc biệt của mảng mới sẽ trùng với phần tử gốc.*

Mệnh đề 7.2. *Nếu tất cả các phần tử trong một mảng gặp nhau thành từng cặp, thì nếu chúng ta giữ một đại diện và loại bỏ cái kia cho mỗi cặp, thì phần tử phần tử trội của mảng mới sẽ trùng với phần tử phần tử trội của mảng ban đầu.*

Chứng minh cho những nhận định trên là nhẹ và chúng ta xin để lại cho bạn đọc. Chúng ta sẽ lưu ý rằng khi xóa hai phần tử khác nhau khỏi một mảng mà không có chức năng phần tử trội, chúng ta có thể nhận được một mảng có chức năng phần tử trội. Ví dụ: nếu chúng ta loại bỏ 3 và 4 (khác nhau) khỏi mảng $\{1, 1, 2, 3, 4\}$) (không chứa phần tử trội), chúng ta nhận được mảng $\{1, 1, 2\}$, trong đó 1 là phần tử trội.

Hãy thử đơn giản hóa điều kiện một lần nữa bằng cách yêu cầu: 1) mảng được đảm bảo chứa một điều kiện phần tử trội và 2) cho phép thay đổi mảng để sau khi chương trình kết thúc, nó (gần như chắc chắn) khác với ban đầu. Hãy giả sử một chút rằng mảng có một số phần tử chẵn. Hãy thử lại để áp dụng chiến lược La Mã, chia mảng thành hai phần bằng nhau: phần 1 - các phần tử có vị trí lẻ và phần 2 - các phần tử của vị trí chẵn. Tuy nhiên, thay vì tìm kiếm điểm phần tử trội trong mỗi chúng, lần này chúng ta sẽ so sánh các yếu tố theo từng cặp: một yếu tố từ bộ phận này và yếu tố kia từ bộ phận kia. Chúng ta xem xét một cặp cụ thể (các phần tử liên tiếp). Nếu hai yếu tố khác nhau, chúng ta loại trừ chúng khỏi việc xem xét thêm và nếu chúng bằng nhau - chúng ta giữ nguyên một yếu tố. Áp dụng điều này cho tất cả các cặp, chúng ta nhận được một mảng mới có nhiều nhất $n/2$ phần tử. Chiến lược tương tự cũng được áp dụng cho anh ta, vì quá trình này tiếp tục cho đến khi thu được một mảng có một phần tử: phần tử phần tử trội (chúng ta nhắc bạn rằng chúng ta giả định rằng chắc chắn có một phần tử trội).

Sự cố xảy ra với một số phần tử lẻ, trong đó một trong các phần tử sẽ không thể được đưa vào một cặp. Vì yếu tố này có thể quan trọng và xác định yếu tố phần tử trội, chúng ta sẽ không muốn vứt bỏ nó. (Ví dụ: AA BB **A**). Mặt khác, chúng ta không muốn luôn giữ nó, vì bằng cách đó chúng ta có thể phá hủy tính chất phần tử trội (Ví dụ: AA **B**).

Một giải pháp khả thi là giữ nguyên, giảm trọng lượng và chỉ tính đến nếu trong mảng mới không có nó thì không thể xác định được trường hợp phần tử trội. Chúng ta nhận được một thuật toán có độ phức tạp tuyến tính, bởi vì ở mỗi bước số lượng phần tử giảm ít nhất một nửa. Nếu chúng ta biểu thị số lượng phép so sánh được thực hiện cho một mảng n phần tử bằng $T(n)$, trong trường hợp xấu nhất, chúng ta thu được các phụ thuộc lặp lại sau:

$T(1) = 0$

$T(n) = T(n/2) + n/2$

Sử dụng định lý cơ bản (xem 1.4.10.) Ta thu được $T(n) \in \Theta(n)$. Việc thực hiện chương trình được đề xuất không sử dụng một mảng bổ sung, nhưng ở mỗi bước sao chép liên tiếp một đại diện của các phần tử của các cặp có cùng phần tử ở đầu cùng một mảng. Dễ dàng nhận thấy rằng điều này không hề nguy hiểm, vì việc sao chép diễn ra "sau lưng". Thật không may, điều này phá hủy mảng ban đầu, làm mất đi cơ hội của chúng ta để kiểm tra xem phần tử được tìm thấy có thực sự là một nguyên nhân phần tử trội hay không.

Chương trình 7.12. Tìm phần tử trội cải tiến 8 (712major8.c)

```
void findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, curCnt;
  char part = 0;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2)
      if (m[i - 1] == m[i])
        m[curCnt++] = m[i];
    if (i == size) {
      m[curCnt++] = m[i - 1];
      part = 1;
    }
    else if (part)
```

```
      m[curCnt] = m[size - 2];
    else if (m[size - 2] == m[size - 1])
      m[curCnt] = m[size - 2];
    else
      curCnt--;
    size = curCnt;
  } while (size > 1);
  *majority = m[0];
}
```

Một khả năng khác là lưu phần tử cuối cùng của mảng với một số phần tử lẻ trong một biến riêng biệt đặc biệt. Nếu trong bước tiếp theo, mảng có một số phần tử lẻ, chúng ta sẽ hủy nó bằng cách viết phần tử cuối cùng mới của mảng lên trên nó. Tại một thời điểm, mảng không có phần tử nào và ứng cử viên cho trường hợp phần tử trội sẽ nằm trong biến đặc biệt. Điều này có hiệu quả không? Sang bước tiếp theo, chúng ta sẽ chỉ mất đi phần tử majorant nếu trong mảng mới có đúng một nửa số phần tử bằng với phần tử majorant và phần tử dành riêng của chúng ta nhất thiết phải bằng phần tử majorant (chúng ta biết rằng chắc chắn có một phần tử phần tử trội trong mảng). Trong mỗi bước tiếp theo, ít nhất một nửa số phần tử sẽ bằng với phần tử trội, vì vậy nếu tại một thời điểm chúng ta thấy mình có một số phần tử lẻ, phần tử còn lại sẽ lại phải bằng phần tử trội, điều đó có nghĩa là chúng ta có thể loại bỏ giá trị cũ một cách an toàn. Sau khi các phần tử của mảng hết, kinh giới sẽ nằm trong biến đặc biệt. Số phần tử lại giảm ít nhất một nửa ở mỗi bước, tức là chúng ta có độ phức tạp tuyến tính.

Chương trình 7.13. Tìm phần tử trội cải tiến 9 (713major9.c)

```
void findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, curCnt;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2)
      if (m[i - 1] == m[i])
        m[curCnt++] = m[i];
      if (size & 1)
        *majority = m[size - 1];
      size = curCnt;
```

```
    } while (size > 0);
}
```

Chúng ta không thể loại bỏ sự cần thiết của một yếu tố bổ sung? Hãy nghĩ xem khi nào chúng ta cần xem xét yếu tố cuối cùng. Không khó để coi rằng nó sẽ không cần thiết nếu số phần tử trong mảng mới là số lẻ, vì trong trường hợp đó không thể có hai ứng cử viên cho trường hợp phần tử trội. Tuy nhiên, nếu con số này là số chẵn, thì có thể không xác định được điều kiện phần tử trội (tức là các phần tử chứa hai giá trị được phân bổ bằng nhau), đó là lý do tại sao anh ta sẽ xác định điều kiện phần tử trội. Vì vậy: trong trường hợp một số phần tử lẻ, phần tử cuối cùng chỉ được thêm vào cuối mảng mới nếu nếu không thì số phần tử của nó sẽ trở thành số chẵn. Do đó, khi chúng ta đến một mảng có số phần tử lẻ, chúng ta sẽ lưu trữ một số lẻ cho đến khi chỉ còn lại một phần tử duy nhất trong đó: majorant. Thuật toán được mô tả là một biến thể đơn giản của thuật toán trước và độ phức tạp của nó lại là tuyến tính: Nếu cần, phần tử cuối cùng của mảng sẽ tiếp nhận các chức năng của biến đặc biệt từ thuật toán trước đó.

Chương trình 7.14. Tìm phần tử trội cải tiến 10 (714major10.c)

```
void findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, curCnt;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2)
      if (m[i - 1] == m[i])
        m[curCnt++] = m[i];
    if (!(curCnt & 1))
      m[curCnt++] = m[size - 1];
    size = curCnt;
  } while (size > 1);
  *majority = m[0];
}
```

Một cách khác có thể (nhưng phức tạp hơn không cần thiết) để giải bài toán về số phần tử lẻ là liên kết một bộ đếm với mỗi phần tử, cho biết nó thực sự đại diện cho bao nhiêu phần tử. Ban đầu, các bộ đếm của tất cả các phần tử được khởi tạo bằng 1. Khi so sánh

hai phần tử `m[i-1]` và `m[i]`, với các bộ đếm lần lượt là `cnt[i-1]` và `cnt[i]`, chúng ta có hai lựa chọn:
1. `m[i-1] == m[i]`
Chúng ta lưu một bản sao của phần tử, với bộ đếm `cnt [i-1] + cnt [i]`.
2. `m[i-1] != m[i]`
2.1. `cnt[i-1] == cnt[i]`
Chúng ta loại bỏ cả hai yếu tố.
2.2. `cnt[i-1] <cnt[i]`
Ta lưu `m[i]` với bộ đếm `cnt[i] - cnt[i-1].`
2.3. `cnt[i-1]> cnt[i]`
Ta lưu `m[i-1]` với bộ đếm `cnt[i-1] - cnt[i].`

Bây giờ chúng ta luôn có thể lưu phần tử cuối cùng trong trường hợp một số phần tử lẻ. Đối với trường hợp là số chẵn thì tất nhiên chúng ta sẽ không giữ lại, vì nó sẽ được gộp thành từng cặp. Dễ dàng nhận thấy rằng thuật toán kết quả lại là tuyến tính và ở mỗi bước, điều kiện phần tử trội được giữ nguyên. Sau đây là một ví dụ triển khai:

Chương trình 7.15. Tìm phần tử trội cải tiến 11 (715major11.c)

```
void findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, curCnt;
  unsigned *cnt = (unsigned *) malloc(size * sizeof(*cnt));
  for (i = 0; i < size; i++) cnt[i] = 1;
  do {
    for (curCnt = 0, i = 1; i < size; i += 2) {
      if (m[i - 1] == m[i]) {
        cnt[curCnt] = cnt[i - 1] + cnt[i];
        m[curCnt++] = m[i];
      }
      else if (cnt[i] > cnt[i - 1]) {
        cnt[curCnt] = cnt[i] - cnt[i - 1];
        m[curCnt++] = m[i];
      }
      else if (cnt[i] < cnt[i - 1]) {
        cnt[curCnt] = cnt[i - 1] - cnt[i];
        m[curCnt++] = m[i - 1];
      }
```

```
        }
        if (size & 1) {
          cnt[curCnt] = cnt[i - 1];
          m[curCnt++] = m[i - 1];
        }
        size = curCnt;
      } while (size > 1);
      free(cnt);
      *majority = m[0];
    }
```

Một giải pháp thay thế để giải quyết vấn đề thời gian $\Theta(n)$ là sử dụng một ngăn xếp (xem **??**). Lần này chúng ta sẽ loại bỏ yêu cầu mảng phải chứa cây kinh giới. Chúng ta bắt đầu với một ngăn xếp trống, trong đó phần tử đầu tiên của mảng nhập vào ở đầu. Sau đó, ở mỗi bước của mảng, phần tử tiếp theo được trích xuất và so sánh với phần tử ở trên cùng của ngăn xếp. Nếu các phần tử giống nhau, phần tử mới sẽ đi vào ngăn xếp. Nếu không, phần tử trên cùng ngăn xếp bị loại trừ, điều này thực tế có nghĩa là từ chối cả hai phần tử. Trong trường hợp ngăn xếp trống, phần tử tiếp theo sẽ nhập vào đó. Quá trình tiếp tục cho đến khi hết các phần tử của mảng. Nếu có điều gì phần tử trội, nó ở trên cùng của ngăn xếp. Thật vậy, các phần tử bị phá hủy đồng thời theo từng cặp, và chỉ khi *chúng khác nhau*. Do đó, không thể hủy phần tử trội vì không có đủ các phần tử không phải phần tử trội trong mảng. Thật vậy, theo định nghĩa, điều kiện phần tử trội hoàn toàn *bằng hơn* một nửa số phần tử.

Tuy nhiên, mảng có thể không chứa phần tử phần tử trội và vẫn có một phần tử ở đầu ngăn xếp, ví dụ:
"ABC" (ngăn xếp: ∅)
"BC" (ngăn xếp: A)
"C" (ngăn xếp: ∅)
" "(Ngăn xếp: C)

Điều này yêu cầu một đường truyền bổ sung qua mảng để kiểm tra xem phần tử có thực sự là vật phần tử trội hay không. Lưu ý rằng trong trường hợp này, điều này có thể xảy ra vì mảng ban đầu không bị phá hủy, nhưng điều này gây tốn kém bộ nhớ ngăn xếp bổ sung. Lưu ý rằng ngăn xếp có thể chứa nhiều hơn một phần tử.

Trong trường hợp này, tất cả các phần tử trong nó sẽ có cùng giá trị, theo lý luận ở trên. Sau đây là một ví dụ triển khai thuật toán:

Chương trình 7.16. Tìm phần tử trội cải tiến 12 (716major12.c)

```
/* Biến, Hàm và định nghĩa ngăn xếp*/
#define STACK_SIZE 100
CDataType stack[STACK_SIZE];
unsigned stIndex;
void stackInit(void) { stIndex = 0; }
void stackPush(CDataType elem) { stack[stIndex++] = elem; }
CDataType stackPop(void) { return stack[--stIndex]; }
CDataType stackTop(void) { return stack[stIndex - 1]; }
char stackIsEmpty(void) { return 0 == stIndex; }

char findMajority(CDataType m[],unsigned size,CDataType *majority
    )
{ unsigned i, cnt;
  stackInit();
  for (stackPush(m[0]), i = 1; i < size; i++) {
    if (stackIsEmpty())
      stackPush(m[i]);
    else if (stackTop() == m[i])
      stackPush(m[i]);
    else
      stackPop();
  }
  if (stackIsEmpty()) return 0;
  for (*majority = stackPop(), i = cnt = 0; i < size; i++)
    if (m[i] == *majority)
      cnt++;
  return(cnt > size / 2);
}
```

Chúng ta có thực sự cần một ngăn xếp không? Không khó để thấy rằng trong thuật toán đề xuất ngăn xếp luôn chứa các phần tử có cùng giá trị. Thật vậy, nếu phần tử hiện tại khác với phần tử ở trên cùng của ngăn xếp (giả sử ngăn xếp không trống), thì nó không thể được thêm vào, mà thay vào đó, phần tử ở trên cùng của ngăn xếp sẽ hủy lẫn nhau. Nhưng sau đó chúng ta có thể loại bỏ ngăn xếp và thay thế nó bằng một cặp biến như: (phần tử hiện tại, bộ đếm).

Lúc đầu, ứng cử viên có giá trị không xác định và bộ đếm có giá trị bằng 0. Thuật toán chuyển tuần tự qua các phần tử của mảng và ở mỗi bước thực hiện kiểm tra và các hành động liên quan sau:
1) Nếu bộ đếm là 0, phần tử hiện tại trở thành một ứng cử viên và bộ đếm trở thành 1.
2) Nếu bộ đếm khác 0:
2.1) Nếu ứng cử viên phù hợp với phần tử hiện tại, bộ đếm sẽ tăng lên 1.
2.2) Nếu khác, bộ đếm giảm đi 1.

Quá trình tiếp tục cho đến khi hết các phần tử của mảng. Nếu cuối cùng bộ đếm trở thành 0, mảng chắc chắn không chứa phần tử trội. Tuy nhiên, nếu nó khác 0, điều đó không có nghĩa là ứng viên đó nhất thiết phải là một trường hợp phần tử trội và cần được kiểm tra tương ứng. Mặt khác, nếu mảng có điều kiện phần tử trội, thì điều này nhất thiết phải là ứng cử viên.

Sau đây là một ví dụ về việc áp dụng thuật toán cho một chuỗi cụ thể có chứa điều kiện khó hiểu.

```
A A A C C B B C C C B C C
  ^
 ?:0
A A A C C B B C C C B C C
  ^
 A:1
A A A C C B B C C C B C C
    ^
   A:2
A A A C C B B C C C B C C
      ^
     A:3
A A A C C B B C C C B C C
        ^
      A:2
A A A C C B B C C C B C C
          ^
        A:1
A A A C C B B C C C B C C
            ^
          ?:0
```

```
A A A C C B B C C C B C C
             ^
           B:1
A A A C C B B C C C B C C
               ^
             ?:0
A A A C C B B C C C B C C
                 ^
               C:1
A A A C C B B C C C B C C
                   ^
                 C:2
A A A C C B B C C C B C C
                     ^
                   C:1
A A A C C B B C C C B C C
                       ^
                     C:2
A A A C C B B C C C B C C
                         ^
                       C:3
```

Trong trường hợp của chúng ta, điều kiện phần tử trội là C và thuật toán xác định chính xác anh ta là một ứng cử viên. Lưu ý rằng nếu chúng ta thay thế, ví dụ: C đầu tiên bằng A, thuật toán sẽ lại chỉ ra C là ứng cử viên cho trường hợp phần tử trội, nhưng trong trường hợp này đơn giản là không có trường hợp phần tử trội. Đó là, cần phải có một cuộc kiểm tra cuối cùng tiêu chuẩn để đảm bảo rằng ứng viên được đề cử trên thực tế là một trường hợp phần tử trội. Sau đây là một ví dụ triển khai:

Chương trình 7.17. Tìm phần tử trội cải tiến 13 (717major13.c)

```
char FindMajority(CDataType m[],unsigned size,CDataType *
    majority)
{ unsigned cnt, i;
  for (i = cnt = 0; i < size; i++) {
    if (0 == cnt) {
      *majority = m[i];
      cnt = 1;
    }
```

```
    else if (m[i] == *majority) cnt++;
    else cnt--;
  }
  if (cnt > 0) {
    for (i = cnt = 0; i < size; i++)
      if (m[i] == *majority)
        cnt++;
    return(cnt > size / 2);
  }
  return 0;
}
```

Lưu ý rằng thuật toán này, không giống như một số thuật toán trước, sử dụng bộ nhớ bổ sung không đổi (cho hai biến) và không thay đổi mảng. Mặc dù nó có thể được coi là một biến thể của cái trước, nhưng chúng ta sẽ cố gắng giải thích và biện minh cho nó từ một quan điểm khác.

Giả sử thuật toán đã bắt đầu và đạt đến một vị trí. Chúng ta có thể giả định rằng anh ta đã chia các phần tử được kiểm tra của mảng thành hai khu vực. Một khu vực chứa các phần tử được nhóm theo cặp, vì vậy các thành viên của mỗi cặp là khác nhau. Các yếu tố khác bằng nhau và ngang bằng với ứng viên chuyên ngành. Có thể dễ dàng nhận thấy rằng nếu mảng chứa điều kiện phần tử trội, thì đại diện của nó chắc chắn sẽ rơi vào nhóm thứ hai. Thật vậy, điều ngược lại là không thể, vì không có đủ các yếu tố khác ngoài yếu tố phần tử trội để kết đôi với anh ta, vì vậy tất cả các cuộc họp của phần tử trội đều rơi vào nhóm đầu tiên. Tuy nhiên, nếu mảng không có phần tử trội, có thể nhóm thứ hai vẫn bao gồm phần tử ứng viên (ví dụ: ABC, C sẽ là một ứng viên). Đó là lý do tại sao ở đây cũng cần có lối đi bổ sung quen thuộc qua khối núi. Nếu nhóm thứ hai trở nên trống rỗng, thì không có gì đảm bảo về phần tử trội trong mảng. Chúng ta sẽ lưu ý rằng mặc dù một số thuật toán tuyến tính đã được xem xét, nhưng các hằng số ẩn không được quên. Ví dụ: thuật toán thứ hai tốt hơn thuật toán của Bloom, Floyd, Pratt, Rivest và Tarjan để tìm giá trị trung bình, hoạt động dưới những ràng buộc chặt chẽ hơn. Thật vậy, ở đây số phép so sánh là $2n$, trong khi trong thuật toán của Bloom, Floyd, Pratt, Rivest và Tarjan là $5{,}43n - 163, n > 32$. Muốn vậy chúng ta phải thêm $n - 1$ phép so

sánh để kiểm tra xem trung vị có phải là phần tử trội, trong đó chúng ta nhận được $6,43n - 164, n > 32$ (xem 7.1.)

Bài tập

▷ **7.6.** Chứng minh rằng nếu một mảng có chứa phần tử trội được sắp xếp, thì phần tử ở giữa của nó bị chiếm bởi phần tử trội.

▷ **7.7.** Chứng minh rằng nếu x là trường hợp phần tử trội của một mảng và mảng được chia thành hai phần gần như bằng nhau thì x là trường hợp phần tử trội của ít nhất một phần trong số chúng.

▷ **7.8.** Chứng minh rằng nếu mảng `m[]` có một giá trị phần tử trội, thì nếu các phần tử `m[i]` và `m[j]` khác nhau và bị loại bỏ, thì giá trị phần tử trội của mảng mới sẽ trùng với giá trị phần tử trội của mảng cũ.

▷ **7.9.** Chứng minh rằng nếu một mảng có một phần tử phần tử trội, thì nếu tất cả các phần tử trong mảng gặp nhau thành từng cặp và nếu một đại diện được giữ lại bằng cách loại bỏ cái kia cho mỗi cặp thì phần tử phần tử trội của mảng mới sẽ trùng với phần tử phần tử trội của mảng cũ.

7.4. Hợp nhất các mảng đã sắp xếp

Cho hai dãy đã sắp xếp A và B là các miền con của mảng `a[]`. bài toán là kết hợp chúng theo một cách thích hợp để chuỗi kết quả C cũng được sắp xếp. Thoạt nhìn, có vẻ như điều này có thể được thực hiện ngay tại chỗ, tức là trực tiếp trong mảng `a[]`. Chúng ta để người đọc tự thấy rằng điều này không dễ dàng như vậy. Trên thực tế, một phương pháp như vậy có tồn tại, nhưng nó khá phức tạp và đòi hỏi sự chú ý đặc biệt và nỗ lực thêm. Thay vào đó, ở đây chúng ta sẽ đề xuất thuật toán cổ điển với một mảng bổ sung `b[]`, trong đó các phần tử của `a[]` ban đầu được sao chép, sau đó hai phân vùng hợp nhất thành một `a[]`.

Trước tiên, chúng ta hãy xem xét trường hợp tổng quát hơn của việc hợp nhất hai mảng đã sắp xếp `a[]` và `b[]` (với n và m phần tử tương ứng) thành một mảng được sắp xếp mới `c[]`. Điều này có thể được thực hiện như sau: Chúng ta khởi tạo 3 chỉ mục: một cho mỗi

mảng. Ta so sánh các phần tử đầu tiên của hai mảng `a[]` và `b[]` rồi chuyển phần nhỏ hơn thành `c[]`, sau đó tăng các chỉ số của `c[]` và mảng chứa phần tử nhỏ hơn. Trong bước tiếp theo, chúng ta lại so sánh các phần tử tương ứng của hai mảng và sao chép mảng nhỏ hơn trong `c[]`, tăng các chỉ số tương ứng. Quá trình kết thúc khi một trong các mảng bị cạn kiệt, sau đó các phần tử khác của mảng chưa được xử lý sẽ được sao chép vào `c[]`. Sau đây là một ví dụ triển khai thuật toán được mô tả (hai vòng lặp while cuối cùng có thể được thay thế bằng `memcpy()`):

Thay vòng lặp chu kỳ lồng nhau

```
i = j = k = 0;
while (i < n && j < m)
  c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
if (i == n)
  while(j < m)
    c[k++] = b[j++];
else
  while(i < n) c[k++] = a[i++];
```

Chúng ta có thể cố gắng tối ưu hóa phân đoạn chương trình trên bằng cách giảm số lượng so sánh được thực hiện trong chu kỳ trong `while` đầu tiên. Điều này có thể đạt được với sự trợ giúp của bộ giới hạn (phần tử dừng bổ sung ∞) trong mảng đầu ra `a[]` và `b[]`, do đó không thể đi qua các cạnh của chúng. Bất lợi của việc triển khai là các so sánh dư thừa được thực hiện khi các phần tử của `a[]` hoặc `b[]` đã cạn kiệt

```
i = j = k = 0; a[m] = b[n] = INFINITY; mn = m + n;
while (k < mn)
  c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
```

Thuật toán mới nhất thiết yêu cầu chính xác $n + m$ phép so sánh, trong khi thuật toán đầu tiên yêu cầu ít hơn nếu sử dụng hàm `memcpy()`. Trên thực tế, hàm `memcpy()` cũng được triển khai nội bộ thông qua một vòng lặp được liên kết với các phép so sánh có liên quan, nhưng việc triển khai nó khá hiệu quả, vì vậy có lẽ tùy chọn sau là phù hợp hơn (tuy nhiên, có một so sánh bổ sung và một số bộ sưu tập ở đây):

Sửa đổi hàm memcpy()

```
i = j = k = 0;
while (i < n && j < m)
  c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
if (i == n)
  memcpy(c+k,b+j,m-j);
else
  memcpy(c+k,a+i,n-i);
```

Thuật toán được mô tả được triển khai tốt nhất bằng cách sử dụng cấu trúc dữ liệu động, vì việc chuyển một phần tử từ A hoặc B sang C được thực hiện đơn giản bằng cách chuyển hướng con trỏ, tức là không cần thêm bộ nhớ cho C và khi hết các phần tử của một chuỗi, các phần tử của phần tử kia được dán một cách tầm thường vào cuối C. Tuy nhiên, mọi thứ không quá rực rỡ, vì các con trỏ yêu cầu thêm bộ nhớ và kết quả là hoạt động của thuật toán A và B bị phá hủy.

Thuật toán trên có thể được tổng quát hóa cho bất kỳ số lượng trình tự nào, mỗi chuỗi duy trì con trỏ chỉ mục của riêng nó với một chức năng tương tự như i và j của việc triển khai ở trên. Chương trình được đề xuất dưới đây là tối thiểu về số lượng so sánh được thực hiện. Các chuỗi được giữ ở trạng thái tĩnh (trong mảng) và bản thân các mảng được tổ chức trong một danh sách tuyến tính động. Tại mỗi bước, các phần tử hiện tại của mảng được so sánh và phần tử tối thiểu được xác định. Sau đó hiển thị và tắt. Nếu một mảng trở nên trống, nó ngay lập tức bị loại khỏi danh sách các mảng và không tham gia vào các phép so sánh sau. Lần này chúng ta sẽ làm việc với các phần tử của kiểu `struct CElem` và chúng ta sẽ so sánh các khóa `key`.

Chương trình 7.18. Trộn hai mảng thứ tự làm một (718mergearr.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 12
#define ARRAYS 6
struct CElem {
  int key;
```

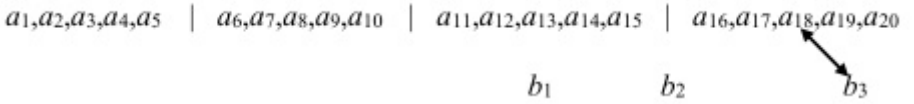
```
  /* một số dữ liệu khác*/
};

struct CList {
  unsigned len, point;
  struct CElem data[MAX];
  struct CList *next;
};

/* Điền những giá trị nguyên ngẫu nhiên */
struct CList *init(unsigned mod)
{ struct CList *head, *p;
  unsigned i, j;
  srand(time(NULL));
  for (head = NULL, i = 0; i < ARRAYS; i++) {
    p = (struct CList *) malloc(sizeof(struct CList));
    p->len = MAX;
    p->point = 0;
    p->data[0].key = (rand() % mod);
    for (j = 1; j < MAX; j++)
      /* Tạo một chuỗi được sắp xếp*/
      p->data[j].key = p->data[j-1].key + (rand() % mod);
    p->next = head;
    head = p;
  }
  return head;
}

void merge(struct CList *head)
{ struct CList *p, *q, *pMin;
  struct CElem k1, k2;
  int i;
  printf("\n");
  p = (struct CList *) malloc(sizeof(struct CList));
  p->next = head;
  head = p;
  for (i = 0; i < MAX*ARRAYS; i++) {
    p = head; pMin = head;
    while (NULL != p->next) {
      k1 = p->next->data[p->next->point];
```

```
            k2 = pMin->next->data[pMin->next->point];
            if (k1.key < k2.key)
               pMin = p;
            p = p->next;
         }
         printf("%8d", pMin->next->data[pMin->next->point].key);
         if (pMin->next->len-1 == pMin->next->point) {
            q = pMin->next;
            pMin->next = pMin->next->next;
            free(q);
         }
         else
            pMin->next->point++;
      }
   }

   void print(struct CList *head)
   { unsigned i;
     for (; NULL != head; head = head->next) {
       for (i = 0; i < MAX; i++)
         printf("%6d", head->data[i].key);
       printf("\n");
     }
     printf("\n");
   }

   int main()
   { struct CList *head;
     head = init(500);
     printf("\nMảng trước khi sắp xếp:\n");
     print(head);
     printf("Kết quả kết hợp:");
     merge(head);
     return 0;
   }
```

Rõ ràng, hàm `merge()` thực hiện $n + m$ số phép so sánh (m và n là độ dài của cả hai dãy). Tại các giá trị gần nhau của m và n, điều này là tốt. Nếu $m = n$ trong trường hợp tổng quát, chúng ta sẽ cần $2n - 1$ phép so sánh. Nhưng điều gì xảy ra ở $m = 1$? Trong trường

hợp này, sử dụng *tìm kiếm nhị phân* (xem **??**) Để tìm vị trí cần chèn B_1, chúng ta có thể giải quyết vấn đề hiệu quả hơn nhiều - phép so sánh $\log_2 n$ sẽ là đủ thay vì $\Theta(n+1)$.

Bài toán mà chúng ta sẽ tự đặt ra là tận dụng tối đa cả hai thuật toán: cụ thể là hợp nhất đơn giản và tìm kiếm nhị phân, biên dịch một thuật toán hoạt động không tệ hơn chúng, $tim = 1$ hoạt động giống như tìm kiếm nhị phân và tại $m = n$ - như một sự hợp nhất đơn giản.



Hình 7.5. Hợp nhất nhị phân của các mảng đã sắp xếp.

Lần này C sẽ phát triển qua lại - giống như một ngăn xếp. Để xác định, giả sử $n > m$ và chia A thành $m+1$ nhóm (dãy con) với số phần tử xấp xỉ bằng nhau. Ta so sánh B_m với phần tử cuối cùng A_k của nhóm áp chót trong A. Nếu B_m nhỏ hơn A_k, thì ta có thể chuyển A_k sang C, cùng với tất cả các phần tử ở bên phải của A_k, nghĩa là, toàn bộ nhóm cuối cùng. Nếu không $(B_m \geq A_k)$ tìm vị trí chèn tương ứng của B_m bằng tìm kiếm nhị phân, sau đó chuyển sang C B_m và các phần tử của A ở bên phải B_m. Vì tìm kiếm nhị phân hoạt động tốt nhất trên các mảng có kích thước, bội của 2, sẽ tốt hơn nếu nhóm cuối cùng chứa $2^{\lfloor \log_2 n/m \rfloor}$ phần tử thay vì n/m. Quá trình tiếp tục cho đến khi hết các phần tử của một dãy, trong đó các phần tử của dãy kia được chuyển sang C. (xem Hình 7.5)

Việc sử dụng tìm kiếm nhị phân ngụ ý chắc chắn truy cập trực tiếp vào từng phần tử của mảng và chắc chắn dẫn đến triển khai tĩnh, như trong hàm ứng dụng `binaryMerge()`, thực hiện hợp nhất nhị phân các mảng theo thuật toán được mô tả. Nó nhận các tham số của hai mảng đã sắp xếp `a[]` và `b[]`, cũng như số phần tử trong mỗi mảng - m và n tương ứng, và trả về một mảng `c[]`, chứa chuỗi đã hợp nhất:

Chương trình 7.19. Trộn hai mảng nhị phân (719binmerge.c)

```
int binarySearch(struct CElem m[], int left,
```

```
                    int right, struct CElem elem)
{ int middle;
  do {
    middle = (left + right) / 2;
    if (m[middle].key < elem.key)
      left = middle + 1;
    else
      right = middle - 1;
  } while (left <= right);
  return right;
}

void binaryMerge(struct CElem a[], struct CElem b[],
                 struct CElem c[], int n, int m)
{ int t, t2, cind, k, j;
  cind = n + m;
  while (n > 0 && m > 0) {
    if (m <= n) {
      t = (int) (log(n / m) / log(2));
      t2 = 1 << t; /* T2 <-- 2^T */
      if (b[m - 1].key < a[n - t2].key) {
        /* Bỏ a[n-t2-1],...,a[n] vào dãy ban đầu */
        cind -= t2;
        n -= t2;
        for (j = 0; j < t2; j++)
          c[cind + j] = a[n + j];
      }
      else {
        k = binarySearch(a, n - t2, n - 1, b[m - 1]);
        for (j = 0; j < n - k - 1; j++)
          c[cind - n + k + j + 1] = a[k + j + 1];
        cind -= n - k - 1;
        n = k + 1;
        c[--cind] = b[--m];
      }
    }
    else {
      t = (int) (log(m / n) / log(2));
      t2 = 1 << t; /* T2 <-- 2^T */
      if (a[n - 1].key < b[m - t2].key) {
```

```
            /* Bỏ b[m-t2-1],...,b[m] vào dãy ban đầu */
            cind -= t2;
            m -= t2;
            for (j = 0; j < t2; j++)
              c[cind + j] = b[m + j];
          }
          else {
            k = binarySearch(b, m - t2, m - 1, a[n - 1]);
            for (j = 0; j < m - k - 1; j++)
              c[cind - m + k + j + 1] = b[k + j + 1];
            cind -= m - k - 1;
            m = k + 1;
           c[--cind] = a[--n];
        }
      }
    }
    if (n == 0)
      for (j = 0; j < m; j++)
        c[j] = b[j];
    else
      for (j = 0; j < n; j++)
        c[j] = a[j];
    }
```

Bài tập

▷ **7.10.** Hãy đề xuất và triển khai một thuật toán để hợp nhất hai chuỗi được sắp xếp từ một mảng tại chỗ, tức là, mà không cần sử dụng một mảng bổ sung.

7.5. Sắp xếp theo hợp nhất

Chúng ta sẽ phát triển một thuật toán để sắp xếp một mảng, và sau đó là một danh sách được liên kết dựa trên *nguyên tắc chia để trị*. Chúng ta sẽ bắt đầu với một mảng. Ý tưởng là chia nó thành hai phần, để sắp xếp chúng riêng biệt (chia), và sau đó hợp nhất chúng thành một mảng được sắp xếp chung (quy tắc). Việc hợp nhất hai phần có thể được thực hiện bằng cách chuyển qua hai mảng con đã được sắp xếp. Chúng ta sẽ thực hiện phép chia ở giữa mảng và

chúng ta sẽ áp dụng đệ quy cùng một thuật toán cho từng phần. Quá trình kết thúc khi đến một mảng con chứa 0 hoặc 1 phần tử.

Rõ ràng, hợp nhất nhị phân ở đây (xem 7.4) Không thể có ích gì cho chúng ta, vì chúng ta hợp nhất các phân vùng với một số phần tử bằng nhau. Chúng ta có một mảng duy nhất `a[]`, được chia thành hai phần mà chúng ta muốn hợp nhất. Chương trình được đề xuất sử dụng một sửa đổi của phương pháp giới hạn. (xem 7.4) Thuật toán tiến hành theo hai bước. Trong bước đầu tiên, nội dung của `a[]` được sao chép vào mảng phụ `b[]`, và các phần tử của phần thứ hai được sao chép theo thứ tự giảm dần. Sau đó, các phép so sánh liên tiếp bắt đầu từ hai đầu đối diện của mảng. Lưu ý rằng phần tử lớn nhất sẽ nằm ở giữa mảng (nó là phần tử cuối cùng trong một trong hai phân vùng được sắp xếp) và sẽ đóng vai trò như một ràng buộc đối với cả hai chỉ mục sau khi một trong các phân vùng bị cạn kiệt. Đây là cách chúng ta nhận được chương trình sau:

Chương trình 7.20. Sắp xếp theo hợp nhất (720mergea.c)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int a[MAX], /* Mảng cơ sở cho sắp xếp*/
b[MAX]; /* Mảng trợ giúp*/
const unsigned n = 100; /*Số phần tử sắp xếp*/

/* Sinh ra tập ví dụ*/
void generate(void)
{ unsigned i;
  for (i = 0; i < n; i++)
  a[i] = rand() % (2*n + 1);
}

/* In danh sách lên màn hình*/
void printList(void)
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%4d", a[i]);
}

/* Sắp xếp */
```

```
void mergeSort(unsigned left, unsigned right)
{ unsigned i, j, k, mid;
  if (right <= left) return; /* kiểm tra xem có gì sắp xếp*/
  mid = (right + left) / 2;
  mergeSort(left, mid); /* Sắp xếp phần bên trái */
  mergeSort(mid + 1, right); /*Sắp xếp phần bên phải*/
  /* Chép các phần tử của a[] vào mảng trợ giúp b[] */
  for (i = mid + 1; i > left; i--)
    b[i - 1] = a[i - 1]; /* Chiều bên phải */
  for (j = mid; j < right; j++)
    b[right + mid - j] = a[j + 1]; /* chiều ngược lại*/
  /* Hợp hai mảng vào a[] */
  for (k = left; k <= right; k++)
    a[k] = (b[i] < b[j]) ? b[i++] : b[j--];
}

int main() {
  generate();
  printf("Trước khi sắp xếp:\n");
  printList();
  mergeSort(0, n-1);
  printf("Sau khi sắp xếp:\n");
  printList();
  return 0;
}
```

Nhược điểm chính của việc triển khai trên là việc sao chép dữ liệu ở mỗi bước. Điều này có thể tránh được bằng cách duy trì một số chức năng hợp nhất. Ý tưởng là xen kẽ hợp nhất `a[]` thành `b[]` với hợp nhất `b[]` thành `a[]`, cũng như hợp nhất tăng dần với hợp nhất theo thứ tự giảm dần. Ví dụ, để có được một dãy được sắp xếp tăng dần trong `a[]`, chúng ta cần hợp nhất hai dãy của `b[]`: dãy đầu tiên được sắp xếp theo thứ tự tăng dần (ký hiệu là `b1[]`) và dãy thứ hai - theo thứ tự giảm dần (`b2[ ]`).. Để có được `b1[]`, chúng ta cần hợp nhất hai chuỗi tương ứng từ phần đầu tiên của một `a[]` (được sắp xếp theo thứ tự tăng dần và giảm dần, tương ứng). Để có được `b2[]`, hai phần khác của dấu `a[]` phải được hợp nhất (lại được sắp xếp theo thứ tự giảm dần và tăng dần, tương ứng), v.v. Cần có 4 hàm hợp nhất khác nhau cho bốn kết hợp có thể có: hợp nhất từ `a[]/b[]`

đến b[]/a[] theo thứ tự tăng dần/giảm dần.

Một cách khác (ít hiệu quả hơn) để tránh sao chép dữ liệu là sử dụng danh sách liên kết. Hợp nhất hai danh sách liên kết đã sắp xếp gần giống như hợp nhất hai mảng đã sắp xếp. Sau đây là một ví dụ về chức năng thực hiện việc sáp nhập.

Chương trình 7.21. Sửa sắp xếp theo hợp nhất (721mergel1.c)

```
struct list  {  /*  Loại danh sách liên kết  */
int value;
struct list  *next;
}  *empty; /*  Phần tử trống */

const unsigned long n = 100;

struct list  *merge(struct list *a, struct list  *b)
{  struct list  *head, * tail ;
/*Giả thiết  hai  danh sách cứa  nhất 1  phần tử */
 tail  = head = empty;
for (;;)  {
  if (a->value < b->value) {
     tail ->next = a;
    a  = a->next;
     tail  =  tail ->next;
    if (NULL == a) {
       tail ->next = b;
      break;
    }
  }
  else {
     tail ->next = b;
    b  = b->next;
     tail  =  tail ->next;
    if (NULL == b) {
       tail ->next = a;
      break;
      }
    }
  }
  return head->next;
}
```

Mọi thứ khó chịu hơn một chút với việc chia danh sách thành hai danh sách con trước khi kháng cáo đệ quy. Việc thiếu quyền truy cập trực tiếp vào các phần tử của một danh sách được liên kết đòi hỏi phải có một đoạn văn rõ ràng qua một nửa các phần tử của nó. Vì mục đích này, nên chỉ định số phần tử trong phân vùng dưới dạng tham số, như được thực hiện trong hàm sau.

Sửa đổi hàm sắp xếp trong 721mergel1.c

```
struct list *mergeSort(struct list *c, unsigned long n)
{ struct list *a, *b;
  unsigned long i, n2;
  /* Nếu danh sách chứa chỉ 1 phần tử thì không làm gì*/
  if (n < 2)
    return c;
  /* Chia danh sách làm hai phần*/
  for (a = c, n2 = n / 2, i = 2; i <= n2; i++)
    c = c->next;
  b = c->next;
  c->next = NULL;
  /* Sắp xếp riêng hai phần sau đó hợp */
  return merge(mergeSort(a, n2), mergeSort(b, n - n2));
}
```

Sử dụng phương pháp giới hạn cho phép tối ưu hóa quá trình sáp nhập. Vì mục đích này, chúng ta sẽ bỏ chỉ báo tiêu chuẩn cuối danh sách `NULL`. Thay vào đó, chúng ta sẽ sử dụng một phần tử đặc biệt z, mà con trỏ tiếp theo sẽ trỏ đến z và `value` của nó sẽ là giá trị tối đa cho phép đối với kiểu (trong trường hợp này là `INFINITY`). Tất cả các danh sách sẽ có z là phần tử cuối cùng của chúng.

Chúng ta đã đề cập ở trên rằng hàm `mergeSort()` cần nhận như một tham số là số lượng các mục trong danh sách. Trong trường hợp chung, nó có thể không được biết trước. Một giải pháp khả thi sau đó là tìm trước số bằng cách duyệt qua danh sách và sau đó gửi nó dưới dạng tham số, như trên. Một cách tiếp cận khả thi khác là tổ chức một chu trình bao gồm hai con trỏ: một con di chuyển ở bước 1 và con thứ hai ở bước 2. Khi con trỏ thứ hai đến cuối danh sách, con trỏ đầu tiên sẽ trỏ vào giữa. Lưu ý rằng nếu được sử dụng làm dấu phân tách `NULL`, thuật toán này sẽ không hoạt động bình thường,

như trong trường hợp số lượng mục chẵn, phần cuối của danh sách sẽ bị bỏ qua. Tuy nhiên, trong trường hợp của chúng ta, trường tiếp theo của z lại trỏ tới z, vì vậy không có gì nguy hiểm. Do đó, chúng ta nhận được phiên bản sau của chương trình:

Chương trình 7.22. Sửa sắp xếp theo hợp nhất (722mergel2.c)

```
#include <stdio.h>
#include <stdlib.h>
#define INFINITY (int)((1 << (sizeof(int)*8 - 1)) - 1)
const unsigned long n = 100;
struct list { /*Loại danh sách liên kết */
  int value;
  struct list *next;
} *z; /* pần tử trống */

/* Tạo ra tập ví dụ */
struct list *generate(unsigned long n)
{ struct list *p, *q;
unsigned long i;
  for (p = z, i = 0; i < n; i++) {
    q = (struct list *) malloc(sizeof(struct list));
    q->value = rand() % (2*n + 1);
    q->next = p;
    p = q;
  }
  return p;
}

void printList(struct list *p) /*In dah sách lên màn hình */
{ for (; p != z; p = p->next)
  printf("%4d", p->value);
}

struct list *merge(struct list *a, struct list *b)
{ struct list *c;
  c = z;

/* Giả thiết hai danh sách chứa ít nhất 1 phần tử */
do {
  if (a->value < b->value) {
```

```
      c->next = a;
    c = a;
    a = a->next;
    }
    else {
        c->next = b;
      c = b;
      b = b->next;
      }
    } while (c != z);
    c = z->next;
    z->next = z;
    return c;
}

struct list *mergeSort(struct list *c)
{ struct list *a, *b;
  /* Nếu danh sách chứa chỉ 1 phần tử: không làm gì cả */
  if (c->next == z)
    return c;
  /* Danh sách chia làm hai phần */
  for (a = c, b = c->next->next->next; b != z; c = c->next)
    b = b->next->next;
  b = c->next;
  c->next = z;
  /* Sắp xếp riêng từng phần sau đó hợp */
  return merge(mergeSort(a), mergeSort(b));
}

int main() {
  struct list *l;
  /* Khới tạo z */
  z = (struct list *) malloc(sizeof(struct list));
  z->value = INFINITY;
  z->next = z;

  l = generate(n);
  printf("Trước khi sắp xếp:\n");
  printList(l);
  l = mergeSort(l); /* giả thiết danh sách chứa ít nhất 1 phần tử */
```

```
  printf("Sau khi sắp xếp:\n");
   printList(1);
  return 0;
}
```

Ai cũng biết rằng mọi chương trình đệ quy đều có một phép lặp tương đương. Đối với một số chương trình, việc chuyển đổi từ đệ quy sang lặp lại rất dễ dàng và đơn giản. Đối với những người khác, đây có thể là một bài toán khá khó khăn. Một cách tiêu chuẩn để chuyển đổi một chương trình đệ quy thành một chương trình lặp là sử dụng một ngăn xếp. May mắn thay, `mergeSort()` cho phép thực hiện lặp đi lặp lại tương đối đơn giản mà không cần ngăn xếp.

Phương thức thực hiện lặp được đề xuất không hoàn toàn tương đương với phương pháp đệ quy và thực hiện các phép chia theo những cách khác nhau. Quá trình này được điều khiển bởi vòng lặp `for` bên ngoài. Trong bước đầu tiên, các cặp phần tử có thứ tự được hình thành, trong - phần thứ hai, trong - phần thứ ba, v.v. Ở mỗi bước tiếp theo, các cặp danh sách có thứ tự liên tiếp được hợp nhất, dẫn đến các danh sách được sắp xếp mới có độ dài gấp đôi so với bước trước đó. Chúng ta sẽ giữ tất cả các danh sách phụ được liên kết trong một danh sách chung. Việc sát nhập diễn ra theo chu trình nội bộ. Một danh sách việc `todo` gồm các phần tử chưa xử lý được hình thành. Vòng lặp `while` nội bộ tuần tự tìm kiếm các cặp danh sách thu được ở bước trước, sau đó hợp nhất chúng. Danh sách kết quả được thêm vào cuối danh sách các phần tử đã xử lý. Việc thực hiện chu trình nội bộ tiếp tục cho đến khi tất cả các mục trong danh sách này được chuyển giao. [Sedgewick-1992]

Chương trình 7.23. Sửa sắp xếp theo hợp nhất (723mergel3.c)

```
struct list  *mergeSort(struct list *c)
{ unsigned long i, n, n2;
  struct list  *a, *b, *head, *todo, *t;
  head = (struct list  *)  malloc(sizeof(struct list));
  head->next = c;
  a  = z;
  for (n = 1;  a  != head->next; n <<= 1) {
    todo = head->next;
    c  = head;
```

```
    while (todo != z) {
      t = todo;
      /* chia mảng a[] */
      for (a = t, i = 1; i < n; i++)
        t = t->next;
      /* chia mảng b[] */
      b = t->next; t->next = z;
      for (t = b, i = 1; i < n; i++)
        t = t->next;
      /* hợp a[] và b[] */
      todo = t->next; t->next = z;
      c->next = merge(a, b);
      /* Bỏ qua mảng đã hợp nhất*/
      for (n2 = n + n, i = 1; i <= n2; i++)
        c = c->next;
    }
  }
  return head->next;
}
```

Hiệu quả của sắp xếp hợp nhất là gì? Nó chỉ ra rằng đây là một trong những thuật toán sắp xếp hiệu quả nhất đã biết với độ phức tạp thời gian trung bình được đảm bảo $\Theta(n.\log_2 n)$. Về mặt lý thuyết, đây là độ phức tạp thuật toán tốt nhất có thể đạt được bằng một thuật toán sắp xếp phổ quát (xem **??**). Như chúng ta đã thấy trong đoạn **??**, Với các giả định bổ sung cho biểu diễn dữ liệu (nhị phân), các thuật toán sắp xếp tuyến tính có thể thu được. Tuy nhiên, chúng không phổ biến và không thể được sử dụng để sắp xếp các số dấu phẩy động, chẳng hạn. Điều này đặt nó bên cạnh hình chóp (xem **??**) Và nhanh (xem **??**), Có cùng độ phức tạp thời gian trung bình. Hơn nữa, tương tự như sắp xếp theo hình chóp, sắp xếp hợp nhất có cùng độ phức tạp trong trường hợp xấu nhất (với hằng số ẩn nhỏ hơn hằng số hình chóp).

Đồng thời, sắp xếp nhanh trong trường hợp xấu nhất có độ phức tạp $\Theta(n^2)$, xếp nó trong số các thuật toán kém hiệu quả nhất như *phương pháp bong bóng*, xem **??** (Cái sau, như đã đề cập, là cơ sở của *sắp xếp nhanh*.) Khi chúng ta muốn đảm bảo độ phức tạp $\theta(n.\log_2 n)$, kể cả trong trường hợp xấu nhất, chúng ta có thể sử dụng *sắp xếp*

theo hình chóp hoặc *sắp xếp hợp nhất*.

Một nhược điểm chính của sắp xếp hợp nhất là nó yêu cầu *bộ nhớ bổ sung* tỷ lệ với n: cho một mảng bổ sung hoặc cho các con trỏ, tương ứng. Các thử nghiệm thực nghiệm cho thấy rằng, trong trường hợp trung bình, phân loại nhanh đập hình chóp từ 2-3 lần (xem [Wirth-1980]). Tốc độ sắp xếp hợp nhất chiếm một vị trí trung gian giữa hai, đó là lý do tại sao nó là một ứng cử viên nặng ký khi tốc độ là quan trọng trong trường hợp xấu nhất, đồng thời có khả năng cấp phát bộ nhớ bổ sung tỷ lệ với n.

Một ưu điểm chính khác của sắp xếp hợp nhất là quyền truy cập vào các phần tử được thực hiện *tuần tự nghiêm ngặt*, đó là lý do tại sao nó phù hợp để sắp xếp danh sách, tệp liên tiếp, v.v. Cuối cùng nhưng không kém phần quan trọng, cần lưu ý rằng việc sắp xếp theo cách hợp nhất là ổn định, tức là nó bảo toàn thứ tự tương đối của các phần tử có khóa bằng nhau. Đây là một thuộc tính quan trọng và không dành riêng cho mọi thuật toán sắp xếp. Ví dụ, phân loại nhanh là không ổn định và đòi hỏi nhiều nỗ lực để làm cho nó ổn định. Một tính năng thú vị khác của sắp xếp bằng cách hợp nhất trong biến thể của nó, khi việc hợp nhất không kiểm tra xem một trong hai danh sách có bị cạn kiệt hay không, là độ phức tạp của nó không phụ thuộc vào sự sắp xếp sơ bộ của các phần tử của tập hợp. Trong phiên bản có sự kiểm tra, sắc lệnh có ảnh hưởng không đáng kể.

Giống như hầu hết các phương pháp sắp xếp hiện đại, sắp xếp hợp nhất hoạt động kém với một số phần tử nhỏ. Cũng như phân loại nhanh, kết quả tốt sẽ thu được khi kết hợp với các phương pháp khác. Một cách tiếp cận tiêu chuẩn là sử dụng sắp xếp chèn khi số phần tử của tập hợp được đề cập giảm xuống dưới 15-20 (con số này là một ví dụ) (xem **??**).

Bài tập

▷ **7.11.** So sánh tốc độ triển khai được đề xuất để sắp xếp bằng cách hợp nhất cho: 100; 1000; 10000; 100.000 phần tử.

▷ **7.12.** Viết chương trình sắp xếp theo hợp nhất, trong đó tập hợp được chia thành:

a) 3 tập hợp con

b) k tập con, $k > 3$

▷ **7.13.** Tính toán mức độ phức tạp của các phương án sáp nhập khác nhau:

(a) tốt nhất

b) trong trường hợp xấu nhất

Tìm chuỗi đầu vào tốt nhất và kém nhất.

▷ **7.14.** Hãy triển khai một biến thể lặp đi lặp lại của sắp xếp bằng cách hợp nhất, hãy sử dụng mảng thay vì danh sách được liên kết.

▷ **7.15.** Hãy xác định bằng thực nghiệm số lượng phần tử mà sắp xếp chèn nên được sử dụng để sắp xếp hợp nhất.

▷ **7.16.** Hãy so sánh thực nghiệm tốc độ sắp xếp bằng cách hợp nhất, kết hợp với:

(a) sắp xếp chèn

b) phương pháp bong bóng

c) lựa chọn trực tiếp

▷ **7.17.** Hãy thực hiện một biến thể sắp xếp bằng cách hợp nhất, lợi dụng của sự sắp xếp tự nhiên có thể có của các phần tử trong tập nguồn (xem 7.4).

7.6. Nâng nhanh lũy thừa

Trong 1.1.1. chúng ta đã xem xét một cách lặp lại đơn giản để tìm x^n (x - số thực, n - tự nhiên). Có một thuật toán nhanh hơn (thực hiện chia tỷ lệ với ít phép nhân hơn) dựa trên công thức sau [Schwertner-1995]:

$$x^n = \begin{cases} x^{\frac{n}{2}}.x^{\frac{n}{2}}, & \text{với } n \text{ là số chẵn} \\ x^{n-1}.x, & \text{với } n \text{ là số lẻ} \end{cases}$$

Không khó để thấy rằng công thức trên dựa trên *phép chia để trị*, tuy ở dạng hơi lạ: ở mỗi bước ta có hai bài toán con, nhưng thực tế chỉ cần giải một bài không tầm thường (đối với *phần chia*).). Trong

trường hợp đầu tiên, vì hai bài toán con trùng nhau, và trong trường hợp thứ hai - vì một bài toán nhỏ. Tuy nhiên, phần *để trị* là cổ điển và dựa trên các giải pháp của hai bài toán con, một giải pháp cho vấn đề ban đầu sẽ thu được.

Chương trình 7.24. Nâng nhanh lũy thừa (724powerc.c)

```
#include <stdio.h>
const double base = 3.14;
const unsigned d = 11;
double power(double x, unsigned n)
{ if (0 == n) return 1;
  else
    if (n % 2)
      return x * power(x, n - 1);
    else
      return power(x * x, n / 2);
}

int main() {
  printf("%lf^%u = %lf\n", base, d, power(base, d));
  return 0;
}
```

Chúng ta sẽ lưu ý rằng công thức trên không cung cấp số nhân tối thiểu để thực hiện nâng theo độ. Ví dụ, đối với $n = 15$, các chương trình trên sẽ thực hiện việc nâng tuần tự như sau:
1) x^1
2) $x^2 = x^1.x^1$
3) $x^3 = x^2.x^1$
4) $x^6 = x^3.x^3$
5) $x^7 = x^6.x^1$
6) $x^{14} = x^7.x^7$
7) $x^{15} = x^{14}.x^1$
tức là với tổng số là 7 phép nhân. Có các giải pháp khác, ví dụ:
1) x^1
2) $x^2 = x^1.x^1$
3) $x^3 = x^2.x^1$
4) $x^4 = x^3.x^1$
5) $x^7 = x^4.x^3$

6) $x^8 = x^7.x^1$
7) $x^{15} = x^8.x^7$

Tuy nhiên, cũng có những chuỗi ngắn hơn chỉ gồm 6 phép nhân, ví dụ:
1) x^1
2) $x^2 = x^1.x^1$
3) $x^3 = x^2.x^1$
4) $x^5 = x^3.x^2$
5) $x^{10} = x^5.x^5$
6) $x^{15} = x^{10}.x^5$
Như :
1) x^1;
2) $x^2 = x^1.x^1$;
3) $x^4 = x^2.x^2$;
4) $x^5 = x^4.x^1$;
5) $x^{10} = x^5.x^5$;
6) $x^{15} = x^{10}.x^5$

Nếu chúng ta chỉ xem xét một loạt các chỉ số theo cấp số nhân:

1, 2, 3, 4, 7, 8, 15
1, 2, 3, 5, 10, 15
1, 2, 4, 5, 10, 15

có thể thấy rằng đây là những hàng trong đó phần tử đầu tiên là 1, và mỗi phần tiếp theo là tổng của hai phần trước nó. Một loạt như vậy được gọi là phụ gia. Thật không may, cách duy nhất đã biết để tìm một chuỗi cộng với độ dài tối thiểu là cạn kiệt hoàn toàn, điều này không được quan tâm trong đoạn này. [Nakov-1998]

Bài tập

▷ **7.18.** Đề xuất một thuật toán heuristic (xem Chương 9) luôn tìm thấy một dãy cộng không dài hơn dãy được đưa ra bởi công thức trên.

7.7. Thuật toán Strassen để nhân nhanh các ma trận

Phép nhân các ma trận là một cơ hội mới để chứng minh các khả năng của nguyên lý La Mã cũ. Chúng ta sẽ nhớ lại rằng thuật toán cổ điển để nhân ma trận $A = (a_{ij})_{m\times n}$ và $B = (b_{ij})_{n\times r}$ bắt đầu trực tiếp từ định nghĩa và được đưa ra bởi công thức:

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}.$$

Trong trường hợp ma trận vuông có kích thước $n \times n$, thời gian tính toán của ma trận C hiển nhiên là $\Theta(n^3)$: ta có n^2 phần tử và việc tính toán mỗi phần tử cần thời gian $\Theta(n)$. Tất nhiên, ở đây, chúng ta giả sử rằng phép cộng và phép nhân có độ phức tạp theo thời gian không đổi $\Theta(1)$, không phụ thuộc vào n.

Thuật toán này đơn giản và hiển nhiên đến nỗi trong một thời gian dài không ai nghĩ rằng có thể có một thuật toán khác và không cố gắng tìm kiếm nó. Tuy nhiên, vào cuối những năm 1960, Strassen đã đề xuất một cải tiến thú vị cho thuật toán, giảm độ phức tạp xuống $\log(n^{\log 7})$. Khám phá của ông gây ngạc nhiên cho giới khoa học và dẫn đến *phương pháp chia để trị* rất xứng đáng, vốn cho đến nay vẫn bị đánh giá thấp như một kỹ thuật lập trình hiệu quả. Hiệu quả to lớn của khám phá Strassen là do phép nhân các ma trận, như một phép toán cơ bản của đại số, được đưa vào nhiều thuật toán số. Bất kỳ cải tiến nào ở đây sẽ tự động làm giảm độ phức tạp tính toán của một số thuật toán khác.

Xét các ma trận A và B có kích thước 2×2:

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

Sử dụng thuật toán tiêu chuẩn, chúng ta thu được:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}, i = 1,2 \text{ và } j = 1,2$$

Có thể thấy rằng chúng ta cần 8 phép nhân và 4 phép cộng. Strassen lưu ý rằng số phép nhân có thể giảm xuống còn 7 như

sau (với P_i, chúng ta có nghĩa là các biến phụ):

$$
\begin{aligned}
P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
P_2 &= (a_{21} + a_{22})b_{11} \\
P_3 &= a_{11}(b_{12} - b_{22}) \\
P_4 &= a_{22}(b_{21} - b_{11}) \\
P_5 &= (a_{11} + a_{12})b_{22} \\
P_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
P_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
c_{11} &= P_1 + P_4 - P_5 + P_7 \\
c_{12} &= P_3 + P_5 \\
c_{21} &= P_2 + P_4 \\
c_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}
$$

Tất nhiên, không có bữa trưa miễn phí: số lượng phép cộng (bao gồm cả phép trừ) tăng lên 18. Điều này đặt ra câu hỏi về lợi thế của phương pháp Strassen trong trường hợp này, do thực tế là các bộ xử lý cũ hơn "nhân" nhiều như hai (hiếm khi là ba.) phép cộng / phép trừ. Tuy nhiên, hầu hết các bộ vi xử lý hiện đại đều thực hiện phép cộng và phép trừ ở cùng một tốc độ. Đó là, phương pháp được trình bày dường như không dẫn đến cải tiến, ít nhất là không trực tiếp. Tuy nhiên, Strassen lưu ý rằng thuật toán có thể được áp dụng một cách đệ quy. Trước khi thảo luận về cách điều này có thể xảy ra, chúng ta sẽ chỉ ra rằng chuỗi hoạt động trên không phải là duy nhất. Một khả năng hiện thực hóa khác của ý tưởng trông như thế

này (xem [Brassard, Bratley-1987]):

$$
\begin{aligned}
P_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
P_2 &= a_{11}b_{11} \\
P_3 &= a_{12}b_{21} \\
P_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\
P_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\
P_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
P_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21}) \\
c_{11} &= P_2 + P_3 \\
c_{12} &= P_1 + P_2 + P_5 + P_6 \\
c_{21} &= P_1 + P_2 + P_4 - P_7 \\
c_{22} &= P_1 + P_2 + P_4 + P_5
\end{aligned}
$$

Tất nhiên, có những cách khác để đạt được hiệu quả tương tự. Chúng ta sẽ không xem xét tất cả chúng, nhưng chúng ta vẫn sẽ bị cám dỗ để trích dẫn một cái khác ([Aho, Hopcroft, Ullman-1987]):

$$
\begin{aligned}
P_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
P_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
P_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
P_4 &= (a_{11} + a_{12})b_{22} \\
P_5 &= a_{11}(b_{12} - b_{22}) \\
P_6 &= a_{22}(b_{21} - b_{11}) \\
P_7 &= (a_{21} + a_{22})b_{11} \\
c_{11} &= P_1 + P_2 - P_4 + P_6 \\
c_{12} &= P_4 + P_5 \\
c_{21} &= P_6 + P_7 \\
c_{22} &= P_2 - P_3 + P_5 - P_7
\end{aligned}
$$

Nếu chúng ta thay thế các phần tử a_{ij}, b_{ij} và c_{ij} bằng các ma trận có kích thước $n \times n$, chúng ta nhận được một thuật toán có thể nhân các ma trận có kích thước $2n$, sử dụng 7 thay vì 8 phép nhân của các ma trận có kích thước $n \times n$. Tuy nhiên, nhân hai ma trận chắc

chắn là một phép toán khó hơn phép cộng. Thật vậy, nếu sử dụng các thuật toán cổ điển từ các định nghĩa của phép cộng, chúng ta cần n^2 tổng cơ bản, thì đối với phép nhân chúng ta cần n^3 tổng cơ bản cộng với n^3 phép nhân sơ cấp. Đó là, lưu một phép nhân có ảnh hưởng nghiêm trọng.

Nhưng tại sao lại dừng ở đó? Nếu n là một số chẵn, chúng ta có thể áp dụng cùng một sơ đồ đệ quy cho các ma trận có kích thước $(n/2) \times (n/2)$. Trong trường hợp n là lũy thừa của 2, quá trình có thể tiếp tục cho đến khi đạt được ma trận có kích thước 2×2. Do đó, chúng ta nhận được một thuật toán đệ quy, mà cây ở mỗi bước có phân nhánh là 7 và được duyệt theo chiều sâu. Độ phức tạp $T(n)$ của thuật toán được mô tả trong trường hợp ma trận vuông cỡ n, bậc 2, được cho bởi sự phụ thuộc lặp lại:

$$T(n) = 7T(n/2) + 18(n/2)^2, n > 2$$

Do đó theo định lý cơ bản (xem 1.4.10.) Ta thu được $T(n) \in \Theta(n^{\log 7})$. Nhưng $\log 7 \approx 2,81 < 3$, tức là Thuật toán Strassen có độ phức tạp về thời gian tốt hơn so với thuật toán cổ điển.

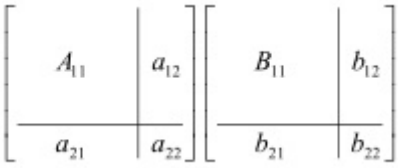
Điều gì sẽ xảy ra nếu chúng ta có ma trận vuông nhưng n không phải là lũy thừa của 2? Một giải pháp tiêu chuẩn cho vấn đề này là bổ sung các ma trận không có hàng và cột vào ít nhất n, đó là lũy thừa của 2. Kỹ thuật này được gọi là phép cộng tĩnh. Điều này có thể dẫn đến việc tăng tối đa kích thước của ma trận lên gấp đôi. Và thực hiện một phép lặp khác, theo sự phụ thuộc lặp lại ở trên, có thể tăng số lần nhân lên tối đa 7 lần.

Kể từ khi thuật toán Strassen được công bố, một số nỗ lực thành công và không thành công đã được thực hiện để cải thiện nó. Điểm yếu của thuật toán nằm ở đâu và chúng có thể được củng cố như thế nào? Một vấn đề đã trở nên rõ ràng: việc bổ sung trước ma trận không hiệu quả, có thể làm chậm nó tới 7 lần. Vấn đề thứ hai cũng đã được giải quyết ngay từ đầu: tại $n = 2$, chúng ta thực tế có sự suy giảm thay vì cải thiện. Và $n = 3, 4, ...$ thì sao? Ở đâu (với n) chúng ta nên ngừng áp dụng đệ quy thuật toán Strassen và sử dụng thuật toán cổ điển ở đâu?

Một giải pháp khả thi cho vấn đề đầu tiên (n không phải là lũy thừa của 2) là hoãn việc thêm ma trận với số không cho đến khi điều

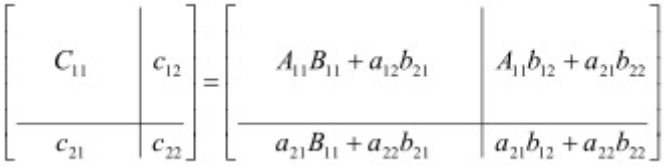
này có thể thực hiện được. Tức là, nếu ban đầu n là chẵn, chúng ta có thể thực hiện một phép lặp của thuật toán. Nếu $n/2$ chẵn, chúng ta có thể làm một cái khác, và nếu nó là lẻ - chúng ta sẽ phải bổ sung nếu cần. Tuy nhiên, chỉ bằng cách bổ sung ngay bây giờ, chúng ta cuối cùng đã tiết kiệm được việc chèn thêm nhiều số không. Kỹ thuật này được gọi là bổ sung năng động.

Mặc dù dẫn đến cải thiện đáng kể, nhưng bổ sung động không phải là phương pháp tối ưu. Huss và Lederman sử dụng một kỹ thuật khác để xử lý các kích thước kỳ lạ: bóc tách động. Ý tưởng là tách một hàng và một cột. Ma trận "bóc tách" thu được có kích thước chẵn n và bước đệ quy của Strassen có thể được áp dụng cho nó. (xem Hình 7.6)



Hình 7.6. Động "bóc tách" của A và B.

Kết quả thu được sau đó phải được kết hợp với ma trận kết quả C_{11} (xem Hình 7.7).



Hình 7.7. Kết quae C là tích của A và B.

Vấn đề thứ hai không thể được tấn công trực tiếp, vì nó yêu cầu so sánh giữa tốc độ của thuật toán cổ điển và tốc độ của thuật toán Strassen tại một n cố định. Điều này đòi hỏi phải tính đến thời gian truy cập vào các phần tử ma trận trong cả hai phương pháp, điều này phụ thuộc vào máy. Một nghiên cứu năm 1996 của Luoma

và Spider cho thấy số lượng bộ nhớ đệm có tầm quan trọng lớn và trong hầu hết các trường hợp thực tế, thuật toán ban đầu của Strasen hoạt động kém hơn thuật toán cổ điển. Họ đưa ra ý tưởng về việc triển khai thuật toán, trong đó các phép tính trung gian được thực hiện ngay trước khi chúng cần thiết, điều này đảm bảo rằng chúng vẫn nằm trong bộ nhớ cache cho đến lần sử dụng tiếp theo.

Một cải tiến bất ngờ khác của thuật toán là Vinograd. Ông đã cố gắng giảm số lượng các phép cộng / trừ từ 18 xuống còn 15, giữ nguyên số phép nhân. Đề án của anh ta có dạng:

$$
\begin{aligned}
S_1 &= a_{21} + a_{22} \\
S_2 &= S_1 \breve{} a_{11} \\
S_3 &= a_{11} \breve{} a_{21} \\
S_4 &= a_{12} \breve{} S_2 \\
T_1 &= b_{12} \breve{} b_{11} \\
T_2 &= b_{22} \breve{} T_1 \\
T_3 &= b_{22} \breve{} b_{12} \\
T_4 &= b_{21} \breve{} T_2 \\
P_1 &= a_{11} b_{11} \\
P_2 &= a_{12} b_{21} \\
P_3 &= S_1 T_1 \\
P_4 &= S_2 T_2 \\
P_5 &= S_3 T_3 \\
P_6 &= T_4 b_{22} \\
P_7 &= a_{22} T_4 \\
U_1 &= P_1 + P_4 \\
U_2 &= U_1 + P_5 \\
U_3 &= U_1 + P_3 \\
c_{11} &= P_1 + P_2 \\
c_{21} &= U_2 + P_7 \\
c_{22} &= U_2 + P_3 \\
c_{12} &= U_3 + P_6
\end{aligned}
$$

Một số nỗ lực khác đã được thực hiện để giảm số lượng phép nhân

và phép cộng trong sơ đồ cổ điển của Strassen ở $n = 2$, nhưng tất cả đều không thành công. Năm 1971, Hopcroft và Kerr đã chứng minh rằng điều này là không thể nếu không sử dụng phép nhân giao hoán (phép nhân ma trận không giao hoán). Sau đó, người ta đã tìm ra một cách để nhân ma trận 3×3 với tối đa 21 phép nhân, thuật toán này đưa ra một thuật toán có độ phức tạp $\Theta(n^{\log_3 21})$, tức là $\Theta(n^{2,771244})$. Có một số cải tiến khác với việc tăng kích thước ma trận và ít cơ hội áp dụng thực tế hơn. Ví dụ, Pan đã tìm ra cách nhân 70×70 ma trận với 143.640 phép nhân sơ cấp (so với 343.000, theo thuật toán cổ điển). Sau đó, các thuật toán thậm chí còn hiệu quả hơn về mặt lý thuyết và thậm chí không thể áp dụng được trong thực tế: $\Theta(n^{2,521813})$ - 1979, $\Theta(n^{2,521801})$ - 1980, $\Theta(n^{2,376})$ - 1986

Bài tập

▷ **7.19.** Thực nghiệm để xác định thời điểm gián đoạn đệ quy và áp dụng thuật toán cổ điển.

▷ **7.20.** Hãy kiểm tra trình tự do Vinograd đề xuất để nhân nhanh các ma trận.

▷ **7.21.** Hãy đề xuất một trình tự khác để nhân nhanh ma trận.

7.8. Nhân nhanh các số dài

Thuật toán cổ điển để nhân các số được học ở trường yêu cầu thời gian của bậc $\Theta(n^2)$ để nhân hai số có n chữ số (chúng ta giả định rằng phép cộng và nhân các số có một chữ số diễn ra trong một thời gian không đổi độc lập với n). Chúng ta đã quá quen với thuật toán này đến nỗi hầu như không ai trong chúng ta nghĩ đến tính hiệu quả và khả năng cải tiến của nó.

Hãy xem *phương pháp chia để trị* có thể mang lại cho chúng ta điều gì. Giả sử chúng ta muốn nhân hai số có n chữ số X và Y, và bây giờ chúng ta coi n là lũy thừa của 2 (chúng ta sẽ hướng dẫn bạn cách thực hiện điều này sau khi trường hợp này không xảy ra). Chia X và Y thành hai phần A, B và C, D, mỗi phần có $n/2$ chữ số sao

cho:

$$X = 2^{n/2}.A + B$$
$$Y = 2^{n/2}.C + D$$

Từ đây cho tích XY, chúng ta nhận được:

$$XY = 2^n.AC + 2^{n/2}.(AD + BC) + BD$$

Như vậy, để tính giá trị của XY theo công thức trên, cần 4 phép nhân các số có $n/2$ chữ số, 3 phép cộng các số có tối đa $n + 1$ chữ số và hai lần dịch sang trái ở $n/2$ vị trí (phép nhân với 2^n và $2^{n/2}$). Nếu chúng ta sử dụng đệ quy cùng một công thức để tính các tích AC, AD, BC và BD, và quá trình tiếp tục cho đến khi chúng ta đạt đến trường hợp cơ bản của hai số có một chữ số, chúng ta nhận được sự phụ thuộc lặp lại:

$$T(1) = 1$$
$$T(n) = 4T(n/2) + cn, n > 1 \text{ và } n - \text{ bậc } 2$$

Áp dụng định lý cơ bản (xem 1.4.10.), Chúng ta thu được $T(n) \in \Theta(n^2)$, tức là độ phức tạp trùng với độ phức tạp của thuật toán ban đầu, có nghĩa là phép chia đơn giản không dẫn đến cải tiến: chúng ta cũng cần một chút tháo vát. Chúng ta hãy nhớ lại thuật toán Strasen ở trên (xem **??**), Trong đó sự cải tiến đến từ việc giảm số lượng phép nhân. Hãy xem liệu chúng ta có thể áp dụng nguyên tắc tương tự ở đây không. Nếu chúng ta xem xét kỹ hơn công thức, chúng ta sẽ thấy rằng chúng ta không nhất thiết phải tính các tích AD và BC: tổng của chúng là đủ. Hãy xem xét công thức:

$$XY = 2^n.AC + 2^{n/2}.[(A - B)(D - C) + AC + BD] + BD$$

Chúng ta sẽ không triển khai thuật toán hoạt động, vì điều này liên quan đến các hoạt động với "số dài" (tức là không phù hợp với các kiểu tiêu chuẩn), điều này sẽ làm lộn xộn mã. (Hơn nữa, một cách triển khai tốt có thể sẽ không sử dụng hệ thống số thập phân mà là hệ thống số tối đa dựa trên lũy thừa của 2, cho phép kết hợp kết quả của phép nhân hai chữ số thành một từ máy. Tuy nhiên, điều

này nằm ngoài phạm vi của sự cân nhắc của chúng ta.) chúng ta sẽ cố gắng làm rõ điều đó bằng một mã giả. Chúng ta giả sử rằng X và Y là các số nguyên có dấu và n là lũy thừa của 2. Trong trường hợp giới hạn $n = 1$, chúng ta sẽ nhân các số trực tiếp. [Aho, Hopcroft, Ullman - 1987]

Giả mã nhân số dài

```
int mult (int X, Y, n)
{
  int s; / * dấu hiệu của tích XY * /
  int m1, m2, m3; / * ba tích * /
  int A, B, C, D; / * một nửa của X và Y * /
  / * kiểm tra trường hợp biên * /
  if (1 == n)
    return (X * Y);
  s = sign(X) *sign(Y);
  X = abs (X);
  Y = abs (Y);
  A = n/2 bit bên trái của X;
  B = n/2 bit phải của X;
  C = n/2 bit bên trái của Y;
  D = n/2 bit phải của Y;
  m1 = mult (A, C, n/2);
  m2 = mult (A-B, D-C, n/2); / * *** * /
  m3 = mult (B, D, n/2);
  return s * (m1 << n + (m1 + m2 + m3) << (n / 2) + m3); / * *** * /
}
```

Chúng ta cũng có thể sử dụng phiên bản thứ hai của công thức, trong đó các dòng được đánh dấu bằng / * *** * / sẽ được thay thế bằng các dòng sau:

```
m2 = mult(A+B,C+D,n/2); /* *** */
return s * (m1<<n + (m2-m1-m3)<<(n/2) + m3); /* *** */
```

Điều gì xảy ra khi n không phải là lũy thừa của 2? Nếu n chẵn thì không có vấn đề gì ở bước đầu tiên và chúng ta có thể chia số thành hai phần có độ dài bằng nhau. Nếu $n/2$ chẵn, chúng ta có thể tiếp tục như vậy, nhưng nó như thể ở một bước nào đó chúng ta sẽ nhận được độ dài lẻ và điều này sẽ không thể thực hiện được. Một giải pháp rõ ràng cho vấn đề là bổ sung số bên trái với số 0 ở đầu.

Mặc dù điều này làm giảm hiệu quả của thuật toán, nhưng phân tích cho thấy rằng độ phức tạp của nó vẫn là $\Theta(n^{\log_2 3})$.

Một điểm thú vị khác đã bị bỏ qua trong phân tích ở trên là thực tế là tổng của hai số có $n/2$ chữ số không nhất thiết cho một số có n chữ số, nhưng có thể cho $(n+1)$ chữ số. Khi đó, tích $(A+B)(C+D)$, trong phiên bản thứ hai của công thức, có thể là $(n+1)$ - ở dạng số (các tích khác AC và BD không có cơ hội như vậy.). Lấy $X = 9999$ và $Y = 9998$. Ta có:

A = 99, B = 99, C = 99 và D = 98
AC = 99.99 = 9801
BD = 99.98 = 9702
(A+B)(C+D) = 198.197 = 39006 (nghĩa là 5-số chữ số)

May mắn thay, mức tăng nhiều nhất là 1 và độ phức tạp cuối cùng của thuật toán và trong trường hợp này sẽ là $\Theta(n^{\log_2 3})$. Chúng ta sẽ không phân tích chi tiết về những trường hợp này. Độc giả tò mò có thể tìm thêm chi tiết trong [Brassard, Bratley - 1987].

Làm gì khi cả hai số có độ dài m và n, $m \neq n$? Để xác định, giả sử rằng $m < n$. Nếu chúng ta điền số ngắn hơn bằng các số 0 ở đầu, chúng ta sẽ nhận được một thuật toán có độ phức tạp $\Theta(n^{log_2 3})$ so với $\Theta(mn)$ trong thuật toán cổ điển. Từ đó có thể dễ dàng thấy rằng thuật toán cổ điển tốt hơn so với các cải tiến ở $m < n^{(\log_2 3)-1}$. Một giải pháp hay là chia các chữ số của số khối có độ dài lớn hơn m. Mỗi khối này được nhân với số đầu tiên bằng cách sử dụng thuật toán cải tiến, sau đó với sự trợ giúp của $\lceil n/m \rceil$ các phép cộng và hiệu số bổ sung, kết quả cuối cùng sẽ thu được. Do đó, tổng độ phức tạp của thuật toán được sửa đổi là $\Theta(mn^{(\log_2 3)-1})$.

Câu hỏi logic được đặt ra: Nếu thuật toán này thực sự tốt như vậy, tại sao chúng ta không học nó ở trường? Vì hai lý do: thứ nhất, nó phức tạp hơn để giải thích và khó áp dụng hơn nhiều, đặc biệt là đối với những con số dài hơn. Khái niệm "đệ quy" là trừu tượng và khó hiểu. Mặt khác, có tính đến các hằng số ẩn, chúng ta có thể mong đợi rằng thuật toán cải tiến tốt hơn thuật toán cổ điển cho các số có ít nhất 500 chữ số. Hầu như không ai thực hiện các phép tính thủ công như vậy ở trường...

Bài tập

▷ **7.22.** Hãy thực hiện thuật toán được đề xuất để nhân nhanh các số dài.

▷ **7.23.** Hãy xác định bằng thực nghiệm, thuật toán nhân các số dài tốt hơn thuật toán cổ điển có bao nhiêu chữ số.

▷ **7.24.** Hãy cải thiện thuật toán nhân nhanh các số dài, vì mục đích này, các số được chia thành ba thay vì hai phần. Chứng tỏ rằng 5 (thay vì 9) phép nhân với các số có độ dài $n/3$ là đủ để tính tích. Xác định độ phức tạp của thuật toán kết quả.

▷ **7.25.** Dựa vào bài toán trước để chứng tỏ rằng phép nhân $2k-1$ là đủ để tính tích trong trường hợp ngắt nhóm gồm k chữ số liên tiếp. Chứng tỏ rằng do đó có sự tồn tại của thuật toán nhân các số dài có độ phức tạp $\Theta(n^x)$ với mọi $x > 1$.

7.9. Bài toán tháp Hà Nội

Bài toán: Có n số đĩa có đường kính khác nhau và ba trụ A, B, C. Các đĩa được xâu vào trụ thứ nhất theo thứ tự giảm dần và tạo thành một hình tháp. (xem Hình 7.8) Chúng phải được chuyển từ trụ đầu tiên sang trụ cuối cùng tuân theo các quy tắc sau:

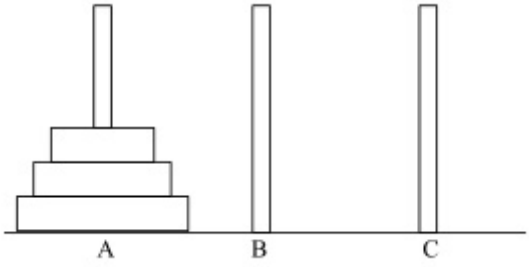
1. Mỗi lượt có thể di chuyển một đĩa và đĩa này phải là đầu của một trong các trụ.

2. Đĩa có đường kính lớn hơn không được xếp chồng lên đĩa có đường kính nhỏ hơn.

Chúng ta sẽ trình bày biến thể đệ quy cổ điển dựa trên phương pháp chia để trị, cũng như một số giải pháp lặp thú vị nhất. Đối với những độc giả tỏ ra quan tâm đến vấn đề này, chúng ta khuyên bạn nên làm quen với [Banchev-1993], nơi một số thuật toán khác được chỉ ra.

Thuật toán 1

Đây là thuật toán cổ điển để giải quyết vấn đề. Ý tưởng là để di chuyển n đĩa từ cột A sang cột C, cần phải di chuyển $n-1$ đĩa từ cột Asang cột B, sau đó di chuyển đĩa có số n (là lớn nhất trong số họ) từ cột A sang cột C và cuối cùng là di chuyển $n-1$ đĩa còn lại từ cột



Hình 7.8. Tháp Hà Nội ban đầu

B sang cột *C*. Do đó, chúng ta tự nhiên giảm vấn đề thành giải hai bản sao nhỏ hơn khác của nó (chia), có liên kết, cùng với một phép toán bổ sung (chuyển đĩa lớn nhất) cho chúng ta giải pháp mong muốn (quy tắc). Việc triển khai thuật toán này trông giống như sau:

Chương trình 7.25. Bài toán tháp Hà Nội (725hanoi.c)

```
#include <stdio.h>
const unsigned n = 4;

void diskMove(unsigned n, char a, char b)
{ printf("Di chuyển đĩa %u tư %c sang %c.\n", n, a, b); }

void hanoi(char a, char c, char b, unsigned numb)
{ if (1 == numb) diskMove(1, a, c);
  else {
    hanoi(a, b, c, numb - 1);
    diskMove(numb, a, c);
    hanoi(b, c, a, numb - 1);
  }
}

int main() {
  printf("Số đĩa: %u\n", n);
  hanoy('A', 'C', 'B', n);
  return 0;
}
```

Sau đây là kết quả mẫu từ việc thực hiện chương trình cho $n = 2, 3$ *và* 4:

```
Số đĩa: 2
Di chuyển đĩa 1 từ A đến B.
Di chuyển đĩa 2 từ A đến C.
Di chuyển đĩa 1 từ B đến C.
Số đĩa: 3
Di chuyển đĩa 1 từ A đến C.
Di chuyển đĩa 2 từ A đến B.
Di chuyển đĩa 1 từ C sang B.
Di chuyển đĩa 3 từ A đến C.
Di chuyển đĩa 1 từ B đến A.
Di chuyển đĩa 2 từ B đến C.
Di chuyển đĩa 1 từ A đến C.
Số đĩa: 4
Di chuyển đĩa 1 từ A đến B.
Di chuyển đĩa 2 từ A đến C.
Di chuyển đĩa 1 từ B đến C.
Di chuyển đĩa 3 từ A đến B.
Di chuyển đĩa 1 từ C sang A.
Di chuyển đĩa 2 từ C sang B.
Di chuyển đĩa 1 từ A đến B.
Di chuyển đĩa 4 từ A đến C.
Di chuyển đĩa 1 từ B đến C.
Di chuyển đĩa 2 từ B đến A.
Di chuyển đĩa 1 từ C sang A.
Di chuyển đĩa 3 từ B đến C.
Di chuyển đĩa 1 từ A đến B.
Di chuyển đĩa 2 từ A đến C.
Di chuyển đĩa 1 từ B đến C.
```

Đối với một bài toán nhất định, có nhiều cách khác nhau để chuyển đổi một giải pháp đệ quy thành một giải pháp lặp lại. Chúng ta sẽ không tập trung vào phương pháp tiêu chuẩn để chuyển đổi đệ quy thành lặp lại bằng cách sử dụng ngăn xếp, mà sẽ chỉ xem xét các thuật toán đủ rõ ràng dựa trên một bộ quy tắc đảm bảo rằng có thể có một bước di chuyển duy nhất ở mỗi bước. Chúng ta sẽ để người đọc tự thấy rằng tất cả các thuật toán sau đây đều xây dựng một giải pháp giống như thuật toán 1. Trên thực tế, nó là một phát biểu khác về cùng một ý tưởng.

Thuật toán 2

Việc xây dựng giải pháp được xác định bởi các quy tắc sau:

1. Nếu n lẻ, mỗi đĩa có số lẻ luôn được chuyển sang phải và những đĩa có số chẵn luôn được chuyển sang trái. Nếu n chẵn thì - chiều ngược lại.

2. Không có hai lần chuyển động liên tiếp nào được thực hiện cùng một đĩa.

3. Không đặt đĩa lớn hơn trên đĩa nhỏ hơn.

Thuật toán 3

Quy tắc:

1. Đối với các lần di chuyển số lẻ, đĩa nhỏ nhất di chuyển sang phải hoặc trái, tùy thuộc vào việc n là lẻ (trái) hay chẵn (phải).

2. Trong trường hợp chuyển động là số chẵn, chuyển động duy nhất có thể được thực hiện trong đó đĩa nhỏ nhất không tham gia.

3. Không đặt đĩa lớn hơn trên đĩa nhỏ hơn.

Thuật toán 4

Một trong các cột được chọn là đã đánh dấu và hướng thay đổi của cột được đánh dấu được cố định. Tại n lẻ, C được đánh dấu ban đầu và hướng thay đổi là sang trái; đối với n chẵn B được đánh dấu và hướng thay đổi được đảo ngược.

1. Nếu có các đĩa trên hai trụ không được đánh dấu, di chuyển đĩa nhỏ hơn trong hai đĩa trên cùng sang đĩa lớn hơn. Nếu một trong các cột chưa được đánh dấu trống, đĩa trên cùng của cột kia sẽ được di chuyển trên đó. Nếu cái kia trống, thì bài toán đã được giải quyết.

2. Vị trí tiếp giáp với cực được đánh dấu theo hướng thay đổi đã được cố định trước được chọn để đánh dấu.

Bài tập

▷ **7.26.** Hãy triển khai thực hiện các thuật toán 2, 3 và 4.

▷ **7.27.** Chứng minh rằng các thuật toán 2, 3 và 4 thực hiện các hành động tương tự như thuật toán 1.

7.10. Tổ chức giải vô địch bóng đá

Một bài toán kinh điển khác, được giải quyết một cách hiệu quả với sự giúp đỡ của sự phân chia để trị, là chuẩn bị một kế hoạch giải đấu kiểu "mọi người chống lại mọi người". Ví dụ:

Giải vô địch bóng đá . Bộ Văn Hóa và Thể thao của một nước ASEAN phải tổ chức một giải vô địch bóng đá với n đội (đánh số từ 1 đến n). Chức vô địch phải được tổ chức trong n vòng nếu n lẻ, và $n-1$ vòng nếu n là chẵn. Cứ hai đội thi đấu với nhau đúng một lần. Tạo một chương trình quản lý chương trình của giải vô địch.

Kết quả phải là một bảng gồm n hàng. Số từ cột thứ j của hàng thứ i của bảng có nghĩa là số của đội đấu với đội có số i trong vòng tròn số j. (Nếu đội i đấu với đội k ở vòng j, điều này đương nhiên có nghĩa là đội k đấu với đội i trong cùng một vòng.)

Nếu đội thứ i nghỉ ở vòng thứ j, thì số thứ j ở hàng thứ i là 0.

Ví dụ: đối với $n = 3$ và $n = 4$, lịch trình tương ứng sẽ như thế này (dọc: đội, ngang: vòng):

2	3	0
1	0	3
0	1	2
3	2	1

2	3	4
1	4	3
4	1	2

Lời giải: Chúng ta sẽ cho phép mình thay đổi một chút hình dạng của kết quả. Thay vì bảng được chỉ định trong điều kiện, chúng ta sẽ tạo một bảng khác, luôn có kích thước $n \times n$ (thậm chí đối với n chẵn), thuộc loại "đội-nhóm", như trên hàng thứ i và cột thứ j sẽ là vòng tròn, trong đó sẽ gặp các đội có số thứ tự i và j ($1 \leq i, j \leq n$). Chúng ta để việc chuyển đổi bảng sang định dạng yêu cầu trong bài toán cho người đọc như một bài tập dễ dàng.

Rõ ràng bài toán không có ý nghĩa gì ở $n = 1$. Tại $n = 2$, đúng một trò chơi được chơi, mất đúng một ngày: xem Bảng 7.1.

Chúng ta giới thiệu ký hiệu viết tắt $i : j$, có nghĩa là "đội ta gặp đội j" ($1 \leq i, j \leq n$). Trong trường hợp bốn đội, giải đấu kết thúc sau 3 ngày. Làm thế nào để xây dựng lịch trình cuộc họp phù hợp? Chúng ta có thể chia các đội thành các cặp, như trong ngày đầu tiên

Đội	1	2
1	0	1
2	1	0

Bảng 7.1. Lịch thi đấu của 2 đội

1: 2 và 3: 4, vào ngày thứ hai 1: 3 và 2: 4, và vào ngày thứ ba - 1: 4 và 2: 3. Ta được Bảng 7.2. Khi xem xét kỹ hơn, chúng ta nhận thấy rằng

Đội	1	2	3	4
1	0	1	2	3
2	1	0	3	2
3	2	3	0	1
4	3	2	1	0

Bảng 7.2. Lịch thi đấu của 4 đội

bảng này chứa gấp đôi bảng cho $n = 2$ và hai lần bảng

$$\begin{matrix} 2 & 3 \\ 3 & 2 \end{matrix}$$

Lưu ý rằng sau này thu được từ

$$\begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix}$$

bằng cách thêm phần tử 2.

Bạn đã biết làm thế nào chúng ta có thể có được kế hoạch giải đấu tương ứng cho $n = 8$? Sao chép bảng cho $n = 4$ bốn lần, sau đó thêm từng phần tử số 4 vào các bảng có kích thước 4 bên ngoài đường chéo chính. (xem Bảng 7.3)

Bảng 7.3 được xác minh trực tiếp. là một lịch thi đấu thực sự: mỗi hàng và mỗi cột chứa một hoán vị của các số từ 0 đến 7 mà không lặp lại, tức là giải đấu kết thúc sau 7 ngày, với mỗi đội thi đấu đúng

Đội	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
3	2	3	0	1	6	7	4	5
4	3	2	1	0	7	6	5	4
5	4	5	6	7	0	1	2	3
6	5	4	7	6	1	0	3	2
7	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

Bảng 7.3. Lịch thi đấu của 8 đội.

một trận mỗi ngày. Bảng là đối xứng, tức là nếu $i : j$ thì $j : i$ đúng, và ngược lại ($1 \leq i, j \leq n$). Có các số không trên đường chéo chính, tức là $i : i$ không đúng với $1 \leq i \leq n$. Theo cách tương tự, chúng ta nhận được bảng cho $n = 16, 32, 64, ...$ và nói chung cho n, bậc chính xác là 2. Điều này dễ dàng theo sau bằng quy nạp. Chúng ta để lại phần chứng minh cho người đọc như một bài tập dễ dàng. Hãy suy nghĩ lại. Cho $n = 2^k$. Để có một bảng cho n, chúng ta cần lấy một bảng cho $n/2$ (phép chia), sau đó sao chép nó 4 lần và thêm một hằng số vào hai trong số các bản sao (quy tắc). Sau đây là một ví dụ triển khai thuật toán được mô tả.

Chương trình 7.26. Lịch thi đấu giải vô địch(726tourn1.c)

```
#include <stdio.h>
#define MAX 100
unsigned m[MAX][MAX];

void copyMatrix(unsigned stX, unsigned stY, unsigned cnt, unsigned
     add)
{ unsigned i, j;
  for (i = 0; i < cnt; i++)
    for (j = 0; j < cnt; j++)
      m[i + stX][j + stY] = m[i + 1][j + 1] + add;
}
```

```
void findSolution(unsigned n) /*Xây dựng bảng*/
{ unsigned i;
  m[1][1] = 0;
  for (i = 1; i <= n; i <<= 1) {
    copyMatrix(i + 1, 1, i, i);
    copyMatrix(i + 1, i + 1, i, 0);
    copyMatrix(1,i + 1, i, i);
  }
}

void print(unsigned n) /* In ra kết quả*/
{ unsigned i, j;
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++)
      printf("%3u", m[i][j]);
    printf("\n");
  }
}

int main() {
  const unsigned n = 8;
  findSolution(n);
  print(n);
  return 0;
}
```

Thật không may, ở dạng này, thuật toán không phù hợp với n, không phải là lũy thừa của 2. Chúng ta để người đọc cố gắng sửa đổi để nó hoạt động với mọi n tự nhiên. Dưới đây chúng ta sẽ xem xét một thuật toán khác hợp lệ cho mỗi n.

Đối với n bảng lẻ, chúng ta sẽ tạo ra như sau: Chúng ta sẽ không điền vào bảng $n \times n$, mà là một bảng với một cột bổ sung khác, tức là $n \times (n+1)$. Chúng ta sẽ điền vào các hàng, bắt đầu từ cột ngoài cùng bên trái và di chuyển sang bên phải. Khi chúng ta điền vào cột $(n+1)$ của hàng hiện tại, chúng ta sẽ chuyển sang hàng tiếp theo. Chúng ta sẽ bắt đầu điền với giá trị 1 trong hàng 1 của cột 1, và chúng ta sẽ điền vào ô tiếp theo với giá trị tiếp theo trong khoảng $[1; n]$. Khi đạt đến giá trị của n, chúng ta sẽ bắt đầu lại từ 1. Bảng 7.4. hiển thị kết quả tại $n = 5$. Tuy nhiên, có thể thấy rằng chúng ta có

thể không điền vào cột thứ $(n+1)$ - nó trùng với cột đầu tiên. (Tại sao?)

Đội	1	2	3	4	5	6
1	1	2	3	4	5	1
2	2	3	4	5	1	2
3	3	4	5	1	2	3
4	4	5	1	2	3	4
5	5	1	2	3	4	5

Bảng 7.4. Bắt đầu điền lịch đấu cho 5 đội.

Sau khi điền xong, chúng ta đặt lại tất cả các phần tử trên đường chéo chính của ma trận (xem Bảng 7.5). Với thậm chí n, chúng ta chỉ

Đội	1	2	3	4	5	6
1	0	2	3	4	5	1
2	2	0	4	5	1	2
3	3	4	0	1	2	3
4	4	5	1	0	3	4
5	5	1	2	3	0	5

Bảng 7.5. Lịch thi đấu của 5 đội.

có thể giới thiệu một đội hư cấu. Tuy nhiên, ở đây, chúng ta sẽ áp dụng một chiến lược khác. Chúng ta sẽ giải bài toán cho $n-1$, là số lẻ, sau đó, trong khi đặt lại đường chéo chính, chúng ta sẽ điền thêm vào hàng và cột thứ n: đội thứ k phải chơi với đội thứ n, vì không có n người đi nghỉ.

Ví dụ, để lập lịch trình cho $n = 4$, trước tiên chúng ta điền vào bảng cho $n = 3$ (xem Bảng 7.6).

Sau đó, chúng ta sẽ đặt lại đường chéo chính và điền vào hàng thứ n như trong Bảng 7.7. Sự thay đổi tương ứng của hàm `findSolution()` như sau

Đội	1	2	3	4
1	1	2	3	1
2	2	3	1	2
3	3	1	2	3

Bảng 7.6. Bắt đầu điền lịch họp cho 3 đội.

Đội	1	2	3	4
1	0	2	3	1
2	2	0	1	2
3	3	1	0	3
4	1	2	3	0

Bảng 7.7. Lịch thi đấu của 4 đội.

Chương trình 7.27. Sửa đổi hàm findSolution() (727tourn2.c)

```
void findSolution(unsigned n) /* Thiết lập bảng */
{ unsigned i;
  unsigned saveN = n;
  if (n % 2 == 0) /* Nếu n chẵn bài toán đưa về n-1 */
    n--;
  /* Điền vào bảng n n - ở đây đảm bảo số lẻ. */
  for (i = 0; i < n * (n + 1); i++)
    m[i % (n + 1)][i / (n + 1)] = i % n + 1;
  /* Phục hồi với giá trị của n */
  n = saveN;

  for (i = 0; i < n; i++) {
    if (n%2 == 0) /* điền vào cột và hàng cuối cùng số chẵn n */
      m[i][n - 1] = m[n - 1][i] = m[i][i];
    m[i][i] = 0; /* Đường chéo chính điền với 0 */
  }
}
```

Giải pháp trên, mặc dù hiệu quả, nhưng không cho phép trả lời trực tiếp cho câu hỏi: Đội i và đội j gặp nhau trong vòng tròn nào? Điều này yêu cầu hoàn thành đầy đủ hoặc ít nhất một phần của

bảng liên quan. Khi xem xét kỹ hơn các bảng trên, có thể rút ra các công thức thích hợp.

Hãy có một số người tham gia chẵn ($n = 2k$). Cho người tham gia A và B có số s và t khác n. Sau đó chơi trong vòng tròn $s + t - 1$ nếu $s + t \leq n$, và trong $s + t - n$ nếu $s + t > n$. Trong một giải đấu cờ vua, ai chơi với quân trắng và ai - với quân đen cũng rất quan trọng. Một giải pháp khả thi là tuân theo quy tắc: người chơi có số nhỏ hơn sẽ chỉ chơi với người quân trắng nếu $s + t$ là số lẻ. Người có số n chơi với người có số l trong vòng tròn $2l - 1$, nếu $2l \leq n$ và trong vòng $2l - n$, khi $2l > n$. Trong một giải đấu cờ vua có số người tham gia từ 1 đến $n/2$, anh ta chơi với quân đen và những người khác với quân trắng. Trong trường hợp có số lượng người tham gia lẻ, chúng ta thêm một người tham gia hư cấu và người chơi cùng anh ta sẽ ở trong vòng kết nối tương ứng.

Mặc dù sơ đồ sau phù hợp hơn để có được câu trả lời trực tiếp cho câu hỏi "i và j đóng trong vòng tròn nào?", Nhưng nó cũng có thể được sử dụng để điền vào bảng do tính đơn giản của công thức. Một lần nữa, chúng ta có thể từ bỏ nhóm hư cấu và điền trực tiếp vào các giá trị tương ứng của trụ cuối cùng. Đây là cách chúng ta nhận được:

Chương trình 7.28. Sửa đổi hàm findSolution() (728tourn3.c)

```
void findSolution(unsigned n) /* Thiết lập bảng */
{ unsigned i, j;
  unsigned saveN = n;
  if (n % 2 == 0) /* nếu n chẵn, bài toán đưa về n-1 */
    n--;

  /* Điền đầy bảng cho n - đây đảm bảo số lẻ. */
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if ((m[i][j] = i + j + 1) > n)
        m[i][j] -= n;

  /* Khôi phục giá trị của n */
  n = saveN;
  for (i = 0; i < n; i++) {
    if (n % 2 == 0) /* điền cột và hàng cuối, số chẵn n */
```

```
            m[i][n - 1] = m[n - 1][i] = m[i][i];
        m[i][i] = 0; /* Đến đường chéo chính bằng 0 */
    }
}
```

Hàm trên khác một chút so với cách lập luận lý thuyết bởi vì chỉ mục của mảng trong C bắt đầu từ 0 thay vì 1.

Bài tập

▷ **7.28.** Hãy chuyển đổi bảng đã xây dựng thành một bảng thuộc loại được yêu cầu trong điều kiện của bài toán từ Giải thể thao vô địch Việt Nam.

▷ **7.29.** Hãy chứng minh tính đúng đắn của thuật toán đầu tiên để biên soạn lịch thi đấu cho n bậc 2.

▷ **7.30.** Hãy thực hiện thuật toán thứ hai để biên soạn lịch thi đấu, giới thiệu một đội giả tưởng cho n chẵn.

▷ **7.31.** Hãy chứng minh tính đúng đắn của thuật toán thứ hai để biên soạn lịch thi đấu cho mỗi n tự nhiên, $n \neq 2$.

▷ **7.32.** Hãy suy ra các công thức đã cho để lập lịch thi đấu cờ vua.

7.11. Sự dịch chuyển tuần hoàn của các phần tử mảng

Vào những năm 1960, Carnigan và Plodger đã gặp phải vấn đề chuyển vị theo chu kỳ các phần tử mảng khi phát triển một trình soạn thảo văn bản cho hệ thống UNIX. Khả năng tính toán khiêm tốn của các máy tính thời đó đặt ra các yêu cầu nghiêm ngặt về hiệu quả của hoạt động này cả về thời gian sử dụng của CPU và RAM cần thiết. Phiên bản gốc của chương trình, dựa trên các danh sách được liên kết, được chứng minh là không hiệu quả và phức tạp, thậm chí còn chứa một số lỗi. Sau khi xem xét kỹ hơn bản chất của bài toán, cả hai đã cố gắng tạo ra một chương trình ngắn, hiệu quả cả về thời gian và bộ nhớ, sau này được sử dụng trong một số trình soạn thảo văn bản khác.

Có tổng cộng năm thuật toán khác nhau để giải quyết vấn đề, tất cả chúng sẽ được thảo luận bên dưới. Hãy để chúng ta ký hiệu mảng bằng m và số phần tử của nó bằng n. Làm thế nào chúng ta có thể thay đổi? Có hai giải pháp khá đơn giản, mỗi giải pháp đều có nhược điểm của nó: thứ nhất là không hiệu quả về bộ nhớ và thứ hai là về mặt thời gian.

Hãy xem giải pháp đầu tiên. Nếu chúng ta có một lượng RAM bổ sung nhất định, chúng ta có thể thực hiện như sau

Thuật toán 1

1. Sao chép k phần tử đầu tiên của m vào mảng thời gian x nào đó.

2. Chuyển $n - k$ phần tử còn lại sang bên trái k vị trí.

3. Sao chép các phần tử từ x trở lại m (ở k vị trí cuối cùng).

Thuật toán trên rõ ràng là không hiệu quả về dung lượng RAM được sử dụng. Đồng thời, rõ ràng là trong trường hợp đặc biệt tại $k = 1$, chúng ta có thể thực hiện hàm `shiftBy1()`, hàm này sẽ yêu cầu bộ nhớ bổ sung liên tục và thời gian có bậc là $\Theta(n)$. Chúng ta nhận được một thuật toán mới:

Thuật toán 2

1. Gọi k lần `shiftBy1()`.

Thật không may, thuật toán được đề xuất, mặc dù hiệu quả trong bộ nhớ, trong trường hợp xấu nhất đòi hỏi thời gian có bậc là $\Theta(n^2)$.

Chẳng lẽ chúng ta không thể tìm ra một thuật toán vừa tiết kiệm thời gian vừa tiết kiệm bộ nhớ, kết hợp những ưu điểm của các thuật toán trên? Nó chỉ ra rằng một thuật toán như vậy tồn tại. Ý tưởng là chuyển phần tử đầu tiên của `m[]` sang một biến tạm thời `tmp`, sau đó chuyển `m[k + 1]` thành `m[1]`, `m[2k + 1]` thành `m[k + 1]`, v.v. ., lấy tất cả các chỉ số của `m[]` modulo `n`. Quá trình tiếp tục cho đến khi đạt đến `m[1]`, trong đó `tmp` được sử dụng thay thế. Trong trường hợp không phải tất cả các phần tử đều đúng vị trí, chúng ta lặp lại quy trình, bắt đầu bằng `m[2]`, v.v. Một ví dụ về triển khai ý tưởng đã trình bày trông như thế này (Ở đây, các chỉ mục được thay đổi vì lập chỉ mục của mảng bắt đầu từ 0):

Chương trình 7.29. Dịch chuyển tuần hoàn của các phần tử mảng (729shift1.c)

```
#include <stdio.h>
#define MAX 100

struct CElem {
  int data;
  /* có thể thêm... */
} m[MAX];

const unsigned n = 10; /* Số phần tử trong mảng */
const unsigned k = 2; /* Số vị trí di chuyển*/

void init(void)
{ unsigned i;
  for (i = 0; i < n; i++)
    m[i].data = i;
}

unsigned gcd(unsigned x, unsigned y)
{ while (y > 0) {
  unsigned tmp = y;
  y = x % y;
  x = tmp;
  }
  return x;
}

void shiftLeft1(unsigned k)
{ /* di chuyển m[] với k vị trí từ trái, sử dụng một biến phụ trợ*/
  unsigned i, ths, next, gcdNK;
  struct CElem tmp;
  for (gcdNK = gcd(n, k), i = 0; i < gcdNK; i++) {
    ths = i; tmp = m[ths];
    next = ths + k;
    if (next >= n)
      next -= n;
    while (next != i) {
      m[ths] = m[next];
      ths = next;
```

```
        next += k;
        if (next >= n) next -= n;
      }
      m[ths] = tmp;
    }
}

void print(void)
{ unsigned i;
  for (i = 0; i < n; i++)
    printf("%d ", m[i].data);
  printf("\n");
}

int main() {
   init ();
   shiftLeft1(k);
  print();
  return 0;
}
```

Chúng ta có thể nhìn mọi thứ theo một cách khác. Gọi A là k phần tử đầu tiên của m, và B là $n - k$ còn lại. Khi đó, phép di chuyển tuần hoàn các phần tử của dãy m sang trái tại k vị trí có thể coi là một phép biến hình AB thành BA. Không giới hạn cộng đồng lý luận, chúng ta chắc chắn có thể coi rằng $k < n/2$, tức là A ngắn hơn B. Chúng ta chia B thành hai phần B_L và B_R, vì B_R chứa k phần tử. Nhiều nhất là A Sau đó, sự chuyển dịch tuần hoàn giảm đến sự biến đổi AB_LB_R thành B_LB_RA. Nếu chúng ta hoán đổi vị trí của A và B_R (mà k trao đổi là đủ cho chúng ta), các phần tử của A sẽ đến vị trí cuối cùng của chúng. Nó vẫn là để giải quyết vấn đề với B_L và B_R - cả hai phần của B. Đối với B, chúng ta nhận được một bài toán mới, tương tự như ban đầu, nhưng với kích thước nhỏ hơn. Chúng ta có thể tiếp tục quy trình một cách đệ quy, cho đến khi giải pháp cuối cùng của nhiệm vụ: chia và trị. Mặc dù thoạt nhìn có vẻ phức tạp nhưng quy trình được mô tả đủ đơn giản và có thể được mô tả lặp đi lặp lại. Grizzly và Mills cung cấp nhận thức lặp đi lặp lại sau:

Chương trình 7.30. Dịch chuyển tuần hoàn của các phần tử mảng (730shift2.c)

```
void swap(unsigned a, unsigned b, unsigned l)
{  /* Thay chỗ mảng con m[a..a+l-1] và m[b..b+l-1] */
   unsigned i;
   struct CElem tmp;
   for (i = 0; i < l; i++) {
      tmp = m[a + i];
      m[a + i] = m[b + i];
      m[b + i] = tmp;
   }
}

void shiftLeft2(unsigned k)
{  /* Di chuyển mảng m[] bằng k vị trí từ trái.
   * quy trình đệ quy được thực hiện lặp đi lặp lại */
   unsigned i, j, p;
   p = i = k;
   j = n - k;
   while (i != j)
      if (i > j) { swap(p - i, p, j); i -= j; }
      else { swap(p - i, p + j - i, i); j -= i; }
   swap(p - i, p, i);
}
```

Vì vậy, bây giờ chúng ta có hai thuật toán hiệu quả về thời gian và hiệu quả về bộ nhớ. Chúng ta sẽ đề xuất một thuật toán hiệu quả và đặc biệt phức tạp khác, cụ thể là thuật toán được sử dụng bởi Carnigan và Plodger, dựa trên nhận dạng toán học đơn giản sau:

$$BA = (A^R B^R)^R.$$

Với A^R ở trên ta ký hiệu mảng thu được từ các phần tử của A, được lấy theo thứ tự ngược lại. Ý nghĩa của B^R và $(A^R B^R)^R$ là tương tự. Sự đồng nhất ở trên cho chúng ta một thuật toán cực kỳ đơn giản và hiệu quả để giải quyết vấn đề, đó là: để có được BA, chúng ta cần đảo ngược A, đảo ngược B, và sau đó đảo toàn bộ mảng. Sau đây là hiện thực của thuật toán được trình bày:

Chương trình 7.31. Dịch chuyển tuần hoàn của các phần tử mảng (731shift3.c)

```
void reverse(unsigned a, unsigned b) /* Đảo ngược mảng con m[a..b]
    */
{ unsigned i, j, k, cnt;
  struct CElem tmp;
  for (cnt = (b-a)/2, k=a, j=b, i=0; i <= cnt; i++, j--, k++) {
    tmp = m[k];
    m[k] = m[j];
    m[j] = tmp;
  }
}

void shiftLeft3(unsigned k)
{ /* Di chuyển mảng m đến k vị trí sang trái, trong ba bước*/
  reverse(0, k - 1);
  reverse(k, n - 1);
  reverse(0, n - 1);
}
```

Ba thuật toán cuối cùng yêu cầu bộ nhớ bổ sung liên tục và được thực thi trong thời gian $\Theta(n)$. Tuy nhiên, ở các giá trị cao hơn của n, chúng không tương đương về tốc độ, vì hằng số trước n khác nhau. Thật vậy, với `ShiftLeft1()` tương đối phức tạp, mỗi phần tử của mảng được đọc và ghi nhớ chính xác một lần, trong khi với `ShiftLeft3()` các phép toán này dường như được thực hiện hai lần (xem [Bentley-1990], [Nakov-1998d]).

Bài tập

▷ **7.33.** Hãy so sánh về mặt lý thuyết thời gian hoạt động của các hàm `shiftLeft1()`, `shiftLeft2()` và `shiftLeft3()`.

▷ **7.34.** Hãy suy ra công thức $BA = (A^R B^R)^R$.

7.12. Câu hỏi và bài tập

▷ **7.35.** *Sắp xếp nhanh tốt hơn*

Sử dụng thuật toán trung vị trung bình (xem 7.2), Sắp xếp lại sắp xếp nhanh (xem 3.1.6.) Để nó luôn chạy trong thời gian $\Theta(n.\log_2 n)$. So sánh độ phức tạp lý thuyết và thực nghiệm với độ phức tạp của

phân loại hình chóp (xem 3.1.9.).

▷ **7.36.** *k-phần tử nhỏ nhất trong một mảng*

Hãy phát triển một thuật toán để tìm k số nhỏ nhất trong một mảng. So sánh các thuật toán sau:

- sắp xếp và hiển thị k số đầu tiên
- tìm phần tử x lớn nhất thứ k bằng thuật toán tuyến tính (xem 7.1.), Phân chia mảng phân vùng trái và phải đối với x và sắp xếp hoàn chỉnh phân vùng bên trái
- xây dựng một kim tự tháp (hàng đợi ưu tiên) và chiết xuất gấp k của mức tối thiểu phần tử hàng đầu của nó.

▷ **7.37.** *Cặp điểm gần nhất*

Có n điểm trong mặt phẳng được cho bởi tọa độ của chúng. Tìm một cặp điểm mà khoảng cách giữa hai điểm là nhỏ nhất. Thuật toán của bạn có hoạt động trong không gian 3D không? Và trong n-chiều?

▷ **7.38.** *Vỏ lồi ở nhiều điểm*

Có n điểm trong mặt phẳng được cho bởi tọa độ của chúng. Tìm một đa giác lồi có một mặt nhỏ nhất (tập lồi nhỏ nhất) có chứa chúng. Thuật toán của bạn có hoạt động trong không gian 3D không? Và trong n-chiều?

▷ **7.39.** *Các ví dụ khác về chia và chinh phục*

Có đúng là các thuật toán được liệt kê dưới đây dựa trên chia và chinh phục không? Tại sao?

-
- Phân loại Hoor nhanh (xem 3.1.6.)
- phân loại theo hình chóp (xem 3.1.9.), Một phần (tại sao?)
- sắp xếp bit (xem 3.2.3.), Biến thể đệ quy
- tìm kiếm nhị phân (xem 4.3.)
- Tìm kiếm Fibonacci (xem 4.4.)
- tìm kiếm nội suy (xem 4.5.)
- Thuật toán Shannon-Fano (xem 10.4.1.)
- Thuật toán Huffman (xem 10.4.2.)

Những thuật toán nào khác được coi là có thể được thêm vào?