NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 1
SỐ HỌC VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 1
SỐ HỌC VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

LỜI NÓI ĐẦU

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

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

Tôi đã làm các quyển sách:

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

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

Hà Nội, ngày 25 tháng 2 năm 2022

Nguyễn Hữu Điển

NHỮNG KÝ HIỆU

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

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

NỘI DUNG

DANH SÁCH CÁC HÌNH

DANH SÁCH CÁC BẢNG

Danh sách chương trình

CHƯƠNG 1
SỐ HỌC VÀ THUẬT TOÁN

1.1. Khái niệm toán học và thuật toán

Tài liệu trong đoạn này nhằm mục đích giúp người đọc làm quen với các ký hiệu, thuật ngữ, khái niệm và các tính chất cơ bản của một số đối tượng toán học được sử dụng trong cuốn sách. Khuyến nghị của chúng ta, ngay cả đối với những người đã có nhiều kinh nghiệm với các thuật toán máy tính, là hãy đọc kỹ chương giới thiệu này để những điều chưa giải thích được từ nó càng ít càng tốt. Đây là điều kiện cần để có thể hiểu đầy đủ hơn về tài liệu. Tất nhiên, không nhất thiết phải nhớ tất cả các thuật ngữ và định nghĩa. Chỉ cho thấy một chút "hiểu biết nhỏ" về ngôn ngữ chặt chẽ và trừu tượng của toán học.

1.1.1. Tập hợp

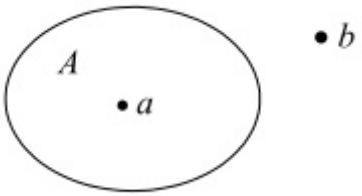
Khái niệm tập hợp là chính và không được định nghĩa chặt chẽ bởi các khái niệm toán học khác. Nó được nhận thức một cách trực quan và thường được làm rõ qua các ví dụ.

Giả thiết rằng một tập hợp được đưa ra khi một hệ thống (tập hợp) các đối tượng được đưa ra, hầu hết thường được thống nhất trên cơ sở một số đặc điểm, thuộc tính chung hoặc một số quy tắc nào đó. Ví dụ, những con ong tạo thành một tập hợp trong một tổ ong, các đỉnh của một đa giác, các đường thẳng đi qua một điểm cho trước, nghiệm nguyên của phương trình $ax^2 + bx + c = 0$, v.v.

Các chữ cái viết hoa của bảng chữ cái Latinh thường được sử dụng làm ký hiệu cho một tập hợp: $A, B, C, ...$ Các đối tượng tham gia vào một tập hợp được gọi là các *phần tử của tập hợp* và thường được đánh dấu bằng các chữ cái Latinh viết thường (hoặc bằng cách đánh chỉ mục thích hợp). Chúng ta sẽ tránh định nghĩa chặt chẽ về tập chỉ mục là gì, nhưng thường đối với các phần tử của tập A, chúng ta sẽ sử dụng chuỗi được lập chỉ mục $a_1, a_2, ..., a_n$ và chúng ta sẽ biểu thị nó theo cách này: $A = \{a_1, a_2, ..., a_n\}$

Thực tế là phần tử $a_i, i = 1, 2, ..., n$, thuộc tập A được ký hiệu là $a_i \in A$ và phần tử không thuộc - bởi $a_i \notin A$. Số phần tử của một tập hợp được gọi là lũy thừa của tập hợp (đối với tập A ở trên số này bằng n), và đôi khi chúng ta sẽ sử dụng $|A|$ để biểu thị lũy thừa. Với $n = 0$, tập được gọi là *rỗng* và được ký hiệu là $\emptyset$. Thông thường,

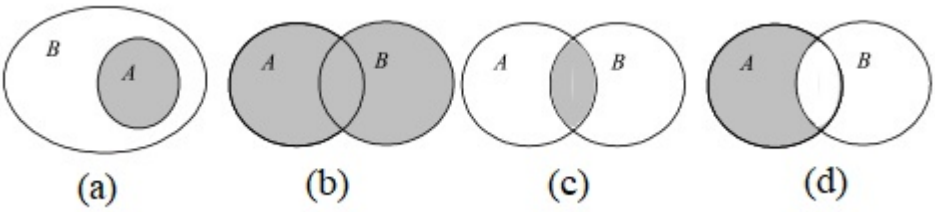
các tập hợp được biểu diễn bằng biểu đồ Venn (xem Hình 1.1.): Mỗi tập hợp A được biểu diễn dưới dạng một vùng đóng và các phần tử $(a, b, ...)$ - là các điểm nằm trong, hoặc bên ngoài nó, tùy thuộc vào việc chúng có thuộc bộ hay không.



Hình 1.1. Sơ đồ Cen

Có nhiều khái niệm, tính chất và phép toán thống nhất trong một lý thuyết tập hợp được xây dựng tốt [Dilova, Stoyanov-1973]. Chúng ta sẽ đề cập ngắn gọn thêm một số trong số đó, sẽ cần trong tài liệu sau.

Định nghĩa 1.1. Nếu tất cả các phần tử của một tập hợp A đã cho là phần tử của một tập hợp B khác, thì A được gọi là một *tập hợp con* của B. Điều này được ký hiệu là $A \subseteq B$ (xem Hình 1.2.). Khi biết thêm rằng B không trùng với A thì tập A được gọi là *tập con* thích hợp (thực) của B. Trong trường hợp này chúng ta sẽ sử dụng kí hiệu $A \subset B$.



Hình 1.2. Tập hợp con (a), tập hợp hợp (b), tập giao (c) và tập hiệu (d) của các tập hợp.

Định nghĩa 1.2. Tập hợp C được gọi là *hợp* của tập A và B nếu nó bao gồm tất cả các phần tử a sao cho $a \in A$ hoặc $a \in B$. Viết $C = A \cup B$.

Định nghĩa 1.3. *Giao* $C = A \cap B$ của hai tập hợp A và B được gọi là tập hợp C, gồm tất cả các phần tử đồng thời thuộc A và B.

Định nghĩa 1.4. *Hiệu* $C = A \setminus B$ của tập A và B được gọi là tập C, gồm tất cả các phần tử thuộc A nhưng không thuộc B.

Định nghĩa 1.5. Một tập hợp được gọi là *hữu hạn* nếu nó chứa một số hữu hạn phần tử. Nếu không nó được gọi là *vô hạn*.

Trước khi chúng ta tiếp tục, đây là một vài định nghĩa quan trọng hơn. Khi xem xét một tập hợp, không quan trọng số lần một phần tử xuất hiện, cũng như thứ tự của các phần tử của nó. Do đó các tập hợp sau là tương đương (trùng nhau):

$$\{a, a, b\} = \{a, b, b\} = \{a, b, a\} = \{a, b\} = \{b, a\}$$

Định nghĩa 1.6. Một tập hợp trong đó sự lặp lại của các phần tử được cho phép được gọi là *đa tập hợp* .

Định nghĩa 1.7. Nếu một thứ tự của các phần tử được nhập vào một tập hợp n phần tử, thì đối tượng kết quả được gọi là một n-bộ có thứ tự (danh sách).

Chúng ta sẽ sử dụng dấu ngoặc tròn thay vì dấu ngoặc nhọn để biểu thị n-bộ có thứ tự. Ví dụ, ba thứ tự (a, b, c) khác với ba thứ tự (a, c, b), v.v.

Bài tập

▷ **1.1.** Các tập hợp $A = \{1, 2, 4, 5, 7\}$ và $B = \{2, 3, 4, 5, 6\}$ được đưa ra. Xác định các tập hợp: $A \cup B, A \cap B, A \setminus B, B \setminus A$.

▷ **1.2.** Hai tập hợp A và B đã cho và biết rằng $A \cap B = A$. Bạn có thể nói gì về tập hợp B?

▷ **1.3.** Toán tử hiệu đối xứng $\oplus$ của A và B được xác định như sau: $A \oplus B = (A \cup B) \setminus (A \cap B)$. Xác định hiệu đối xứng của các tập $A = \{1, 2, 4, 5, 7\}$ và $B = \{2, 3, 4, 5, 6\}$.

▷ **1.4.** Một phép toán $\bullet$ được gọi là đối xứng nếu $A \bullet B = B \bullet A$. Phép toán nào sau đây là đối xứng: $A \cup B, A \cap B, A \setminus B, B \setminus A, A \oplus B$?

1.1.2. Tập hợp số

Các con số là cơ sở cho tất cả các loại tính toán toán học, cũng như các thuật toán là cơ sở của tin học máy tính. Theo trình tự thời gian (ban đầu chỉ có 3 "số" được dùng để đếm: một, hai và nhiều), chúng được coi là hình thức đầu tiên của tư duy trừu tượng và nhiều người coi chúng như một thứ gì đó kỳ diệu và đặc biệt.

Các con số có liên quan đến hầu hết mọi thuật toán và chương trình. Đó là lý do tại sao các vấn đề liên quan đến việc trình bày, lưu trữ và sử dụng máy tính của họ là vô cùng quan trọng.

Định nghĩa 1.8. Tập hợp các *số tự nhiên* (chúng ta sẽ ký hiệu là $\mathbb{N}$) chứa các số mà chúng ta đếm được: $0, 1, 2, 3, ...$

Có tranh cãi về việc số 0 có nên được coi là một số tự nhiên hay không. Nó trừu tượng hơn các số tự nhiên khác và bị thiếu trong nhiều hệ thống số cũ: ví dụ, trong hệ thống số La Mã, việc biểu diễn các số bắt đầu bằng một. Mặt khác, trong toán học rời rạc, việc giới thiệu $\mathbb{N}$ thường được thực hiện từ đầu. Điều này là do khi làm việc với các tập hợp n phần tử, thường phải đưa tập hợp rỗng vào định nghĩa.

Định nghĩa 1.9. Định nghĩa 1.9. Tập hợp các *số nguyên* $\mathbb{Z}$ bao gồm: $..., -3, -2, -1$ (số nguyên âm), 0 (số không), $1, 2, 3, ...$ (số nguyên dương).

Trong bộ nhớ máy tính, chúng được biểu diễn dưới dạng một chuỗi các bit - mã chuyển tiếp, mã ngược hoặc mã bổ sung [Shishkov-1995]. Ngày nay, mã bổ sung thường được sử dụng nhiều nhất, vì nó dễ dàng thu gọn vào phép cộng, thuận tiện cho việc thực hiện trên máy tính.

Các trình biên dịch khác nhau của các loại C và phạm vi định nghĩa của chúng có thể khác nhau. ANSIC (American National Standards Institute [ANSIC]) không xác định các phạm vi cụ thể (Bảng 1.1 làm ví dụ cho thấy các phạm vi cho Borland C cho DOS), mà chỉ xác định các mối quan hệ, ví dụ:

$$|\texttt{short}| \leq |\texttt{int}| \leq |\texttt{long}|$$

Theo mặc định, kích thước của `int` là một từ máy (điều này giải thích tại sao loại DOS là 2 byte và loại Windows là 4).

Một cách chuẩn để nhận giá trị lớn nhất của một loại, chẳng hạn như `unsigned`, là dựa trên biểu diễn bên trong của các số trong mã bổ sung: `(unsigned)(-1)`

Loại	**Giá trị**	**Kích thước**
char	$-128, \ldots, 127$	8 bit có dấu
unsigned char	$0, \ldots, 255$	8 bit không dấu
short int	$-32768, \ldots, 32767$	16 bit có dấu
int	$-32768, \ldots, 32767$	16 bit có dấu
long int	$-2147483648, \ldots, 2147483647$	32 bit có dấu
unsigned short int	$0, \ldots, 65535$	16 bit không dấu
unsigned long int	$0, \ldots, 4294967295$	32 bit không dấu

Bảng 1.1. Các kiểu số nguyên trong Borland C dành cho DOS.

Ở cột giữa của Bảng 1.1. phạm vi giá trị có thể chấp nhận một biến của kiểu tương ứng được đưa ra. Mỗi kiểu được đặc trưng bởi một số nguyên tối đa và tối thiểu mà nó có thể lưu trữ. Bất kỳ phép gán nào vượt quá giá trị tối đa (hoặc tối thiểu) của kiểu được gọi là tràn và có thể dẫn đến kết quả thảm hại cho từng thuật toán và chương trình.

Định nghĩa 1.10. *Số hữu tỉ* là những số thuộc loại $\frac{p}{q}$, trong đó p và q là các số nguyên và q là số dương. Tập hợp các số hữu tỉ được kí hiệu là Q.

Định nghĩa 1.11. *Số thực* là những số có thể viết dưới dạng:

$$x = n + 0, d_1 d_2 d_3 \ldots,$$

với n là một số nguyên và d_i là các chữ số thập phân từ 0 đến 9.

Các số $0, d_1 d_2 d_3 \ldots$ được gọi là số thập phân. Trình bày ở dạng trên có nghĩa là bất đẳng thức

$$n + \frac{d_1}{10} + \frac{d_2}{100} + \cdots + \frac{d_k}{10^k} \leq x \leq n + \frac{d_1}{10} + \frac{d_2}{100} + \cdots + \frac{d_k}{10^k} + \frac{1}{10^k}$$

với mọi $k \in \mathbb{N}, k \geq 0$.

Lưu ý rằng dãy chữ số d_i có thể là vô hạn. Điều này có thể xảy ra đối với cả số hữu tỉ và số vô tỉ (tức là số không hữu tỉ). Ví dụ, số $1/3 = 0,333333.....$ Ở đây số 3 được lặp lại vô hạn và được gọi là chu kỳ của phân số. Viết thêm $1/3 = 0,(3)$ và đọc "không nguyên và ba trong khoảng thời gian". Dấu chấm có thể dài hơn một chữ số, ví dụ $1/7 = 0,(142857)$.

Ví dụ về một số thực vô tỷ và không tuần hoàn, tức là không được biểu diễn chính xác dưới dạng $p/q (p, q \in \mathbb{N}, q > 0)$, là tỷ số giữa chu vi hình tròn với đường kính của nó - hằng số π:

$$\pi = 3,141592653\ldots.$$

Giống như bất kỳ số thực nào, π có thể được tính gần đúng bằng một số hữu tỉ - ví dụ 355/113, sẽ xác định nó với độ chính xác sau dấu thập phân, nhưng sau một số chữ số nhất định (đối với ví dụ đã chọn, nó là 6 chữ số sau dấu dấu thập phân) độ chính xác này bị mất. Một giá trị gần đúng hữu ích khác là 22/7.

Trong bộ nhớ máy tính, số thực về mặt lý thuyết có thể được biểu diễn theo ba cách: cố định, tự nhiên và dấu phẩy động [Shishkov-1995]. Trong thực tế, số dấu phẩy động được sử dụng phổ biến nhất theo tiêu chuẩn IEEE ((Institute of Electrical & Electronics Engineers). Các nhận xét tương tự cũng áp dụng cho các kiểu thực cũng như kiểu số nguyên. Trong Bảng 1.2. phạm vi của các loại thực tế trong Borland C cho DOS được hiển thị

Loại	**Giá trị**	**Kích thước**
float	3,4.10 -38 , ..., 3,4.10 38	32 bit
double	1,7.10 -308 , ..., 1,7.10 308	64 bit
long double	3,4.10 -4932 , ..., 1,1.10 4932	80 bit

Bảng 1.2. Các kiểu số th trong Borland C.

Khi làm việc với số thực, phải cẩn thận để tránh hiện tượng tràn và mất độ chính xác sau dấu thập phân (underflow): Khi biểu diễn với một số bit cố định, độ chính xác bị mất khi làm tròn (kết quả là chúng ta chỉ có thể sử dụng một số lượng bit giới hạn). Khi biểu

diễn một số thực dưới dạng phân số hữu tỉ, độ chính xác bị mất đi so với tính gần đúng trong biểu diễn: 1/3 là phân số thập phân vô hạn tuần hoàn và không thể biểu diễn chính xác bằng một số thực hữu hạn.

Do đó, đối với mỗi kiểu dữ liệu biểu diễn một số thực, tồn tại số ϵ nhỏ nhất ($\epsilon > 0$) sao cho bất kỳ số nào (hoặc kết quả của một phép tính số học) nhỏ hơn ϵ về giá trị tuyệt đối được làm tròn thành 0.

Bài tập

▷ **1.5.** Ch minh rằng rằng các số thực có thể được biểu diễn dưới dạng số dấu phẩy động thực tiêu chuẩn trên thực tế là một tập con hữu hạn của các số hữu tỉ.

1.1.3. Phép chia hết và chia có dư

Gọi m và n là các số nguyên, $m \neq 0$. Khi đó tồn tại các số nguyên q và r ($0 \leq r < m$) sao cho $n = q.m + r$. Số q được gọi là *thương* của phép chia số nguyên n/m, và r được gọi là *phần dư*. Nếu phần dư r của phép chia số nguyên là 0, ta nói rằng m chia cho n (n là bội của m) và viết $m|n$. Trong ngôn ngữ C, phép chia với phần nguyên và phần dư (khi làm việc với số nguyên) được thực hiện với sự trợ giúp của các phép toán / và %. Để tránh "nhầm lẫn", chúng ta sẽ sử dụng hai ký hiệu này trong các văn bản toán học:

$$q = n/m;$$
$$r = n\%m;$$

Định nghĩa 1.12. Khi $(n - m)\%z = 0$, ta nói rằng n có thể *so sánh* với m *modulo* z và viết $n \equiv m \pmod{z}$.

Phép chia cho thương và dư có thể dùng để tìm số chữ số của số tự nhiên n. Thuật toán như sau: Chia liên tiếp (nếu có thể) một số nguyên n cho 10. Rõ ràng, với mỗi phép chia như vậy, các chữ số của n giảm đi một. Như vậy, số chữ số của n được xác định bằng số phép chia ta thực hiện cho đến khi n bằng 0

Chương trình 1.1. Số chữ số (101digits.c)

```
#include <stdio.h>
```

```
unsigned m, n = 424267;

int main(void) {
  unsigned digits;
  m = n;
  for (digits = 0; n > 0; n /= 10, digits++);
  printf("So nhung chu so cua %u la %u\n", m, digits);
  getchar();
  return 0;
}
```

Bài tập

▷ **1.6.** Tìm thương và dư của phép chia m cho n nếu (m,n) là: $(7,3),(-7,3),(7,-3),(-7,-3),3,7),(-3,7),(3,-7),(-3,-7)$.

▷ **1.7.** Đối với các mục tiêu đã cho, m và n ($m \neq 0$) cố gắng chứng minh sự tồn tại và tính duy nhất của biểu diễn $n = q.m + r, 0 \leq r < m$, ($q, r$ số nguyên). [Siderov-1995]

▷ **1.8.** Đề xuất giải thuật tìm số chữ số của một số thực. Những vấn đề gì phát sinh?

1.1.4. Tính tổng và tích

Các số $a_1, a_2, \ldots, a_n$ được cho trước. Một trong ba ký hiệu sau đây thường được sử dụng nhất để biểu thị tổng của chúng $S = a_1 + a_2 + \cdots + a_n$:

$$S = \sum_{i=1}^{n} a_i, S = \sum_{1 \leq i \leq n} a_i, \text{ hoặc } c$$

Có thể cho một hàm $R(x)$ tạo ra các giá trị của i - khi đó tổng được viết dưới dạng:

$$S = \sum_{i:R(x)} a_i,$$

Hàm rút gọn sau đây của trong ngôn ngữ C tìm tổng S_n của n số tự nhiên đầu tiên:

Chương trình 1.2. Tính tổng cách 1 (102sum1.c)

```
#include <stdio.h>
unsigned sum(unsigned n)
{ unsigned i, s = 0;
  for (i = 1; i <= n; i++) s += i;
  return s;
}
int main(void) {
    unsigned s;
    s=sum(1000);
    printf("Tong la %u\n",s);
    getchar();
    return 0;
}
```

Tương tự ta có thể tính tổng mà các phần tử nằm trong mảng n thành phần.

Chương trình 1.3. Tính tổng cách 2 (103sum1.c)

```
#include <stdio.h>
int sum(unsigned n)
{ unsigned i;
int a[n];
for (i = 0; i < n; i++) a[i]= i;
int s = 0;
for (i = 0; i < n; i++) s += a[i];
return s;
}

int main(void) {
    unsigned s;
    s=sum(100);
    printf("Tong la %u\n",s);
    getchar();
    return 0;
}
```

Trong ví dụ đầu tiên, chu kỳ là dư thừa - tổng có thể được tìm

trực tiếp bằng công thức cấp số cộng:

$$S_n = \frac{n(n+1)}{2}$$

Trong chương trình thứ hai - một công thức trực tiếp như vậy không thể tồn tại, bởi vì các phần tử của mảng có thể là các số nguyên tùy ý.

Trường hợp tổng lồng nhau, ta sử dụng chu kỳ lồng nhau

$$\begin{aligned}\sum_{j=1}^{n}\sum_{i=1}^{m} a_j b_i &= a_1b_1 + a_1b_2 + \cdots + a_1b_n + \cdots + a_nb_1 + \cdots + a_nb_n \\ &= a_1(b_1 + \cdots + b_m) + a_2(b_1 + \cdots + b_m) + \cdots + a_n(b_1 + \cdots + b_m) \\ &= (a_1 + \cdots + a_n)(b_1 + \cdots + b_m) \\ &= \sum_{j=1}^{n} a_j \sum_{i=1}^{m} b_i. \end{aligned} \tag{1.1}$$

Các đầu vào như vậy có thể được tính toán bằng cách sử dụng các chu kỳ đầu vào:

Chu kỳ lồng nhau

```
unsigned i, j;
int result = 0;
for (j = 1; j <= n; j++)
   for (i = 1; i <= m; i++)
      result += a[i] * b[j];
```

Hai thuộc tính thú vị khác hợp lệ với số tổng là:

$$\sum_{i:R(x)}\sum_{j:S(x)} a_{ij} = \sum_{j:S(x)}\sum_{i:R(x)} a_{ij} \tag{1.2}$$

$$\sum_{R(x)} a_i + \sum_{S(x)} a_i = \sum_{S(x)||R(x)} a_i + \sum_{S(x)\&\&R(x)} a_i. \tag{1.3}$$

Ở đây với $S(x)||R(x)$ và $S(x)\&\&R(x)$, chúng ta đã biểu thị tương ứng, liên hợp và giao điểm của các giá trị được tạo bởi các hàm S và R, có thể chấp nhận tham số i và j.

Trong tích các số $P = a_1.a_2.a_3.....a_n$ ký hiệu toán học được sử dụng là:

$$P = \prod_{i=1}^{n} a_i, P = \prod_{1 \le i \le n} a_i, \text{ hoặc } P = \prod_{i=1..n} a_i$$

Một hàm của ngôn ngữ C để tìm tích các phần tử của mảng $a[]$, với n phần tử, có thể có dạng sau:

Tính tích của một mảng số trong C.

Chương trình 1.4. Tính tích (104mult.c)

```
#include <stdio.h>
int mult(unsigned n)
{ unsigned i;
  int a[n];
  for (i = 0; i < n; i++) a[i]= i+1;
  int s = 1;
  for (i = 0; i < n; i++) s = s*a[i];
  return s;
}
int main( ) {
  unsigned s;
    s = mult(13);
    printf("Tich la %u\n",s);
    getchar();
    return 0;
}
```

Bài tập

▷ **1.9.** Tìm ra công thức tổng của một cấp số cộng.

▷ **1.10.** Chứng minh các tính chất (1.1), (1.2) và (1.3) của tổng.

▷ **1.11.** Công thức và chứng minh cho một tính chất sản phẩm tương tự như (1.1), (1.2) và (1.3).

▷ **1.12.** Sử dụng các thuộc tính (1.1), (1.2) và (1.3), kiểm tra tính hợp lệ của đẳng thức:

$$\sum_{i=1..n} a_i + \sum_{i=n..m} a_i = \sum_{i=1..m} a_i + a_n, \quad 1 \le n \le m.$$

1.1.5. Tính lũy thừa, logarit và căn

Định nghĩa 1.13. Nếu x là số thực và y là số tự nhiên thì tung độ được xác định như sau:

$$x^y = x.x.\ldots.x(y \text{ lần })$$

Khi $y < 0$ thì $x^y = 1/x^{-y}$.

Các thuộc tính sau là hợp lệ ($x \neq 0$):

$$x^y = x^{y-1}.xx^y \quad = x^{y+1}/xx^{y_1+y_2} = x^{y_1}.x^{y_2}x^{y_1.y_2} \quad = (x^{y_1})^{y_2}$$

Việc thực hiện đơn giản để tính x^y là bằng cách thực hiện các phép nhân liên tiếp y:

Chương trình 1.5. Tính lũy thừa (105power.c)

```
#include <stdio.h>
float power(float x, unsigned y)
{ float res = x;
  unsigned i;
  for (i = 1; i < y; i++) res *= x;
  return res;
}
int main(void) {
    float s;
  s=power(3,7);
    printf("Luy thua la %f\n",s);
    getchar();
    return 0;
}
```

Sau đó (xem 7.5.) Chúng ta sẽ xem xét một số cách khác, thú vị hơn và hiệu quả hơn để tìm x^y.

Nếu $x^n = y$ (n là số tự nhiên, $n > 1$) thì x được gọi là căn bậc n của y và viết $x = \sqrt[n]{y}$. Phép toán trong đó gốc thứ y của y nhận được từ một y cho trước được gọi là *phptnhcn*. Đối với $n = 2$, thay vì căn bậc hai của y, hãy nói *căn bậc hai* hoặc chỉ *căn y* và viết $\sqrt{y}$. Sử dụng thao tác lấy căn, việc tăng một số có thể được xác định một cách hợp lý:

$$x^{\frac{p}{q}} = \sqrt[q]{x^p}.$$

Trường hợp thú vị nhất là khi x và y là các số thực ($x > 1$). Gọi y được biểu diễn dưới dạng $y = d, d_1 d_2 d_3 ...$ và xét ràng buộc sau:

$$x^{d+\frac{d_1}{10}+\frac{d_2}{100}+\cdots+\frac{d_k}{10^k}} \leq x^y \leq x^{d+\frac{d_1}{10}+\frac{d_2}{100}+\cdots+\frac{d_k}{10^k}} + \frac{1}{10^k}.$$

Rõ ràng, nó định nghĩa x^y là một số thực duy nhất - chúng ta có thể nhận được bất kỳ số chữ số nào của nó sau dấu thập phân, miễn là chúng ta đã chọn một số k đủ lớn.

Chúng ta sẽ xem xét một chức năng quan trọng khác sẽ cần thiết trong tài liệu sau này, đặc biệt là trong phân tích độ phức tạp của một thuật toán. Phương trình $y = b^x$ với $b \neq 1, b > 0, y > 0$ có nghiệm duy nhất là x. Nghiệm này được gọi là *logarit của y tại cơ số b* và được ký hiệu là $\log_b y$. Dưới đây là một số tính chất của hàm logarit ($x > 0, y > 0, b > 0, b \neq 1, c > 0, c \neq 1$):

$$x = b^{\log_b x} = \log_b b^s \tag{1.4}$$

$$\log_b(xy) = \log_b x + \log_b y \tag{1.5}$$

$$\log_b x^y = y \log_b x \tag{1.6}$$

$$\log_b x = \frac{\log_c x}{\log_c b} \tag{1.7}$$

Ở phần sau của cuốn sách, chúng ta thường sẽ bỏ qua cơ số của logarit khi nó là 2 và chúng ta sẽ chỉ đơn giản là viết $\log x$ thay vì $\log_2 x$. Chúng ta cũng sẽ sử dụng ký hiệu $\ln x$ để biểu thị một logarit tự nhiên là $\log_e x$: dựa trên số Hepper $e = 2.71828...$ (xem 1.1.6.).

1.1.6. Bài tập

▷ **1.13.** Để chứng minh các thuộc tính nêu trên của mức độ, bắt đầu từ định nghĩa.

▷ **1.14.** Chứng minh các tính chất (1.4), (1.5) và (1.6) của lôgarit, bắt đầu từ định nghĩa.

1.1.7. Giai thừa và hồi qui

Thừa số của $n, n \in \mathbb{N}$ (viết $n!$) Là tích của các số tự nhiên từ 1 đến n:

$$n! = 1.2.\ldots.n = \prod_{i=1}^{n},$$

theo định nghĩa $0! = 1$.

Trong ngôn ngữ C để tính $n!$ không đặc biệt khó khăn:

Chương trình 1.6. Giai thừa (106factorial.c)

```
#include <stdio.h>
const unsigned n = 10;
unsigned long factoriel(unsigned n)
{ unsigned i;
  unsigned long r = 1;
  for (i  = 2;  i  <= n; i++) r  *= i;
  return r;
}
int main() {
  printf("%u! = %lu\n", n, factoriel(n));
  return 0;
}
```

Đối với một số hàm toán học và dãy số, định nghĩa lặp lại tương ứng thuận tiện và rõ ràng hơn nhiều. Điều này có nghĩa là xác định hàm bằng chính nó hoặc tính toán từng phần tử kế tiếp của chuỗi các giá trị của hàm, sử dụng các giá trị của hàm trước đó. Trong trường hợp của chúng ta, định nghĩa lặp lại của một giai thừa trông giống như sau:

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0 \end{cases}$$

Tương tự với giai thừa cho tổng của n số tự nhiên S_n đầu tiên, chúng ta thu được định nghĩa truy hồi:

$$S_n = \begin{cases} 0, & n = 0 \\ S_{n-1} + n, & n > 0 \end{cases}$$

Chúng ta sẽ thấy ở phần sau (xem 1.2.) Điều đó để tính toán cho mỗi hàm toán học lặp lại, một *hàm đệ quy* tương ứng của C có thể được viết.

Bài tập

▷ **1.15.** Đề xuất công thức truy hồi để nâng x thành lũy thừa $y (x \in \mathbb{R}, y \in \mathbb{N})$.

▷ **1.16.** Đề xuất công thức tìm ước số chung lớn nhất của hai số tự nhiên.

1.1.8. Ma trận

Ma trận là một bảng số hình chữ nhật (nói chung là một bảng hình chữ nhật gồm các đối tượng ngẫu nhiên).

Các số m và n xác định *số chiều* ($m \times n$) của ma trận: tức là chứng tỏ rằng nó bao gồm m hàng và n cột ($n \geq 1, m \geq 1$). Khi $m = n$ ma trận được gọi là *hình vuông*. Mỗi phần tử a_{ij} của ma trận được đặc trưng bởi một chỉ số kép: số đầu tiên trong đó xác định số hàng và số thứ hai - số bậc thang nơi nó nằm (xem Hình 1.3.).

$$A_{m\times n} = \begin{array}{|c|c|c|c|} \hline a_{11} & a_{12} & \ldots & a_{1n} \\ \hline a_{21} & a_{22} & \ldots & a_{2n} \\ \hline \multicolumn{4}{|c|}{\ldots} \\ \hline a_{m1} & a_{m2} & \ldots & a_{mn} \\ \hline \end{array}$$

Hình 1.3. Ma trận $m \times n$

Ma trận được sử dụng rộng rãi trong đại số, trong giải hệ phương trình, trong hoạt hình máy tính, trong đó ma trận có kích thước $2 \times 2, 3 \times 3, 4 \times 4$ được sử dụng để dịch và quay các đối tượng, để đạt được các hiệu ứng đồ họa khác nhau (chẳng hạn như phối cảnh xem trong địa hình ba chiều), v.v. Việc sử dụng ma trận trong các trường hợp này và các trường hợp khác dẫn đến các thuật toán hiệu quả hơn đáng kể và do đó, năng suất cao hơn [Ayres-1962].

Trong ngôn ngữ C một bảng chữ nhật ta có thể biểu diễn như một mảng

```
int A[m][n];
```

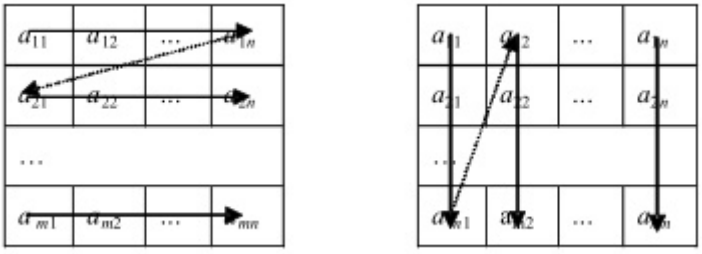
Trường hợp tổng quát hơn cũng có thể xảy ra khi các phần tử của ma trận là các đối tượng tùy ý của một số kiểu `struct` data được xác định trước, ví dụ:

Loại dữ liệu ma trận

```
struct data {
int a;
int b;
...
} A[m][n];
```

Nhập vào và in ra một ma trận

Phần tử ma trận có thể được đọc/in theo nhiều cách khác nhau. Hai đơn giản nhất là ma trận *hàng* hoặc *cột* (Hình 1.1.1d).



Hình 1.4. Thu thập thông tin các phần tử của ma trận: (a) theo hàng và (b) theo cột.

Nhập dữ liệu ma trận

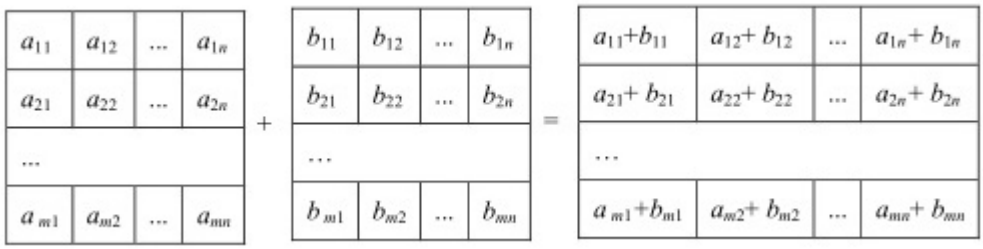
```
//Nhập theo hàng
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++) scanf("%d", &A[i][j]);

//Nhập theo cột
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++) scanf("%d", &A[j][i]);

// In theo hàng
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) printf("%.3d", A[i][j]);
  printf("\n");
```

Tổng hai ma trận

Tổng của hai ma trận $A_{m\times n}$ và $B_{m\times n}$ là ma trận thứ ba $C_{m\times n}$ sao cho $c_{ij} = a_{ij} + b_{ij}$ (với $i = 1, 2, ..., m, j = 1, 2, ..., .n$), xem Hình 1.5.



Hình 1.5. Tổng các ma trận.

Tổng hai ma tận

```
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
     C[i][j] = A[i][j] + B[i][j];
```

Tích hai ma trận

Tích của hai ma trận $A_{m\times n}$ và $B_{n\times p}$ là ma trận thứ ba $C_{m\times p}$ mà:

$$c_{ij} = \sum_{k=1}^{n}(a_{ik}b_{kj}), \text{ với mỗi } i = 1, 2, ..., m \text{ và } j = 1, 2, ..., p.$$

Thực hiện bên dưới, áp dụng trực tiếp công thức từ định nghĩa và thực hiện phép nhân đơn giản $m.p.n$. Nếu $n > m$ và $n > p$ thì số phép nhân nguyên tố bị giới hạn bởi n^3.

Tích hai ma trận

```
for (i = 0; i < m; i++)
  for (j = 0; j < p; j++) {
    C[i][j] = 0;
    for (k=0;k<n;k++)
          C[i][j] += A[i][k]*B[k][j];
  }
```

Tích của ma trận có nhiều ứng dụng - trong việc giải các hệ phương trình, trong thống kê, trong lý thuyết đồ thị và các ứng dụng khác. Có một phương pháp để nhân ma trận số, thực hiện phép nhân đơn giản $n^{\log 7}$ ($\approx n^{2,81}$), đồng thời thực hiện các phép cộng bổ sung. Phương pháp được gọi là *phương pháp Strasen* để nhân

nhanh các ma trận sẽ được xem xét trong 7.6., Như một minh chứng của kỹ thuật *chia và trị*.

Bài tập

▷ **1.17.** Một ma trận được cho trước `unsigned a[MAX][MAX]`. Viết một hàm

`void fillMatrix (unsigned a[][MAX], unsigned n),`

điền vào các phần tử của `a[][]` như sau:

0	20	19	17	14
1	0	18	16	13
2	5	0	15	12
3	6	8	0	11
4	7	9	10	0

▷ **1.18.** Một ma trận không dấu được đưa ra `a[MAX][MAX]`. Viết một hàm hiển thị phần tử xoắn ốc của nó, ví dụ với $n = 5$, chúng ta có:

1	16	15	14	13
2	17	24	23	12
3	18	25	22	11
4	19	20	21	10
5	6	7	8	9

1.1.9. Tìm số chữ số của một tích

Bài toán: Cho trước các số nguyên $a_1, a_2, ..., a_n$. Tìm số chữ số của tích $P = a_1.a_2.....a_n$.

Đặc biệt, nếu $a_i = i$, với $i = 1, 2, ..., n$, thì hãy tìm số chữ số của $P = 1, 2...n = n!$.

Một giải pháp rõ ràng là thực hiện phép nhân và sử dụng thuật toán 1.1.1, tìm số chữ số của kết quả. Đối với ví dụ đã chọn, điều này có nghĩa là tính toán $n!$. Hàm $n!$ tuy nhiên, nó đang phát triển quá nhanh, trong khi số chữ số đang tăng chậm hơn nhiều. 10! là 3628.800, và con số của nó chỉ là bảy. Ở tuổi 20! kết quả vượt quá giá trị lớn nhất cho một kiểu số nguyên trong C.

Chúng ta sẽ áp dụng một thuật toán hiệu quả hơn dựa trên mối quan hệ giữa số chúng ta nhân và số chữ số trong tích kết quả:

Số chữ số của P bằng $\left[1+\sum_{i=1}^{n}\log_{10}a_i\right]$, trong đó $[x]$ biểu thị số nguyên lớn nhất nhỏ hơn hoặc bằng x.

Để giải thích tính hợp lệ của công thức trên, chúng ta sẽ chú ý đến thực tế là số chữ số của mỗi số nguyên P bằng $[1+\log_{10}(P)]$, cũng như tính chất của hàm logarit:

$$\log P = \log(a_1.a_2....a_n) = \log a_1 + \log a_2 + ... + \log a_n.$$

Sau đây là thực hiện trên ngôn ngữ C để tìm các chữ số của $n!$

Chương trình 1.7. Số chữ số của tích (110digitsnf.c)

```
#include <stdio.h>
#include <math.h>
const unsigned long n = 123;

int main(void)
{ float digits = 0;
  unsigned i;
  for (i = 1; i <= n; i++) {
     digits += log10(i);
  }
//Số [x] đưa ra số nguyên dài
  printf("So chu so cua %lu! la %lu\n", n,
                    (unsigned long) digits + 1);
  getchar();
  return 0;
}
```

Bài tập

▷ **1.19.** Chứng minh rằng với mọi số tự nhiên P có số chữ số thập phân của nó bằng $[1+\log_{10}(P)]$.

1.2. Số nguyên tố

Số nguyên tố đã được xem xét trong toán học từ thời cổ đại và gắn liền với nhiều vấn đề thú vị vượt xa biên giới của nó. Trong khoa học máy tính, chúng được sử dụng trong mã hóa, lưu trữ, v.v.

Định nghĩa 1.14. Một số tự nhiên được gọi là đơn giản nếu không có ước nào khác ngoài 1 và chính nó, và số 1 không được coi là số nguyên tố. Nếu nó không đơn giản, nó được gọi là hợp chất.

Dãy số nguyên tố bắt đầu như sau:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, ...$$

Người ta đã chứng minh (Euclid - 300 TCN) rằng các số nguyên tố là vô số.

Như đã đề cập, một trong những ứng dụng của số nguyên tố là mã hóa dữ liệu được truyền qua mạng "không an toàn" (ví dụ: các giao dịch tài chính trên Internet). Một số thuật toán để mã hóa thông tin được truyền (ví dụ RSA [RSA-78]) sử dụng tích các số nguyên tố lớn. Để "phá vỡ" kênh truyền tải thông tin như vậy, bản thân các số nguyên tố phải được biết đến chứ không chỉ là tích của chúng. Nếu nhìn vào số 55, chúng ta có thể đoán ngay nó phân hủy như thế nào - là tích của 5 và 11. Đối với số 4853, chúng ta sẽ khó nhớ lại các ước số đơn giản của nó (211 và 23), nhưng chúng ta có thể biên dịch một chương trình. mà tìm thấy chúng. Nhiều số nguyên tố lớn hiện được biết đến - ví dụ: nếu chúng ta nhân hai số nguyên tố ngẫu nhiên có 100 chữ số và chỉ có tích kết quả, thì việc khôi phục các số bằng bất kỳ thuật toán nào sẽ mất một thời gian dài không thể chấp nhận được (ví dụ: gấp nhiều lần so với yêu cầu hoàn thành giao dịch ngân hàng).

Khi chúng ta xem xét số lượng các số nguyên tố, một số câu hỏi nảy sinh:

- Có bao nhiêu số nguyên tố trong khoảng $[a, b]$ cho trước?
- Phần vô cực là số nguyên tố?
- Có công thức tìm số nguyên tố thứ n không?

Nếu ta kí hiệu $\pi(x)$ với tất cả các số nguyên tố không vượt quá một số tự nhiên x thì việc tìm công thức tính $\pi(x)$ chính xác sẽ trả lời được ba câu hỏi trên. Thật không may, một công thức như vậy vẫn chưa tồn tại (và không chắc sẽ được tìm thấy - xem 6.2.), Nhưng có những công thức để tính gần đúng $\pi(x)$, chẳng hạn (hãy nhớ lại rằng $\ln x \equiv \log_e x$):

Định lý 1.1 (cho số nguyên tố). $\pi(x) \simeq \dfrac{x}{\ln(x-a)}$, *trong đó a là hằng số dương tùy ý, nhỏ hơn x. (Giá trị gần đúng nhất đạt được là* $a = 1$.*)*

Hệ quả 1.1. *Số nguyên tố thứ n xấp xỉ* $[n.\ln(n)]$. *Một xấp xỉ tốt hơn đạt được với* $[n[(\ln(n) + \ln(\ln n - 1))]$.

Hệ quả 1.2. *Xác suất để một số x là xấp xỉ* $\dfrac{1}{\ln x}$.

Trước khi chuyển sang các thuật toán để kiểm tra xem một số có phải là số nguyên tố hay không, chúng ta sẽ đề cập đến một số tính chất và định lý thú vị hơn cho các số nguyên tố [Số nguyên tố-1] [Số nguyên tố-2]:

Giả thuyết của Goldbach

1. Mọi số nguyên $n > 2$ đều có thể được biểu diễn dưới dạng tổng của hai số nguyên tố.
2. Mọi số nguyên $n > 17$ đều có thể được biểu diễn dưới dạng tổng của ba số nguyên tố khác nhau.
3. Mỗi số nguyên có thể được biểu diễn dưới dạng tổng của tối đa sáu số nguyên tố.
4. Mọi số nguyên lẻ $n > 5$ đều có thể được biểu diễn dưới dạng tổng của ba số nguyên tố.
5. Mỗi số chẵn có thể được biểu diễn dưới dạng hiệu của hai số nguyên tố.

Định lý 1.2. *Có vô số số rất nguyên tố loại* $n^2 + m^2$ *và* $n^2 + m^2 + 1$.

Giả thuyết. Có vô số số nguyên tố dạng $n^2 + 1$.

Định lý 1.3 (Operman). *Luôn luôn có một số nguyên tố giữa* n^2 *và* $(n + 1)^2$.

Bài tập

▷ **1.20.** 1. Tại sao việc tìm một công thức chính xác cho $\pi(x)$ sẽ trả lời được cả ba câu hỏi về số nguyên tố đã lập ở trên?
2. Viết chương trình kiểm tra các giả thuyết của Goldbach.
3. Viết chương trình kiểm tra định lý Operman.

1.2.1. Kiểm tra một số có phải là số nguyên tố

Một thuật toán hiển nhiên, hệ quả trực tiếp của định nghĩa, là như sau: chúng ta kiểm tra xem mỗi số trong khoảng $\left[2, \frac{p}{2} - 1\right]$ có chia hết cho p hay không và nếu chúng ta tìm thấy một thì theo đó p là hợp số.

Có một số "công thức" để kiểm tra xem một số có đơn giản hay không, nhưng trong thực tế, chúng không thể áp dụng được, vì việc triển khai chúng đòi hỏi nhiều tài nguyên tính toán hơn so với thuật toán vừa mô tả. Một ví dụ là sau đây

Định lý 1.4 (Wilson). *Số p là số nguyên tố nếu và chỉ khi* $(p-1)! \equiv -1 \pmod{p}$.

Nó phải tính toán $(p-1)!$, khó thực hiện hơn nhiều và đòi hỏi nhiều phép tính hơn so với việc thực hiện $\frac{p}{2} - 1$ phép chia trong thuật toán trên.

Dễ dàng thấy rằng không cần thiết phải kiểm tra tất cả các số có đến $\frac{p}{2} - 1$: chỉ cần kiểm tra chia hết cho đến $\sqrt{p}$ (bao gồm) là đủ. Điều này là do bất cứ khi nào p có một ước số x, $x > \sqrt{p}$, thì p được biểu diễn dưới dạng $p = x.y, y < \sqrt{p}$, tức là còn một số chia nhỏ hơn $\sqrt{p}$. Sau đây là cách triển khai thuật toán này:

Chương trình 1.8. Số chữ số (111prime.c)

```
#include <stdio.h>
#include <math.h>

const unsigned n = 23;

// Trả lại  1 thì $n$ là nguyên tố; 0 thì $n$ la hợp số
char isPrime(unsigned n)
{ unsigned i = 2;
  if (n == 2) return 1;
  while (i <= sqrt(n)) {
    if (n % i == 0) return 0;
    i++;
  }
  return 1;
```

```
}

int main(void) {
  if (isPrime(n))
    printf("So %u la nguyen to.\n", n);
  else
    printf("So %u la hop so.\n", n);
}
```

Chúng ta có thể mở rộng kết quả cuối cùng: để tìm rằng p là số nguyên tố, nó đủ để đảm bảo rằng nó không chia hết cho bất kỳ số nguyên tố nào khác trong khoảng $[2, \sqrt{p}]$ (Chúng ta để lại như một bài tập cho bạn đọc để chứng minh rằng kết luận sau là đúng). Vì vậy, nếu chúng ta có k số nguyên tố đầu tiên (ký hiệu là P_i, với $i = 1, 2, ..., k$), chúng ta sẽ có thể kiểm tra xem có số nào trong khoảng $[2, (P_k)^2]$ Thì đơn giản. Triển khai trong ngôn ngữ C như sau:

Chương trình 1.9. Kiểm tra số nguyên tố (112preproc.c)

```
#include <stdio.h>
const unsigned n = 25;
//Số lượng số nguyên tố mà ta cần phải tính
#define K 25
unsigned prime[K] = {
  2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
  47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};

// Kiểm tra có phải chăng là số nguyên tố
//bằng cách kiểm tra có ước số trong danh sách prime[]
char checkprime(unsigned n)
{ unsigned i = 0;
  while (i < K && prime[i] * prime[i] <= n) {
    if (n % prime[i] == 0) return 0;
    i++;
  }
  return 1;
}

int main(void)
{ if (checkprime(n))
```

```
    printf("So %u la so nguyen to. \n", n);
  else
    printf("So %u la hop so. \n", n);
  return 0;
}
```

Bài tập

▷ **1.21.** Viết chương trình kiểm tra xem một số có phải là số nguyên tố hay không, sử dụng định lý Wilson. Có cần thiết phải tính $(p-1)!$, với điều kiện chúng ta chỉ quan tâm đến modulo p phần dư của nó không?

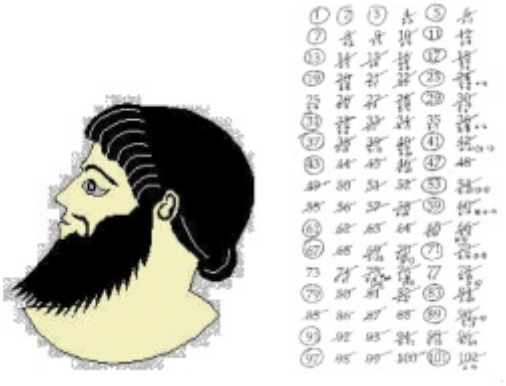
▷ **1.22.** Để chứng minh rằng để chứng minh rằng p là số nguyên tố, chỉ cần biết chắc chắn rằng nó không chia hết cho bất kỳ số nguyên tố nào khác trong khoảng $[2, \sqrt{p}]$.

1.2.2. Sàng Eratosten. Tìm số nguyên tố trong khoảng

Bài toán: Tìm tất cả các số nguyên tố nhỏ hơn hoặc bằng một số tự nhiên n. Một cách tiếp cận rõ ràng để giải quyết vấn đề là kiểm tra xem có số tự nhiên nào nhỏ hơn n là đơn giản hay không. Do đó, n lần kiểm tra được thực hiện, và đối với mỗi số k, tối đa k số lần kiểm tra tính chia hết sẽ được yêu cầu.

Với điều kiện có đủ bộ nhớ, chúng ta có thể áp dụng một phương pháp nhanh hơn để tìm các số nguyên tố trong một khoảng gọi là *sàng Eratosthenes*. Phương pháp này được đặt theo tên của người tạo ra nó - Eratosthenes of Siren (275-195 trước Công nguyên) - người đầu tiên dự đoán đường kính chính xác của Trái đất. Ông cũng được biết đến là người đã làm việc tại Thư viện Alexandria trong nhiều năm.

Như tên cho thấy, "sàng" là một phương pháp lập trình loại trừ tất cả các phần tử của một tập hợp hữu hạn mà chúng ta không quan tâm [Rakhnev, Garov, Gavrilov-1995] [Reingold, Nivergelt, Deo-1980]. Nó có thể được minh họa bằng một cái rây lọc spaghetti - nước mà chúng ta muốn "tống khứ" hết đi và spaghetti vẫn còn. Theo cách tương tự, sàng lọc của Eratosthenes "ép" các số tổng hợp, khiến chúng trở nên đơn giản.



Hình 1.6. Eratosthenes và lưới của mình

Trong trường hợp này, ý tưởng của cách tiếp cận này xuất phát từ đoạn trước: một số là đơn giản nếu không có ước số nguyên tố nào khác (ngoại trừ chính nó).

Lưới của Eratosthenes:

Chúng ta viết các số từ 2 đến n liên tiếp:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \ldots, n$$

Chúng ta tìm thấy số đầu tiên chưa được đánh dấu và chưa được đánh dấu – đó là 2. Chúng ta đánh dấu nó, sau đó gạch bỏ mỗi số thứ hai sau nó:

$$(2), 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \ldots, n$$

Tiếp theo, chúng ta tìm lại số đầu tiên chưa được đánh dấu và chưa được đánh dấu: đây là số 3. Chúng ta đánh dấu nó và gạch bỏ tất cả các số trong dãy, bội số của 3:

$$(2), (3), \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \ldots, n$$

Sau đó, đã đến lúc cho số 5 – chúng ta đánh dấu nó và gạch bỏ cứ sau mỗi ngày 5:

$$(2), (3), \cancel{4}, (5), \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \ldots, n$$

Do đó, tất cả các số tổng hợp đều được "sàng lọc" và chúng ta luôn chắc chắn rằng số i tối thiểu chưa được đánh dấu và chưa được

đánh dấu là đơn giản. Quá trình tiếp tục cho đến khi không có số nào bị gạch chéo hoặc không có dấu - khi đó tất cả các số được đánh dấu là số nguyên tố và tất cả các số bị gạch bỏ đều là hợp số. Dưới đây là sơ đồ chi tiết hơn để thực hiện sàng lọc bằng phương pháp sàng Eratosthenes:

Thuật toán sàng Eratosthenes

1. Khởi tạo mảng sieve[] với các số không. Sau này, khi gạch bỏ một số, chúng ta sẽ viết 1 ở vị trí tương ứng trong mảng i. Chúng ta khởi tạo $i = 2$.
2. Chúng ta tăng i cho đến khi sieve[i] trở thành 0. Khi đó số i là đơn giản và chúng ta in ra.
3. Đánh dấu bằng 1 tất cả các giá trị sieve[k], cho $k = i, 2i, 3i, ..., (n/i).i$ (tức là tất cả các bội số của i giá trị).
4. Nếu $i \leq n$ thì chúng ta quay lại bước 2, ngược lại thì kết thúc.

Chương trình 1.10. Danh sách số nguyên tố (113sieve.c)

```
#include <stdio.h>
#define MAXN 30000
const unsigned n = 200;
char sieve[MAXN];
//Tìm và in ra  số nguyên tố đến $n$
void eratosten(unsigned n)
{ unsigned j,  i  = 2;
  while (i  <= n) {
    if (sieve[i] == 0) {
      printf("%5u", i);
      j  = i;
      while (j  <= n) {
        sieve[j] = 1;
        j  += i;
      }
    }
    i++;
  }
}

int main(void) {
  unsigned i;
  for (i  = 0;  i  < n;  i++) sieve[i] = 0;
```

```
    eratosten(n);
    return 0;
}
```

Kết quả thực hiện chương trình:

```
   2   3   5   7   11   13   17   19   23   29   31   37   41   43   47
53   59   61   67   71   73   79   83   89   97   101   103   107   109
113   127   131   137   139   149   151   157   163   167   173   179
181   191   193   197   199
```

Ngoài ra còn có một thuật toán thậm chí còn hiệu quả hơn để tìm kiếm tất cả các số nguyên tố trong một khoảng thời gian. Nó không yêu cầu bộ nhớ (mảng sieve[N]) với kích thước của khoảng, cũng như không cần thiết phải thu thập dữ liệu toàn bộ khoảng ở mỗi bước.

Trong việc triển khai preproc.c từ đoạn trước, chúng ta đã nhập trước mảng số prime[], chứa k số nguyên tố đầu tiên và với sự trợ giúp của nó, chúng ta đã kiểm tra xem một số trong khoảng $[2, (P_k)^2]$ có số nguyên tố.

Bây giờ chúng ta sẽ áp dụng lược đồ đã sửa đổi sau: chúng ta sẽ bắt đầu với một danh sách trống, mà chúng ta sẽ điền tuần tự. Ví dụ, đặt nó vào số nguyên tố đầu tiên 2, chúng ta có thể tìm thấy tất cả các số nguyên tố trong khoảng $[3, 2^2]$ - chẳng hạn như 3 và chúng ta thêm nó vào danh sách. Hơn nữa, đã có các số nguyên tố đến 3, chúng ta có thể tìm thấy tất cả các số nguyên tố trong phạm vi $[4, 3^2]$ - đây là 5 và 7, chúng ta cũng thêm vào danh sách. Chúng ta tiếp tục quá trình này cho đến khi các số nguyên tố được thêm vào *prime*[] để kiểm tra xem mỗi số trong khoảng $[2, n]$ có phải là số nguyên tố hay không. Sau đây là một ví dụ triển khai:

Kiểm tra một số nguyên tố có nằm trong khoảng $[2, n]$ không?

Chương trình 1.11. Số nguyên tố trong một khoảng (114proc.c)

```
#include <stdio.h>
#define MAXN 10000
//Tìm số nguyên tố đến $n$
const unsigned n = 500;
unsigned primes[MAXN], pN = 0;
```

```
char isPrime(unsigned n)
{ unsigned i = 0;
  while (i  < pN && primes[i] * primes[i] <= n) {
    if (n % primes[i] == 0) return 0;
    i++;
  }
  return 1;
}

void findPrimes(unsigned n)
{ unsigned i = 2;
  while (i  < n) {
    if (isPrime(i)) {
      primes[pN] = i;
      pN++;
      printf("%5u", i);
    }
    i++;
  }
}

int main(void) {
  findPrimes(n);
  printf("\n");
  return 0;
}
```

Trong trường hợp chúng ta đang tìm n số nguyên tố đầu tiên, phương pháp sàng Eratosthenes cho kết quả rất tốt. Tuy nhiên, khi tìm các số nguyên tố trong khoảng $[a, b]$ cho $a(a >> 1)$ đủ lớn, tốt hơn nên sử dụng một thuật toán khác - một phiên bản cải tiến của thử nghiệm ngay lập tức.

Ba số nguyên tố đầu tiên là 2, 3 và 5. Do đó, bất kỳ số tự nhiên nào cũng có thể được biểu diễn dưới dạng:

$$n = 30.q + r, r \in [0..29] \text{ hoặc } r \in [0, \pm1, \pm2, ..., \pm14, 15].$$
$$30.q = 2.3.5.q$$

Khi đó trong số 30 số liên tiếp, chỉ có 8 số có thể là số nguyên tố,

tức là chúng ta chỉ cần kiểm tra $4k/15$ số, cụ thể là:

$$30.q \pm 1, 30.q \pm 7, 30.q \pm 11, 30.q \pm 13$$

Phương pháp được đề xuất dẫn đến tăng tốc gấp đôi so với phương pháp xác minh trực tiếp tất cả các con số trong khoảng thời gian xác định.

Bài tập

▷ **1.23.** Thuật toán có thể được cải thiện: ở bước 3) tìm kiếm bắt đầu từ $k = i^2$, trong khi tăng dần với giá trị i được giữ nguyên. Chứng minh cho kết quả này và sửa đổi việc triển khai cho phù hợp. So sánh với bản gốc.

▷ **1.24.** Để chứng minh thuật toán sàng.

▷ **1.25.** Để cải thiện thuật toán tìm kiếm các số nguyên tố trong một khoảng, vì mục đích này, 4 số nguyên tố đầu tiên được xem xét.

1.2.3. Phân tích một số thành tích thừa số nguyên tố

Định lý 1.5 (Định lý cơ bản của số học)**.** *Mọi số tự nhiên $P(P > 1)$ đều có thể được biểu diễn (thừa số) một cách duy nhất dưới dạng $P_1^{q_1}.P_2^{q_2}.....P_n^{q_n}$, trong đó $P_1 < P_2 < \ldots < P_n$ và P là các số nguyên tố và q_i là các số nguyên dương. [Siderov-1995].*

Dưới đây là một số ví dụ:

$$520 = 2^3.5^1.13^1$$
$$64 = 2^6$$
$$2345 = 5^1.7^1.67^1$$

Thuật toán để có được sự phân rã như vậy cần thiết cho một số tác vụ tính toán (ví dụ: Thuật toán 2 của 1.1.5.) Là như sau:

Hãy phân tích số P.

1) Ta đặt $i = 2$.

2) Ta đặt $k = 0$. Trong khi P chia hết cho i, ta thực hiện phép chia và tăng k lên một. Chúng ta chuyển sang 3).

3) Nếu $k > 0$, thì ta thu được số hạng tiếp theo của phân thức - đó là i^k. Chuyển sang 4)

4) Nếu $P > 1$, ta tăng i lên một và trở về 2).

Chúng ta để nó cho người đọc để chứng minh rằng thuật toán đang hoạt động bình thường. Sau đây là nhận thức đầy đủ của C:

Chương trình 1.12. Phân tích số thành tích thừa số nguyên tố (115numdev.c)

```
#include <stdio.h>
//Số để phân tích ra  thừa số
unsigned n = 435;

int main(void) {
  unsigned how, i, j;
  printf("%u = ", n);
  i  = 1;
  while (n != 1)  {
    i++;
    how = 0;
    while ((n % i)  == 0)  {
      how++;
      n = n /  i;
    }
    for (j  = 0;  j  < how; j++) printf("%u ", i);
  }
  printf("\n");
  return 0;
}
```

Bài tập

▷ **1.26.** Chứng minh thuật toán được đề xuất để phân tích nhân tử.

1.2.4. Tìm số lượng số không trong kết quả phép nhân

Bài toán: Cho trước một số số nguyên $a_1, a_2, ..., a_n$. Chúng ta tìm số lượng các số không mà tại đó tích $P = a_1.a_2.....a_n$.

Như trong 1.1.2, phép nhân là không mong muốn và không phải lúc nào cũng dẫn đến kết quả mong muốn. Để giải quyết vấn đề,

chúng ta sẽ chú ý đến thực tế sau: Các số duy nhất có tích kết thúc bằng 0 là 2 và 5, hoặc tích của bội số của 2 với bội số của 5.

Thuật toán giải quyết vấn đề mà không cần thực hiện phép nhân như sau:

1. Với mỗi $a_i (i = 1, 2, ..., n)$ ta biểu diễn a_i dưới dạng $a_i = 2^{M_i}.5^{N_i}.b_i, \quad b_i\%2 \neq 0, b_i\%5 \neq 0.$
2. Kết quả của sản phẩm sẽ là $P = 2^{\sum_{i=1..n} M_i}.5^{\sum_{i=1..n} N_i}.c$, ($c$ là hằng số), và số lượng các số không ở cuối sản phẩm sẽ là nhỏ nhất của $\sum_{i=1}^{n} M_i$ và $\sum_{i=1}^{n} N_i$.

Ví dụ, đối với chuỗi:

$$25, 4, 20, 11, 13, 15$$

sau khi phân hủy chúng ta sẽ nhận được:

$$2^0.5^2.1, 2^2.5^0.1, 2^2.5^1.1, 2^0.5^0.11, 2^0.5^0.13, 2^0.5^1.3,$$

mà cho 4 số không. Ta có thể kiểm tra tính đúng đắn của kết quả thu được một cách trực tiếp: $25.4.20.11.13.15 = 4290000$.

Chúng ta sẽ để việc triển khai thuật toán vừa được mô tả như một bài tập cho người đọc. Nó sẽ là một sửa đổi nhỏ của thuật toán đã được xem xét để phân tách số lượng các ước số nguyên tố từ đoạn trước.

Chúng ta sẽ kết thúc với một công thức để tìm số lượng các số không ở cuối $n!$. Số này bằng $\sum_{k=1}^{[\log_5 n]} \left[\frac{n}{5^k}\right]$. Công thức được suy ra như một hệ quả của sơ đồ tổng quát hơn, sau khi tính xem có bao nhiêu lần các chữ số 2 và 5 tham gia làm ước của n số tự nhiên đầu tiên.

Chương trình 1.13. Tìm số 0 làm phép nhân (116factzero.c)

```
#include <stdio.h>
const unsigned n = 10;
int main(void) {
  unsigned zeroes = 0, p = 5;
  while (n >= p) {
    zeroes += n / p;
    p *= 5;
  }
```

```
    printf("So luong so 0 cua %u! la %u\n", n, zeroes);
    return 0;
}
```

Bài tập

▷ **1.27.** Để chứng minh thuật toán được đề xuất để tìm số lượng các số không mà tại đó phép nhân kết thúc.

▷ **1.28.** Để thực hiện thuật toán được đề xuất tìm số lượng các số không tại đó thuật toán kết thúc.

▷ **1.29.** Biện minh cho công thức tìm số lượng các số không kết thúc bằng $n!$

1.3. Số mensen và số hoàn thiện

1.3.1. Số mensen

Định lý 1.6. *Một số nguyên tố được gọi là số nguyên tố Mersenne nếu nó có thể được biểu diễn dưới dạng* $2^p - 1$, *trong đó p là số nguyên tố.*

Hiện tại, 39 giá trị của p đã được biết, trong đó các số $2^p - 1$ là Mersenne:

2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917

37 giá trị đầu tiên của p số tương ứng với lũy thừa của 37 số Mersenne đầu tiên. Hai số cuối cùng là của Mersen (lần lượt được phát hiện vào năm 1999 và 2001), nhưng chúng không nhất thiết phải là số 38 và 39, vì chúng được tìm thấy bằng máy tính mà không cần kiểm tra tất cả các số nhỏ hơn. các giá trị của p. nghĩa là, có thể có số Mersenne nhỏ hơn.

Số Mersenne tìm thấy một số ứng dụng: để tìm kiếm các *số hoàn hảo*, để tìm kiếm các số nguyên tố rất lớn và các ứng dụng khác. Chúng vừa có giá trị thực tế vừa mang giá trị biểu tượng. Khi số nguyên tố thứ 23 của Mersen được tìm thấy vào năm 1963, Trường

Toán học của Đại học Illinois đã rất tự hào về khám phá của mình đến nỗi tất cả các bức thư được gửi kèm theo một con tem bổ sung là "$2^{11213} - 1$ là số nguyên tố" (xem Hình1.7).

Hình 1.7. $2^{11213} - 1$ là số nguyên tố

Một số số nguyên tố lớn nhất được tìm thấy cho đến nay là Mersenne:

$2^{13466917} - 1$ với 8107892 chữ số (?? số nguyên tố Mersenne thứ 39)

$2^{6972593} - 1$ với 2098960 chữ số (?? số nguyên tố Mersenne thứ 38)

$2^{3021377} - 1$ với 909526 chữ số (số nguyên tố Mersenne thứ 37)

$2^{2976221} - 1$ với 895932 chữ số (số Mersenne thứ 36)

$2^{1398269} - 1$ với 420921 chữ số (số Mersenne thứ 35)

Việc công bố các giải thưởng tiền mặt lớn là một hiện tượng ngày càng phổ biến đối với các kết quả khoa học hoặc thực nghiệm nghiêm túc. Ví dụ, có những dự án phổ biến trên Internet để tìm kiếm số nguyên tố lớn tiếp theo (và rất có thể nó sẽ là số nguyên tố của Mersen - xem bên dưới). Những dự án như vậy được tài trợ bởi các tổ chức nghiên cứu nổi tiếng, và việc tham gia vào một dự án như vậy giống như một cuộc xổ số: người may mắn trúng số nguyên tố thứ 38 của Mersen thì trúng 50.000 USD. Giải thưởng cho những con số tiếp theo bắt đầu từ 250.000 đô la [Số nguyên tố-3].

Một thuật toán hiển nhiên để tìm kiếm các số Mersenne như sau: liên tiếp với mỗi số nguyên tố p = 2, 3, 5, 7, 11,... chúng ta kiểm tra xem M = 2 p - 1 có đơn giản không. Tuy nhiên, thuật toán này cực kỳ

kém hiệu quả và trong thực tế không cho kết quả khả quan, áp dụng cho số lượng lớn. Chìa khóa để tìm ra các số lớn như vậy là định lý Lucas năm 1870, sau đó được Lehmer sửa đổi, và ứng dụng của nó cũng yêu cầu một chương trình nhân các số nhanh (một phương pháp nhân các số như vậy được tìm thấy sau này và dựa trên phép biến đổi Fourier nhanh chóng [Guinier -1991]).

Phương pháp Lucas - Lehmer (1930) bao gồm các nội dung sau:

Trình tự lặp lại được xác định:

$$E_1 = 4$$
$$E_{n+1} = (E_n)^2 - 2$$

Một số thành viên đầu tiên của loạt bài này là:

$$4, 14, 194, 37634, ...$$

Định lý 1.7 (Lucas-Lemer). *Số tự nhiên $m = 2^p - 1$ là số nguyên tố Mersenne (với p lẻ) nếu và chỉ khi:*

$$(E_{p-1})\%(2^p - 1) = 0$$

Chúng ta cho phép người đọc triển khai một chương trình để tìm một vài số nguyên tố Mersenne đầu tiên. Trong đoạn tiếp theo, chúng ta sẽ xem xét một trong những ứng dụng của số nguyên tố Mersenne - để tìm kiếm các số hoàn hảo.

Bài tập

▷ **1.30.** Viết chương trình tìm n số Mersenne đầu tiên mà không sử dụng định lý Lucas-Lemmer.

▷ **1.31.** Viết chương trình tìm n số Mersenne đầu tiên sử dụng định lý Lucas-Lemmer. Để so sánh hiệu quả với bài toán trước đó.

▷ **1.32.** $2^n - 1$ có thể là số nguyên tố nếu n không phải là số nguyên tố không?

1.3.2. Số hoàn thiện

Định nghĩa 1.15. Một số tự nhiên n được gọi là *hoàn hảo* nếu nó bằng tổng của tất cả các ước của nó (bản thân n không được coi là ước).

3 số hoàn hảo đầu tiên là:

$$\begin{aligned} 6 &= 1+2+3, \\ 28 &= 1+2+4+7+14, \\ 496 &= 1+2+4+8+16+31+62+124+248. \end{aligned}$$

Số lượng các số hoàn hảo tăng rất nhanh:

$$8128, 33550336, 8589869056, \ldots$$

Vì vậy, nếu chúng ta cố gắng tìm kiếm chúng bằng một thuật toán trực tiếp (liên tiếp với từng $n = 1, 2, 3, \ldots$ để kiểm tra xem nó có hoàn hảo hay không), chúng ta sẽ không thể tìm được nhiều hơn số hoàn hảo thứ 5 cho hợp lý. thời gian tính toán. Sau đây là sự hỗ trợ của chúng ta:

Định lý 1.8 (Euler). *Nếu $2^p - 1$ là số nguyên tố thì $2^{p-1}(2^p - 1)$ là hoàn hảo.*

Theo định lý trên, một chương trình tìm n số hoàn hảo đầu tiên có thể được biên soạn trực tiếp: chỉ cần biết lũy thừa p_i của n số Mersenne đầu tiên (10 lũy thừa đầu tiên là $2, 3, 5, 7, 13, 17, 19, 31, 61, 89$). Vấn đề nảy sinh bởi vì ngay cả ở các giá trị nhỏ của n, các số hoàn hảo được yêu cầu vượt quá giá trị lớn nhất cho phép đối với một kiểu số nguyên trong ngôn ngữ C. Do đó, chúng ta sẽ không sử dụng các kiểu tiêu chuẩn, nhưng sẽ thực hiện các phép toán cần thiết với các số "lớn": nhân một số với 2 và tăng một. Chúng ta sẽ giữ số "lớn" trong mảng một chiều number[]: mỗi phần tử của nó đại diện cho một chữ số thập phân. Để thuận tiện, các chữ số của số được sắp xếp theo thứ tự ngược lại trong mảng: chữ số đầu tiên được viết trong number[k-1] (nếu số có k chữ số), chữ số thứ hai trong number[k-2], v.v. Chữ số k cuối cùng được viết bằng number[0]. Thuật toán để tăng một số được viết bằng number[] như sau:

Tăng một đơn vị

```
i = 0; number[i]++;
while(10==number[i]){number[i]=0; number[++i]++;}
if (i == k) k++;
```

Vì trong tìm kiếm các số hoàn hảo chỉ thực hiện phép nhân với 2 nên kết quả sẽ là lũy thừa của cặp số, tức là chữ số cuối cùng không được là 9, thì hai dòng cuối cùng, bắt đầu từ dấu kiểm **while** (10 == number [i]), là thừa.

Tương tự, chúng ta nhân một số lớn với hai:

Nhân một số lớn với hai

```
unsigned i, carry = 0, temp;
for (i = 0; i < k; i++) {
temp = number[i] * 2 + carry;
number[i] = temp % 10;
carry = temp / 10;
}
if (carry > 0) number[k++] = carry;
```

Sau đây là một triển khai hoàn chỉnh tìm 10 số hoàn hảo đầu tiên, với lũy thừa của mười số nguyên tố Mersenne đầu tiên được đặt làm hằng số trong mảng mPrimes[]

Chương trình 1.14. Số hoàn thiện (117perfect.c)

```
#include <stdio.h>
#define MN 10
unsigned mPrimes[MN]={2,3,5,7,13,17,19,31,61,89 };
unsigned k, number[200];

void doubleN(void)
{ unsigned i, carry = 0, temp;
  for (i = 0; i < k; i++) {
    temp = number[i] * 2 + carry;
    number[i] = temp % 10;
    carry = temp / 10;
  }
  if (carry > 0) number[k++] = carry;
}

void print(void)
{ unsigned i;
  for (i = k; i > 0; i--) printf("%u", number[i-1]);
  printf("\n");
}
```

```
void perfect(unsigned s, unsigned m)
{ unsigned i;
  k = 1; number[0] = 1;
  for (i = 0; i < m; i++) doubleN();//đây là ước số có dạng $2^i$
  number[0]--;//những chữ số cuối cùng chắc chắn giữa $\{2,4,8,6\}$
  for (i = 0; i < m - 1; i++) doubleN();
  printf("So hoan hao thu %2u la = ", s);
  print();  // In ra lần lượt các số
}

int main() {
  unsigned i;
  for (i=1; i<= MN; i++) perfect(i, mPrimes[i-1]);
  return 0;
}
```

Kết quả thực hiện chương trình:

Số hoàn hảo đầu tiên là = 6
Số hoàn hảo thứ 2 là = 28
Số hoàn hảo thứ 3 là = 496
Số hoàn hảo thứ 4 là = 8128
Số hoàn hảo thứ 5 là = 33550336
Số hoàn hảo thứ 6 là = 8589869056
Số hoàn hảo thứ 7 là = 137438691328
Số hoàn hảo thứ 8 là = 2305843008139952128
Số hoàn hảo thứ 9 là = 2658455991569831744654692615953842176
Số hoàn hảo thứ 10 là = 191561942608236107294793378084303638130997321548168

Lưu ý: Rõ ràng, định lý trên chỉ cho các số hoàn hảo chẵn (thực tế là tất cả các số hoàn hảo chẵn). Câu hỏi về việc liệu có những số hoàn hảo lẻ vẫn chưa được trả lời (có lẽ là không). Người ta đã chứng minh rằng nếu có thì chúng phải có ít nhất 300 chữ số và nhiều ước.

Bài tập

▷ **1.33.** Chứng minh rằng tổng các giá trị nghịch đảo của các ước (bao gồm cả 1 và số chính nó) của mỗi số hoàn hảo là 2. Ví dụ, với 6 ta có: $1/1 + 1/2 + 1/3 + 1/6 = 2$.

▷ **1.34.** Viết chương trình tìm tất cả các số mà tổng giá trị nghịch đảo của các ước (kể cả 1 và chính số đó) là 2. Chúng có hoàn hảo không?

1.4. Những hệ số đa thức, tam giác Pascal và giai thừa

Định nghĩa 1.16. Cho một tập hợp n phần tử đã cho. Số của tất cả các tập con k phần tử có thể có của nó được gọi là hệ số nhị thức và được biểu thị và tính toán như sau:

$$C_n^k = C_n^k = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1}. \tag{1.8}$$

Chúng ta sẽ lưu ý rằng các ký hiệu C_n^k và C_n^k không hoàn toàn tương đương và ký hiệu sau tổng quát hơn ký hiệu trước, vì nó thường cho phép các giá trị thực của n, nơi không có ý nghĩa tổ hợp. Tuy nhiên, đối với mục đích của chúng ta, các ký hiệu này là tương đương và chúng ta sẽ sử dụng chúng song song. Nếu n là số tự nhiên, ta có thể viết (1.8) là:

$$C_n^k = C_n^k = \frac{n!}{k!(n-k)!}. \tag{1.9}$$

Chúng ta sẽ lưu ý rằng đây chính xác là công thức cung cấp số tổ hợp không lặp lại n phần tử của lớp k (xem 1.3.3.) Và đây không phải là ngẫu nhiên (Tại sao?).

Hệ số nhị thức có một số ứng dụng, ví dụ như trong khai triển của $(a+b)^n$, do đó tên của chúng là:

$$(a+b)^n = C_n^0 a^n b^0 + C_n^1 a^{n-1} b^1 + \dots + C_n^n a^0 b^n = \sum_{i=0}^{n} C_n^i a^{n-i} b^i.$$

Có một số thuộc tính và công thức liên quan đến hệ số nhị thức [Knuth-1/1968]. Chúng ta sẽ xem xét một số trong số chúng trong đoạn này.

Trong Bảng 1.3. các giá trị của C_n^k được hiển thị, với $0 \le k \le n < 10$.

n	C_n^0	C_n^1	C_n^2	C_n^3	C_n^4	C_n^5	C_n^6	C_n^7	C_n^8	C_n^9
0	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0	0	0
3	1	3	3	1	0	0	0	0	0	0
4	1	4	6	4	1	0	0	0	0	0
5	1	5	10	10	5	1	0	0	0	0
6	1	6	15	20	15	6	1	0	0	0
7	1	7	21	35	35	21	7	1	0	0
8	1	8	28	56	70	56	28	8	1	0
9	1	9	36	84	126	126	84	36	9	1

Bảng 1.3. Hệ số đa thức

Một số tính chất của hệ số nhị thức trở nên rõ ràng nếu chúng ta phân tích Bảng 1.3 kỹ hơn, ví dụ:

$$C_n^0 = C_n^n = 1, \tag{1.10}$$
$$C_n^k = C_n^{n-k}, \tag{1.11}$$
$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}. \tag{1.12}$$

Nếu chúng ta loại bỏ các phần tử 0 khỏi Bảng 1.3. ở trên chúng ta sẽ nhận được một tam giác được gọi là tam giác Pascal:

$n = 0$ 1
$n = 1$ 1 1
$n = 2$ 1 2 1
$n = 3$ 1 3 3 1
$n = 4$ 1 4 6 4 1
$n = 5$ 1 5 10 10 5 1

Hình 1.8. Tam giác Pascal

Số đầu tiên và số cuối cùng trong mỗi hàng là 1 (theo sau từ thuộc tính (1.10) và (1.11)), và lấy nhau dưới dạng tổng của hai số ở

trên nó (thuộc tính (1.12)). Do đó, các hệ số của bậc n có thể được tìm thấy nếu chúng ta có bậc $(n-1)$. Nếu chúng ta tìm $C_n^k = C_n^k$ và bắt đầu từ hàng đầu tiên, chúng ta có thể tìm thấy tất cả các hàng cho đến thứ n và giá trị ở vị trí thứ k sẽ chính xác là C_n^k. Lưu ý rằng khi tính toán các giá trị trong hàng i, chúng ta chỉ sử dụng các giá trị của hàng $i-1$, chứ không sử dụng các giá trị trước đó.

Sau đây là cách triển khai trực tiếp trong ngôn ngữ C: chúng ta sẽ giữ hàng được tính cuối cùng trong mảng lastLine[], và các tham số n và k cho C_n^k bắt buộc được định nghĩa ở đầu chương trình dưới dạng hằng số.

Chương trình 1.15. Tam giác Pascal (118pascalt.c)

```
#include <stdio.h>
// Cỡ lớn nhất của tam giác
#define MAXN 1000
unsigned n = 7;
unsigned k = 3;
unsigned long lastLine[MAXN + 1];

int main(void) {
  unsigned i, j;
  lastLine[0] = 1;
  for (i = 1; i <= n; i++) {
    lastLine[i] = 1;
    for (j = i - 1; j >= 1; j--)
      lastLine[j] += lastLine[j - 1];
  }
  printf("C(%u,%u) = %lu\n", n, k, lastLine[k]);
  return 0;
}
```

Chúng ta sẽ đề xuất một thuật toán khác dựa trên công thức (1.8) và trình bày một sơ đồ hữu ích để giảm các phân số hữu tỉ có tử số và mẫu số lớn. Nếu chúng ta thử tính toán trực tiếp trên (1.9) $n!$ và $k!(n-k)!$, trong nhiều trường hợp thu được kết quả trung gian lớn, trong khi kết quả cuối cùng không phải là một số lớn như vậy: ví dụ, với $n = 100$ và $k = 2$, chúng ta sẽ phải tính 100! và 2!.98! (các số trên 150 chữ số), và kết quả của C_{100}^2 chỉ là 4950.

Thuật toán 2 để tìm C_n^k (với thừa số hóa)

1. Như một ví dụ, chúng ta sẽ xem xét C_7^3. Chúng ta sẽ biểu diễn tử số của công thức (1.8) dưới dạng tích của các số, mỗi số chúng ta sẽ phân tích thành các thừa số nguyên tố. Bây giờ chúng ta có thể phân tích toàn bộ tác phẩm: đối với ví dụ $7! = 1.2.3...7 = 1.2.3.2^2.5.(2.3).7 = 2^4.3^2.5^1.7^1$.
2. Tương tự, chúng ta phân tích mẫu số: $3!.(7-3)! = 1.2.3.1.2.3.2^2 = 2^4.3^2$.
3. Viết tắt tử số và mẫu số: $\frac{2^4.3^2.5^1.7^1}{2^4.3^2} = \frac{5^1.7^1}{1}$.
4. Sau khi rút gọn ta thực hiện các phép nhân còn lại ở tử số và được kết quả như ý: $5^1.7^1 = 35$.

Việc thực hiện thuật toán 2 như sau.

Chương trình 1.16. Tính hệ số Newton (119cnk.c)

```
#include <stdio.h>
#define MAXN 500
unsigned long n = 7;
unsigned long k = 3;
unsigned long pN, primes[MAXN], counts[MAXN];

void modify(long int x, int how)
{ unsigned i;
  for (i = 0; i < pN; i++)
    if (x == primes[i]) {counts[i] += how; return;}
  counts[pN] = how;
  primes[pN++] = x;
}

void solve(unsigned long start, unsigned long end, unsigned long
    inc)
{ unsigned long prime, mul, how, i, max;
  for (i = start; i <= end; i++) {
    mul = i;
    prime = 2;
    while (mul != 1) {
      for (how = 0; mul % prime == 0; mul /= prime, how++);
      if (how > 0) modify(prime, inc * how);
      prime++;
    }
  }
```

```
}

long int calc(void)
{ int i, j;
  long int result = 1;
  for (i = 0; i < pN; i++)
    for (j = 0; j < counts[i]; j++) result *= primes[i];
  return result;
}

int main(void) {
  printf("C(%lu,%lu)= ", n, k);
  pN = 0;
  if (n - k < k) k = n - k;
  solve(n - k + 1, n, 1);  // phân tích tử số (n-k+1),...,n
  solve(1, k, -1);      // phân tích mẫu số 1,...,k
  printf("%lu\n", calc());
  return 0;
}
```

Một cách tiếp cận thuật toán như vậy (tính toán thừa số, tức là thực hiện các hành động với các số ở dạng thừa số nhân với thừa số nguyên tố) được áp dụng thành công không chỉ khi tìm C_n^k, mà luôn luôn khi thu được các giá trị trung gian lớn ở tử số và mẫu số, trong khi bản thân kết quả không phải là quá tuyệt vời.

Lưu ý: Trong chương trình trên, nếu cần, chúng ta áp dụng công thức "đảo ngược" dữ liệu đầu vào. Đồng thời, những vết cắt rõ ràng được thực hiện đầu tiên và chỉ sau đó là sự phân hủy.

Bài tập

▷ **1.35.** Chứng minh các tính chất (1.10), (1.11) và (1.12) bằng cách sử dụng định nghĩa hệ số của nhị thức. Ví dụ, bằng chứng của (1.10) có thể được thực hiện như sau:

▷ **1.36.** Đưa ra các chứng minh tổ hợp của các tính chất (1.10), (1.11) và (1.12).

▷ **1.37.** Thuật toán đề xuất 1 (pascalt.c) yêu cầu một mảng lastLine[] với $n+1$ phần tử để tìm C_n^k và ở các giá trị cao hơn của n, việc cấp

phát nhiều bộ nhớ như vậy sẽ không thể thực hiện được. Có thể sửa đổi nó theo cách sau: vòng lặp bên trong, lấp đầy hàng tiếp theo của tam giác, nên được thực hiện không phải từ 1 đến i, mà từ 1 đến k, vì đoạn của tam giác không còn quan tâm đến chúng ta. Lưu ý rằng khi k gần với n, chúng ta có thể áp dụng tính chất (1.11) và tìm C_n^{n-k} thay cho C_n^k.

1.5. Hệ số đếm và sự biến đổi hệ

Nói chung, chúng ta có thể định nghĩa hệ thống số như một tập hợp các dấu hiệu đồ họa và các quy tắc để biểu diễn các số. Trong các thời đại khác nhau của sự phát triển lịch sử của nhân loại, nhiều hệ thống số khác nhau đã được sử dụng, phổ biến nhất ngày nay là hệ thống số Ả Rập. Nó sử dụng một hoặc nhiều ký hiệu đồ họa $0, 1, 2, 3, 4, 5, 6, 7, 8$ và 9, được gọi là số, để đại diện cho các số tự nhiên. Thế giới Ả Rập khác với mười biểu tượng đồ họa được đề cập ở trên, nhưng nó là cùng một hệ thống số.). Số chữ số dùng để viết số sẽ được gọi là cơ sở của hệ thống. Ví dụ, hệ thống chữ số Ả Rập là hệ thập phân (sử dụng 10 chữ số). Cũng có những khái niệm khái quát về cơ số, cho phép nó là một số thực (môđun khác 0 và 1) tùy ý (hoặc thậm chí phức) với một dấu. Trong trường hợp chúng ta có $p < 0$ đối với cơ số p của hệ thống số, các số âm sẽ được viết mà không sử dụng dấu "-" một cách rõ ràng.

Các hệ thống khác đóng một vai trò đặc biệt trong số học máy tính - hệ nhị phân, hệ thập lục phân và một phần là hệ bát phân. Hệ thống số nhị phân sử dụng các chữ số 0 và 1 để viết số. Ví dụ: số thập phân $11_{(10)}$ sẽ giống như $1011_{(2)}$ (Chúng ta sẽ sử dụng dấu ngoặc đơn và một phông chữ nhỏ để chỉ ra cơ sở của hệ thống số). . Cần có 16 chữ số để viết số trong hệ thống số thập lục phân. Do đó, ngoài các chữ số từ 0 đến 9 của hệ thống số thập phân, các chữ cái in hoa A, B, C, D, E và F được sử dụng, với các giá trị $10, 11, 12, 13, 14$ và 15, tương ứng. số $254_{(10)}$ được biểu thị là $FE_{(16)}$.

Bằng phương pháp phân tích toán học, có thể chỉ ra rằng hệ thống số tối ưu là hệ thống có cơ số $e = 2,718281828...$(số e định nghĩa như $a_n = \left(1 + \frac{1}{n}\right)^n$ với $n \to \infty$). Ở đây, tối ưu có nghĩa là tỷ lệ tốt nhất giữa độ dài bản ghi của một số trong hệ thống tương ứng

và số chữ số được sử dụng bởi hệ thống.

Vì số e là số vô tỷ (nó không thể được biểu diễn dưới dạng tỷ số của hai số nguyên), làm việc với hệ thống này là không thuận tiện và theo nghĩa này, chúng ta có thể coi các hệ thống số tối ưu dựa trên 3 và 2. Từ quan điểm toán học, hệ thống số bậc ba là thích hợp hơn vì số 3 gần với e hơn. Tuy nhiên, từ quan điểm kỹ thuật thuần túy, hệ thống số nhị phân được ưu tiên hơn. Hệ thống số nhị phân cực kỳ thuận tiện cho việc thực hiện kỹ thuật - ở 0 điện áp là $0V$ được so sánh và ở $1-5$ V. Hầu như tất cả các máy tính hiện đại đều hoạt động trong hệ thống số nhị phân. Trước đây ở Liên Xô cũng đã có những máy móc hoạt động với hệ số đối xứng bậc ba, ví dụ như Setun, được tạo ra vào năm 1960 tại Moscow, sử dụng các số $-1, 0$ và 1.

Ý nghĩa của hệ thập lục phân được xác định bởi thực tế rằng 16 là lũy thừa của $2 (16 = 2^4)$. Cần 4 bit (chữ số nhị phân - 0 và 1) để mã hóa các số từ 0 đến 15. Để tạo điều kiện thuận lợi cho quá trình xử lý của chúng, các bit nhị phân này được nhóm thành các byte - nhóm 8 bit. Dễ dàng nhận thấy rằng các số từ 0 đến 255 có thể được viết trong một byte, tức là 256 số khác nhau. Vì $256 = 16^2$ nên mỗi byte có thể được biểu diễn bằng hai chữ số thập lục phân liên tiếp. Việc chuyển từ hệ nhị phân sang hệ thập lục phân rất đơn giản - số nhị phân được chia từ phải sang trái thành các tứ phân (4 bit liên tiếp) và nếu cần thì bên trái được bổ sung bằng các số không, sau đó mỗi tứ phân được chuyển đổi riêng rẽ thành một hệ thập lục phân tương ứng chữ số.

Chúng ta sẽ minh họa phương pháp vừa được mô tả bằng cách chuyển đổi số $10101000110111_{(2)}$ sang hệ thống số thập lục phân. Chúng ta chia nó thành các sổ ghi chép:

$$10|1010|0011|0111$$

và thêm các số không vào nó:

$$0010|1010|0011|0111$$

Bây giờ chúng ta mã hóa mỗi sổ ghi chép bằng chữ số thập lục phân tương ứng của nó và thu được $2A37_{(16)}$.

Tình hình cũng tương tự khi chuyển từ hệ thống số nhị phân sang hệ bát phân, trong trường hợp đó số được chia thành các bộ ba (ba chữ số nhị phân liên tiếp). Dưới đây chúng ta sẽ xem xét một phương pháp tổng quát hơn để chuyển đổi giữa các hệ thống số.

Lưu ý rằng bản ghi số không thể xác định rõ ràng nó được viết trong hệ thống số nào. Ví dụ, đối với số 153, chúng ta chỉ có thể nói rằng nó có ít nhất là 6. Thực tế, dựa trên ý nghĩa của các số 1, 5 và 3, chúng ta có thể nói rằng số đó được viết trong một hệ thống dựa trên ít nhất 3 , bởi vì Có 3 chữ số khác nhau, có thể được hiểu là 0, 1 và 2. Để tránh những hiểu lầm như vậy, theo thông lệ, chúng ta chỉ định trong ngoặc sau số đó bằng phông chữ nhỏ hơn, như trên, cơ sở của hệ thống được sử dụng .

Hệ thống chữ số Ả Rập là vị trí, tức là giá trị của các chữ số không được xác định chặt chẽ và thay đổi tùy thuộc vào vị trí của chữ số. Ví dụ, trong số 123, số 3 có giá trị là 3, trong khi ở số 34, nó có giá trị là 30. Cũng có các hệ thống số không vị trí (xem 1.1.7.), Trong đó giá trị của mỗi dấu hiệu được cố định nghiêm ngặt và không phụ thuộc vào vị trí mu.

Nói chung có hai phương pháp để viết một số trong một hệ thống số: *kỹ thuật số* và *đa thức*. Ghi âm kỹ thuật số thường được ưu tiên là ngắn hơn. Trong trường hợp này, các chữ số tương ứng của số được viết liền kề nhau và giá trị của chúng tăng từ phải sang trái theo một cấp số nhân hình học với một phần p, trong đó p là cơ sở của hệ thống số được sử dụng. Ký hiệu số có dạng $a_n a_{n-1} ... a_0(p)$, trong đó $a_i (1 \leq i \leq n)$ là một chữ số. Trong một bản ghi đa thức, số có dạng:

$$A = a_n p^n + a_{n-1} p^{n-1} + ... a_1 p + a_0$$

Kí hiệu đa thức rất hữu ích khi chuyển từ hệ thống số này sang hệ thống số khác, như chúng ta sẽ thấy bên dưới.

Bài tập

▷ **1.38.** Tại sao cơ sở của hệ thống số phải là môđun khác 0 và 1?

▷ **1.39.** Để trình bày các số 17 và -17 trong một hệ thống số, hãy dựa vào: 2; s8; 16.

▷ **1.40.** Để trình bày các số 17 và -17 trong một hệ thống số, dựa

vào $-2; -8; -16$.

▷ **1.41.** Không cần thông qua hệ thập phân, chuyển đổi các số nhị phân: $111, 110100, 1110100101, 10010101, 10101010101$ và 10111110101 sang hệ thống số thập lục phân.

▷ **1.42.** Không cần thông qua hệ thập phân, chuyển các số nhị phân: $11, 11001, 1010101, 111111, 1010101000, 10101101000$ và 11010111000 thành một hệ thống số bát phân.

▷ **1.43.** Kí hiệu đa thức của một số trong hệ thống số đối xứng có dạng như thế nào?

▷ **1.44.** Biểu diễn các số 17 và -17 trong một hệ số đối xứng bậc ba.

1.5.1. Chuyển hệ cơ số 10 sang cơ số p

Dựa vào cách biểu diễn một số ở dạng đa thức, ta có thể xây dựng thuật toán chuyển từ hệ thập phân sang hệ số p như sau: chia số A cho p theo thương và dư cho đến khi A trở thành 0, sau đó viết ngược lại số dư. đơn đặt hàng. biên nhận của họ (Tại sao?). Ví dụ, khi chuyển đổi số 29 thành một hệ thống số nhị phân, chúng ta nhận được (với chữ in nghiêng dưới mỗi phép chia là phần dư tương ứng):

$$29 : 2 = 14 : 2 = 7 : 2 = 3 : 2 = 1 : 2 = 0$$

$$\mathbf{1 \quad 0 \quad 1 \quad 1 \quad 1}$$

Ta lấy các số dư còn lại thu được theo thứ tự ngược lại ta được: $29_{(10)} = 11101_{(2)}$.

Hàm convert () thực hiện chuyển đổi và trả về kết quả là một chuỗi ký tự:

Hàm chuyển đổi convert() trong 120base.c

```
char getChar(char n) //Trả về ký hiệu tương ứng với $n$
{ return (n < 10) ? n + '0' : n + 'A' - 10; }

void reverse(char *pch)
{ char *pEnd;
```

```
  for (pEnd = pch + strlen(pch) - 1; pch < pEnd; pch++, pEnd--) {
    char c = *pch;
    *pch = *pEnd;
    *pEnd = c;
  }
}
void convert(char *rslt, unsigned long n, unsigned char base)
//Chuyển đổi số nguyên hệ thập phân n (n >= 0)
//thành hệ đếm có cơ sở
{ char *saveRslt = rslt;
  while (n > 0) {
    * rslt ++ = getChar((char)(n % base));
    n /= base;
    }
  * rslt = '\0';
  reverse(saveRslt);
}
```

Nếu số nhỏ hơn 1, lại có một ký hiệu ở dạng đa thức, trong trường hợp đó sẽ thu được một đa thức *bậc âm* của p. Khi chuyển A sang hệ số p, mọi thứ hoàn toàn ngược lại với trường hợp của một số tự nhiên: Ta *nhân* A với p, tách *các phần nguyên* và viết chúng theo thứ tự nhận. Ví dụ: chuyển đổi 0,125 thành một hệ thống số nhị phân trông giống như sau:

$$0,125,2 = \mathbf{0},25.2 = \mathbf{0},5.2 = \mathbf{1},0$$

Ta được $0,125_{(10)} = 0,001_{(2)}$. Ở đây một vấn đề khác nảy sinh - ký hiệu của A trong hệ số thứ p có thể là một phân số vô hạn (không tuần hoàn). Do đó, sẽ là khôn ngoan nếu giới thiệu một số chính xác số chữ số tối đa sau dấu thập phân. Hàm convertLessThan1()chuyển đổi số $A(0 \leq A < 1)$ thành cnt chữ số gần nhất sau dấu thập phân, bỏ qua các chữ số sau.

Hàm chuyển đổi convertLessThan1() trong 120base.c

```
void convertLessThan1(char *rslt,
    double n,
    unsigned char base,
    unsigned char cnt)
//Chuyển một số 0<= n <1 vào hệ đếm với cơ số
```

```
//không lớn hơn cnt của số chữ số sau dấu phẩy thập phân.
{
  while (cnt--) {
      //Phải chăng ta không nhân 0?
      if (fabs(n) < EPS) break;
      //Nhận chữ số tiếp theo
      n *= base;
      *rslt++ = getChar((char)(int)floor(n));
      n -= floor(n);
  }
  *rslt = '\0';
}
```

Sau khi thực hiện các hàm chuyển đổi riêng biệt cho các trường hợp $A > 1$ và $0 \leq A < 1$, nó vẫn tổ hợp chúng theo cách tự nhiên, cho phép chuyển đổi bất kỳ số thực nào, kể cả các số âm. Chúng ta nhận được hàm convertReal()

Hàm chuyển đổi convertReal() trong 120base.c

```
void convertReal(char *rslt, double n,unsigned char base, unsigned
    char cnt)
//Biến đổi một số thực n thanh hệ đếm cơ số cơ bản
{ double integer, fraction;
  // Tìm dấu của số
  if (n < 0) {
    *rslt++ = '-';
    n = -n;
  }
//chia phần nguyên và phần thập phân
fraction = modf(n, &integer);
// Chuyển đổi phần nguyên
convert(rslt, (unsigned long)integer, base);
// Đặt dấu chấm thập phân theo dấu phẩy
if ('\0' == *rslt) *rslt++ = '0';
  else rslt += strlen(rslt);
  *rslt++ = '.';
// Chuyển phần thập phân
convertLessThan1(rslt, fraction, base, cnt);
  if ('\0' == *rslt) {
  *rslt++ = '0';
```

```
    *rslt = '\0';
    }
}
```

Bài tập

▷ **1.45.** Tìm kí hiệu thập phân của số $126_{(8)}; 10101_{(2)}; 3F2B_{(16)}; 3CB_{(14)}$.

▷ **1.46.** Tìm kí hiệu thập phân của số $0,233_{(8)}; 0,01_{(2)}; 0,34_{(16)}; 0,2A_{(14)}$.

▷ **1.47.** Tìm kí hiệu thập phân của số $126,233_{(8)}; 10101.01_{(2)}; 3F2B,34_{(16)}; 3CB,2A_{(14)}$.

▷ **1.48.** Để chứng minh thuật toán được đề xuất để chuyển đổi từ p-cơ số sang thập phân.

1.5.2. Chuyển hệ cơ số p vào cơ số 10. Sơ đồ Horner

Việc chuyển đổi từ chữ số p sang hệ thống số thập phân được rút gọn để tính giá trị của đa thức từ ký hiệu đa thức của A, vì tất cả các phép toán số học được thực hiện trong hệ thống số thập phân. Ví dụ, đối với $12734_{(8)}$, chúng ta nhận được như sau:

$$12734_{(8)} = 1.8^4 + 2.8^3 + 7.8^2 + 3.8 + 4 = 5596_{(10)}$$

Lưu ý rằng chúng ta cần 10 phép nhân để tính toán. Chúng ta có thể giảm đáng kể con số này nếu chúng ta giữ các lũy thừa đã được tính ở mức 8, thay vì đếm lại chúng mỗi lần. Tuy nhiên, có một phương pháp khác tổng quát hơn cho phép chúng ta tính giá trị của đa thức bậc n với không quá n phép nhân *n công thức của Horner*. Ý tưởng là sử dụng (2) thay vì loại (1) trong phép tính.

(1) $P_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$

(2) $P_n(x) = a_n + x(a_{n-1} + x(a_{n-2} + \cdots + x(a_2 + x(a_1 + xa_0))...))$
- công thức của Horner

Một ví dụ về triển khai cách tiếp cận này là hàm calculate():

Hàm tính toán calculate() trong 120base.c

```
char getValue(char c) //Trả về giá trị của ký hiệu
  { return (c >= '0' && c <= '9') ? c - '0' : c - 'A' + 10; }
unsigned long calculate(const char *numb, unsigned char base)
//Tìm giá trị thập phân của số numb, được cho trong hệ đếm với cơ s
    ố numb>=0
{ unsigned long result;
  for (result = 0; '\0' != *numb; numb++)
  result = result*base + getValue(*numb);
  return result;
}
```

Theo cách tương tự, chúng ta có thể nhận ra phép biến đổi một số không âm nhỏ hơn 1. Trường hợp này không khác nhiều so với trường hợp biến đổi một số tự nhiên. Sự khác biệt duy nhất là ở đây đa thức có bậc âm của p.

Hàm chuyển đổi calculateLessThan1() trong 120base.c cho số âm

```
double calculateLessThan1(const char *numb, unsigned char base)
//Tìm giá trị thập phân của số numb (0<numb<1), cho trong hệ số đế
    m với cơ số
{ const char *end;
  double result;
  for (end=numb+strlen(numb)-1, result=0.0; end>= numb; end--)
  result = (result + getValue(*end)) / base;
  return result;
}
```

Cuối cùng, vẫn tổ hợp hai trường hợp để có được một hàm chuyển đổi một số thực tùy ý với một dấu:

Hàm tổ hợp calculateReal() trong 120base.c cho số âm

```
double calculateReal(char *numb, unsigned char base)
//Tìm giá trị thập phân của số thực numb, cho trong hệ đếm với cơ s
    ố cơ sở
{ char *pointPos;
  char minus;
  double result;
  //Kiểm tra dấu âm
  if ('-' == *numb) {
```

```
    minus = -1;
    numb++;
    }
    else
    minus = 1;
if (NULL == (pointPos = strchr(numb, '.')))
    return calculate(numb, base); //Không có phần thập phân
//Tính phần nguyên
    *pointPos = '\0';
    result  = calculate(numb, base);
    *pointPos = '.';
    //thêm vào phần thập phân
    result  += calculateLessThan1(pointPos+1, base);
    return minus*result;
}
```

Bài tập

▷ **1.49.** Tìm kí hiệu thập phân của số $126_{(8)}; 10101_{(2)}; 3F2B_{(16)}; 3CB_{(14)}$.

▷ **1.50.** Tìm kí hiệu thập phân của số $0,233_{(8)}; 0,01_{(2)}; 0,34_{(16)}; 0,2A_{(14)}$.

▷ **1.51.** Tìm kí hiệu thập phân của số $126,233_{(8)}; 10101.01_{(2)}; 3F2B,34_{(16)}; 3CB,2A_{(14)}$.

▷ **1.52.** Để chứng minh thuật toán được đề xuất để chuyển đổi từ hệ thống số p sang số thập phân.

1.6. Chữ số la mã

Nhiều hệ thống được thảo luận ở trên, mặc dù khác nhau về cơ sở của chúng, đều thuộc cùng một lớp: vị trí. Đối với họ, cùng một hình có các giá trị khác nhau tùy thuộc vào vị trí của nó. Tuy nhiên, có những hệ thống số khác, được gọi là không có vị trí, trong đó giá trị của các chữ số riêng lẻ là cố định và không phụ thuộc vào vị trí của nó. Một ví dụ kinh điển về vấn đề này là hệ thống chữ số La Mã. Nó sử dụng 7 chữ cái Latinh viết hoa để biểu thị các số tự nhiên

trong phạm vi [1; 3999] (các giá trị tương ứng của chúng được cho trong ngoặc): I (1), V (5), X (10), L (50), C (100), D (500) và M (1000). Ví dụ, số 1989 được ghi là MCMLXXXIX.

1.6.1. Biểu diễn số thập phân thành chữ số La mã

Chúng ta sẽ không đi sâu chi tiết vào các thuộc tính của hệ thống số La Mã. Chúng ta sẽ chỉ lưu ý rằng khi một hình nhỏ hơn đứng trước một hình lớn hơn, giá trị của nó sẽ bị trừ đi giá trị của hình lớn hơn. Nếu không, hãy thêm:

$$XI = 10 + 1 = 11$$
$$IX = 10 - 1 = 9$$

Mỗi chữ số có thể được theo sau bởi bất kỳ chữ số nào nhỏ hơn. Tuy nhiên, có những giới hạn đối với số lượng chữ số liên tiếp của cùng một giá trị, cũng như chữ số nhỏ hơn có thể đứng trước chữ số lớn hơn nào. Điều này được thấy rõ trong Bảng 1.4.

1(I)	2(II)	3(III)	4(IV)	5(V)	6(VI)	7(VII)	8(VIII)	9(IX)
10(X)	20(XX)	30(XXX)	40(XL)	50(L)	60(LX)	70(LXX)	80(LXXX)	90(XC)
100(C)	200(CC)	300(CCC)	400(CD)	500(D)	600(DC)	700(DCC)	800(DCCC)	900(CM)

Bảng 1.4. Ghi lại một số số bằng chữ số La mã.

Chương trình 1.17. Chuyển số thập phân thành La Mã và ngược lại (121rom2dec.c)

```
#define MAX_ROMAN_LEN 20
#include <stdio.h>
#include <string.h>
const char *roman1_9[] = {"", "A", "AA", "AAA", "AB", "B", "BA", "
    BAA", "BAAA", "AC"};
const char *romanDigits[] = {"IVX", "XLC", "CDM", "M" };
const char *roman2test = "MCMLXXXIX";
void getRomanDigit(char *rslt, char x, unsigned char power)
{  const char *pch;
   for (pch = roman1_9[x]; '\0' != *pch; pch++)
   * rslt ++ = romanDigits[power][*pch - 'A'];
   * rslt  = '\0';
}
```

```
//Chuyển số cơ số 10  thành số La Mã
char *decimal2Roman(char *rslt, unsigned x)
{ unsigned char power;
  char buf[10];
  char oldRslt[MAX_ROMAN_LEN];
  for (*rslt  = '\0', power = 0; x > 0; power++, x /= 10) {
  getRomanDigit(buf, (char)(x % 10), power);
  strcpy(oldRslt, rslt );
  strcpy(rslt,buf);
  strcat(rslt,oldRslt);
  }
  return rslt;
}
//Chuyển số La Mã  thành số cơ số 10
unsigned roman2Decimal(const char *roman, char *error)
{ unsigned rslt, value, old;
  const char *saveRoman = roman;
  char buf[MAX_ROMAN_LEN];
  old = 1000; rslt  = 0;
  while ('\0' != *roman) {
  switch (*roman++) {
    case 'I': value =    1; break;
    case 'V': value =    5; break;
    case 'X': value =   10; break;
    case 'L': value =   50; break;
    case 'C': value =  100; break;
    case 'D': value =  500; break;
    case 'M': value = 1000; break;
    default:
    *error = 1;
    return (unsigned)(-1);
  }
   rslt  += value;
  if (value > old)
     rslt  -= 2*old;
  old = value;
  }
  return (*error = strcmp(saveRoman,decimal2Roman(buf,rslt)))
    ? (unsigned)(-1) :  rslt ;
}
```

```
int main(void) {
  unsigned decimal;
  char error;
  decimal = roman2Decimal(roman2test,&error);
  if (error) printf("chuong trinh bi loi!");
  else printf("Ket qua chuyen doi %u", decimal);
  return 0;
}
```

Bài tập

▷ **1.53.** Viết bằng số La mã: 10; 19; 159; 763; 1991; 1979; 1997; 2002

▷ **1.54.** Viết dưới dạng số La mã: $0; -10; 0,28; 3,14; 1/7$.

▷ **1.55.** Chứng minh thuật toán được đề xuất để chuyển đổi một số thập phân thành một hệ thống chữ số La Mã.

1.6.2. Chuyển đổi chữ số La Mã sang số thập phân

Ở đây mọi thứ đơn giản hơn. Lần này chúng ta sẽ xem xét các con số từ trái sang phải, chúng ta sẽ tính toán giá trị thập phân của chúng, và sau đó chúng ta sẽ tích lũy nó một số tiền. Nếu nó chỉ ra rằng chữ số trước đó có giá trị thập phân thấp hơn giá trị hiện tại, điều này có nghĩa là chữ số trước đó đáng lẽ không được thêm vào mà phải bị trừ đi. Vì trong trường hợp này, chúng ta đã cộng nó một lần, chúng ta nên trừ nó đi hai lần:

```
if (value> old) rslt -- 2 * old;
```

Ví dụ, đối với số 19 (XIX), con số I phải được trừ đi tổng, trong khi đối với số 21 (XXI) - nó phải được thêm vào.

Một vấn đề khác nảy sinh: làm thế nào để kiểm tra xem dãy số La Mã được cho dưới dạng tham số của roman2Decimal() có phải là một số La Mã chính xác hay không? Việc kiểm tra này rất phức tạp, nhưng chúng ta có thể làm cho mọi thứ đơn giản hơn nhiều. Với mục đích này, chỉ cần thực hiện phép biến đổi nghịch đảo và so sánh kết quả thu được với số ban đầu là đủ. Chi tiết có thể được nhìn thấy từ việc thực hiện được đề xuất.

Hãy xem nội dung hàm số

```
unsigned roman2Decimal(const char *roman, char *error)
```

trong chương trình trên.

Bài tập

▷ **1.56.** Viết các chữ số La Mã trong hệ thập phân: DCLXXXIV, DCCLXIV, LX, LXX, LXXX, XL, XXL, XXXL.

▷ **1.57.** Để chứng minh thuật toán được đề xuất để chuyển đổi một số La Mã thành một hệ thống số thập phân.

▷ **1.58.** Viết chương trình kiểm tra dãy số La Mã có phải là chữ số La Mã viết đúng mà không qua hệ thống số thập phân hay không. Để làm được điều này, hãy quan sát kỹ các mẫu trong Bảng 1.4.

1.7. Hồi quy và lặp lại

Một câu nổi tiếng trong lập trình dân gian là: "Để xác định khái niệm đệ quy, trước tiên chúng ta phải xác định khái niệm đệ quy." Có rất nhiều "định nghĩa" như vậy trong thế giới UNIX (ví dụ, GNU là viết tắt của GNU không là UNIX, WINE là viết tắt của WINE không là Emulator, v.v.).

Định nghĩa 1.17. Một đối tượng được gọi là *đệ quy* nếu nó được chứa trong chính nó hoặc được định nghĩa bởi chính nó.

Trong tin học máy tính, đệ quy là một trong những kỹ thuật lập trình mạnh mẽ nhất: nó định nghĩa các thuật toán một cách trang nhã, tạo ra các cấu trúc dữ liệu thuận tiện và linh hoạt.

Trong nhiều trường hợp, việc sử dụng đệ quy có thể dẫn đến các thuật toán không hiệu quả. Mục đích của đoạn này, ngoài việc giới thiệu đệ quy như một cách tiếp cận thuật toán cơ bản, sẽ là khám phá câu hỏi khi nào ứng dụng của nó hữu ích trong thực tế.

Phương tiện của biểu thức đệ quy trong chương trình C là các hàm. Nếu trong phần thân của một hàm P đã cho có tham chiếu đến chính nó, chúng ta nói rằng nó trực tiếp đệ quy. Một kịch bản "phức tạp hơn" cũng có thể xảy ra: hàm P_1 chuyển thành hàm P_2,

P_2 thành P_3, ..., P_n thành P_1. Trong trường hợp này chúng ta nói rằng P_1, cũng như P_2, P_3, ..., P_n, là các hàm đệ quy gián tiếp (gián tiếp).

Khi lập trình các hàm đệ quy, một số điều kiện quan trọng phải được xem xét:

1. bài toán chúng ta đang xem xét nên được chia thành các bài toán con mà (đệ quy) cùng một thuật toán có thể được áp dụng. Kết hợp các giải pháp của tất cả các bài toán phụ sẽ dẫn đến một giải pháp của bài toán ban đầu.
2. Thực hiện một thuật toán đệ quy, chúng ta phải chắc chắn rằng sau một số bước hữu hạn, chúng ta sẽ đạt được một kết quả cụ thể, tức là phải có một số hữu hạn các trường hợp đơn giản (ít nhất một) mà lời giải của chúng có thể được tìm thấy trực tiếp. Thông thường người ta gọi chúng là đáy của đệ quy.
3. Tất cả các bài toán phụ của bài toán phải "khao khát" một trong những trường hợp đơn giản này, tức là sau một số lần gọi đệ quy hữu hạn để đạt đến đáy của đệ quy (Tương tự, trong các công thức truy hồi, cũng như trong suy luận quy nạp, cần phải có một cơ số.).

Một vài ví dụ đầu tiên về các hàm C đệ quy mà chúng ta sẽ trình bày sẽ chứng minh việc tính toán các hàm toán học được xác định tuần hoàn.

1.7.1. Tính giai thừa

Định nghĩa về giai thừa, cũng như phép tính lặp lại của hàm, chúng ta đã xem xét trong 1.1.1. Để thực hiện một hàm tính giai thừa đệ quy, chúng ta phải chắc chắn rằng hai điều kiện chính được đáp ứng:

- Dưới cùng của đệ quy là trường hợp đơn giản $n = 0$ - khi đó giá trị của hàm là 1.
- Trong tất cả các trường hợp khác, chúng ta giải quyết bài toán con cho $n - 1$ và nhân kết quả với n. Do đó tham số đệ quy giảm đơn điệu và sẽ đạt đến đáy của đệ quy sau một số bước hữu hạn (giữa n và 0 có một số lượng tự nhiên hữu hạn).

Sau đây là một hàm của C, tính toán đệ quy $n!$:

Chương trình 1.18. Tính giai thừa (121factrec.c)

```
#include <stdio.h>
const unsigned n = 6;
unsigned long fact(unsigned i)
{ if (i < 2) return 1;
return i * fact(i - 1);
}
int main(void) {
printf("%u! = %lu \n", n, fact(n));
return 0;
}
```

Theo cách tương tự, chúng ta có thể tính toán một cách đệ quy tổng của n số tự nhiên đầu tiên (trực tiếp bằng định nghĩa truy hồi từ 1.1.1.):

Tính tổng đệ quy

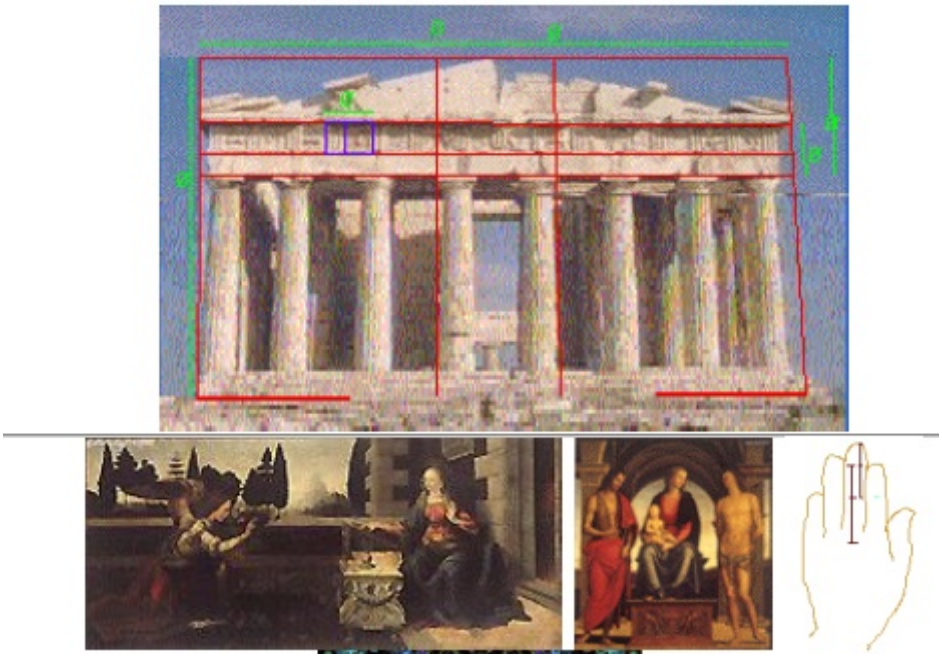
```
unsigned long sum(unsigned n){
if (0 == n) return 0;
else return n + sum(n - 1);
}
```

Có thể nhận thấy một số điều trong các triển khai trên: khi một hàm toán học được xác định lặp lại được đưa ra, việc thực hiện một hàm C đệ quy tương ứng không khó, trong khi thuật toán lặp (tuần tự) để giải trong một số trường hợp, đặc biệt là khi hàm nhiều hơn phức tạp, không phải là quá rõ ràng. (xem 1.2.2.)

Mặt khác, bộ nhớ bị tiêu tốn nhiều hơn. Điều này là do mỗi lần gọi đệ quy đến hàm trong ngăn xếp hệ thống sẽ cấp phát bộ nhớ mới cho các đối số và biến cục bộ, cũng như cho các kết quả được trả về bởi hàm. Trong các ví dụ trên, n sẽ là bắt buộc. sizeof (không dấu) byte bộ nhớ cho n cuộc gọi đệ quy. Để so sánh: chương trình tìm n! từ 1.1.1. đã sử dụng một biến duy nhất trong đó kết quả được tích lũy, tức là bộ nhớ kém hơn nhiều lần. Tuy nhiên, trong những ví dụ này, bộ nhớ không phải là một yếu tố quan trọng, vì sự phát triển nhanh chóng của $n!$ chúng ta sẽ nhận được tràn số nguyên dài không dấu chứ không phải là thiếu bộ nhớ.

1.7.2. Dãy Phibonacci

Số Fibonacci là một trong những ví dụ yêu thích của một số cuốn sách và sách hướng dẫn về thuật toán cho sự sang trọng của đệ quy như một kỹ thuật lập trình. Vì lý do này, đôi khi chúng ta sẽ không cưỡng lại sự cám dỗ để đi sâu vào chi tiết hơn về lịch sử, bản chất và tính chất của chúng, mà chúng ta tin rằng sẽ không khiến người đọc khó chịu.



Hình 1.9. Số Fibonacci trong kiến trúc và nghệ thuật

Bất cứ ai đã từng tham gia vào lập trình ít nhất một chút đều không tránh khỏi việc gặp phải những con số Fibonacci. Tại sao? Có một loạt các thuật toán và bài toán lập trình, dường như không liên quan theo bất kỳ cách nào, dựa trên những con số này. Sẽ không quá lời khi nói rằng sự hiện diện và vai trò của chúng trong khoa học máy tính, toán học và tự nhiên nói chung là thực sự gây sốc. Chỉ cần biết cách quan sát, người ta có thể tìm thấy các số Fibonacci xung quanh mình: từ cách sắp xếp các đàn chim đang bay, qua các hình nón và bài thơ, bánh hướng dương và các bản giao hưởng,

nghệ thuật cổ đại (xem Hình 1.9) Và máy tính hiện đại, đến tận hệ thống năng lượng mặt trời và các sàn giao dịch chứng khoán.

Chính xác thì những con số nổi tiếng này là gì? Đây là những phần tử của số sau:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...,$$

Mỗi phần tử của chuỗi được lấy là tổng của hai phần trước, hai phần đầu theo định nghĩa lần lượt là 0 và 1. các công thức hợp lệ:

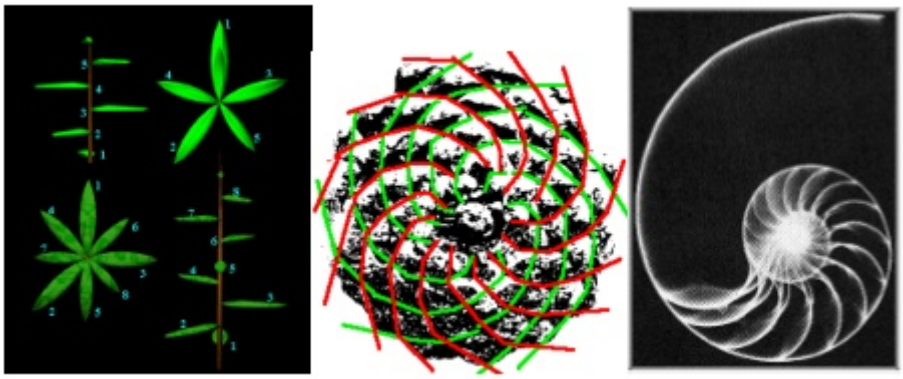
$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2} \end{aligned}$$

Hình 1.10. Leonardo Fibonacci.

Nhà nghiên cứu đầu tiên của dãy số trên là Leonardo Pisano (Leonardo of Pisa), hay còn được biết đến với cái tên Leonardo Fibonacci (Filius Bonaccii, tức là con trai của Bonacio), người vào năm 1202 trong cuốn sách Liber Abacci (Sách đếm) đã đưa ra bài toán nổi tiếng của mình đối với thỏ. . Bài toán là tìm số thỏ sẽ có được từ một cặp trong một năm với các điều kiện sau:

- mỗi cặp thỏ đậu quả sẽ tăng thêm hai con mỗi tháng;
- thỏ mới kết trái khi được một tháng tuổi;
- Thỏ không bao giờ chết.

Như vậy, trong một tháng, chúng ta sẽ có hai cặp thỏ, sau đó là 3 cặp, tháng sau chúng ta sẽ có 5 cặp (một cặp thỏ mới ban đầu và một - của cặp đã nhận được trong tháng đầu tiên), vào tháng sau. chúng ta sẽ có tổng cộng 8 cặp đôi, v.v. Rõ ràng là quá trình được mô tả có thể diễn ra vô thời hạn. (xem Hình 1.12).



Hình 1.11. Số Fibonacci trong tự nhiên.

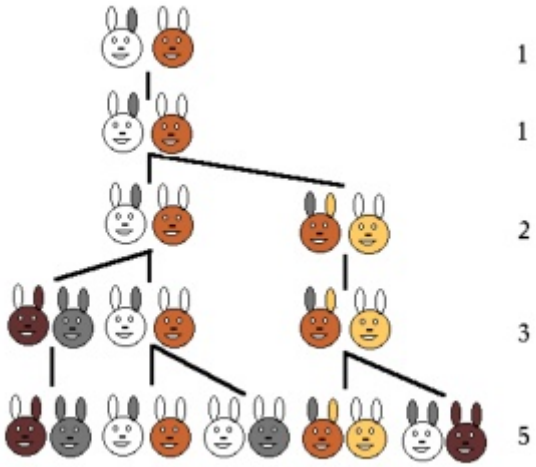
Sau đó vào năm 1611, Kepler, không biết về nghiên cứu của Fibonacci, đã khám phá lại một số nghiên cứu của mình về sự sắp xếp các lá của một số cây trên thân cây (Hình 1.11). Nói chung, như đã đề cập ở trên, số Fibonacci có bản chất cực kỳ phổ biến, rất có thể là kết quả của các quá trình tương tự như của bài toán Fibonacci đối với thỏ.

Mối quan hệ giữa số Fibonacci và thuật toán là gì? Mối liên hệ đầu tiên là chính Leonardo Fibonacci, người đã siêng năng nghiên cứu các tác phẩm của al-Khwarizmi (từ tên mà thuật toán từ bắt nguồn, như chúng ta đã đề cập trong chương giới thiệu). Sau đó, vào năm 1845, Lame sử dụng dãy Fibonacci trong quá trình nghiên cứu thuật toán Euclid nổi tiếng (xem 1.2.3) để tìm ước số chung lớn nhất. Kể từ đó, số Fibonacci dần dần bắt đầu chiếm vị trí xứng đáng trong việc phát triển và nghiên cứu các thuật toán khác nhau. Mối liên hệ giữa dãy Fibonacci và số sau rất thú vị

$$\phi = \frac{1+\sqrt{5}}{2} = 1,61803$$

Số của nó có một lịch sử rất thú vị và đã được mọi người biết đến từ thời cổ đại. Euclid gọi nó là tỷ số của *số hữu hạn so với giá trị trung bình* và định nghĩa nó là tỷ số của một cặp số A và B trong đó tỷ lệ có giá trị:

$$\frac{A}{B} = \frac{A+B}{A}(= \phi)$$

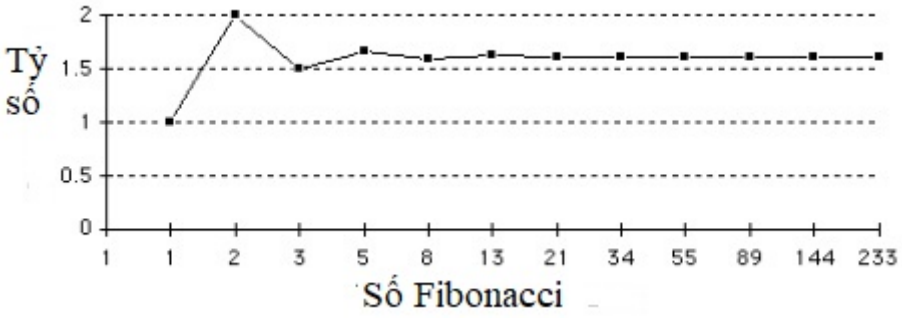


Hình 1.12. Bài toán những con thỏ.

Sau đó, vào thời kỳ Phục hưng, số ϕ được coi là một tỷ lệ thần thánh, và vào thế kỷ 19, nó có tên gọi cuối cùng như ngày nay: tỷ lệ vàng. Ký hiệu được chấp nhận ϕ xuất phát từ chữ cái đầu tiên trong tên của nhà điêu khắc Hy Lạp cổ đại Phidias, người thường sử dụng tỷ lệ này trong các tác phẩm điêu khắc của mình. Không khó để nhận thấy sự tương đồng giữa các định nghĩa về số Fibonacci và tỷ lệ vàng được đề xuất ở trên. Nó chỉ ra rằng tỷ lệ của hai số Fibonacci liên tiếp có xu hướng theo tỷ lệ vàng ϕ, và điều này đôi khi được xác định là giới hạn của tỷ lệ của hai số Fibonacci liên tiếp. (xem Hình1.12.)

Lưu ý: Trước khi chúng ta chuyển sang các thuật toán tìm số Fibonacci, cần lưu ý rằng đôi khi có một định nghĩa khác, theo đó phần tử 0 của dãy Fibonacci là 1, không phải 0. Tất nhiên, định nghĩa sau không quá quan trọng. và người đọc, nếu cần, có thể dễ dàng

thay đổi các triển khai chương trình sau theo định nghĩa được ưu tiên.



Hình 1.13. Mối quan hệ của hai số Fibonacci liên tiếp.

Có lẽ cách thực hiện tự nhiên nhất của một hàm tìm số Fibonacci thứ n có được trực tiếp từ định nghĩa lặp lại được đưa ra ở trên:

Chương trình 1.19. Tính Fibonacci (123fibrec.c)

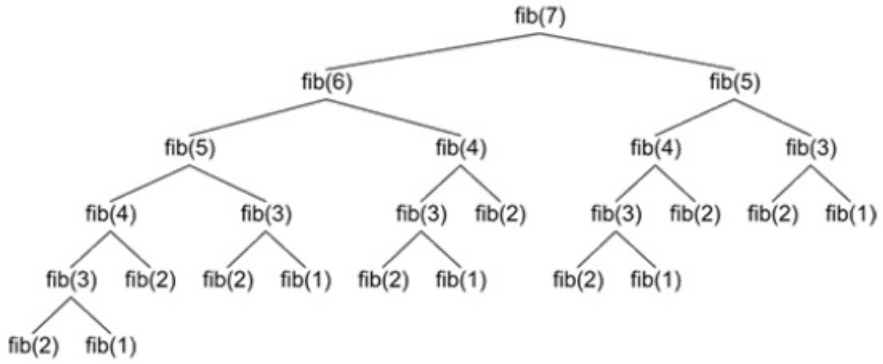
```
 // da oprawq w teksta
// fib(7) dawaha razlichno - da checkna koe e po definiciqta!!!
#include <stdio.h>
const unsigned n = 7;

unsigned long fib(unsigned n)
{ if (n < 2) return 1;
    else return fib(n - 1) + fib(n - 2);
}

int main() {
  printf("fib(%u) = %lu\n", n, fib(n));
  return 0;
}
```

Mặc dù đủ đơn giản và có vẻ tự nhiên, nhưng nhận thức như vậy là cực kỳ kém hiệu quả. Người đọc có thể thử hàm trên, ví dụ với $n = 40$. Mỗi nghịch đảo đệ quy, ngoại trừ những nghịch đảo tầm thường (đối với n = 0 và 1), dẫn đến hai nghịch đảo nữa. Do đó cây bài toán con phát triển theo cấp số nhân. Đồng thời, nó hoàn

toàn vô nghĩa, vì hầu hết thời gian các giá trị đã được tính toán của hàm đều được tính toán (xem Hình **??**).



Hình 1.14. Cây gọi hàm đệ quy.

Số Fibonacci là một ví dụ cổ điển trong đó việc sử dụng đệ quy không thành công. Vấn đề bắt nguồn từ thực tế là các tính toán hoàn toàn không cần thiết được thực hiện, tức là nhiều thành viên của dòng được tính vài lần. Ví dụ, khi tính F_10, chúng ta sẽ tính F_8 hai lần - một lần khi tính $F_10 = F_9 + F_8$ và sau đó - cho $F_9 = F_8 + F_7$. Ở các cấp độ đệ quy sâu hơn, tình hình thậm chí còn tồi tệ hơn. Ở đây, tất nhiên, đệ quy có thể cho kết quả tốt hơn đáng kể với chi phí của bộ nhớ bổ sung cần thiết. Phương pháp (được gọi là ghi nhớ) được thảo luận trong Chương 8 - Tối ưu hóa động.

Không khó để thấy rằng chúng ta có thể không sử dụng đệ quy bằng cách tính toán các số Fibonacci tuần tự: bắt đầu với số đầu tiên, chúng ta chỉ giữ lại hai phần tử được tính cuối cùng của chuỗi, vì chỉ chúng ta mới cần chúng để lấy phần tử tiếp theo. Do đó, chúng ta thu được thuật toán lặp sau:

Thuật toán lặp

```
unsigned long fibIter(unsigned n)
{ unsigned long fn = 1, fn_1 = 0,  fn_2;
  while (n--) {
    fn_2 = fn_1;
    fn_1 = fn;
    fn  = fn_1 + fn_2;
    }
```

```
  return fn_1;
}
```

Không khó để thấy rằng chúng ta không thực sự cần ba biến và chỉ có thể giải quyết hai biến như thế này:

Chương trình 1.20. Tính Fibonacci không đệ quy (124fibiter.c)

```
 //n->const, FibIter -> fibIter
#include <stdio.h>
const unsigned n = 7;

unsigned long fibIter(unsigned n)
{ unsigned long f1 = 0, f2 = 1;
  unsigned i = 0;
  while (i < n) {
    f2 = f1 + f2;
    f1 = f2 - f1;
    i++;
  }
  return f2;
}

int main() {
  printf("fib(%u) = %lu\n", n, fibIter (n));
  return 0;
}
```

Trong trường hợp chúng ta đang tìm n số Fibonacci đầu tiên, cách tốt nhất là sử dụng tùy chọn cuối cùng với kết quả trung gian. Nhưng chúng ta đang tìm số Fibonacci thứ n như thế nào?

Có một công thức không lặp lại cho phần tử chung của chuỗi Fibonacci, được gọi là công thức Moaver (xem 1.4.8):

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1=\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

Thoạt nhìn, công thức của Moaver có vẻ hiệu quả hơn so với phiên bản lặp lại - ở đây, thành viên thứ n của chuỗi dường như được tìm thấy trực tiếp. Thật không may, việc sử dụng nó gắn liền

với việc làm việc với các số thực, yêu cầu root và thao tác mở rộng không kém phần khó khăn, điều này làm thay đổi mọi thứ. Tuy nhiên, đối với công thức Moaver n đủ lớn, nó được ưu tiên hơn, đặc biệt nếu sử dụng phương pháp phân loại nhanh để phân loại (xem 7.3., [Nakov-1998c]). Chúng ta sẽ lưu ý rằng quá trình root của 5 có thể được thực hiện trước và kết quả có thể được đặt thành một hằng số trong chương trình. Điều này sẽ tránh tính toán nó trong quá trình thực thi, đây là một quá trình lặp lại chậm vì nó xảy ra với các hàng.

Chúng ta sẽ thảo luận về các thuật toán để tìm số Fibonacci thứ n trong Chương 8 (xem 8.1.), Ngoài việc áp dụng các kỹ thuật cụ thể cho *Tối ưu hóa động*, chúng ta sẽ xem cách chúng được kết hợp với công thức Moaver và mở rộng quy mô nhanh.

Bài tập

▷ **1.59.** Chứng tỏ rằng tỷ lệ của hai số Fibonacci liên tiếp có xu hướng theo tỷ lệ vàng.

▷ **1.60.** Để chứng minh sự tương đương của hai thực nghiệm lặp đi lặp lại của một hàm để tìm số Fibonacci thứ n: với ba và với hai biến.

1.7.3. Ước số chung lớn nhất và thuật toán Euclid

Ước chung lớn nhất (GCD) của hai số là một ví dụ điển hình khác về việc minh họa đệ quy như một kỹ thuật lập trình.

Định nghĩa 1.18. Hai số tự nhiên a và b đã cho. Ta nói rằng d là ước chung lớn nhất của a và b nếu nó là số tự nhiên lớn nhất chia cả a và b. Được viết chính thức, điều sau có nghĩa là đáp ứng hai điều kiện sau:

1) $d|a, d|b$

2) Nếu $d_1|a$ và $d_1|b$ thì $d_1|d$.

Trong định nghĩa, chúng ta đã sử dụng ký hiệu chia hết: $d|a$ có nghĩa là d chia một số nguyên a không có dư. Hai điều kiện của định nghĩa cũng có thể được hiểu như sau: Mỗi ước chung d_1 của a và b chia hết và d.

Chúng ta sẽ ký hiệu ước số chung lớn nhất của a và b bằng $GCD(a,b)$ hoặc đơn giản là (a,b).

Ví dụ: $(12,8) = (8,12) = (8,4) = 4, (1,10) = (7,10) = 1$.

Định nghĩa 1.19. Nếu GCD của hai số nguyên dương bằng 1 thì các số đó được gọi là *nguyên tố cùng nhau*.

Công thức sau đây hợp lệ cho ước chung lớn nhất của hơn hai số:

$$GCD(a_1, a_2, ..., a_n) = GCD(a_1, a_2, ..., a_{n-1}), a_n)$$

Chúng ta sẽ xem xét hai thuật toán để tìm $GCD(a,b)$, được đặt tên theo Euclid. Đầu tiên trong số này được gọi là thuật toán trừ Euclide:

1) Nếu $a > b$, thực hiện 4), nếu không thực hiện 2).

2) Nếu $a = b$ thì sau đó chúng ta đã tìm được $GCD(a,b)$ - đây là giá trị của b và chúng ta kết thúc. Nếu $a \neq b$ ta thực hiện 3).

3) Gán $b = b - a$ và quay lại bước 1).

4) Gán $a = a - b$ và quay lại bước 1).

Chúng ta sẽ không đi sâu vào thuật toán này chi tiết hơn, vì thuật toán tiếp theo chúng ta sẽ xem xét là hiệu quả hơn nhiều. Nó được gọi là *thuật toán Euclid với phép chia* (nó đã được đề cập trong phần giới thiệu của cuốn sách):

Thuật toán Euclid với phép chia

Chia số nguyên a cho b được q_1 và số dư r_1. Sau đó chia b cho phần dư thu được, rồi chia r_1 cho phần dư r_2, v.v., cho đến khi phần dư bằng không:

$$\begin{aligned} a &= q_1.b + r_1 \\ b &= q_2.r_1 + r_2 \\ r_1 &= q_3.r_2 + r_3 \\ &..... \\ r_{k-1} &= q_{k+1}.r_k + r_{k+1}, \text{ vì } r_{k+1} = 0 \end{aligned}$$

Phần dư khác không cuối cùng r_k sẽ là GCD cần thiết. Thuật toán cũng có một biến thể đệ quy tương ứng dựa trên thuộc tính

sau của GCD:

$$GCD(a,b) = GCD(b, a\%b)$$

Chương trình 1.21. Tìm ước số chung lớn nhất không đệ quy (125gcditer.c)

```
#include <stdio.h>

const unsigned a = 28;
const unsigned b = 49;

unsigned gcd(unsigned a, unsigned b)
{ unsigned swap;
  while (b > 0) {
    swap = b;
    b = a % b;
    a = swap;
  }
  return a;
}

int main(void) {
  printf("NOD(%u,%u) = %u\n", a, b, gcd(a, b));
  return 0;
}
```

Chương trình 1.22. Tìm ước số chung lớn nhất đệ quy (126gcdrec.c)

```
#include <stdio.h>
const unsigned a = 28;
const unsigned b = 49;

unsigned gcd(unsigned a, unsigned b)
{ return (b == 0) ? a : gcd(b, a % b);
}

int main(void) {
  printf("NOD(%u,%u) = %u\n", a, b, gcd(a, b));
  return 0;
}
```

Có thể thấy rằng hàm đệ quy ngắn và chặt chẽ hơn rất nhiều. Tuy nhiên, bộ nhớ được tiêu thụ bởi nó nhiều hơn: đối với mỗi lệnh gọi đệ quy trong ngăn xếp, bộ nhớ được cấp phát cho các tham số chính thức, cũng như cho kết quả được trả về bởi hàm.

Thuật toán Euclid nâng cao

Đôi khi, chẳng hạn như trong số học mô-đun, thuật toán Euclide được sử dụng để tìm hai số nguyên bổ sung nhân x và $y(x, y \in \mathbb{Z})$, sao cho:

$$d = GCD(a, b) = ax + by$$

Bài tập

▷ **1.61.** 1. Để tìm GCD của: $(10, 5)$; (5.10); $(15, 25)$; $(25, 15)$; $(7, 8, 9)$; $(3, 6, 9)$; $(158, 128, 256)$; $(64, 28, 72, 18)$.

2. Viết chương trình rút gọn phân số thường gặp. Ví dụ, ở đầu vào $10/15$ để cung cấp cho $2/3$.
3. Chứng minh rằng $(a_1, a_2, ..., a_n) = (a_1, a_2, ..., a_{n-1}), a_n)$.
4. Để thực hiện thuật toán trừ Euclide.
5. Chứng minh rằng $(a, b) = (b, a\%b)$.
6. Thuật toán và chương trình phải được sửa đổi như thế nào để đối với các số nguyên a và b đã cho, ngoài GCD, tìm được hai thừa số nguyên x và $y(x, y \in \mathbb{Z})$ mà $(a, b) = ax + by$?
7. Đề xuất và thực hiện một thuật toán khác để tìm GCD, dựa trên định lý cơ bản của số học. Để so sánh hiệu quả của nó với hiệu quả của việc thực hiện được đề xuất ở đây tại 2; Số 5; 100; 1000 số.
8. (Bài toán Poisson về ba bình) Cho ba bình có dung tích a, b và c, a, b và c là các số tự nhiên, $c > a > b$, $c \geq a + b - 1$, bình lớn nhất là đầy và hai cái còn lại trống rỗng. Đo d lít, với d là số tự nhiên và $0 < d \leq a$. Chỉ cho phép các tràn như vậy từ tàu này sang tàu khác, trong đó việc đổ đầy tối đa và/hoặc làm rỗng tối đa của một trong các tàu diễn ra.

1.7.4. Bội số chung nhỏ nhất

Định nghĩa 1.20. Hai số nguyên a và b đã cho. Số tự nhiên nhỏ nhất $d(d > 0)$ sao cho $a|d$ và $b|d$ được gọi là bội chung nhỏ nhất LCM) của a và b.

Chúng ta sẽ đánh dấu ít nhất bội chung của a và b bằng $LCM(a,b)$ hoặc đơn giản là $[a,b]$.

Ví dụ: $[6,15] = [15,6] = 30, [1,10] = 10, [5,10] = 10, [5,12] = 60$.

Khi chúng ta tìm LCM trên nhiều hơn hai số, có sự phụ thuộc tương tự như trong LCM:

$$LCM(a_1, a_2, ..., a_n) = LCM(LCM(a_1, a_2, ..., a_{n-1}), a_n)$$

Ít nhất một bội số chung có thể được tìm thấy bằng cách sử dụng mối quan hệ hiện có giữa nó và LCM, đó là:

$$LCM(a,b) = \frac{ab}{LCM(a,b)}$$

Dựa trên các tính chất trên, chúng ta sẽ thực hiện LCM của n số. Các số được cho trong mảng A[] và được cho cùng với số của chúng, dưới dạng tham số của hàm đệ quy lcm()

Chương trình 1.23. Bội chung nhỏ nhất (127lcm.c)

```
 #include <stdio.h>
const unsigned n = 4;
const unsigned A[] = { 10, 8, 5, 9 };
unsigned gcd(unsigned a, unsigned b)
{ return (0 == b) ? a : gcd(b, a % b);
}
unsigned lcm(unsigned a[], unsigned n)
{ unsigned b;
if (2 == n)
return(a[0] * a[1]) / (gcd(a[0], a[1]));
else {
b = lcm(a, n - 1);
return(a[n - 1] * b) / (gcd(a[n - 1], b));
}
}

 int main(void)
{ printf("%u\n", lcm(A, n));
return 0;
}
```

Kết quả thực hiện chương trình:

360

Bài tập

▷ **1.62.** Để tìm: $[10, 15]$; $[15, 10]$; $[7, 8, 9]$; $[3, 6, 9]$; $[158, 128, 256]$; $[64, 28, 72, 18]$.

▷ **1.63.** Chứng minh rằng $[a_1, a_2, ..., a_n] = [a_1, a_2, ..., a_{n-1}], a_n]$.

▷ **1.64.** Chứng minh rằng $[a, b] = \dfrac{ab}{(a, b)}$.

▷ **1.65.** Đề xuất và thực hiện một thuật toán khác để tìm NOC, dựa trên định lý cơ bản của số học. Để so sánh hiệu quả của nó với hiệu quả của việc thực hiện được đề xuất ở đây tại 2; Số 5; 100; 1000 số.

1.7.5. Trả lại giá trị từ đệ quy và dùng biến

Trong tất cả các hàm đệ quy C được trình bày cho đến nay, chúng ta chỉ thực hiện các thao tác trước lệnh gọi đệ quy. Tuy nhiên, trong thực tế, thường phải thực hiện các phép toán sau khi quay trở lại từ đệ quy.

Bài toán: In các chữ số của số tự nhiên n, đã cho trong hệ số thập phân, tuần tự từ chữ đầu tiên đến chữ số cuối cùng.

Ví dụ, với $n = 7892$, bạn sẽ cần in: $7, 8, 9, 2$.

Vấn đề trong bài toán này là xác định chữ số đầu tiên của dãy số. Mặt khác, việc tìm chữ số cuối cùng và loại bỏ nó có thể chỉ với hai phép toán: $n\%10$ và $n/10$. Chúng ta đưa ra thuật toán sau: ở mỗi bước, chúng ta tìm chữ số cuối cùng và viết nó vào một mảng số nguyên. Sau khi viết tất cả các chữ số trong mảng, chúng ta hiển thị nó theo thứ tự ngược lại:

Chương trình 1.24. Giá trị trả về của đệ quy (128print.c)

```
#include <stdio.h>
unsigned n = 7892;
int main(void)
{ unsigned dig[20], i, k = 0;
  while (n > 0) {
  dig[k] = n % 10;
```

```
  n = n / 10;
  k++;
  }
  for (i = k; i > 0; i--) printf("%u ", dig[i-1]);
  printf("\n");
  return 0;
}
```

Sử dụng đệ quy trong tác vụ này một lần nữa sẽ dẫn đến một giải pháp thanh lịch hơn nhiều. Chúng ta sẽ sử dụng thuộc tính sau: để in 7892, chúng ta cần in các chữ số của 789 và sau đó là chữ số 2. Nói chung, để in số n, chúng ta cần in (đệ quy!) Số $n/10$ và sau đó là chữ số cuối cùng $n\%10$. Do đó, sử dụng ngăn xếp đệ quy, chúng ta không cần phải giới thiệu một mảng một cách rõ ràng. Dựa trên kết luận cuối cùng, chúng ta sẽ biên dịch hàm tương ứng của C:

Chương trình 1.25. Giá trị trả về của đệ quy (129printrec.c)

```
#include <stdio.h>
unsigned n = 7892;
void printN(unsigned n)
{ if (n >= 10) printN(n / 10);
  printf("%u ", n % 10);
}

int main() {
  printN(n); printf("\n");
  return 0;
}
```

Phần dưới cùng của đệ quy, khi chúng ta ngừng gọi đệ quy, là $n \leq 9$. Các phép toán biến rất quan trọng trong đệ quy. Cho đến nay, chỉ có các biến-tham số và có thể là các biến cục bộ tạm thời đã thay đổi trong phần thân của một hàm đệ quy. Các phép toán với các biến toàn cục, trước và sau khi trả về từ đệ quy, rất quan trọng trong việc thiết kế các thuật toán đệ quy. Chúng ta sẽ minh họa sự thay thế các biến đối số bằng các biến toàn cục bằng một vài ví dụ: Tìm $n!$ nó có thể được thực hiện như sau bằng cách thay thế đối số của hàm bằng một biến toàn cục i (Ưu điểm của cách triển khai đệ quy này so với những cách được thảo luận cho đến nay là không có

bộ nhớ ngăn xếp nào được sử dụng):

Chương trình 1.26. Giá trị trả về của đệ quy (122factrec.c)

```
#include <stdio.h>
const unsigned n = 6;
unsigned i;
unsigned long fact(void)
{ if (i == 1) return 1;
  return --i * fact();
}

int main(void) {
  i = n + 1;
  printf("%u! = %lu \n", n, fact());
  return 0;
}
```

Chúng ta sẽ xem xét một ví dụ đơn giản khác: Đối với một số tự nhiên n cho trước in ra theo thứ tự tăng dần và giảm dần các số $10^k (1 \leq k \leq n)$. Ví dụ, đối với$n = 5$, hàng sẽ được in:

$$10, 100, 1000, 10000, 100000, 100000, 10000, 1000, 100, 10$$

Chúng ta sẽ giải quyết vấn đề một cách đệ quy theo ba cách khác nhau. Với chúng, chúng ta sẽ minh họa khả năng hoán đổi cho nhau của các biến toàn cục và các tham số biến:

Phương án 1. Tất cả các biến (ngoại trừ đầu vào n) là tham số của hàm đệ quy:

Chương trình 1.27. Giá trị trả về của đệ quy (130print1.c)

```
#include <stdio.h>
const unsigned n = 5;
void printRed(unsigned k, unsigned long result)
{ printf("%lu ", result);
  if (k < n) printRed(k + 1, result * 10);
  printf("%lu ", result);
}

int main(void) {
  printRed(1, 10);
```

```
  printf("\n");
  return 0;
}
```

Phương án 2. Tham số bộ đếm k có thể được xuất dưới dạng biến toàn cục:

Chương trình 1.28. Giá trị trả về của đệ quy (131print2.c)

```
#include <stdio.h>
const unsigned n = 5;
unsigned k = 0;
void printRed(unsigned long result)
{ k++;
  printf("%lu ", result);
  if (k < n) printRed(result * 10);
  printf("%lu ", result);
}
int main(void) {
  printRed(10);
  printf("\n");
  return 0;
}
```

Phương án 3. Nếu chúng ta sửa đổi kết quả kết quả trước và sau cuộc gọi đệ quy, nó cũng có thể được xuất dưới dạng biến toàn cục:

Chương trình 1.29. Giá trị trả về của đệ quy (132print3.c)

```
#include <stdio.h>
const unsigned n = 5;
unsigned long result = 1;
unsigned k = 0;
void printRed(void)
{ k++;
  result *= 10;
  printf("%lu ", result);
  if (k < n) printRed();
  printf("%lu ", result);
  result /= 10;
}
int main(void) {
  printRed();
```

```
    printf("\n");
    return 0;
}
```

Trong đoạn tiếp theo, khi xem xét các thuật toán tổ hợp cơ bản (cũng như nói chung trong tài liệu bên dưới), chúng ta sẽ tiếp tục sử dụng đệ quy trong tất cả các giống "kỳ lạ" của nó.

Bài tập

▷ **1.66.** Để đề xuất một lý do có thể cho việc triển khai factrec.c khác nhau trong Borland C cho DOS và Microsoft Visual C ++ cho Windows. Bạn có thể đưa ra một lựa chọn "an toàn" không?

▷ **1.67.** Để đưa ra giả định về kết quả của việc thực hiện đoạn phân đoạn dưới đây. Bạn có mong đợi sự khác biệt trong Borland C cho DOS và Microsoft Visual C ++ cho Windows không? Dự đoán của bạn có khớp với kết quả thực tế không?

```
unsigned i = 1;
printf("%u %u", ++i, i);
```

Và bạn nghĩ gì về mảnh vỡ:

```
unsigned i = 1;
printf("%u %u", i, ++i);
```

▷ **1.68.** Giả sử giá trị của biến x sau khi thực hiện đoạn chương trình dưới đây. Bạn có mong đợi sự khác biệt trong Borland C cho DOS và Microsoft Visual C ++ cho Windows không? Dự đoán của bạn có khớp với kết quả thực tế không?

```
unsigned x, a = 3, b = 5;
x = a+++b;
```

▷ **1.69.** Dựa trên kết quả của các tác vụ trước đó, đưa ra các đề xuất để viết mã dễ di chuyển và rõ ràng nhất.

1.8. Thuật toán đếm cơ sở

Nhiều bài toán chúng ta sẽ xem xét trong cuốn sách này liên quan đến việc tìm ra giải pháp tối ưu (cực hạn). Trong những tình

huống như vậy, cách tiếp cận phổ biến là khám phá tất cả các giải pháp có thể chấp nhận được cho vấn đề và chọn giải pháp tối ưu - một hoặc một số. Việc tạo ra các đối tượng tổ hợp (sử dụng các thuật toán tổ hợp) tương ứng chính xác với lược đồ này.

Tuy nhiên, trong các bài toán trong đoạn này, chúng ta sẽ không quan tâm đến giải pháp cực hạn, mà chỉ quan tâm đến việc tạo ra tất cả các giải pháp: tất cả các cách có thể để sắp xếp đối tượng, tất cả các cách có thể để chọn đối tượng, v.v. Ví dụ, nếu một bộ bài 52 lá được đưa ra, chúng ta có thể quan tâm đến tất cả các cách có thể để chọn 5 lá bài sao cho chúng liên tiếp và giống nhau (theo ngôn ngữ của poker: thuần túy xô). Một ví dụ khác là tính số hoặc tạo tất cả các số điện thoại có sáu chữ số có thể có, với điều kiện bổ sung là số 9 không tham gia vào chúng.

Chúng ta sẽ xem xét các cấu hình tổ hợp nổi tiếng nhất: cả thuật toán tạo và công thức tìm số của chúng. Mặc dù thoạt nhìn, các thuật toán được xem xét không trực tiếp giải quyết bất kỳ vấn đề thực tế nào ngoài các bài toán tổ hợp thuần túy, nhưng chúng (bao gồm cả kết hợp với các kỹ thuật thuật toán khác) tỏ ra cực kỳ hữu ích trong việc giải quyết một số vấn đề tối ưu hóa.

1.8.1. Hoán vị

Định nghĩa 1.21. Ta xét một tập n phần tử $A = \{a_1, a_2, ..., a_n\}$. Mọi tập có n có thứ tự thu được với các phần tử của A, mỗi phần tử của A tham gia đúng một lần được gọi là *hoán vị*.

Tập hợp tất cả các hoán vị có thể có được ký hiệu là P_n, và số của chúng là bởi $|P_n|$. Không khó để chứng tỏ rằng $|P_n| = n!$.

Ví dụ, nếu cho trước tập hợp $\{a, b, c\}$ có 3 phần tử, thì tất cả các hoán vị có thể có (tức là tất cả n-torques có thứ tự có thể có) như sau:

$$(a, b, c)$$
$$(a, c, b)$$
$$(b, a, c)$$
$$(b, c, a)$$
$$(c, a, b)$$
$$(c, b, a)$$

Đến cuối phần này, các phần tử của tập hợp chúng ta làm việc với sẽ là các số nguyên từ 1 đến n. Điều này không hạn chế tính phổ biến của các thuật toán được xem xét và chúng sẽ có giá trị đối với các tập tùy ý (bất kỳ tập n phần tử nào là đẳng cấu với tập gồm n số tự nhiên đầu tiên).

Sinh ra hoán vị

Để tạo ra các hoán vị trên máy tính, chúng ta sẽ sử dụng một thuật toán đệ quy:

Thuật toán 1:

Chúng ta đặt từng phần tử ở vị trí đầu tiên theo thứ tự tuyến tính, sau đó ở $n-1$ vị trí còn lại, chúng ta đặt tất cả các hoán vị có thể có của $n-1$ còn lại thành phần. Do đó, chúng ta có được sơ đồ đệ quy sau để tạo ra các hoán vị:

```
//Đặt phần tử vào vị trí i
void permute(i) {
  if (i >= n) {//Tìm được một sắp xếp tuyến tính
  //những phần tử và in nó ra
  printPerm();
} else
for (k = 0; k < n; k++)
  if (!used[k]){//Nếu phần tử k chưa dùng đến
  used[k] = 1;//tìm được phần tử k để dùng nó
  position[i] = k; // phần tử của vị trí i là k
  permute(i+1);
  used[k] = 0; //bỏ đánh dấu phần tử k đã dùng
  }
}
```

Hàm *permute*() có một tham số duy nhất: vị trí mà chúng ta sẽ "đặt" một phần tử. Với vòng lặp bên trong, chúng ta lần lượt đặt tất cả các phần tử hiện không tham gia theo thứ tự vào vị trí thứ i và tiếp tục đệ quy cho vị trí thứ $(i+1)$. Chúng ta sẽ nhập một mảng used[] để chúng ta có thể xác định xem một phần tử có được sử dụng hay không: nếu phần tử đã tham gia vào hoán vị, thì used[k] == 1 và used[k] == 0, ngược lại (tức là . nó sẽ được "đặt"). Phần cuối của đệ quy ở i -- n - khi đó tất cả các phần tử được "đặt" và chúng ta có một hoán vị, chúng ta sẽ in ra.

Đây là cách thực hiện đầy đủ, in tất cả $n!$ hoán vị trên một tập hợp có n phần tử:

Chương trình 1.30. Tính hoán vị (133permute.c)

```
 #include <stdio.h>
#define MAXN 100
const unsigned n = 3;
char used[MAXN];
unsigned mp[MAXN];
void print(void)
{ unsigned i;
  for (i = 0; i < n; i++) printf("%u ", mp[i] + 1);
  printf("\n");
  }
 void permute(unsigned i)
{ unsigned k;
if (i >= n) { print(); return; }
for (k = 0; k < n; k++) {
  if (!used[k]) {
       used[k] = 1; mp[i] = k;
       permute(i+1); //Nếu có nghĩa tiếp tục sinh permute(i+1)
       used[k] = 0;
       }
  }
}
int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) used[i] = 0;
  permute(0);
  return 0;
}
```

Kết quả thực hiện chương trình:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Hãy xem bình luận ở trên có chứa if. Nếu một bài toán tối ưu được đưa ra, lời giải của nó được rút gọn để tạo ra các hoán vị, có thể làm gián đoạn thế hệ hiện tại, nếu chắc chắn rằng nó sẽ không dẫn đến một lời giải tối ưu. Trong tài liệu dưới đây (xem Chương 6), chúng ta sẽ xem xét vấn đề này một cách chi tiết và tạm thời chúng ta sẽ giới hạn bản thân trong một ví dụ cụ thể về sự gián đoạn như vậy.

Bài toán: Tìm tất cả các số điện thoại có mười chữ số trong đó mỗi chữ số tham gia đúng một lần và sao cho k_i là chữ số thứ i của số điện thoại thì $\sum_{1\leq i\leq 10 i\neq k} k_i = 20$.

Rõ ràng là tất cả các số điện thoại có thể được tạo ra bằng cách hoán vị. Do đó, nếu khi đặt một vài chữ số đầu tiên theo thứ tự tuyến tính, chúng ta nhận thấy rằng tổng vượt quá 20, thì sẽ không có ý nghĩa gì khi tiếp tục tạo, vì chúng ta sẽ không thể nhận được tổng chính xác.

Chương trình trên tạo ra các hoán vị theo thứ tự từ vựng. Điều này có nghĩa là cứ hai hoán vị liên tiếp $(i_1, i_2, ..., i_n)$ và $(j_1, j_2, ..., j_n)$ thì tồn tại một số $k(1 \leq k < n)$ sao cho $i_p = j_p, p = 1, 2, ..., k-1$ và $i_k < j_k$. Chúng ta có thể coi hoán vị là các số trong một hệ thống số tương ứng: một hoán vị lớn hơn về mặt từ vựng so với hoán vị khác nếu nó lớn hơn dưới dạng một số.

Có thể thực hiện quá trình sinh theo cách khác: thu được các hoán vị có kích thước $k+1$ từ các hoán vị có kích thước k, sử dụng cơ chế đệ quy. Do đó, không cần thêm bộ nhớ cho mảng used[] đã sử dụng. Điều này có thể được thực hiện, chẳng hạn, với thuật toán đệ quy sau [Nakov-1998]:

Thuật toán 2.

1) Tại $n = 1$, chúng ta tạo ra một hoán vị duy nhất: (1).

2) Xét một hoán vị $(p_1, p_2, \ldots, p_k)$ gồm $k(1 \leq k < n)$ phần tử. Chúng ta đặt phần tử $(k+1)$ ở vị trí $1, 2, ..., (k+1)$, tương ứng:

$$
\begin{gathered}
(p_{k+1}, p_1, p_2, \ldots, p_k) \\
(p_1, p_{k+1}, p_2, \ldots, p_k) \\
\ldots \\
(p_1, p_2, \ldots, p_k, p_{k+1})
\end{gathered}
$$

Như vậy, lặp lại bước 2) với tất cả các hoán vị của k phần tử, ta thu được tất cả các hoán vị của $k+1$ phần tử.

Việc triển khai được đề xuất dưới đây là một sửa đổi của ***Thuật toán 2.*** Chúng ta để nó như một bài tập để người đọc xem xét mối quan hệ giữa chúng, thoạt nhìn có vẻ không trực tiếp như vậy.

Chương trình 1.31. Tính hoán vị 2 (134permswap.c)

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 3;
unsigned a[MAXN];

void print(void)
{ unsigned i;
  for (i = 0; i < n; i++) printf("%u ", a[i] + 1);
  printf("\n");
}

void permut(unsigned k)
{ unsigned i, swap;
  if (k == 0) print();
  else {
    permut(k - 1);
    for (i = 0; i < k - 1; i++) {
      swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;
      permut(k - 1);
      swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;
    }
  }
}

int main(void) {
  unsigned i;
  printf("\n\n");
  for (i = 0; i < n; i++) a[i] = i;
  permut(n);
  return 0;
}
```

Kết quả thực hiện chương trình:

1 2 3
2 1 3
3 2 1
2 3 1
1 3 2
3 1 2

Bài tập

▷ **1.70.** Chứng minh rằng số hoán vị của một tập hợp n phần tử là n!

▷ **1.71.** Tìm các hoán vị của các phần tử của \{a, b, c, d\} theo cách thủ công, sử dụng:

a) thuật toán 1

b) thuật toán 2

So sánh các kết quả.

▷ **1.72.** Để chứng minh thuật toán 1 và thuật toán 2.

▷ **1.73.** Chứng tỏ rằng permswap.c thực sự là một triển khai của *Thuật toán 2*.

▷ **1.74.** Để thực hiện một thuật toán để tạo lặp đi lặp lại các hoán vị.

1.8.2. Mã hóa và giải mã

Đôi khi, ví dụ, khi tạo các hoán vị ngẫu nhiên bằng các thuật toán heuristic (Chương 9) hoặc ghi nhớ (Chương 8), cần phải mã hóa và giải mã rõ ràng các hoán vị. Điều này có nghĩa là tại mỗi hoán vị khác nhau, một số tự nhiên duy nhất được so sánh để sau này có thể tái tạo duy nhất từ nó. Ví dụ, hãy xem xét các hoán vị có thứ tự từ vựng của ba phần tử:

$$(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1).$$

Trên mỗi chúng, chúng ta có thể so sánh một số tự nhiên duy nhất từ 0 đến 5. Nói chung, mỗi hoán vị của một tập n phần tử đã cho tương ứng với một số nhị vị giữa 0 và $n! - 1$. Một thuật toán để xác định nó bằng một hoán vị nhất định (mã hóa hoán vị) và phép

toán nghịch đảo (giải mã) có thể được biên dịch theo các sơ đồ sau [Shishkov-1995]:

Mã hóa

1) Cho perm[] là một mảng có độ dài n chứa các phần tử của hoán vị mà ta cần mã hóa.

2) Cho pos = result = 0, p[] là một mảng có n phần tử, $p[i] = i + 1, i = 0, 1, ..., n - 1$.

3) Nếu $pos < n$, thì chúng ta loại trừ phần tử chiếm vị trí thứ r trong p và gán cho r, trong đó i là chỉ số mà perm[pos] == p[i]. Nếu không, thuật toán kết thúc và result là kết quả bắt buộc.

4) Gán $result = result * (n - pos) + r$.

5) pos ++ và chuyển sang 3).

Giải mã

1) ho số num và chúng ta phải khôi phục lại hoán vị từ nó. Gọi k = n-1, p[] là một mảng n sao cho p[i - 1] = i, với $i = 1, 2, ..., n$.

2) Trong khi $k \geq 0$, chúng ta lặp lại:

```
m = n - k;
perm [k] = num% m;
if (k> 0) num / = m;
k--;
```

3) Gán $k = 0$. Trong khi $k < n$ ta lặp lại:

```
m = perm [k];
perm [k] = p [m];
nếu (k <n)
for (i = m + 1; i <n; i ++) p [i-1] = p [i];
k ++;
```

Sau đây là mã nguồn hoàn chỉnh của chương trình thực hiện các toán tử trên:

Chương trình 1.32. Số chữ số (135codeperm.c)

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

const unsigned n = 6;
const unsigned perm[MAXN] = { 5, 3, 6, 4, 2, 1 };
```

```
const unsigned long code = 551;

unsigned long codePerm(unsigned n, unsigned perm[])
{ unsigned p[MAXN], i, pos;
  unsigned long r, result;
  result = 0;
  for (i = 0; i < n; i++) p[i] = i + 1;
  for (pos = 0; pos < n; pos++) {
    r = 0;
    while (perm[pos] != p[r]) r++;
    result = result * (n - pos) + r;
    for (i = r + 1; i < n; i++) p[i - 1] = p[i];
  }
  return result;
}

void decodePerm(unsigned long num, unsigned n, unsigned perm[])
{ unsigned long r, m, k;
  unsigned i, p[MAXN];
  for (i = 0; i < n; i++) p[i] = i + 1;
  k = n;
  do {
    m = n - k + 1;
    perm[k - 1] = num % m;
    if (k > 1) num /= m;
  } while (--k > 0);
  k = 0;
  do {
    m = perm[k]; perm[k] = p[m];
    if (k < n)
      for (i = m + 1; i < n; i++) p[i - 1] = p[i];
  } while (++k < n);
}

int main(void) {
  unsigned i;
  printf("Mot to hop ma hoa nhu la %lu \n", codePerm(n, perm));
  printf("Mo ma hoa cua so la %lu: ", code);
  decodePerm(code, n, perm);
  for (i = 0; i < n; i++) printf("%u ", perm[i]);
```

```
    printf("\n");
    return 0;
}
```

Kết quả thực hiện chương trình:

Hoán vị đã cho được mã hóa là 551

Ta giải mã hoán vị tương ứng với số 551: 5 3 6 4 2 1

Bài tập

▷ **1.75.** 1. Sử dụng thuật toán trên, mã hóa thủ công các hoán vị: $(2,3,1,4), (5,3,2,4,1), (3,6,4,1,5,2)$.
2. Sử dụng thuật toán trên, tìm thủ công các hoán vị của 5 phần tử tương ứng với mã: $3, 13, 27, 87, 119$.

1.8.3. Hoán vị lặp lại

Bây giờ chúng ta hãy xem xét một tập đa thay vì tập A (nghĩa là có thể cho các phần tử lặp lại tham gia vào A). Lấy ví dụ, tập đa $A = \{1,1,2,3\}$ của 4 phần tử. Chúng ta quan tâm đến việc tạo ra tất cả các *hoán vị có lặp lại*. Vì vậy, một lần nữa chúng ta đang tìm kiếm tất cả các n-bộ có thứ tự khác nhau được tạo thành bởi các phần tử của A. Lưu ý rằng khi các vị trí thay đổi các phần tử giống nhau - ví dụ: hai đầu tiên trong hoán vị $(1,1,2,3)$, không mới hoán vị xảy ra. Nói chung, hai hoán vị được coi là khác nhau khi có một vị trí trong đó các phần tử khác nhau được đặt.

Tập hợp tất cả các hoán vị có lặp lại được ký hiệu là $\widetilde{P}_n^{s_1,s_2,...,s_k}$, trong đó s_i là số phần tử loại $i (i = 1,2,...,k)$. Số tất cả các hoán vị có lặp lại được cho bởi công thức:

$$|\widetilde{P}_n^{s_1,s_2,...,s_n}| = \frac{n!}{s_1!s_2!....s_k!}, \quad n = \sum_{i=1}^{k} s_i. \tag{1.13}$$

Với $s_1 = 2, s_2 = 1, s_3 = 1$ là số cần thiết $|\widetilde{P}_4^{2,1,1}|$ sẽ bằng $\frac{4!}{2!1!1!} = 12$.

Bài tập

▷ **1.76.** Viết hoán vị các phần tử của tập đa $\{1,1,2,3\}$.

▷ **1.77.** Viết chương trình tìm tất cả các hoán vị có lặp. Có thể sử dụng việc thực hiện hoán vị mà không lặp lại làm cơ sở không? Nếu vậy, những thay đổi nào là cần thiết. Nếu không - tại sao?

▷ **1.78.** Viết chương trình tính số hoán vị có lặp với các tham số liên quan đã cho. Sử dụng thừa số hóa.

▷ **1.79.** Để thực hiện các thuật toán mã hóa và giải mã hoán vị có lặp lại.

▷ **1.80.** Chứng minh công thức (1.13).

1.9. Chỉnh hợp

1.9.1. Các dạng chỉnh hợp và cách sinh ra

Các thuật toán sau đây mà chúng ta sẽ xem xét là để tạo ra các biến thể có và không lặp lại. Mọi lập trình viên phải viết các vòng lặp lồng nhau - khi chúng là hai, ba hoặc một số xác định trước, thật dễ dàng. Tuy nhiên, khi số của chúng không được biết trước, ví dụ bài toán chúng ta đang giải yêu cầu n chu trình lồng nhau, thì cách tiếp cận tiêu chuẩn không thể áp dụng được. Hãy xem xét đoạn sau:

Ví dụ 1.1. Chỉnh hợp có lặp

Chỉnh hợp có lặp

```
for (a1 = 1; a1 <= k; a1++)
  for (a2 = 1; a2 <= k; a2++)
    for (a3 = 1; a3 <= k; a3++)
          ...
      for (an = 1; an <= k; an++)
        printf(" %u %u %u ... %u;", a1, a2, a3, ..., an);
```

Kết quả của việc thực thi $n = 2$ và $k = 3$, tương đương với 2 chu kỳ lồng nhau cho i từ 1 đến 3, sẽ là:

1 1; 1 2; 1 3; 2 1; 2 2; 2 3; 3 1; 3 2; 3 3;

Ví dụ 1.2. Chỉnh hợp không lặp

Chỉnh hợp không lặp

```
for (a1 = 1; a1 <= k; a1++)
  for (a2 = 1; a2 <= k; a2++) if (a2 != a1)
    for (a3 = 1; a3 <= k; a3++) if ((a3 != a1) && (a3 != a2))
    ...
      for (an = 1; an <= k; an++)
        if ((an!=a1)&&(an!=a2)&&(an!=a3)&&...&&(an!=an-1))
          printf("%u %u %u ... %u;", a1, a2, a3, ... ,an);
```

Kết quả cho $n = 2$ và $k = 3$ bây giờ sẽ là:

1 2;1 3;2 1;2 3;3 1;3 2;

Sự khác biệt giữa hai biến thể của thế hệ là ở thế hệ thứ hai không có số lặp lại.

Kết quả của đoạn đầu tiên là tạo ra tất cả các biến thể với sự lặp lại của n phần tử của lớp thứ k và kết quả của đoạn thứ hai - các biến thể không có sự lặp lại của n phần tử của lớp thứ k. Chúng ta sẽ định nghĩa các khái niệm chính xác hơn:

Cho là một tập hợp A với n phần tử.

Định nghĩa 1.22. Một biến thể có sự lặp lại của n phần tử của hạng thứ k được gọi là danh sách nhiều phần tử có k được tạo thành bởi các phần tử của A (không nhất thiết phải khác).

Tập hợp tất cả các biến thể có lặp lại được ký hiệu là $\widetilde{V}_n^k$ và số phần tử của nó là n^k. Chúng ta để lại phần chứng minh của phần sau cho người đọc như một bài tập dễ dàng.

Định nghĩa 1.23. Một biến thể không có sự lặp lại của n phần tử của lớp k được gọi là bất kỳ danh sách k phần tử có thứ tự nào được tạo thành bởi các phần tử của A.

Số lượng các biến thể khác nhau không lặp lại là $|V_n^k = \frac{n!}{(n-k)!}|$

Ta thấy ngay rằng với $k = n$ thì tập các biến thiên không lặp lại trùng với tập các hoán vị không lặp lại.

Chúng ta sẽ tạo ra tất cả các biến thể với sự lặp lại trên n và k đã cho, cho tập A, bao gồm các số tự nhiên từ 1 đến n. Điều này, tất nhiên, không thể được thực hiện với các vòng lặp lồng nhau, như trong hai phân đoạn ở trên. Chúng ta sẽ tập trung vào cách tiếp cận mà chúng ta đã sử dụng để tạo ra các hoán vị. Sự khác biệt sẽ là chúng ta sẽ đặt từng phần tử vào từng vị trí, chứ không phải chỉ một phần tử chưa được sử dụng cho đến nay (vì cho phép lặp lại). Điều này giúp loại bỏ sự cần thiết của một mảng used[] để đánh dấu các phần tử được sử dụng và chúng ta dễ dàng tiếp cận một chương trình tương tự như permute.c:

Chương trình 1.33. Chỉnh hợp (136variate.c)

```
#include <stdio.h>
#define MAXN 100
/*chinh hop tu  n  lay  k phan tu*/
const unsigned n = 4;
const unsigned k = 2;

int taken[MAXN];

void print(unsigned i)
{ unsigned k;
  printf("( ");
  for (k = 0; k <= i - 1; k++) printf("%u ", taken[k] + 1);
  printf(")\n");
}

/*Hoi quy*/
void variate(unsigned i)
{ unsigned j;
  /*Neu sai if (i>=k) thi return; o day (chi in print(i);
  neu muon co the sinh ra tat ca do dai 1,2, ..., k, khong chi co do
    dai k */
  if (i >= k) { print(i); return; }
  for (j = 0; j < n; j++) {
    /* if (allowed(k)) { */
    taken[i] = j;
    variate(i + 1);
  }
}
```

```
int main(void) {
  variate(0);
  return 0;
}
```

Kết quả thực hiện chương trình:

```
(1 1)
(1 2)
(1 3)
(1 4)
(2 1)
(2 2)
(2 3)
(2 4)
(3 1)
(3 2)
(3 3)
(3 4)
(4 1)
(4 2)
(4 3)
(4 4)
```

Bài tập

▷ **1.81.** Để sửa đổi chương trình trên để tạo ra các biến thể mà không lặp lại.

▷ **1.82.** 2. Viết các biến 3 phần tử của các phần tử của tập hợp $\{a, b, c, d, e\}$:

- với sự lặp lại;
- không lặp lại.

▷ **1.83.** Chứng minh rằng số biến đổi k phần tử có lặp lại trên một tập hợp n phần tử là nk.

▷ **1.84.** Chứng minh rằng số biến đổi k phần tử không lặp lại trên một tập hợp n phần tử là $n!/(N-k)!$

▷ **1.85.** Để thực hiện một thuật toán để tạo lặp đi lặp lại các biến thể có / không lặp lại.

▷ **1.86.** Để thực hiện các thuật toán mã hóa và giải mã các biến thể có và không có lặp lại.

1.10. Tổng bằng không

Bài toán. Cho các số $a_1, a_2, ..., a_n$. Đặt các phép toán "+" và "-" giữa các số a_i và a_{i+1}, với $i = 1, 2, ..., n-1$ sao cho kết quả sau khi tính biểu thức thu được bằng 0.

Ví dụ, đối với các số tự nhiên từ 1 đến 8, một số giải pháp khả thi cho bài toán sau đây:

$$1+2+3+4-5-6-7+8=0$$
$$1+2+3-4+5-6+7-8=0$$
$$1+2-3+4+5+6-7-8=0$$
$$1+2-3-4-5-6+7+8=0$$

Lời giải: Chúng ta sẽ tạo ra tất cả các loại biến thể bằng cách lặp lại $n-1$ phần tử của lớp thứ hai, tức là tất cả các loại có thứ tự $(n-1)$-bộ, bao gồm 0 và 1 (tương ứng với một dấu dương và một dấu âm ở phía trước của số tương ứng). Với mỗi $(n-1)$-bộ như vậy, chúng ta sẽ kiểm tra xem nó có phải là một nghiệm của bài toán hay không, và với mục đích này, chúng ta sẽ tính toán giá trị của biểu thức tương ứng. Cách thực hiện đầy đủ của giải pháp này như sau, vì dữ liệu đầu vào được đặt ở đầu dưới dạng hằng số - n là số lượng các số và mảng $a[]$ chứa chính các số:

Chương trình 1.34. Tổng bằng 0 (137sumzero.c)

```
#include <stdio.h>
#include <math.h>

/* So luong so trong day */
const unsigned n = 8;
/* Day so */
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
/* Tim tong */
```

```
int sum = 0;

void checkSol(void)
{ unsigned i;
  int tempSum = 0;
  for (i = 0; i < n; i++) tempSum += a[i];
  if (tempSum == sum) { /* Tim duoc nghiem => In no ra*/
    for (i = 0; i < n; i++)
      if (a[i] > 0) printf("+%d ", a[i]);
        else printf("%d ", a[i]);
    printf(" = %d\n", tempSum);
  }
}

void variate(unsigned i)
{ if (i >= n) {
    checkSol();
    return;
  }
  a[i] = abs(a[i]);  variate(i + 1);
  a[i] = -abs(a[i]); variate(i + 1);
}

int main(void) {
  variate(0);
  return 0;
}
```

Kết quả thực hiện chương trình:

$$
\begin{aligned}
+1+2+3+4-5-6-7+8 &= 0\\
+1+2+3-4+5-6+7-8 &= 0\\
+1+2-3+4+5+6-7-8 &= 0\\
+1+2-3-4-5-6+7+8 &= 0\\
+1-2+3-4-5+6-7+8 &= 0\\
+1-2-3+4+5-6-7+8 &= 0\\
+1-2-3+4-5+6+7-8 &= 0\\
-1+2+3-4+5-6-7+8 &= 0\\
-1+2+3-4-5+6+7-8 &= 0\\
-1+2-3+4+5-6+7-8 &= 0\\
-1-2+3+4+5+6-7-8 &= 0\\
-1-2+3-4-5-6+7+8 &= 0\\
-1-2-3+4-5+6-7+8 &= 0\\
-1-2-3-4+5+6+7-8 &= 0
\end{aligned}
$$

Bài tập

▷ **1.87.** Để giải bài toán, nếu trong phép cộng và phép trừ, có thể đặt một dấu nhân.

▷ **1.88.** Đề xuất một cách để lưu một số phép tính. Ví dụ, khi quá trình tạo có thể bị gián đoạn sau vị trí thứ k, i. tiêu chí nào có thể đảm bảo rằng tổng 0 sẽ không nhận được, bất kể phép tính số học nào sẽ được đặt trên $n-k$ vị trí khác?

1.11. Tổ hợp

Nếu cho đến nay thứ tự của các yếu tố đã quan trọng, thì trong đoạn này nó sẽ không còn khiến chúng ta quan tâm nữa. Cho một tập hợp A có n phần tử.

Định nghĩa 1.24. Một tổ hợp không có sự lặp lại của n phần tử thuộc lớp k được gọi là tập hợp con k phần tử của A.

Định nghĩa 1.25. Một tổ hợp có sự lặp lại của n phần tử của lớp k được gọi là một tập hợp nhiều phần tử có chứa các phần tử của A.

Ví dụ: đối với $n = 4, k = 2$ của tập bốn phần tử a, b, c, d thì các kết hợp không có sự lặp lại của lớp thứ hai là $\{a,b\},\{a,c\},\{a,d\},\{b,c\},\{b,d\},\{c,d\}$ và các kết hợp có lặp lại là:

$$\{a,a\},\{a,b\},\{a,c\},\{a,d\},\{b,b\},\{b,c\},\{b,d\},\{c,c\},\{c,d\},\{d,d\}.$$

Tổ hợp được sử dụng rộng rãi trong các bài toán tối ưu hóa. Trong tương lai chúng ta sẽ gặp chúng nhiều lần, vì vậy các thuật toán cho thế hệ của chúng rất quan trọng. Một số bài toán liên quan đến tổ hợp được đưa ra trong phần 1.5.

Số lượng các kết hợp không có sự lặp lại, mà chúng ta đã thấy cách chúng ta có thể tính toán bằng cách tính toán trong 1.1.5., Là

$$C_n^k = C_n^k = \frac{n(n-1)...(n-k+1)}{k(k-1)...1} = \frac{n!}{k!(n-k)!}$$

và các tổ hợp có lặp lại:

$$\widetilde{C}_n^k = \frac{(n+k-1)!}{k!(n-1)!}.$$

Việc tạo các tổ hợp không lặp lại sẽ được thực hiện theo sơ đồ của thuật toán tạo hoán vị. Chúng ta tuân thủ nó, bởi vì khi được tạo theo thứ tự từ điển, sẽ cực kỳ thuận tiện nếu bạn cần "cắt" một số giải pháp đã tạo. Như đã đề cập, sự cắt giảm như vậy xảy ra khi giải quyết các vấn đề thực tế, trong đó việc nghiên cứu một nhánh lớn của các cấu hình tổ hợp có thể là vô nghĩa.

Chương trình 1.35. Tìm tổ hợp (138comb.c)

```
#include <stdio.h>
#define MAXN 20
/*Tìm tất cả các tổ hợp của n phần tử của lớp k*/
const unsigned n = 5;
const unsigned k = 3;
unsigned mp[MAXN];
void print(unsigned length)
{ unsigned i;
  for (i = 0; i < length; i++) printf("%u ", mp[i]);
  printf("\n");
}
void comb(unsigned i, unsigned after)
```

```
{ unsigned j;
 if (i > k) return;
 for (j = after + 1; j <= n; j++) {
   mp[i - 1] = j;
   if (i == k) print(i);
   comb(i + 1, j);
 }
}
int main() {
  printf("C(%u,%u): \n", n, k);
  comb(1, 0);
  return 0;
}
```

Kết quả thực hiện chương trình:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

Bài tập

▷ **1.89.** Để sửa đổi chương trình trên để nó tạo ra các kết hợp có lặp lại.

▷ **1.90.** Viết các tổ hợp 3 phần tử của các phần tử của tập hợp $\{a, b, c, d, e\}$:

- với sự lặp lại
- không lặp lại

▷ **1.91.** Giải thích mối quan hệ giữa số tổ hợp không lặp lại của n phần tử hạng k và hệ số của nhị thức.

▷ **1.92.** Chứng minh rằng số tổ hợp k phần tử không lặp lại trên một tập n phần tử là $n!/(k!(n-k)!)$.

▷ **1.93.** Chứng minh rằng số tổ hợp gồm k phần tử có lặp lại trên một tập n phần tử là $(n+k-1)!/(k!(n-1)!)$. Đưa ra hai chứng minh: một đại số và một tổ hợp.

▷ **1.94.** Để thực hiện một thuật toán để tạo lặp đi lặp lại các kết hợp có / không lặp lại.

▷ **1.95.** Để thực hiện các thuật toán mã hóa và giải mã kết hợp có / không lặp lại.

1.12. Biểu diễn số thành tổng

1.12.1. Tạo ngắt số dưới dạng tổng của các số đã cho

Bài toán: Tìm tất cả các biểu diễn không có thứ tự (phân tích) có thể có của n trên một số tự nhiên n đã cho dưới dạng tổng các số tự nhiên (không nhất thiết khác nhau). Ví dụ, số 5 có thể được chia theo 7 cách sau:

$$5 = 5$$
$$5 = 4 + 1$$
$$5 = 3 + 2$$
$$5 = 3 + 1 + 1$$
$$5 = 2 + 2 + 1$$
$$5 = 2 + 1 + 1 + 1$$
$$5 = 1 + 1 + 1 + 1 + 1$$

Thuật toán mà chúng ta sẽ nhận ra sự phân rã là đệ quy:

devNum(0) = {}

devNum (n) = {k} + devNum (n - k), k = n, n - 1, . . . , 1.

Chúng ta phải cẩn thận và tránh tạo ra các lỗi lặp đi lặp lại, chẳng hạn như

$$5 = 3 + 2$$
$$5 = 2 + 3$$

Cách sau rất dễ thực hiện: chúng ta sẽ yêu cầu mỗi phép cộng tiếp theo nhỏ hơn hoặc bằng lần trước. Hàm đệ quy thực hiện devNum có hai đối số: *n* (số break) và *pos* - một biến cho biết số lần bị phá vỡ cho đến nay:

Chương trình 1.36. Biểu diễn thành tổng số (139devnum.c)

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 7;
unsigned mp[MAXN + 1];
void print(unsigned length)
{ unsigned i;
  for (i = 1; i < length; i++)
    printf("%u+", mp[i]);
  printf("%u\n", mp[length]);
}
void devNum(unsigned n, unsigned pos)
{
  if (0 == n)
    print(pos-1);
  else {
    unsigned k;
    for (k = n; k >= 1; k--) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos-1])
        devNum(n-k, pos+1);
    }
  }
}
int main() {
   mp[0] = n+1;
   devNum(n, 1);
   return 0;
}
```

Kết quả thực hiện chương trình:

```
7
6+1
5+2
5+1+1
```

4+3
4+2+1
4+1+1+1
3+3+1
3+2+2
3+2+1+1
3+1+1+1+1
2+2+2+1
2+2+1+1+1
2+1+1+1+1+1
1+1+1+1+1+1+1

Nếu nhiệm vụ chỉ tìm kiếm số lần ngắt khác nhau, thì (như với các cấu hình tổ hợp khác) không cần thiết phải tạo và đếm chúng hoàn toàn. Một cách tiếp cận như vậy sẽ cực kỳ không hiệu quả. Có một công thức lặp lại cho phép tính trực tiếp con số này. Trên thực tế, những thứ với công thức này không đơn giản như vậy và cần phải sử dụng tính năng tối ưu hóa động, vì vậy chúng ta sẽ xem xét nó ở phần sau, trong 8.3.6.

Bài tập

▷ **1.96.** Để triển khai một thuật toán tạo ngắt theo thứ tự từ vựng.

▷ **1.97.** Để triển khai một phiên bản lặp đi lặp lại của thuật toán để tạo ra sự cố.

1.12.2. Sinh ra tất cả biểu diễn một số như là tích của các số tự nhiên

Sự khác biệt với đoạn trước, cả về thuật toán và cách triển khai, là tối thiểu. Thay vì devNum (n-k, cnt + 1), chúng ta sẽ gọi đệ quy devNum (n / k, cnt + 1), không phải cho mọi k, mà chỉ cho những người có $n\%k == 0$. Điều kiện để tiếp tục phân tích (vòng lặp for) sẽ là $k > 1$, không phải $k \geq 1$, tức là phần dưới cùng của đệ quy sẽ là $k == 1$, không phải $k == 0$ (sau này dễ dàng giải thích: 0 và 1 là đồng dạng của các phép toán cộng và nhân). Sau đây là cách triển khai đã sửa đổi:

Chương trình 1.37. Biểu diễn thành tích số (140devnum2.c)

```
#include <stdio.h>
#define MAXLN 20 /* Hệ số: tối đa log2n (tối thiểu 2) */
const unsigned n = 50; /* Một con số chúng ta sẽ tách ra */
unsigned mp[MAXLN];
void print(unsigned length)
{ unsigned i;
  for (i = 1; i < length; i++) printf("%u * ", mp[i]);
  printf("%d\n", mp[length]);
}
void devNum(unsigned n, unsigned pos) {
if (1 == n)
  print(pos-1);
else {
unsigned k;
  for (k = n; k > 1; k--) {
    mp[pos] = k;
    if (mp[pos] <= mp[pos-1] && n % k == 0)
      devNum(n / k, pos+1);
    }
  }
}
int main() {
   mp[0] = n + 1;
   devNum(n, 1);
   return 0;
}
```

Kết quả thực hiện chương trình:

```
50
25 * 2
10 * 5
5 * 5 * 2
```

Bài tập

▷ **1.98.** Để thực hiện một thuật toán để tạo ra các ngắt theo thứ tự từ vựng.

▷ **1.99.** Để triển khai một phiên bản lặp đi lặp lại của thuật toán để

tạo ra sự cố.

1.12.3. Sinh ra tất cả biểu diễn một số như là tổng của các số tự nhiên

Thú vị hơn là trường hợp chúng ta phải chia một số tự nhiên là tổng của các số tự nhiên đã cho. Ví dụ, chúng ta có tem bưu chính BGN 2, 5 và 10 và chúng ta phải gửi một bưu kiện có giá trị BGN 20. Tất cả các khả năng (tổng cộng là 6) để hình thành số tiền này là:

$$
\begin{aligned}
20 &= 10 + 10 \\
20 &= 10 + 5 + 5 \\
20 &= 10 + 2 + 2 + 2 + 2 + 2 \\
20 &= 5 + 5 + 5 + 5 \\
20 &= 5 + 5 + 2 + 2 + 2 + 2 + 2 \\
20 &= 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2
\end{aligned}
$$

Thuật toán giải bài toán này tương tự như thuật toán ngắt một số dưới dạng tổng các số tự nhiên. Hãy để các số chúng ta có thể sử dụng trong bảng phân tích nằm trong mảng given[gN]. Chúng ta sẽ thực hiện một chu trình cho p = 0,1, ..., gN-1, và trong lệnh gọi đệ quy thay vì *p*, chúng ta sẽ giảm *n* bởi giá trị tương ứng của given[p]:

Chương trình 1.38. Biểu diễn thành tích số (141devnum3.c)

```
#include <stdio.h>
#define MAX_ADDS 100
/* Tổng sẽ biểu diễn*/
const unsigned n = 15;
/*Số giá trị đồng xu khác nhau */
const unsigned gN = 3;
/* Giá trị của đồng xu */
const unsigned given[] = { 2, 3, 5 };
unsigned mp[MAX_ADDS];
void print(unsigned length)
{ unsigned i;
  for (i = 1; i < length; i++)
     printf("%u + ", mp[i]);
  printf("%d\n", mp[length]);
```

```
}
void devNum(unsigned n, unsigned pos)
{ unsigned k, p;
  for (p = gN; p > 0; p--) {
    k = given[p - 1];
    if (n > k) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos - 1])
        devNum(n - k, pos + 1);
    }
   else if (n == k) {
     mp[pos] = k;
    if (mp[pos] <= mp[pos - 1])
    print(pos);
   }
  }
}
int main() {
   mp[0] = n + 1;
   devNum(n, 1);
   return 0;
}
```

Kết quả thực hiện chương trình:

```
5 + 5 + 5
5 + 5 + 3 + 2
5 + 3 + 3 + 2 + 2
5 + 2 + 2 + 2 + 2 + 2
3 + 3 + 3 + 3 + 3
3 + 3 + 3 + 2 + 2 + 2
3 + 2 + 2 + 2 + 2 + 2 + 2
```

Bài tập

▷ **1.100.** Có thể tổng quát thuật toán thành số thực không? Nếu không: tại sao? Nếu có: những thay đổi nào là cần thiết?

▷ **1.101.** Để thực hiện một thuật toán để tạo ra các ngắt từ vựng.

▷ **1.102.** Để triển khai một biến thể lặp đi lặp lại của thuật toán tạo ra sự cố.

▷ **1.103.** Đề xuất và thực hiện thuật toán ngắt một số dưới dạng tích của các số đã cho.

1.12.4. Phân hoạch một tập hợp

Nhiệm vụ chúng ta sẽ xem xét là tìm số lần ngắt có thể có của một tập hợp, tức là biểu diễn của tập hợp như một hợp của các tập hợp con không rỗng không giao nhau của nó. Ví dụ, đối với tập hợp $A = \{1, 2, 3\}$ các ngắt sẽ là:
$\{1, 2, 3\}$
$\{1, 2\}, \{3\}$
$\{1, 3\}, \{2\}$
$\{1\}, \{2, 3\}$
$\{1\}, \{2\}, \{3\}$

Số Bell và Stirling

Mặc dù có một số công thức trực tiếp cho số lượng cấu hình tổ hợp được thảo luận ở trên, nhưng ở đây mọi thứ phức tạp hơn một chút.

Số lần ngắt cho một tập hợp có n phần tử bằng số thứ n của Bell B (n). Số Bell được định nghĩa như sau:

$$B(n) = \sum_{i=0}^{n} St(n, k),$$

trong đó $St(n, k)$ là số chuỗi của loại thứ hai, được định nghĩa đệ quy như sau:

$$St(n, k) = \begin{cases} St(n-1, k-1) + kSt(n-1, k) & \text{với } k = 1, 2, ..., n; \\ 1 & \text{với } n > 0, k - 1; \\ 0 & \text{với } k = 0; \\ 0 & \text{với } n = 0. \end{cases}$$

Số Stirling đại diện cho số lượng biểu diễn của một tập hợp n phần tử như một hợp của chính xác k tập hợp con không rỗng của nó.

Chúng ta có thể tính toán số Stirling theo một số cách. Một là sử dụng trực tiếp định nghĩa đệ quy và thực hiện một hàm đệ quy tương ứng. Tuy nhiên, một số giá trị sẽ được tính toán nhiều lần,

đây là một giải pháp không hiệu quả. Chúng ta đã gặp phải vấn đề tương tự khi tính toán chuỗi Fibonacci. Chúng ta sẽ tìm các số Stirling và Bell theo cách lặp: Đầu tiên chúng ta sẽ tìm các số Stirling M[i]=St (n, i), i = 0,1, ..., n. Sau đó, chúng ta sẽ tìm số Bell thứ n là tổng của các số Stirling tương ứng.

Số Stirling xác định một tam giác tương tự như tam giác Stirling của Pascal. Quan sát này trực tiếp dẫn chúng ta đến ý tưởng nhận ra một thuật toán tương tự như một thuật toán mà chúng ta đã sử dụng cho các hệ số nhị thức.

					1					
				1		1				
			1		3		1			
		1		7		6		1		
	1		15		25		10		1	
1		31		90		65		15		1
...	...	...	...	...	...	...	...	...	...	...

Hình 1.15. Tam giác Stirling

Chương trình 1.39. Số phân hoạch tập hợp (142bell.c)

```
#include <stdio.h>
#define MAXN 100
const unsigned long n = 10;
unsigned long M[MAXN+1];
void stirling (unsigned n)
{ unsigned i, j;
  if (0 == n) M[0] = 1;
    else M[0] = 0;
  for (i = 1; i <= n; i++) {
    M[i] = 1;
    for (j = i-1; j >= 1; j--) M[j] = j*M[j]+M[j-1];
  }
}
unsigned long bell(unsigned n)
{ unsigned i;
  unsigned long result = 0;
  for (i = 0; i <= n; i++) result += M[i];
  return result;
```

```
}
int main() {
    stirling (n);
    printf("bell(%lu) = %lu\n", n, bell(n));
    return 0;
}
```

Kết quả thực hiện chương trình:

```
bell(10) = 115975
```

Bài tập

▷ **1.104.** Tìm số Stirling thứ năm theo cách thủ công.

▷ **1.105.** Để triển khai một biến thể đệ quy của thuật toán tìm số Stirling.

▷ **1.106.** Tìm ra các công thức phản ánh mối quan hệ giữa các số trong tam giác Stirling. So sánh với những gì trong tam giác Pascal của 1.1.5.

▷ **1.107.** Viết một chương trình để tìm tất cả các điểm ngắt của một tập hợp đã cho dưới dạng hợp nhất của các tập hợp con không rỗng của nó.

1.13. Đánh giá và độ phức tạp của thuật toán

Chúng ta sẽ kết thúc chương mở đầu này với phần giới thiệu ngắn gọn về bộ máy toán học cần thiết cho việc nghiên cứu đầy đủ bất kỳ thuật toán máy tính nào. Chủ đề đánh giá và độ phức tạp của các thuật toán là quan trọng và không nên bỏ qua. Các chỉ định và đặc tính mà chúng ta sẽ trình bày sẽ được tìm thấy gần như liên tục trong tài liệu bên dưới.

Khi xem xét một thuật toán máy tính, chúng ta thường quan tâm đến ba thuộc tính của nó:

- đơn giản (và sang trọng)
- tính đúng đắn
- tốc độ

Trong khi cái đầu tiên trong số này có thể được "đo lường" bằng trực giác (và hơi chủ quan), hai cái sau đòi hỏi phân tích chuyên

sâu hơn nhiều. Các kỹ năng chúng ta sẽ có được trong đoạn này sẽ cho phép chúng ta xác định dễ dàng và chính xác hiệu quả của một thuật toán, so sánh các thuật toán, để tăng tốc độ của chúng thông qua các thay đổi được đo lường và chính xác.

Hãy xem đoạn chương trình sau:

Các dòng lệnh tính toán

```
1) n = 100;
2) sum = 0;
3) for (i = 0; i < n; i++)
4) for (j = 0; j < n; j++)
5) sum++;
```

Chúng ta quan tâm đến việc chương trình trên sẽ hoạt động nhanh như thế nào. Những gì chúng ta có thể làm bằng thực nghiệm là kiểm tra xem mất bao lâu để hoàn thành công việc của nó. Để nghiên cứu hành vi của nó một cách tổng quát hơn, chúng ta có thể thực hiện nó cho các giá trị khác của n. Bảng 1.5. cho thấy mối quan hệ giữa lượng dữ liệu đầu vào và tốc độ thực thi.

Kích thước đầu vào	**Thời gian thực hiện**
10	0,000001 giây
100	0,0001 giây
1000	0,01 giây
10000	1,071 giây
100000	106,543 giây
1000000	10663,6 giây

Bảng 1.5. sự phụ thuộc giữa kích thước dữ liệu đầu vào và tốc độ thực thi.

Từ Bảng 1.5 ta thấy rằng khi ta tăng *n* lên mười lần thì thời gian thực hiện của chương trình tăng lên 100 lần.

Chúng ta hãy xem xét phân đoạn trên sâu hơn. Dòng 1) và 2) có một khởi tạo tĩnh mất một thời gian không đổi. Hãy để chúng ta biểu thị nó bằng a. Đối với các hoạt động i = 0 và i ++, cũng như đối với việc kiểm tra i <n, một lần nữa yêu cầu thời gian không đổi (mỗi trong số chúng đại diện cho một số lượng lệnh không đổi của bộ xử

lý), chúng ta sẽ ký hiệu nó bằng b, c , d, tương ứng. Trong dòng 4) thời gian cần thiết cho các phép toán j = 0, j <n và j ++ được ký hiệu là e, f, g. Cuối cùng, hoạt động trên dòng 5) cũng yêu cầu một thời gian không đổi: giả sử là h.

Với các ký hiệu được giới thiệu theo cách này, không khó để tính tổng thời gian hoạt động của chương trình cho một giá trị tùy ý của n:

$$\begin{aligned} a+b+n.c+n.d+n.(e+n.f+n.g+n.h) = \\ = a+b+n.c+n.d+n.e+n.n.f+n.n.g+n.n.h = \\ = n^2.(f+g+h)+n.(c+d+e)a+b \end{aligned}$$

Nhớ lại rằng a, b, c, d, e, f, g, h là các hằng số.

Hãy biểu thị:

$$\begin{aligned} i &= f+g+x \\ j &= c+d+e \\ k &= a+b \end{aligned}$$

(ở đây i và j không liên quan gì đến các biến được sử dụng trong đoạn trên). Do đó, thuật toán được thực thi trong thời gian:

$$i.n^2+j.n+k$$

Các hằng số i, j và k quan trọng đối với tốc độ của thuật toán, nhưng không có ý nghĩa quyết định. Trong thực tế, khi chúng ta nghiên cứu tính hiệu quả của một thuật toán, chúng ta không quan tâm đến chúng. Các hằng số này phụ thuộc chủ yếu vào hiệu suất máy của chương trình của chúng ta, cũng như tốc độ của máy mà nó chạy.

Hơn nữa, khi chúng ta nghiên cứu hành vi của thuật toán của mình, chúng ta có thể bỏ qua ngay cả các đơn thức $j.n$ và k và chỉ để lại một đơn thức trong đó n tham gia nhiều nhất. Mục đích của việc "đơn giản hóa" này là chỉ để lại tính năng quan trọng nhất cho một thuật toán nhất định, tức là chức năng mà thời gian thực thi phụ thuộc ở mức độ lớn nhất, tức là phát triển nhanh nhất khi lượng dữ liệu đầu vào tăng lên.

Hãy xem xét hai hàm cho biết thời gian thực hiện của hai thuật toán A_1 và A_2 đã cho phụ thuộc vào kích thước n của dữ liệu đầu vào: $f = 2.n^2$ và $g = 200.n$. Dễ dàng nhận thấy rằng mặc dù hệ số của g lớn hơn rất nhiều so với của f nhưng khi n vượt qua một giá trị cố định nào đó (trong trường hợp này là $n > 100$) thì thuật toán A_2 sẽ giải bài toán nhanh hơn A_1. Hơn nữa, n càng tăng thì mối quan hệ giữa thời gian thực hiện của hai thuật toán càng tăng theo hướng có lợi cho A_2. Về mặt tiệm cận, thuật toán A_2 nhanh hơn và độ phức tạp của nó là tuyến tính, trong khi thuật toán A_1 là bậc hai.

Trong khai báo dưới đây, chúng ta sẽ dựa trên những quan sát gần đây của mình trên những nền tảng vững chắc hơn.

Bài tập

▷ **1.108.** Cho ba thuật toán có độ phức $5n^2 - 7n + 13, 3n^2 + 15n + 100$ và $1000n$. Cái nào trong số chúng nên được sử dụng cho dữ liệu đầu vào lên đến: 100; 1000; 10000; 1000000?

▷ **1.109.** Có thể cho đa thức $5n^3 - 5n^2 + 5$ xác định độ phức tạp của một thuật toán không? Và đa thức $5n^3 - 5n^2 - 5$? Còn $-5n^3 - 5n^2 + 5$ thì sao?

1.13.1. Lượng dữ liệu đầu vào

Giả sử một bài toán trong đó kích thước của dữ liệu đầu vào được xác định bởi một số nguyên n. Hầu hết tất cả các nhiệm vụ chúng ta sẽ xem xét có thuộc tính này. Chúng ta sẽ làm rõ vấn đề sau bằng cách xem xét một số ví dụ:

Ví dụ 1.3. Một mảng có n phần tử được cho và chúng ta muốn sắp xếp nó. Kích thước của dữ liệu đầu vào được xác định bởi số n phần tử của mảng.

Ví dụ 1.4. Hai số tự nhiên a và b đã cho, ta tìm ước chung lớn nhất của chúng. Ở đây kích thước của dữ liệu đầu vào được xác định bởi số lượng các chữ số nhị phân (bit) lớn hơn trong số các số a và b: tức là từ $\log_2(max(a, b))$ (. Khi chúng ta giới thiệu bộ máy nghiên cứu tính hiệu quả của các thuật toán dưới đây, chúng ta sẽ thấy lý do tại sao chúng ta chọn cực đại của hai số.

Ví dụ 1.5. Một đồ thị được đưa ra và chúng ta tìm cây bao trùm của nó (xem 5.1.).

Trong trường hợp này, rất thuận tiện để mô tả kích thước của đầu vào với hai tham số: số đỉnh và số cạnh của đồ thị.

1.13.2. Ký hiệu tiệm cận

Khi chúng ta quan tâm đến độ phức tạp của một thuật toán, chúng ta thường quan tâm đến việc nó hoạt động như thế nào ở kích thước đủ lớn n của dữ liệu đầu vào. Khi chính thức ước tính độ phức tạp của các thuật toán, chúng ta sẽ quan tâm đến hành vi của chúng tại n, có xu hướng đến vô cùng. Chúng ta sẽ mô tả độ phức tạp của một thuật toán với các hàm có dạng $f : \mathbb{N} \to \mathbb{N}$. (Nhớ lại rằng $\mathbb{N}$ biểu thị tập các số tự nhiên: $0, 1, 2, ...$). Đôi khi chúng ta sẽ làm việc với các hàm được xác định trên một tập con của $\mathbb{N}$, chẳng hạn chỉ hợp lệ với n chẵn hoặc từ một giá trị nhất định trở đi. Chúng ta cũng sẽ sử dụng các hàm thực, ví dụ $\log n$, và về nguyên tắc, chúng ta sẽ ngụ ý hạn chế của chúng đối với $\mathbb{N}$. Các trường hợp sau không phải là rất "thuần túy" theo quan điểm lý thuyết, nhưng chúng tiết kiệm được rất nhiều khó khăn.

Định nghĩa 1.26. $O(F(n)) = \{f(n) | \exists c(c > 0), \exists n_0(c) : \forall n > n_0 : 0 \leq f(n) \leq c.F(n)\}$

Nghĩa là $O(F(n))$ là tập các hàm f trong đó có một hằng số $c(c > 0)$ sao cho $f(n) \leq c.F(n)$, với mọi *giá trị đủ lớn* của n, tức là tồn tại một hằng số n_0 (có thể phụ thuộc vào c) mà bất đẳng thức trên áp dụng với mọi $n > n_0$. Do đó $O(F)$ xác định tập hợp tất cả các hàm *phát triển không nhanh hơn* F.

Khi xem xét độ phức tạp của các thuật toán, hai ký hiệu cơ bản hơn được sử dụng: $\Theta(F)$ và $\Omega(F)$.

Định nghĩa 1.27. $\Omega(F(n)) = \{f(n) | \exists c(c > 0), \exists n_0(c) : \forall n > n_0 : f(n) \geq c.F(n) \geq 0\}$

Nghĩa là, $\Omega(F)$ là tập các hàm $f(n)$ mà $f(n) \geq c.F(n)$ với mọi $n > n_0$. Do đó $\Omega(F)$ bao gồm tất cả các *hàm tăng không chậm hơn* F.

Định nghĩa 1.28. $\Theta(F(n)) = \{f(n)|\exists c_1(c_1 > 0), \exists c_2(c_2 > 0), \forall n_0(c_1, c_2) : \forall n \geq n_0 : 0 \leq c_1.F(n) \leq f(n) \leq c2.F(n)\}$

Nó dựa trực tiếp từ định nghĩa rằng $\Theta(F) = O(F) \cap \Omega(F)$. Nghĩa là, $\Theta(F)$ chứa tất cả các hàm phát triển nhanh như F (lên đến một cấp số nhân không đổi).

Các ký hiệu sau ít được sử dụng hơn (lưu ý sự bất đẳng thức nghiêm ngặt trong định nghĩa.):

Định nghĩa 1.29. $o(F(n)) = \{f(n)|\forall c(c > 0), \exists n_0(c) : n > n_0 : 0 \leq f(n) < c.F(n)\}$

Nó dựa trực tiếp từ định nghĩa rằng $o(F) = O(F) \setminus \Theta(F)$. Nghĩa là $o(F)$ chứa tất cả các hàm *phát triển chậm* hơn F (lên đến một cấp số nhân không đổi).

Lưu ý rằng trong khi $3n^2 \in O(n^2)$ và $3n^2 \in O(n^3)$, thì $3n^2 \in o(n^3)$, nhưng $3n^2 \neq o(n^2)$.

Định nghĩa 1.30. $\omega(F(n)) = \{f(n)|\forall c(c > 0), \exists n_0(c) : n > n0 : 0 \leq c.F(n) < f(n)\}$

Nó dựa trực tiếp từ định nghĩa rằng $\omega(F) = \Omega(F) \setminus \Theta(F)$. Nghĩa là, $\omega(F)$ chứa tất cả các hàm *phát triển nhanh* hơn F (lên đến một cấp số nhân không đổi).

Trong các phần tiếp theo, chúng ta sẽ xem xét các thuộc tính hữu ích và ví dụ về cách chúng ta có thể đánh giá độ phức tạp của một thuật toán trong thực tế.

Dưới đây chúng ta sẽ sử dụng ký hiệu chuẩn để chỉ ra thuộc tập "$\in$" để chỉ ra rằng $f \in \xi(F)$, trong đó ξ là một trong những hàm tiệm cận trên. Tuy nhiên, khi nào thuận tiện, chúng ta sẽ thay bằng dấu bằng. Mặc dù điều này thoạt nghe có vẻ lạ, nhưng nó mang lại cho chúng ta lợi thế là có thể viết các phụ thuộc lặp lại cho hàm tiệm cận $T(n)$ có dạng:

$$T(n) = T(n-1) + O(n)$$

Bài tập

▷ **1.110.** Chứng minh rằng $\Theta(F) = O(F) \cap (F)$.

▷ **1.111.** Chứng minh rằng $o(F) = O(F) \setminus \Theta(F)$.

▷ **1.112.** Chứng minh rằng $\omega(F) = \Omega(F) \setminus \Theta(F)$.

1.13.3. Tính chất và ví dụ của $O(F)$

Cho đến gần đây, ký hiệu $O(F)$ được sử dụng phổ biến nhất để đánh giá độ phức tạp của các thuật toán và chương trình. Gần đây, một ước tính chính xác hơn về $\Theta(F)$ đã được ưa chuộng hơn, nhưng điều này đòi hỏi nỗ lực bổ sung trong việc phân tích các thuật toán và vẫn chưa phải là một cách tiếp cận được chấp nhận rộng rãi. Trong các chương tiếp theo, chúng ta thường sử dụng ký hiệu $\Theta(...)$ và chỉ trong các trường hợp đặc biệt - các ký hiệu khác.

Một số tính chất của O (F):

1. Tính phản xạ: $f \in O(f)$
2. Tính bắc cầu: $f \in O(g), g \in O(h) \Rightarrow f \in O(h)$
3. Tính đối xứng chuyển vị: $f \in \Omega(g) \Leftrightarrow g \in O(f)$
4. Có thể bỏ qua các hằng số: Với mỗi $k > 0, k.F \in O(F)$.
5. n, nâng cao hơn, phát triển nhanh hơn: $n^r \in O(n^s)$, với $0 \leq r \leq s$.
6. Sự gia tăng của tổng các hàm được xác định bởi tiệm cận nhanh nhất tăng dần (điều này được biểu thị bằng max):

   $$f + g \in O(max(f, g)),$$

   có thể được viết như sau:

   $$f \in O(g) \Rightarrow f + g \in O(g)$$

   hoặc là:

   $$O(c_1 f + c_2 g) \in O(\max(f, g)), \text{ với } c_1, c_2 > 0.$$

7. Nếu $f(n)$ là đa thức bậc d trở xuống thì $f \in O(n^d)$.
8. Tích của các hàm số:

   $$f \in O(F) \text{ và } g \in O(G) \Rightarrow f.g \in O(F.G)$$

Ta có thể chứng minh theo định nghĩa các tính chất trên:

Để khẳng định rằng hàm số f thuộc $O(g)$, cần và đủ để tìm một số tự nhiên n_0 và một hằng số dương c sao cho với mọi tự nhiên $n(n > n_0)$ thì bất phương trình $f(n) \leq cg(n)$.

Tính chất tiệm cận (giả sử tồn tại giới hạn):

1. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} \in \mathbb{R}^* \Rightarrow f(n) \in O(g(n)), g(n) \in O(f(n))$.
2. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin O(f(n))$.
3. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \Rightarrow f(n) \notin O(g(n)), g(n) \in O(f(n))$.

Các thuộc tính này có thể được sử dụng như một cách thay thế để xác minh rằng một hàm thuộc nhóm các hàm kiểu $O(...)$.

Ví dụ 1.6. Xét các hàm $\log n$ và $\sqrt{n}$. Sử dụng quy tắc Lopital, nhận được

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/n}{1/(2\sqrt{n})} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0.$$

Phương trình cuối cùng suy ra là $\log n \in O(\sqrt{n})$, nhưng $\sqrt{n} \notin O(\log n)$.

Ví dụ 1.7. Các ví dụ khẳng định

- $10n \in O(n)$; $10n \in O(n^2)$; $10n \in O(n^4)$;
- $10n \in O(3n^4 - 10n^2 + 7)$; $10n + 3 \in O(n)$;
- $4n^2 - 5n + 2 \in O(n^2)$; $4n^3 + 5n^2 + 5 \in O(n^3)$;
- $\sqrt{n} \in O(n)$; $\log n \in O(\sqrt{n})$.

Ví dụ 1.8. Các ví dụ phủ định

$4n^2 - 5n + 2 \notin O(n^3)$; $\sqrt{n} \notin \Omega(n)$; $\log n \notin \Omega(\sqrt{n})$.

Bài tập

▷ **1.113.** Kiểm tra theo định nghĩa rằng $5n^3 - 5n^2 + 5 \in \Omega(n^3)$.

▷ **1.114.** Chứng minh rằng $\sqrt{n} \in \Omega(n^{0,48})$.

▷ **1.115.** Chứng minh các tính chất tổng quát trên của $\Omega(F)$.

▷ **1.116.** Chứng minh tính chất tiệm cận trên của $\Omega(F)$.

1.13.4. Tính chất và ví dụ về Θ

Một số thuộc tính của $\Theta(F)$***:***

1. Tính phản xạ: $f \in \Theta(f)$
2. Tính bắc cầu: $f \in \Theta(g), g \in \Theta(h)\Theta f \in \Theta(h)$
3. Tính đối xứng: $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$
4. Có thể bỏ qua các hằng số: Với mỗi $k > 0$, $k.F \in \Theta(F)$.
5. Sự gia tăng của tổng các hàm được xác định bởi sự phát triển nhanh nhất của chúng:

   $$f + g \in \max(\Theta(f(n)), \Theta(g(n))),$$

   có thể được viết như sau:

   $$f \in \Theta(g) \Rightarrow f + g \in \Theta(g)$$

   hoặc là:

   $$\Theta(c_1 f(n) + c_2 g(n)) \in \max(\Theta(f(n)), \Theta(g(n))), \text{ với } c_1, c_2 > 0.$$

6. Nếu $f(n)$ là đa thức bậc chính xác d thì $f \in \Theta(n^d)$.

Định nghĩa 1.31. Một quan hệ đồng thời có các thuộc tính phản xạ, đối xứng và tính nhanh là *quan hệ tương đương*.

Chúng ta hãy xác định quan hệ $R =$"thuộc $\Theta(...)$" được cho dưới dạng $(f, g) \in R$ nếu và chỉ khi $f \in \Theta(g)$. Từ các tính chất của $\Theta(...)$ và từ định nghĩa 1.3.2. theo đó R là một quan hệ tương đương.

Chứng minh các tính chất trên theo định nghĩa:

Để khẳng định rằng hàm số f thuộc $\Theta(g)$, cần và đủ để tìm một số tự nhiên n_0 và hai hằng số dương c_1 và c_2 sao cho với mọi số tự nhiên $n(n > n_0)$ thì nó có giá trị là bất đẳng thức $c_1.g(n) \leq f(n) \leq c_2.g(n)$.

Ví dụ 1.9. Lấy ví dụ hàm $\frac{n^2}{2} - 3n$. Ta sẽ chứng tỏ rằng nó thuộc tập hàm $\Theta(n^2)$. Ta tìm các hằng số c_1, c_2 và n_0 sao cho $c_1.n^2 \leq n^2/2 - 3n \leq c_2.n^2 (n > n_0)$. Chia cho n^2 (với ràng buộc bổ sung $n > 0$) ta được: $c_1 \leq 1/2 - 3/n \leq c_2$. Một lựa chọn có thể là $c_1 = 0, c_2 = 1/2$

và $n_0 = 5$. Chúng ta để độc giả thực hiện kiểm tra tương ứng. Chúng ta sẽ lưu ý rằng các giá trị c_1, c_2 và n_0 này không có nghĩa là duy nhất. Ví dụ, chúng ta có thể chọn $c_1 = 1/20, c_2 = 1$ và $n_0 = 100$. Chúng phụ thuộc nhiều vào các hệ số của đa thức đã xét và đối với đa thức khác trong trường hợp chung chúng sẽ khác nhau đáng kể.

Ví dụ 1.10. Xét hàm số $2n^3$. Ta sẽ chứng tỏ rằng nó không thuộc tập các hàm)$\Theta(n^2)$. Thật vậy, nếu giả sử ngược lại thì phải tồn tại một hằng số c_2 sao cho $6n^3 \leq c_2.n^2$. Nhưng nếu chúng ta chia cho n^2 (với giới hạn bổ sung $n > 0$) hai vế của bất đẳng thức, chúng ta nhận được bất đẳng thức $6n \leq c_2$, và do đó: $n \leq c_2/6$. Vì vậy, nó chỉ ra rằng n nhỏ hơn một số hằng số. Rõ ràng, không thể tìm thấy n sao cho với mọi $n, n > n_0, n \leq c_2/6$ thỏa mãn: bên trái của bất đẳng thức ta có số không giới hạn ở đỉnh và bên phải là hằng số.

Tính chất tiệm cận:

1. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} \in \mathbb{R}^* \Rightarrow f(n) \in \Theta(g(n)), g(n) \in \Theta(f(n))$.
2. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin \Theta(f(n))$, nghĩa là $f(n) \in o(g(n))$.
3. $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \Omega(g(n)), f(n) \notin \Theta(g(n))$, nghĩa là $f(n) \in \omega(g(n))$.

Các thuộc tính này có thể được sử dụng như một cách thay thế để xác minh rằng một hàm thuộc về một lớp hàm cụ thể.

Ví dụ 1.11. Đánh giá khẳng định.
$10n \in \Theta(n)$; $4n^2 - 5n + 2 \in \Theta(n^2)$;
$4n^2 - 5n + 2 \in \Theta(6n^2 - n + 1)$; $\log_{10} n \in \Theta(log_2 n)$.

Ví dụ 1.12. Đánh giá phủ định.
$10n \notin \Theta(1)$; $10n \notin \Theta(n^2)$;
$4n^2 - 5n + 2 \notin \Theta(n^3)$; $4n^2 - 5n + 2 \notin \Theta(n + 1)$.

Định nghĩa 1.32. Độ phức tạp $\Theta(1)$ được gọi là *hằng số*, $\Theta(\log n)$ - *logarit*, $\Theta(n)$ - *tuyến tính*, $\Theta(n^2)$ - *bậc hai*, $\Theta(c^n)$ - *hàm mũ*. Khi hàm f là một đa thức thì độ phức tạp $\Theta(f)$ được gọi là *đa thức*.

Bài tập

▷ **1.117.** Kiểm tra theo định nghĩa rằng $5n^3 - 5n^2 + 5 \in \Theta(n^3)$.

▷ **1.118.** Chứng minh rằng $\sqrt{n} \notin \Theta(n^{0,48})$.

▷ **1.119.** Chứng minh các tính chất tổng quát trên của $\Theta(F)$.

▷ **1.120.** Chứng minh tính chất tiệm cận trên của $\Theta(F)$.

▷ **1.121.** Kiểm tra xem quan hệ $R =$"có thuộc $\Theta(...)$" (được cho dưới dạng $(f,g) \in R$ nếu và chỉ khi $f \in \Theta(g)$) là quan hệ tương đương.

▷ **1.122.** Chỉ ra các khẳng định đúng:
a) $4n^3 + 5n^2 + 5 \in \Omega(n^4)$;
b) $\log_2 n \notin \Omega(\sqrt{n})$;
c) $4n^3 + 5n^2 + 5 \in \Omega(7n^4 - 10)$;
d) $n \in \Omega(\sqrt{n})$;
e) $\log_2 n \in O(\sqrt{n})$;
f) $10n + 3 \in O(n)$;
g) $10n \in O(3n^4 - 10n^2 + 7)$;
h) $5n + 1 \in O(\sqrt{n})$.

▷ **1.123.** Chỉ ra các khẳng định đúng:
a) $\Theta(F) = O(F) \cap \Omega(F)$;
b) $\omega(F) = \Omega(F) \setminus \Theta(F)$;
c) $o(F) = O(F) \setminus \Theta(F)$;
d) $O(c_1 f + c_2 g) = O(\max(f,g))$;
e) $f \in \Omega(g) \Rightarrow g \in \Theta(f)$;
f) $f \in O(g) \Rightarrow g \in \Omega(f)$;
g) $f \in \Omega(g) \Leftrightarrow g \notin O(f)$.

▷ **1.124.** Chỉ ra các khẳng định đúng:
a) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) \in O(g(n)), g(n) \in O(f(n))$;
b) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin O(f(n))$;
c) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = +\infty \Rightarrow f(n) \notin O(g(n)), g(n) \in O(f(n))$;
d) $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) \in \Theta(g(n))$;

e) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in \Omega(g(n)), g(n) \notin \Theta(f(n))$.

1.13.5. Hàm tiệm cận và số thực

Các quan hệ giữa các hàm tiệm cận được thảo luận ở trên có các tính chất gần với các quan hệ cổ điển trên các số thực. Các liên kết được cho trong Bảng 1.6.

Tuy nhiên, mối quan hệ chỉ là một phần, vì các hàm tiệm cận không có một tính chất quan trọng của số thực: tam phân. Nghĩa là, với mọi số thực a và b có đúng một trong các quan hệ hợp lệ: $a < b, a = b$ và $a > b$. Ví dụ, đối với các hàm n và $n^{1+\cos n}$ không có quan hệ nào trong số các quan hệ trên là hợp lệ. Chúng ta để lại chứng minh của quan hệ sau cho người đọc.

quan hệ tiệm cận	**quan hệ số thực**
$f \in O(g)$	$a \leq b$
$f \in \Omega(g)$	$a \geq b$
$f \in \Theta(g)$	$a = b$
$f \in o(g)$	$a < b$
$f \in \omega(d)$	$a > b$

Bảng 1.6. Mối liên hệ giữa quan hệ về tiệm cận của hàm số và số thực.

Bài tập

▷ **1.125.** Để đưa ra một ví dụ khác về một cặp hàm số tiệm cận không so sánh được.

▷ **1.126.** Một trong hai hàm $f(n)$ và $g(n)$ có cần thiết là hợp của hai hàm, một trong hai hàm đó phải tuần hoàn, để f và g có tiệm cận không?

Tăng các hàm cơ bản

Các hàm sau đây thường được sử dụng nhất khi ước lượng độ phức tạp của các thuật toán: $c, \log n, n, n\log n, n^2, n^3, n^k, 2^n, n!, n^n$. Ở

đây chúng ta đã sắp xếp chúng để tăng trưởng nhanh hơn. Để người đọc hiểu rõ hơn về tốc độ tăng trưởng của chúng, chúng ta áp dụng Bảng 1.7, Hiển thị các giá trị của các hàm ở các giá trị khác nhau của đối số n của chúng.

Trực tiếp từ Bảng 1.7 người ta thấy rằng:

- Hàm lũy thừa phát triển nhanh hơn hàm lũy thừa:
  $n^k \in O(b^n)$, với mỗi $b > 1, k \geq 0, n$ - tự nhiên.
- Hàm số logarit tăng chậm hơn lũy thừa:
  $\log_b n \in O(n^k)$, với mỗi $b > 1, k > 0, n$ - tự nhiên.

Hàm số	**Giá trị**				
	$n = 1$	$n = 2$	$n = 10$	$n = 100$	$n = 1000$
5	5	5	5	5	5
$\log_n$	0	1	3,32	6,64	9,96
n	1	2	10	100	1000
$n\log n$	0	2	33,2	664	9966
n^2	1	4	100	10000	10^6
n^3	1	8	1000	10^6	10^9
2^n	2	4	1024	10^{30}	10^{300}
$n!$	1	2	3628800	10^{157}	10^{2567}
n^n	1	4	10^{10}	10^{200}	10^{3000}

Bảng 1.7. Tăng một số hàm tiệm cận được sử dụng phổ biến hơn.

Bài tập

▷ **1.127.** Chứng minh rằng $n!$ phát triển nhanh hơn c^n, với mọi $c > 0$.

▷ **1.128.** Sắp xếp hàm $c^n, c > 0$ theo thứ tự tăng dần của các hàm thường dùng nhất (xem Bảng 1.7).

1.13.6. Xác định độ phức tạp của một thuật toán

Chúng ta sẽ xem xét một số thuộc tính cơ bản, với sự trợ giúp của chúng ta sẽ có thể xác định độ phức tạp của một thuật toán đối

với một triển khai C nhất định. Chúng ta sẽ sử dụng T(mã) để chỉ ra mức độ phức tạp của cả một hoạt động riêng lẻ và một đoạn chương trình. Đôi khi, khi mã được ngụ ý, chúng ta sẽ sử dụng ký hiệu $T(n)$ để chỉ ra rõ ràng rằng độ phức tạp là một hàm của biến n. Chúng ta sẽ làm việc độc quyền với ký hiệu $O(...)$. Chúng ta để người đọc tìm ra vị trí có thể thay thế $O(...)$ bằng ước lượng chính xác hơn $\Theta(...)$.

Phép tính cơ bản

Độ phức tạp của một phép toán cơ bản là một hằng số, tức là $O(1)$. Không dễ để định nghĩa một phép toán cơ bản là gì. Trong các trường hợp khác nhau, chúng ta sẽ cho phép mình thay đổi định nghĩa. Về nguyên tắc, hoạt động cơ bản là hoạt động được thực hiện trong một thời gian không đổi, bất kể số lượng dữ liệu được xử lý. Các phép toán cơ bản trong trường hợp chung là, ví dụ, chiếm đoạt, cộng, nhân, v.v. Tuy nhiên, khi làm việc với các số có 100 chữ số, thật khó chấp nhận phép nhân như một phép toán cơ bản. Sẽ là không tốt nếu sử dụng các hàm lượng giác ($\sin, \cos$, v.v.), số mũ, logarit và các hàm thư viện khác trong C, được tính theo hàng trong C là không tốt cho các phép toán cơ bản. Việc gọi một hàm như vậy gây ra một chu kỳ để tính giá trị cần thiết. Mặt khác, các hàng này được tính toán cho đến khi đạt được độ chính xác nhất định, và số lần lặp để tính giá trị cần thiết có thể được coi là độc lập với n. Điều này có ảnh hưởng đến việc đánh giá độ phức tạp của thuật toán hay không và như thế nào thì tùy thuộc vào từng trường hợp và cần được xem xét cụ thể.

Trình tự các toán tử

Độ phức tạp về thời gian của một chuỗi các toán tử được xác định bởi một toán tử chậm hơn. Nếu toán tử s_1 với độ phức tạp F_1 được theo sau bởi toán tử s_2 với độ phức tạp F_2, chúng ta có thể viết:

$$T(s_1) \in O(F_1), T(s_2) \in O(F_2) \Rightarrow T(s_1; s_2) \in O(\max(F_1, F_2))$$

Điều này tương đương với quy tắc:

$$f_1 + f_2 \in (\max(f_1, f_2))$$

Kết hợp các toán tử

Khi một toán tử được bao gồm trong phạm vi của một toán tử khác, độ phức tạp được tính như một tích của độ phức tạp của

chúng, tức là

$$T(s_1) \in O(F_1), T(s_2) \in O(F_2) \Rightarrow T(s_1\{s_2\}) \in O(F_1.F_2))$$

Điều này tương đương với quy tắc:

$$f_1.f_2 \in O(f_1.f_2)$$

Cấu trúc mệnh đề if

```
if (p)
   s1;
else
   s2;
```

Nếu độ phức tạp của p, s_1 và s_2 là $O(P), O(F_1), O(F_2)$, thì độ phức tạp của đoạn được hiển thị là tối đa $(O(P), O(F_1), O_2))$, tức là độ phức tạp của hàm phát triển nhanh nhất giữa P, F_1 và F_2. Ở đây chúng ta giả sử rằng điều kiện p không phải là hằng số Boolean, tức là tùy thuộc vào đầu vào, nó có thể vừa đúng vừa sai.

$$T(p) \in O(P), T(s_1) \in O(F_1), T(s_2) \in O(F_2)$$

$$\Rightarrow T(\texttt{if (p) } s_1\texttt{; else } s_2) \in \max(O(P), O(F_1), O(F_2))$$

Chúng ta sẽ cố gắng chứng minh tài sản này. Giả sử rằng điều kiện p là đúng. Khi đó *cấu trúc if* tương đương với dãy $p; s_1$ (Lưu ý rằng điều kiện đã được kiểm tra.). Theo quy tắc của trình tự, độ phức tạp trong trường hợp này là $\max(O(P), O(F_1))$. Theo cách tương tự, chúng ta hiểu rằng độ phức tạp trong trường hợp p sai là cực đại $(O(P), O(F_2))$. Vì không biết trước chuỗi nào trong hai chuỗi sẽ được thực thi, chúng ta có thể hạn chế mức độ phức tạp của *cấu trúc if* từ phía trên, giả sử trường hợp nghiêm trọng hơn. Đây là cách chúng ta nhận được:

$$T(\texttt{if (p) } s_1\texttt{; else } s_2) \in \max(\max(O(P), O(F_1)), \max(O(P), O(F_2)))$$

$$= \max(O(P))), O(F_1), O(F_2))$$

Theo cách tương tự, sự phức tạp của các cấu trúc chuyển mạch được suy ra.

Chu trình lặp

Hãy nhìn vào chu kỳ:

```
fact = 1;
  for (i = 1; i <= n; i++)
    fact *= i;
```

Chúng ta có thể giả sử rằng phần thân của chu trình mất một khoảng thời gian c không đổi, không phụ thuộc vào n. Độ phức tạp của toán tử vòng lặp for là $O(n)$. Sau đó, theo quy tắc thành phần về độ phức tạp của toàn bộ chu trình, chúng ta thu được $O(c.n)$, tức là $O(n)$. Ở đây chúng ta phải thêm độ phức tạp của lần khởi tạo ban đầu trước chu trình (có độ phức tạp $O(1)$), trong đó, theo quy tắc trình tự, chúng ta nhận được: $O(1+n)$. Cuối cùng, độ phức tạp hóa ra là $O(n)$.

Chu trình lồng nhau

```
sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    sum++;
```

Có thể dễ dàng suy ra độ phức tạp của hai hoặc nhiều chu trình lồng nhau với các bộ đếm độc lập lẫn nhau. Trong trường hợp hai chu trình lồng nhau từ đoạn trên, nó là $f \in n.O(g)$, trong đó g là độ phức tạp của chu trình bên trong. Nhưng $g \in O(n)$, thì $f \in O(n.n)$, tức là $f \in O(n^2)$.

Ở đây và bên dưới, chúng ta thường khởi tạo một tổng biến đặc biệt bằng 0, và sau đó chúng ta sẽ sử dụng sum++ làm toán tử bên trong nhất trong các vòng lặp. Điều này cho phép người đọc tò mò kiểm tra lý thuyết lý thuyết của chúng ta trong thực tế. Với mục đích này, chỉ cần quan sát giá trị của sum sau khi thực hiện phân mảnh đối với các giá trị khác nhau của n là đủ.

```
sum = 0;
for (i = 0; i < n-1; i++)
  for (j = i+1; j < n; j++)
    sum++;
```

Trong ví dụ này, trong bước đầu tiên của chu kỳ ngoài, $i = 0$, chu kỳ bên trong sẽ được thực hiện $n-1$ lần. Trong bước thứ hai, đối với $i = 1$, chu trình bên trong sẽ được thực hiện $n-2$ lần, sau

đó là $n-3$ lần, v.v., và cuối cùng, đối với $i = n-2$, nó sẽ chỉ được thực hiện một lần. Ta có một cấp số cộng với phần tử đầu tiên 1, phần tử cuối cùng $n-1$ và bước 1. Như vậy sum++ sẽ được thực hiện $\frac{n(n-1)}{2}$ lần, tức là độ phức tạp của phân mảnh sẽ là $O(n^2)$. Chúng ta giả sử rằng việc thực thi sum ++ có độ phức tạp không đổi $O(1)$.

Ví dụ 1.13. Đoạn mã

```
sum = 0;
for (i = 0; i < n*n; i++)
   sum++;
```

Độ phức tạp là $O(n^2)$. (Tại sao?)

Ví dụ 1.14. Đoạn mã

```
sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i == j)
      for (k = 0; k < n; k++)
        sum++;
```

Câu lệnh **if** sẽ được thực hiện n^2 lần, nhưng chỉ n lần kết quả kiểm tra i == j là đúng. Vì độ phức tạp của chu trình trong cùng là tuyến tính nên chúng ta thu được tổng độ phức tạp $O(n^2)$. Tất nhiên, ước lượng $O(n^3)$ cũng đúng, nhưng nó không chính xác hơn. Độ phức tạp ở đây là $\Theta(n^2)$.

Ví dụ 1.15. Đoạn mã

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i == j)
      break;
```

Làm thế nào nó có thể được chỉ ra rằng độ phức tạp trong đoạn trên một lần nữa là $O(n^2)$ chứ không phải ít hơn? Lưu ý rằng, không giống như ví dụ trước, không có chu trình tuyến tính bổ sung!

Ví dụ 1.16. Đoạn mã

```
sum = 0;
for (i  = 1;  i  <= n;  i++)
  for (j  = 1;  j  <= i*i;  j++)
    sum++;
```

Chu trình bên trong sẽ được thực hiện n lần. Từ công tức

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

chúng ta nhận được rằng độ phức tạp của phân mảnh là $O(n^3)$. Trên thực tế, chúng ta có thể quyết định rằng độ phức tạp của chu trình ngoài là $O(n)$ và độ phức tạp của chu trình trong là $O(n^2)$, bởi vì giá trị lớn nhất của i là n. Do đó với $i == n$ chúng ta có$i * i == n^2$ và một lần nữa chúng ta đạt đến độ phức tạp $O(n^3)$.

Ví dụ 1.17. Đoạn mã

```
sum = 0;
for (i  = 0;  i  < n;  i++)
  for (j  = 0;  j  < i*i;  j++)
    for (k = 0;  k < j*j;  k++)
      sum++;
```

Một lần nữa, có thể tính toán độ phức tạp bằng cách phân tích số lượng hoạt động trong chu kỳ trong cùng. Tuy nhiên, trong thực tế, có thể đạt được kết quả tương tự và dễ dàng hơn nhiều, chỉ sử dụng các thuộc tính cơ bản của ký hiệu $O(...)$: Trong ví dụ trước, chúng ta đã tính toán độ phức tạp của hai chu trình lồng nhau bên ngoài - $O(n^3)$. Bản thân chu trình cuối cùng có độ phức tạp $O(n^4)$ - điều này dễ thấy nếu chúng ta xét trường hợp $i = n$, thì giới hạn trên của k là $j^2 = i^4 = n^4$. (Độ phức tạp tổng thể của phân mảnh là gì và tại sao?)

Độ phức tạp lôgarit

Hãy xem phân đoạn chương trình:

```
cho (sum = 0, h = 1;  h <n; h *  = 2)
tổng ++;
```

Ở đây h nhận các giá trị $1, 2, 4, ..., 2^k, ...$ cho đến khi nó đạt đến n. Do đó, sum ++ được thực hiện $\lfloor \log_2 n \rfloor$ lần và độ phức tạp của thuật toán là $O(log_2 n)$. Lưu ý rằng độ phức tạp được đặt bởi một logarit, là một hàm thực. Như đã lưu ý ở trên, đây không phải là vấn đề và chúng ta sẽ hiểu được xấp xỉ / ràng buộc số nguyên thích hợp.

Tìm kiếm nhị phân có độ phức tạp thuật toán như vậy, trong đó ở mỗi bước, khoảng *thời gian tìm kiếm* được chia thành hai (gần như) phần bằng nhau. Chúng ta sẽ không đề cập đến thuật toán này, vì chúng ta sẽ xem xét nó chi tiết hơn trong phần sau của cuốn sách (xem 4.3.).

Dưới đây, thay vì $O(\log_2 n)$, chúng ta sẽ viết $O(\log n)$. Một mặt, bỏ cơ số ở lôgarit nhị phân là một quy ước tiêu chuẩn. Mặt khác, nó không quan trọng trong ký hiệu tiệm cận do tính chất (4) của 1.1.1. độ, logarit, căn bậc n. Nếu chúng ta thích làm việc với cơ sở khác $c (c > 0, c \neq 1)$ thì không có gì thay đổi, bởi vì $\log_2 x = \log - cx / \log_c 2$, nhưng $\log_c 2$ là một hằng số và bị bỏ qua trong ký hiệu tiệm cận. Như vậy $O(\log_2 n) \equiv O(log_c n)$.

Thuật toán đệ quy

Phân tích đệ quy trong trường hợp tổng quát là không hề nhỏ. Thông thường, độ phức tạp của thuật toán phụ thuộc vào kiểu $T(n) = f(T(n-1))$. Để tìm ra loại phức tạp rõ ràng, cần phải giải quyết sự phụ thuộc thường xuyên, nói chung là khó. May mắn thay, trong hầu hết các trường hợp thực tế thú vị, điều này không quá khó và đôi khi có thể được thực hiện bằng các phương tiện khác.

Phép tính giai thừa

Hãy xem xét chức năng:

```
unsigned fact(unsigned n)
{ if (n < 2)
    return 1;
  return n*fact(n-1);
}
```

Trong trường hợp này, đệ quy tương đương với một vòng lặp for duy nhất, từ đó chúng ta có thể dễ dàng thu được $O(n)$ về độ phức tạp.

Tính số Fibonacci

Tuy nhiên, mọi thứ không phải lúc nào cũng đơn giản như vậy. Lấy ví dụ, các số Fibonacci mà chúng ta đã hiển thị trong 1.2.2 có thể được tìm thấy cực kỳ kém hiệu quả với đệ quy:

```
unsigned fib(unsigned n)
{
  if (n < 2)
     return 1;
  return fib(n-1) + fib(n-2);
}
```

Tại $n = 0$ và $n = 1$, chúng ta có độ phức tạp về thời gian không đổi: một lần kiểm tra cơ bản và trả về một kết quả. Trong trường hợp khác, chúng ta có kiểm tra lại tương tự, nhưng lần này được theo sau bởi hai tham chiếu đệ quy. Nói chung, các công thức hợp lệ:

```
T(0) = T(1) = O(1)
T(n) = T(n - 1) + T(n - 2) + O(1), n\ge 2
```

Các phụ thuộc trên rất giống với định nghĩa của số Fibonacci:

$$f_0 = f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Ngay sau đó $T(n) \geq f_n$. Tuy nhiên, đối với số Fibonacci, bất đẳng thức có giá trị: $\left(\frac{3}{2}\right)^{n-1} \leq f_n \leq 2^n, n \geq 1$ (Chứng minh một chút bằng quy nạp.). Do đó, nó chỉ ra rằng $T(n)$ phát triển theo cấp số nhân.

Tuy nhiên, nếu chúng ta đủ tháo vát để không tính toán lại một số thứ mà chúng ta đã xem xét, chúng ta có thể giảm độ phức tạp của thuật toán xuống $O(n)$:

```
unsigned long f[MAX] = {0,0,0,... };
unsigned long fib(unsigned n)
{
  if (0 == f[n])
    if (n < 2)
      f[n] = 1;
```

```
    else
      f[n] = fib(n-1) + fib(n-2);
    return f[n];
}
```

Bài tập

▷ **1.129.** Nhớ lại rằng đối với việc xây dựng **if**-then-**else** chúng ta đã có

$$T(p) \in O(P), T(s_1) \in O(F_1), T(s_2) \in O(F_2)$$

$$\Rightarrow T(\texttt{if } (p)s_1\texttt{;else } s_2) \in \max(O(P), O(F_1), O(F_2)).$$

Tìm một biểu thức thuộc loại này cho công tắc cấu trúc `switch`.

▷ **1.130.** Chứng minh rằng độ phức tạp của đoạn chương trình từ Ví dụ 1.13 ở trên là $\Theta(n^2)$.

▷ **1.131.** Chứng minh rằng độ phức tạp của đoạn chương trình từ Ví dụ 1.15 ở trên là $\Theta(n^2)$.

▷ **1.132.** Để đưa ra ước tính kiểu $\Theta(...)$ cho độ phức tạp của đoạn chương trình từ Ví dụ 1.17.

▷ **1.133.** Có đúng không khi thay thế $O(...)$ bằng $\Theta(...)$ trong suốt 1.4.4.?

▷ **1.134.** Chứng minh rằng đối với các số Fibonacci thì đẳng thức tồn tại:

$$\left(\frac{3}{2}\right)^{n-1} \leq f_n \leq 2^n, n \geq 1$$

▷ **1.135.** Xác định độ phức tạp $\Theta(...)$ của đoạn sau:

```
for (i = 0; i < 2*n; i++)
  for (j = 0; j < 2*n; j++)
    if (i<j) for (k = 0; k < 2*n; k++)
      break;
```

▷ **1.136.** 7. Xác định độ phức tạp $\Theta(...)$ của đoạn sau:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
  for (j = i+1; j < i*i; j++)
    sum++;
```

▷ **1.137.** Xác định độ phức tạp $\Theta(...)$ của đoạn sau:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i==j)
      for (k = 0; k < n; k++)
        break;
```

▷ **1.138.** Xác định độ phức tạp $\Theta(...)$ của đoạn sau:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++)
    sum++;
```

▷ **1.139.** Xác định độ phức tạp $\Theta(...)$ của hàm trib():

```
unsigned trib(unsigned n)
{
  if (n < 3)
    return 1;
  if ((n % 2) == 1)
    return trib(n-1) + trib(n-2) + trib(n-3);
  else
    return trib(n / 3) + trib(n / 2);
}
```

▷ **1.140.** Xác định độ phức tạp $\Theta(...)$ của hàm streep():

```
unsigned fib(unsigned n)
{
  if (n < 2)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

```
void streep(unsigned n) {
  fib(fib(n));
}
```

▷ **1.141.** Xác định độ phức tạp $\Theta(...)$ của đoạn sau:

```
int n = 10;
int i;
for (i = 0; i < n; (n = i)++);
```

1.14. Phương trình đặc trưng

Hãy quay lại các số Fibonacci. Lần này chúng ta sẽ không tập trung vào sự phức tạp của việc thực hiện đệ quy ngây thơ của đoạn trước, mà là giải quyết sự phụ thuộc lặp lại được thiết lập bởi định nghĩa của số Fibonacci. Kết quả là, chúng ta có được công thức Moavr nổi tiếng. Nhưng trước đó - một lý thuyết nhỏ ...

1.14.1. Phương trình thuần nhất tuyến tính với nghiệm đơn giản

Trong quá trình phân tích độ phức tạp của các thuật toán máy tính, thường thu được các phụ thuộc thường xuyên của kiểu:

$$a_0T(n) + a_1T(n-1) + \cdots + a_kT(n-k) = 0 \tag{1.14}$$

Ở đây $T(n)$ là một hàm chưa biết. Để có thể ước tính độ phức tạp của thuật toán tiệm cận, chúng ta cần tìm toàn bộ hoặc ít nhất một phần dạng rõ ràng của $T(n)$, tức là để giải quyết sự phụ thuộc thường xuyên. Cách tiếp cận tiêu chuẩn là giảm vấn đề thành giải một phương trình thuần nhất tương ứng. Cho $T(n) = x^n$, trong đó ta nhận được:

$$a_0x^n + a_1x^{n-1} + \cdots + a_kx^{n-k} = 0 \tag{1.15}$$

Một nghiệm của phương trình trên, cũng như của bất kỳ phương trình thuần nhất nào, là $x = 0$, không được quan tâm. Để tìm các nghiệm còn lại, ta chia cho $x^n - k$ (với $x \neq 0$) và thu được đa thức bậc k - đa thức đặc trưng của sự phụ thuộc hồi quy ban đầu:

$$a_0x^k + a_1x^{k-1} + \cdots + a_k = 0 \tag{1.16}$$

Phương trình kết quả là một đa thức bậc k. Theo định lý cơ bản của đại số, nó có đúng k căn (không nhất thiết khác và không nhất thiết thực, tức là có thể phức). Hãy ký hiệu chúng bằng $\alpha_i (1 \leq i \leq k)$. Sau đó chúng ta có:

$$0 = a_0 x^k + a_1 x^{k-1} + \cdots + a_k = a_0 (x - \alpha_1)(x - \alpha_2)...(x - \alpha_k) \quad (1.17)$$

Nếu chúng ta nhân phương trình (1.17) với x^{n-k}, chúng ta nhận được rằng các số $\alpha_i (1 \leq i \leq k)$ không chỉ là nghiệm của phương trình (1.16) mà còn của (1.15). Quay trở lại vị trí $T(n) = x^n$, ta thu được α_i^n là nghiệm nguyên của sự phụ thuộc hồi quy ban đầu (1.14).

Phương trình (1.14) thuần nhất và có vô số nghiệm. Nếu các số α_i khác nhau, thì tập hợp các số α_i^n biểu diễn một hệ nghiệm cơ bản của (1.14), tức là tất cả các nghiệm khác thu được dưới dạng kết hợp tuyến tính của α_i^n. Như vậy nghiệm tổng quát của (1.14) có dạng:

$$T(n) = c_1 \alpha_1^n + c_2 \alpha_2^n + \cdots + c_k \alpha_k^n \quad (1.18)$$

Ở đây các hệ số $c_1, c_2, ..., c_n$ là các hằng số được xác định duy nhất bởi các điều kiện biên. Dưới đây chúng ta sẽ xem cách này hoạt động như thế nào trong thực tế.

Ví dụ 1.18. Hãy xem xét các số Fibonacci thỏa mãn sự phụ thuộc lặp lại:

$$T(n) = T(n-1) + T(n-2)$$

Phương trình đặc trưng tương ứng của loại (1.17) là:

$$x^2 = x + 1$$

và gốc rễ của nó là:

$$\alpha_{1,2} = \frac{1 \pm \sqrt{5}}{2}$$

Chúng ta đang tìm kiếm một giải pháp thuộc loại (1.18), tức là:

$$T(n) = c_1 \alpha_1^n + c_2 \alpha_2^n \quad (1.19)$$

Chúng ta xác định các hằng số c_1 và c_2 từ các điều kiện biên $T(0) = 0$ và $T(1) = 1$, thay thế vào (1.19), ở đó chúng ta nhận được

hệ thống:

$$T(0) = 0 = c_1 + c_2$$
$$T(1) = 1 = c_1\alpha_1 + c_2\alpha_2$$

chúng ta rút ra:

$$c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$$

Bây giờ chúng ta thay thế trong (1.19) và thu được dạng rõ ràng của $T(n)$. Công thức kết quả được gọi là công thức Moaver.

$$T(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Công thức trên cho thấy rõ ràng rằng số Fibonacci tăng theo cấp số nhân khi tăng n. Thoạt nhìn, công thức có vẻ lạ - nó chứa các số vô tỉ. Chúng ta giao nó cho người đọc để đảm bảo rằng chúng được viết tắt và với mỗi n tự nhiên sẽ thu được các số Fibonacci tương ứng.

-

1.14.2. Phương trình thuần nhất tuyến tính với nhiều nghiệm

Trong trường hợp phương trình (1.17) có nhiều nghiệm, nghiệm tổng quát có dạng hơi khác so với (1.18). Nếu α là căn kép thì căn đó cũng sẽ là $n\alpha^n$. Lưu ý rằng gốc này không thể nhận được dưới dạng kết hợp tuyến tính của các gốc khác, tức là nó về cơ bản là mới và phải được đưa vào hệ thống cơ bản của các quyết định. Nếu căn bậc ba thì ngoài $n\alpha^n$ ta nên thêm $n^2\alpha^n$. Nói chung, trong trường hợp một căn bậc hai, các căn α sẽ là tất cả các đơn thức $n^s\alpha^n, 0 \le s \le p$.

Đổi tên các gốc $\alpha_i (1 \le i \le k)$ để loại bỏ nhiều gốc. Gọi các nghiệm nguyên khác nhau (1.16) là $\beta_1, \beta_2, ..., \beta_l, 1 \le l \le k$. Gọi q_j là bội của nghiệm $\beta_j, 1 \le j \le l$. Khi đó, nghiệm chung của (1.14) sẽ giống như sau:

$$T(n) = \sum_{j=1}^{l}\sum_{r=0}^{q_j-1} c_{j,r+1} n^r \beta_j^n. \tag{1.20}$$

Các số $c_{j,r+1}$ là các hệ số được xác định rõ ràng, như trên, theo giới hạn các điều kiện.

Ví dụ 1.19.

$$T(n) = \begin{cases} 3 & n = 0; 1 \\ 5 & n = 2 \\ 4T(n-1) - 5T(n-2) + 2T(n-3) & n \geq 2 \end{cases}$$

Phương trình đặc tính tương ứng có dạng:

$$x^3 = 4x^2 - 5x + 2$$

Hoặc, nếu chúng ta chuyển mọi thứ sang bên trái:

$$x^3 - 4x^2 + 5x - 2 = 0$$

Giải quyết nó, chúng ta nhận được:

$$x_1 = x_2 = 1, x_3 = 2$$

Chúng ta tìm kiếm dạng rõ ràng của $T(n)$ ở dạng (1.20):

$$T(n) = c_{1.1}.1^n + c_{1.2}.n.1^n + c_{2.1}.2^n \tag{1.21}$$

Chúng ta xác định các hằng số $c_{1,1}, c_{1,2}$ và $c_{2,1}$ từ các điều kiện biên cho $n = 0, 1$ và 2. Chúng ta nhận được hệ thống:

$$\begin{aligned} T(0) &= 3 = c_{1.1} + c_{2.1} \\ T(1) &= 3 = c_{1.1} + c_{1.2} + 2c_{2.1} \\ T(2) &= 5 = c_{1.1} + 2c_{1.2} + 4c_{2.1} \end{aligned}$$

ở đâu:

$$c_{1.1} = 1, c_{1.2} = -2, c_{2.1} = 2$$

Bây giờ chúng ta thay thế trong (1.21) và nhận được:

$$T(n) = 1 - 2n + 2^{n+1}$$

1.14.3. Phương trình tuyến tính không thuần nhất

Trong quá trình phân tích thuật toán thường phải phân tích các phương trình không thuần nhất. Hãy xem xét nhiệm vụ sau:

Bài toán: Hàm của C được đưa ra:

```
float P(unsigned i, unsigned j)
{ if (0 == i)
    return 1.0;
  else if (0 == j)
    return 0.0;
  else
    return p * P(i - 1, j) + (1 - p) * P(i, j - 1);
}
```

Chúng ta muốn ước tính độ phức tạp về thời gian của phép nghịch đảo P (n, n) dưới dạng một hàm của n.

Lời giải:

Chúng ta sẽ giải quyết vấn đề tổng quát hơn về ước lượng độ phức tạp của hàm đối với các tham số i và j, và sau đó chúng ta sẽ xem điều gì xảy ra với $i = j = n$.

Dễ dàng thấy rằng hàm P( ) ở trên tính giá trị của hàm $P(i, j)$ cho bởi các phương trình truy hồi:

$$\begin{aligned} P(0, j) &= 1, j = 1, 2, \ldots, n \\ P(i, 0) &= 0, i = 1, 2, \ldots, n \\ P(i, j) &= p.P(i-1, j) + (1-p).P(i, j-1), i > 0, j > 0 \end{aligned}$$

Ở đây độ phức tạp phụ thuộc vào hai tham số nên khó tính toán, tuy nhiên có thể dễ dàng thấy rằng mọi thứ đối xứng với i và j. Do đó, giả sử $k = i + j$, chúng ta thu được:

$$\begin{aligned} T(1) &= c \\ T(k) &= 2T(k-1) + d, k > 1 \end{aligned}$$

Lưu ý rằng ở đây chúng ta không cố gắng tìm dạng rõ ràng của hàm $P(i, j)$, mà để tìm ước lượng tiệm cận của việc thực hiện chương trình của hàm P(). Điều này cho chúng ta quyền không đi sâu quá giá trị của các hằng số c và d, trừ khi nó thực sự cần thiết. Chúng ta

sẽ thấy bên dưới rằng trong trường hợp của chúng ta, các *giá trị* cụ thể của chúng không liên quan.

Trước tiên, chúng ta hãy xem xét một cách đơn giản để có được phương trình đặc trưng trong trường hợp của chúng ta, trong đó một hằng số có liên quan đến quan hệ truy hồi. Chúng ta ghi lại sự phụ thuộc lặp lại cho hai phần tử liên tiếp của chuỗi:

$$T(k) = 2T(k-1) + d$$
$$T(k+1) = 2T(k) + d$$

Chúng ta lấy chúng ra theo nhóm, trong đó hằng số được giảm xuống, và chúng ta nhận được:

$$T(k+1) - T(k) = 2.[T(k) - T(k-1)]$$

tương đương với:

$$T(k+1) - 3T(k) + 2T(k-1) = 0.$$

Phương trình đặc trưng tương ứng là:

$$x^2 - 3x + 2 = 0$$

hoặc là

$$(x-1)(x-2) = 0,$$

trong đó người ta thấy trực tiếp nghiệm của phương trình là $\alpha_1 = 2$ và $\alpha_2 = 1$. Tức là chúng ta tìm kiếm dạng rõ ràng của T (k) trong biểu mẫu:

$$T(k) = c_1\alpha_1^k + c_2\alpha_2^k$$

Hoặc sau khi thay thế:

$$T(k) = c_1 2^k + c_2$$

Các hệ số c_1 và c_2 có thể được tìm thấy bằng cách sử dụng các giá trị của $T(1)$ và $T(2)$. Chúng ta nhận được hệ thống:

$$T(1) = c = 2c_1 + c_2$$
$$T(2) = 2c + d = 4c_1 + c_2$$

ở đâu: $c_1 = (c+d)/2; c_2 = -d$.

Từ $c_1 \neq 0$ lập tức có $T(k) \in \Theta(2^k)$. (Tại sao?) Bây giờ chúng ta hãy quay lại ứng dụng $k = i + j$, nơi chúng ta nhận được $\Theta(2^{i+j})$. Chúng ta hãy nhớ lại rằng chúng ta muốn ước tính độ phức tạp của đoạn chương trình trong địa chỉ $P(n,n)$, tức là cho $i = j = n$. Do đó ta nhận được $\Theta(2^{2n})$ hay cuối cùng là: $\Theta(4^n)$.

Định lý 1.9. *Xét phương trình không thuần nhất:*

$$a_0T(n) + a_1T(n-1) + \cdots + a_kT(n-k) = b_1np_1(n) + b_2np_2(n) + \cdots + bs^np_s(n) \tag{1.22}$$

Ở đây $b_1, b_2, ..., b_s$ là các hằng số thực khác nhau và $p_1(n), p_2(n), ..., p_s(n)$ là các đa thức của n. Gọi lũy thừa của các đa thức lần lượt là $d_1, d_2, ..., d_s$. Khi đó phương trình đặc trưng của sự phụ thuộc hồi quy (1.22) có dạng:

$$(a_0x^k + a_1x^{k-1} + \cdots + a_k)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1}...(x-b_s)^{d_s+1} = 0 \tag{1.23}$$

Phương trình kết quả (1.23) được tiếp tục giải như trong trường hợp thuần nhất.

Ví dụ 1.20. Xem xét sự phụ thuộc thường xuyên:

$$T(n) = \begin{cases} 0 & n = 0 \\ 2T(n-1) + n + 2^n & n \geq 1 \end{cases}$$

Rõ ràng đây là sự phụ thuộc lặp lại không đồng nhất:

$$T(n) - 2T(n-1) = n + 2^n, \tag{1.24}$$

như $b_1 = 1, b_2 = 2, p_1(n) = n, p_2(n) = 1$, và do đó: $d_1 = 1, d_2 = 0$. Khi đó phương trình đặc trưng tương ứng có dạng:

$$(x-2)(x-1)^2(x-2) = 0.$$

Nghiệm của phương trình là $x_1 = x_2 = 1, x_3 = x_4 = 2$. Chúng ta tìm dạng tường minh của $T(n)$:

$$T(n) = c_{1.1}.1^n + c_{1.2}.n.1^n + c_{2.1}.2^n + c_{2.2}.n.2^n \tag{1.25}$$

Để tìm các hằng số, lúc này điều kiện biên sẽ không đủ. Cần tính giá trị của $T(n)$ cho ba giá trị tiếp theo. Đây là cách chúng ta có được hệ thống:

$$\begin{aligned}
T(0) &= 0 = c_{1.1} + c_{2.1} \\
T(1) &= 3 = c_{1.1} + c_{1.2} + 2c_{2.1} + 2c_{2.2} \\
T(2) &= 12 = c_{1.1} + 2c_{1.2} + 4c_{2.1} + 8c_{2.2} \\
T(3) &= 35 = c_{1.1} + 3c_{1.2} + 8c_{2.1} + 24c_{2.2}
\end{aligned}$$

Chúng ta giải nó và nhận được $c_{1,1} = -2, c_{1,2} = -1, c_{2,1} = 2$ và $c_{2,2} = 1$. Bây giờ chúng ta thay thế vào (1.25) và cho $T(n)$ chúng ta cuối cùng đến:

$$T(n) = -2 - n + 2^{n+1} + n.2^n. \tag{1.26}$$

Hệ thống trên có bốn ẩn số, khiến nó khó giải quyết (mặc dù không quá khó). Chúng ta có thể tìm các hệ số theo một cách khác - bằng cách thay (1.25) cho (1.24), chúng ta nhận được:

$$c_{2,2}2^n - c_{1,2}n + (2c_{1,2} - c_{1,1}) = 2^n + n. \tag{1.27}$$

Cân bằng các hệ số trước 2^n ở cả hai vế của đẳng thức, ta được $c_{2.2} = 1$. Tương tự, cân bằng các hệ số trước n cho ta $c_{1.2} = -1$. Bây giờ, biết $c_{1,2}$, từ phương trình của các số hạng tự do, chúng ta nhận được $c_{1,1} = -2$. Hằng số $c_{1,2}$ không thể đạt được theo cách này, nhưng nó xảy ra trực tiếp nếu chúng ta sử dụng điều kiện biên.

Giả sử rằng phương trình của ví dụ cuối cùng thu được là kết quả của việc phân tích một số thuật toán. Điều chúng ta quan tâm trong trường hợp này không phải là dạng rõ ràng của $T(n)$ như là một ước lượng tiệm cận của nó. Tất nhiên, khi biết dạng tường minh (1.26) của $T(n)$, chúng ta ngay lập tức nhận được rằng $T(n) \in \Theta(n2^n)$. Tuy nhiên, chúng ta có thể thiết lập điều này với ít nỗ lực hơn. Thật vậy, rõ ràng từ (1.25) rằng $T(n) \in O(n2^n)$. Nếu chúng ta hài lòng với kết quả này, chúng ta có thể dừng lại ở đó. Tuy nhiên, nếu chúng ta muốn có được một ước lượng có dạng $\Theta(...)$, thì điều này là không đủ, vì hệ số $c_{2,2}$ ở phía trước của hàm phát triển nhanh nhất $n2^n$ có thể là 0. Nhớ lại rằng từ (1.27) nó trực tiếp theo đó $c_{2.2} = 1$, tức là $c_{2.2} \neq 0$ và do đó ta kết luận rằng $T(n) \in \Theta(n2^n)$.

Nếu $c_{2.2}$ bằng 0, chúng ta sẽ kiểm tra hệ số $c_{2.1}$ trước hàm phát triển nhanh nhất tiếp theo 2^n, v.v.

Cần lưu ý hai điều:

1) *Không nhất thiết phải tìm tất cả các hệ số*. Trong trường hợp của chúng ta, nó đủ để chỉ ra rằng $c_{2,2} \neq 0$.

2) *Các điều kiện biên không nhất thiết phải phù hợp*. Chưa có nơi nào chúng ta sử dụng điều kiện $T(0) = 0$, có nghĩa là kết quả $T(n) \in \Theta(n2^n)$ sẽ hợp lệ với bất kỳ giá trị nào của $T(0)$. Tất nhiên, chúng ta không thể mong đợi điều này xảy ra với mọi trường hợp phụ thuộc loài lặp lại (1.22). Nhớ lại rằng $c_{2,1}$ không thể thu được bằng cách cân bằng các hệ số trong (1.26) và do đó phụ thuộc vào $T(0)$. Và $c_{2.1}$ có thể là hệ số đứng trước hàm phát triển nhanh nhất, trong đó $T(0)$ là vấn đề quan trọng.

Bài tập

▷ **1.142.** Xác định độ phức tạp của một thuật toán được đưa ra bởi sự phụ thuộc lặp lại:

a) $T(1) = 1, T(n) = 4T(n-1) - 2n, n \geq 2$

b) $T(1) = 0, T(n) = 2T(n-1) + n + 2n, n \geq 2$

c) $T(0) = 1, T(n) = 2T(n-1) + n, n \geq 1$

d) $T(1) = 1, T(n) = 2T(n-1) + 3n, n \geq 2$

e) $T(1) = 2, T(2) = 3, T(n) = 2T(n-1) - T(n-2), n \geq 3$

1.15. Các kỹ thuật đặc biệt để phân tích thuật toán

1.15.1. Sử dụng phong vũ biểu

Hãy xem lại đoạn chương trình sau:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++)
    sum++;
```

Chúng ta đã xác định độ phức tạp của nó ở trên bằng cách sử dụng các thuộc tính của tổng. Chúng ta có thể tiếp cận nó theo cách

khác: chúng ta chọn lệnh thích hợp (phong vũ biểu) và theo dõi số lần nó được thực thi. Điều này giải phóng chúng ta khỏi lo lắng về việc phân tích tất cả các hướng dẫn khác không liên quan đến hướng dẫn đã chọn. Làm thế nào để chọn một phong vũ biểu? Đây phải là một lệnh được thực thi ít nhất thường xuyên như bất kỳ lệnh nào khác trong chương trình. Trong đoạn chương trình trên, một ứng cử viên thích hợp cho điều này là sum ++. Nhân tiện, giá trị của tổng biến sau khi thực thi đoạn sẽ cho chúng ta số lần thực thi lệnh sum ++

1.15.2. Phân tích khấu hao

Khi phân tích các thuật toán máy tính, chúng ta thường kiểm tra hành vi của chúng trong trường hợp xấu nhất hoặc trung bình, và chúng ta hầu như không bao giờ quan tâm đến cách chúng hoạt động tốt nhất. Một kỹ thuật phổ biến được sử dụng để phân tích một thuật toán nhằm xác định độ phức tạp của nó trong trường hợp xấu nhất là giả định rằng tại mỗi bước của nó, sự trùng hợp tồi tệ nhất có thể xảy ra. Điều này cho chúng ta độ phức tạp chính xác $O(...)$, nhưng không phải lúc nào cũng cung cấp cho chúng ta một ước lượng chính xác cho $\Theta(...)$: kết quả thường khá bi quan và trong thực tế, thuật toán hoạt động nhanh hơn nhiều ngay cả trong trường hợp xấu nhất. Lý do là không phải lúc nào trường hợp xấu nhất cũng có thể xảy ra ở mọi bước. Điều này thường đòi hỏi những trường hợp nhẹ hơn vài lần. Do đó, độ phức tạp tổng thể có thể tốt hơn đáng kể so với dự đoán. Hãy xem xét chức năng sau đây làm ví dụ:

Chu kỳ lồng nhau

```
void addOne(char c[], unsigned m)
{ int i;
  for (i = 0; 1 != c[i] && i < m; i++)
    c[i] = 1 - c[i];
}
```

Hàm nhận dưới dạng tham số một mảng có m phần tử chứa các số không và đơn vị, và coi nó như một bản ghi nhị phân của một số tự nhiên, chữ số nhỏ nhất nằm trong phần tử 0. Nó thêm một đơn

vị, trong mảng là bản ghi nhị phân của số lượng tăng lên. Trong trường hợp tràn, nhận được 0. Độ phức tạp của thuật toán là gì? Rõ ràng, trường hợp khó nhất là khi số chứa m đơn vị (tức là khi xảy ra tràn), vì khi đó tất cả các phần tử của mảng đều được chuyển qua. Độ phức tạp trong trường hợp này rõ ràng là $O(m)$ và $\Theta(m)$, tương ứng. Giả sử chúng ta sử dụng addOne() như một bộ đếm và gọi nó n lần, với giá trị $n = 2^m - 1$. Tổng thời gian phức tạp là bao nhiêu? Giả sử ở mỗi bước là trường hợp xấu nhất, chúng ta đạt đến độ phức tạp $O(n.m)$, tức là $O(n.\log_2 n)$. Đây là một đánh giá đúng, nhưng phân tích kỹ hơn cho thấy chúng ta không thể lần nào tiếp cận được một vụ án nghiêm trọng như vậy. Chúng ta sẽ lưu ý rằng trong quá trình đếm, chúng ta nhận được mọi số nhị phân trong khoảng $[0; 2^m - 1]$ đúng một lần. Một nửa số này là số chẵn và đối với chúng, chúng ta sẽ chỉ có một vòng quay mỗi chu kỳ. Một phần tư số còn lại (lẻ) có bit áp chót được đặt thành 0 và đối với chúng, chúng ta sẽ có hai lần quay mỗi chu kỳ. 1/8 số khác sẽ có 0 ở bit áp chót (và 1 ở hai bit cuối cùng) và sẽ yêu cầu ba lượt mỗi chu kỳ, v.v. Và chỉ trong một trường hợp, chúng ta sẽ có 1 ở tất cả các vị trí, tức là trường hợp càng nặng thì càng ít xảy ra. Chúng ta để bạn đọc chứng tỏ rằng tổng số vòng quay của chu trình là $2n - 1$. Một suy luận khả thi khác lại dẫn chúng ta đến cùng một kết quả tiệm cận là chúng ta có thể đếm đến n với thời gian $\Theta(n)$, đây là một ước lượng chính xác hơn nhiều.

1.15.3. Định lý cơ bản

Trong phân tích các thuật toán chia để trị (xem Chương 7), các phụ thuộc lặp lại thường phát sinh tùy thuộc vào loại:

$$T(n) = a.T(n/b) + c.n^k$$

Định lý 1.10. *Gọi là sự phụ thuộc hồi quy* $T(n) = a.T(n/b) + c.n^k$, $n > n_0, a \geq 1, b > 1, k \geq 0, c > 0, n_0 \geq 1$ *và* a, b, k, c, n_0 *là các số nguyên. Giải pháp của nó được đưa ra bởi công thức:*

$$T(n) = \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^{\log_k a}) & a > b^k. \end{cases}$$

Ví dụ 1.21. Xem xét sự phụ thuộc thường xuyên:

$$T(1) = 1$$
$$T(n) = 3T(n/2) + n, n \geq 2.$$

Sử dụng Định lý 1.10, ta thu được: $a = 3, b = 2, c = 1, k = 1$. Ta có: $3 = a > b^k = 2$. Do đó ta rơi vào trường hợp thứ ba và trực tiếp thu được độ phức tạp $\Theta(n^{\log 3})$.

Ví dụ 1.22. Xem xét sự phụ thuộc thường xuyên:

$$T(1) = d$$
$$T(n) = 4T(n/2) + n^2, n \geq 2.$$

Sử dụng Định lý 1.10, ta được: $a = 4, b = 2, c = 1, k = 2$. Ta có: $4 = a = b^k = 2^2$. Từ đây chúng ta rơi vào trường hợp thứ hai và thu được trực tiếp độ phức tạp $\Theta(n^2.\log_2 n)$. Lưu ý rằng giá trị của d là bao nhiêu không quan trọng.

Ví dụ 1.23. Xem xét sự phụ thuộc thường xuyên:

$$T(1) = d$$
$$T(n) = 2T([\sqrt{n}]) + log_2 n, n \geq 2.$$

Ta đặt $m = \log_2 n$ và nhận được:

$$T(2^m) = 2T(2^{m/2}) + m$$

Bây giờ chúng ta đặt $S(m) = T(2^m)$:

$$S(m) = 2S(m/2) + m$$

Bây giờ chúng ta có thể sử dụng Định lý 1.10: $a = 2, b = 2, c = 1, k = 1$. Ta có: $2 = a = b^k = 2^1$. Ta rơi vào trường hợp thứ hai và nhận được $\Theta(m.log_2 m)$. Chúng ta quay lại các giả định:

$$T(n) = T(2^m) = S(m) = \Theta(m.\log_2 m) = \Theta(\log_2 n.\log_2(\log_2 n))$$

Bài tập

▷ **1.143.** Xác định độ phức tạp của một thuật toán nếu được cung cấp bởi sự phụ thuộc đệ quy:

a) $T(1) = 2, T(n) = 4T(n/3) + n^{\log_3 4}, n \geq 2$.

b) $T(1) = 0 v T(2n+1) = T(2n) = T(n) + \log_2 n, n \geq 2$.

c) $T(1) = 4 v T(n) = 2T([\sqrt{n}]) + \log_2 n, n \geq 2$.

1.15.4. Các vấn đề về ký hiệu tiệm cận

Mặc dù chúng cho phép chúng ta đánh giá mức độ phức tạp của các thuật toán và chương trình trên cơ sở lý thuyết đúng đắn, chúng ta nên hơi nghi ngờ về các ước lượng tiệm cận. Đặc điểm chính của họ là họ quan tâm đến hành vi của thuật toán với mức tăng n không giới hạn. Tuy nhiên, không có số lượng lớn vô hạn trong thế giới máy tính thực. Điều này có nghĩa là ước lượng do hàm tiệm cận cung cấp cho chúng ta có thể không đủ: ví dụ, vì nó ẩn các hằng số, hoặc vì chúng ta không quan tâm đến các giá trị lớn như vậy của n. Giả sử chúng ta muốn tính giá trị của một hàm, nhưng chỉ đối với các đối số trong phạm vi $[0, 65535]$ và chúng ta có hai thuật toán: tuyến tính $(200000n)$ và bậc hai $(2n^2)$. Về mặt tiệm cận, thuật toán tuyến tính sẽ nhanh hơn thuật toán bậc hai. Điều này đúng, nhưng từ một nơi nào đó trở đi. Thật không may cho chúng ta đây là $n = 100000 > 65535$. Vì vậy, đối với tất cả các đối số thực tế của hàm, thuật toán bậc hai sẽ nhanh hơn.

Trong tương lai, chúng ta sẽ thường xuyên đối mặt với vấn đề này và học cách sử dụng nó. Ví dụ, sắp xếp nhanh (xem 3.1.6.) Trong trường hợp giữa có độ phức tạp $\Theta(n.\log_2 n)$ và sắp xếp chèn (xem 3.1.2.) Có $\Theta(n^2)$. Đối với các giá trị đủ lớn của số phần tử được sắp xếp n, việc sắp xếp nhanh sẽ nhanh hơn nhiều, nhưng đối với 10-20 phần tử, sắp xếp bằng cách chèn là tốt hơn. Thoạt nhìn, điều này hầu như không quan trọng, bởi vì với số lượng phần tử nhỏ như vậy, việc sắp xếp nhanh vẫn nhanh ở mức chấp nhận được và việc sử dụng tính năng phân loại chèn là điều khó chấp nhận. Tuy nhiên, xem xét kỹ hơn sẽ thấy rằng phân loại nhanh liên tục nhìn vào các phân vùng nhỏ, với khoảng cách tích tụ. Do đó, hai thuật toán thường được kết hợp để đạt được sự cải thiện đáng kể.

Bài tập

▷ **1.144.** Ba thuật toán có độ phức tạp đã cho là: $5n^2 - 7n + 13, 3n^2 + 15n + 100$ và $1000n$. Đối với mỗi thuật toán để xác định một khoảng giá trị n mà nó nhanh hơn hai giá trị còn lại.

▷ **1.145.** Lời phê bình về ký hiệu tiệm cận có chính đáng không?

1.16. Các câu hỏi và bài tập

1.16.1. Các bài toán về số, chuỗi, hàm

Sử dụng các bài toán được nêu dưới đây, ngoại trừ để giải quyết vấn đề toán học cụ thể và thử nghiệm với các sơ đồ triển khai khác nhau (lặp lại, đệ quy). Xác định độ phức tạp của thuật toán của bạn và sử dụng nó để dự đoán tốc độ chuyển đổi có thể xảy ra đối với các đầu vào khác nhau.

▷ **1.146.** Viết chương trình tìm tất cả các số bằng tổng các thừa số của các chữ số của nó.

Gợi ý: Điều quan trọng là phải thu hẹp phạm vi (để tìm giới hạn trên) mà tìm kiếm những con số như vậy. Ví dụ: không có ý nghĩa gì khi kiểm tra các số có 9 chữ số trở lên vì $9, 9! = 3265920$, rõ ràng là nhỏ hơn bất kỳ số nào có 9 chữ số.

Định nghĩa 1.33. Hai số nguyên tố p và q được gọi là "sinh đôi" nếu $p = q + 2$. Người ta chứng minh rằng có vô số cặp số nguyên tố "sinh đôi". 4 đầu tiên trong số đó là $\{3, 5\}; \{5, 7\}; \{11, 13\}; \{17, 19\}$.

▷ **1.147.** Tìm n cặp số nguyên tố đầu tiên của các cặp sinh đôi.

Định nghĩa 1.34. Tổng của dãy giá trị nghịch đảo của các số sinh đôi

$$S = 1/3 + 1/5 + 1/5 + 1/7 + 1/11 + 1/13 + 1/17 + 1/19 + \cdots$$

là một hằng số được gọi là hằng số Bran và xấp xỉ bằng $1,902160578$.

▷ **1.148.** Tìm tổng các số nghịch đảo của n số nguyên tố đôi một.

▷ **1.149.** *Phương trình Diophantine của 2 biến*
Phương trình $ax + by = c$ được cho trước, trong đó a, b và c là các số nguyên. Tìm x và y nếu chúng là số nguyên.

Gợi ý: Gọi d là ước chung lớn nhất của a và b. Nếu d không chia c, thì không có lời giải cho bài toán. Sử dụng thuật toán Euclid nâng cao để tìm x và y. Bạn có thể tìm thêm giải pháp nếu bạn có một giải pháp cho vấn đề (x_0, y_0) [Rakhnev, Garov, Gavrilov-1995] không?

▷ **1.150.** *x%y*
Thực hiện các phép toán của phép chia số nguyên và phần dư của phép chia số nguyên, chỉ sử dụng các phép toán chuẩn `+, -, *, /` trên các số thực.

▷ **1.151.** *Tích của hai số nguyên tố*
Một số tự nhiên n đã cho. Tìm số tự nhiên nhỏ hơn n có thể được biểu diễn dưới dạng tích của hai số nguyên tố.

▷ **1.152.** *Tổng các số*
Với số tự nhiên n tìm số tự nhiên nhỏ nhất $m, m > n$ có tổng các chữ số bằng tổng các chữ số của n. Các chữ số của n được đặt làm phần tử của một mảng và có thể lên đến 2000.

▷ **1.153.** *Số lượng không thể*
n số nguyên đã cho. Để tìm một trong số chúng mà không thể được biểu diễn bằng tổng của một số những số khác.

▷ **1.154.** $n + 2 = 2m$
Tìm hợp số n nhỏ nhất sao cho $n + 2 = 2m$, với m là số lẻ.

▷ **1.155.** *Con số may mắn*
Từ danh sách $1, 2, 3, ...$ mỗi số thứ hai liên tiếp bị loại trừ. Vậy có $1, 3, 5, 7, \ldots$. Sau đó, mọi thứ ba bị loại khỏi danh sách mới nhận được $(1, 3, 7, 9, 13$vẫn còn), v.v. In kết quả sau k số bước.

Gợi ý: Sử dụng một sửa đổi của phương pháp rây.

▷ **1.156.** *Cặp đôi nguyên tố cùng nhau*
n số tự nhiên đã cho. Tìm số cặp số lớn nhất trong số các số đã cho để các số trong mỗi cặp là nguyên tố cùng nhau.

▷ **1.157.** *Đa thức Hermitian*
Viết hàm tìm giá trị của đa thức Hermitian $H_n(x)$:

$$H_0(x) = 1;$$
$$H_1(x) = 2x;$$
$$H_n(x) = 2x.H_{n-1}(x) - 2(n-1).H_{n-2}(x), n > 1.$$

▷ **1.158.** *Số tribonacci*
Số tribonacci được xác định bằng công thức lặp lại sau:

$$F(n) = F(n-1) + F(n-2) + F(n-3), \text{ biết } F(1) = F(2) = F(3) = 1$$

Phần đầu của chuỗi Tribonacci trông như sau:

$$1, 1, 1, 3, 5, 9, 17, ...$$

Để biên dịch một chương trình, cho trước n, hãy tìm số thứ n của "Tribonachi". Viết thực hiện đệ quy và lặp lại tương ứng. Cái nào trong hai cái hiệu quả hơn và tại sao? Độ phức tạp của thực hiện đệ quy là gì? Hãy tìm công thức để tìm số thứ n của "Tribonachi".

▷ **1.159.** *Số Fibonacci của thứ tự p*
Số Fibonacci của thứ tự p được định nghĩa là:

$$f_{i+1}^{p} = f_i^{(p)} + \cdots + f_{i-p}^{(p)}, i \geq p;$$
$$f_p^{(p)} = 1;$$
$$f_i^{(p)} = 0, 0 \leq i \leq p;$$

Như vậy các số Fibonacci thông thường được định nghĩa là các số Fibonacci bậc 1. Viết chương trình tìm k số Fibonacci đầu tiên có bậc p, p và k là các số tự nhiên.

▷ **1.160.** *Tổng những nghiệm*
Viết chương trình tính tổng $S = \sum_{i=1}^{n} \frac{1}{\sqrt{i}}$ cho số tự nhiên n cho trước. Có thể lưu một số phép tính được thực hiện bằng cách triển khai tầm thường không?

▷ **1.161.** *Chống số đầu tiên*
Số chống đầu tiên là số tự nhiên có nhiều ước hơn bất kỳ số tự nhiên nào khác trước nó. 6 số đầu tiên của loại này là:

1: (có một ước số duy nhất 1)

2: (1, 2)

4: Có 3 ước (1, 2, 4)

6: Có 4 ước số (1, 2, 3, 6)

12: Có 6 ước số (1, 2, 3, 4, 6, 12)

24: Có 8 ước số (1, 2, 3, 4, 6, 8, 12, 24)

Tìm n số đầu tiên như vậy trên một số tự nhiên n cho trước. Bạn có thể tìm thấy mối quan hệ nào giữa chúng không?

▷ **1.162.** *Số nguyên tố dạng* $4k + 3$
Một số tự nhiên n đã cho. Tìm n số nguyên tố đầu tiên có dạng $4k + 3, k \in \mathbb{N}$.

▷ **1.163.** *Công thức cho n!*
Người ta biết rằng có một công thức để tìm số tiền:

$$S = 1 + 2 + 3 + \cdots + n$$

Tương tự, chúng ta coi tích của n số đầu tiên:

$$P = 1.2.3.....n = n!$$

Tìm công thức để tìm $n!$ Điều đó không thực hiện tất cả n phép nhân.

Gợi ý: Nếu bạn không thể tìm ra công thức như vậy trong vài giờ, tốt hơn hết bạn nên từ bỏ: Thực tế, cho đến nay vẫn chưa có ai thành công. Có những công thức, chẳng hạn như công thức Stirling, tìm thấy $n!$ với một ước lượng khá tốt:

$$n! \approx n^n e^{-n} \sqrt{2\pi n}$$

Tuy nhiên, lợi ích của chúng là lý thuyết hơn là thực tế.

▷ **1.164.** *Các khoảng với tổng n*
Với một số tự nhiên n đã cho, hãy tìm khoảng $[a, b]$ sao cho tổng các số

$$a + (a + 1) + (a + 2) \cdots + b$$

bằng n. Ví dụ, với $n = 1986$, một số khoảng thỏa mãn là:

$$[160, 171], [495, 498] \text{ và } [661, 663].$$

▷ **1.165.** *Hàm $\varphi(n)$*
Cho n là một số tự nhiên. Theo $\varphi(n)$, chúng ta có nghĩa là số các số nhỏ hơn n nguyên tố cùng nhau với n. Tìm $\varphi(n)$ với một số nguyên dương n cho trước. Giải thích của bạn cho thực tế là với $n > 2\varphi(n)$ luôn là một số chẵn? Cố gắng tìm công thức chính xác cho $\varphi(n)$.

▷ **1.166.** *Bộ ba Pythagore*
Một số tự nhiên n đã cho. Tìm tất cả các bộ ba Pitago của các số a, b, c (số tự nhiên), $c < n$, trong đó $a^2 + b^2 = c^2$.

▷ **1.167.** *Palindrome bậc*
Cho là một số tự nhiên n. Palindrome bậc $P(n)$ của n được định nghĩa như sau:

1) Nếu n là palindrome thì $P(n) = 1$.

2) Nếu n không phải là palindrome, thì $P(n) = 1 + P(s)$, với s là tổng của n với ảnh phản chiếu của nó, tức là với n, viết qua lại.

Ví dụ: độ palindrome của 36 là 2. Đối với hai bước, chúng ta nhận được palindrome:

$$36 + 63 = 99$$

Tương tự $P(48) = 3$:

$$48 + 84 = 132$$
$$132 + 231 = 363$$

Tìm $P(n)$ với n bất kỳ. Hãy thử chương trình của bạn với tất cả các số tự nhiên từ 1 đến 250. Nó tìm thấy gì cho trường hợp đặc biệt $P(196)$?

▷ **1.168.** *Bộ tứ Pythagore*
Một số tự nhiên n đã cho. Tìm n (n là số tự nhiên) các nhóm nguyên a, b, c, d khác nhau sao cho:

$$a^2 + b^2 + c^2 = d^2$$

Ví dụ: $(1, 2, 2, 3)$.

▷ **1.169.** *Bộ ba số tự nhiên*
Một số tự nhiên n đã cho. Tìm n (n là số nguyên) các bộ ba khác nhau của các số tự nhiên a, b, c sao cho $\sqrt{a^2+b^2}$, $\sqrt{a^2+c^2}$, $\sqrt{b^2+c^2}$, cũng là các số tự nhiên.

▷ **1.170.** *Tổng ngắn nhất của các lập phương*
Một số tự nhiên n đã cho. Tìm biểu diễn của n dưới dạng tổng của các hình lập phương bằng cách sử dụng số lượng nhỏ nhất mà các vật sưu tầm được. Ví dụ:

$$567 = 8^3 + 3^3 + 3^3 + 1^3 (\text{độ dài } 4),$$

nhưng cũng có một giải pháp ngắn hơn:

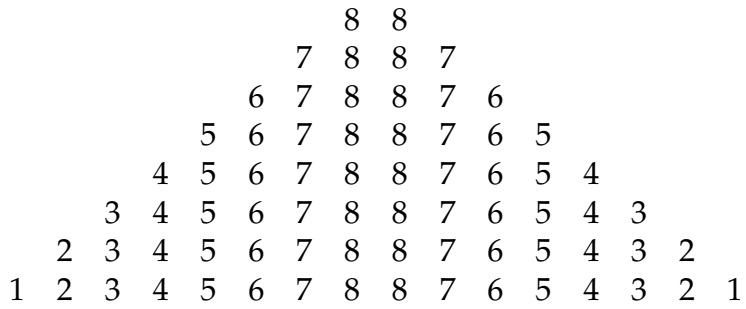
$$567 = 7^3 + 6^3 + 2^3$$

▷ **1.171.** *Biểu diễn 2^n*
Tìm n nhỏ nhất để chín chữ số đầu tiên của số 2^n là 123454321.

▷ **1.172.** *Kim tự tháp số*
Viết hàm sao cho một số tự nhiên n cho trước, hiển thị một hình chóp có chiều cao n thuộc loại 1 hoặc 2 (xem Hình 1.16-1.17).

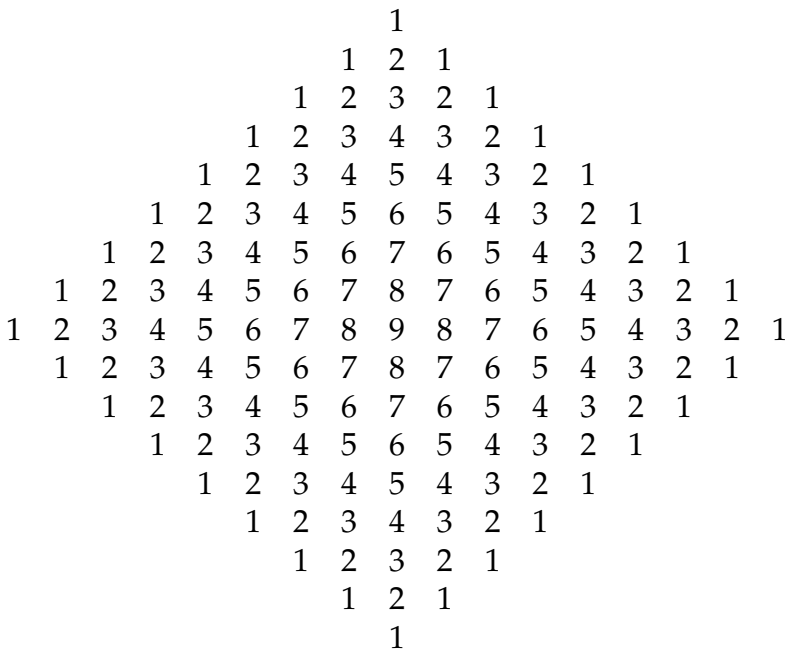
```
                    9
                  8 9 8
                7 8 9 8 7
              6 7 8 9 8 7 6
            5 6 7 8 9 8 7 6 5
          4 5 6 7 8 9 8 7 6 5 4
        3 4 5 6 7 8 9 8 7 6 5 4 3
      2 3 4 5 6 7 8 9 8 7 6 5 4 3 2
    1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
```

Hình 1.16. Kim tự tháp loại 1



Hình 1.17. Kim tự tháp loại 2

▷ **1.173.** *Quảng trường*
Viết một hàm, cho trước một số tự nhiên n, hiển thị một hình vuông có chiều cao n (xem Hình1.18).



Hình 1.18. Quảng trường

1.16.2. Bài toán ma trận và bài toán chung

▷ **1.174.** *Hình vuông ma thuật*
Hình vuông ma thuật của hàng n là một bảng có kích thước $n \times n$,

trong mỗi ô có một số tự nhiên từ 1 đến n^2 được viết sao cho tổng các số được viết trên mỗi đường ngang, dọc hoặc đường chéo chính là như nhau và bằng $n\,\dfrac{n^2+1}{2}$. Ví dụ, với $n = 5$, một hình vuông ma thuật khả dĩ được cho trong Hình 1.19.

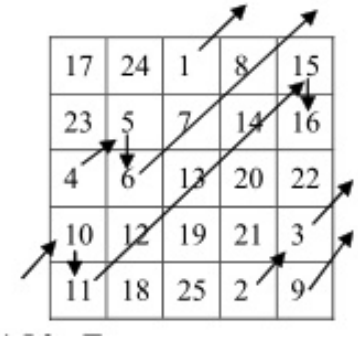
17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Hình 1.19. Hình vuông ma thuật

Gợi ý: Đối với n lẻ ($n \geq 3$), một thuật toán khả thi để điền là như sau:

1) Chúng ta bắt đầu từ giữa dòng đầu tiên và viết số 1 vào đó.

2) Ở mỗi bước, chúng ta di chuyển theo đường chéo lên trên và sang phải và viết số tiếp theo. Nếu chúng ta "đi ra ngoài" bên ngoài bảng trên, chúng ta di chuyển đến hàng cuối cùng, và nếu chúng ta "đi ra ngoài" bên phải, chúng ta di chuyển đến trụ đầu tiên của nó. Nếu trong quá trình di chuyển, chúng ta thấy mình ở trong một ô đã được lấp đầy, thì chúng ta "đi xuống" ô bên dưới nó. Hình 1.20 hiển thị cách một phần của điền bắt đầu, bắt đầu từ 1:



Hình 1.20. Xây dựng một hình vuông kỳ diệu.

Đối với n chẵn, cũng có các thuật toán để thu được một hình vuông ma thuật, nhưng chúng phức tạp hơn một chút [Phép thuật-1].

▷ **1.175.** *Cờ vua*
Xác định số quân cờ tối đa có thể đặt trên một bàn cờ có kích thước $m \times n$ (n, m - các số tự nhiên cho trước) để không có hai quân nào bị tấn công. Để giải quyết công việc tương tự, nếu con số được đặt là: vua; đứng đầu; ngựa; nhân viên văn phòng.

▷ **1.176.** *Từ bốn chữ cái*
Cho trước hai từ gồm bốn chữ cái A_0 và A_n và một từ điển có bốn chữ cái A_i, với $i = 1, 2, ..., n-1$. Có thể rút ra từ A_0 từ từ A_n bằng một chuỗi các từ trong từ điển không:

$$A_0 - A_1 - A_2 - ... - A_n,$$

vì cứ hai từ liên tiếp A_i và A_{i+1}, cho $i = 0, 1, ..., n-1$, có đúng một chữ cái khác nhau. Kiểm tra quyết định của bạn với các từ "bay" và "voi" và một từ điển phù hợp.

▷ **1.177.** *Ma trận chuyển đổi*
Một ma trận số nguyên $A_{n \times n}$ với các phần tử a_{ij} được cho trước. Tìm ma trận chuyển vị của nó $A' = \{a'_{ij}\}$ của ma trận đã cho, $a'_{ij} = a_{ji}$, với $1 \leq i, j \leq n$.

1.16.3. Bài toán tổ hợp

Trong một số tác vụ tiếp theo, bạn muốn tạo các đối tượng có các thuộc tính nhất định, chẳng hạn (xem Bài toán 1.43.) Số PIN hợp lệ. Trong các tác vụ như vậy, không nên tạo cấu trúc tổ hợp tổng quát hơn (các biến thể có lặp lại cho ví dụ đã chọn) và sau đó loại trừ các đối tượng không hợp lệ, nhưng nên biên dịch một lược đồ chỉ tạo các đối tượng hợp lệ.

▷ **1.178.** *Số điện thoại*
Năm 1930, một nhóm nghiên cứu từ Phòng thí nghiệm Bell được giao nhiệm vụ đề xuất cách thiết kế số điện thoại của Hoa Kỳ. Chúng được yêu cầu phải có giá trị ít nhất cho đến cuối thế kỷ 20. Sau khi ước tính sẽ có bao nhiêu trụ điện thoại, các nhà nghiên cứu tập trung

vào sơ đồ số điện thoại sau:
$(a_1a_2a_3) - b_1b_2b_3 - c_1c_2c_3c_4$, trong đó
$a_1 \in \{2, 3, ..., 9\}$
$a_2 \in \{0, 1\}$
$a_3 \in \{0, 1, ..., 9\}$
$b_1, b_2 \in \{2, 3, ..., 9\}$
$b_3, c_1, c_2, c_3, c_4 \in \{0, 1, ..., 9\}$

Tìm số điện thoại khác nhau có thể có được từ sơ đồ trên.

▷ **1.179.** *Hình vuông La tinh*
Hình vuông Latinh của hàng n là một bảng có kích thước n sao cho:

- trong mỗi ô của bảng có một số tự nhiên từ 1 đến n.
- Các số từ mỗi hàng và mỗi cột là hoán vị của các số từ 1 đến n.

Ví dụ, với $n = 4$, hình vuông của Hình1.21. là tiếng Latinh.

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

Hình 1.21. Hình vuông La tinh

Để tạo một hình vuông Latinh bởi một số tự nhiên n cho trước.

▷ **1.180.** *Tích không đổi*
$2n$ số tự nhiên đã cho. Biết rằng có đúng m cặp số (a_i, b_i) sao cho tích $a_i.b_i$ bằng chính số đó (với $i = 1, 2, ..., m$). Tìm và in ra tất cả m các cặp số như vậy.

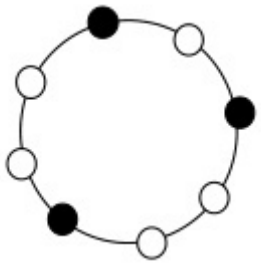
▷ **1.181.** *Ăn tối tại Princess Anna*
Công chúa Anna dự định mời 15 người bạn của mình đi ăn tối. Trong 35 ngày, họ sẽ đến thăm cô ấy đúng 3 lần một ngày. Có thể chọn ba người để trong mỗi 35 ngày, Anna ăn tối với ba người khách khác nhau không? Tạo một "lịch trình" khả thi.

▷ **1.182.** *Giao điểm của các đường chéo trong một n-giác đều.*
Cho số tự nhiên $n, n > 2$. Tìm số giao điểm của các đường chéo

trong một vòng lặp n đều. Ví dụ, không có đường chéo nào trong một tam giác và số này bằng không. Hình vuông có hai đường chéo và tương ứng là một giao điểm, và đối với một hình ngũ giác đều, con số này là 5.

▷ **1.183.** *Bóng trong hộp*
$2t + 1$ hòn bi được cho và phải chia đều vào ba hộp sao cho tổng số bi ở hai hộp bất kì lớn hơn số bi ở hộp thứ ba. Viết một chương trình tạo ra tất cả các phân phối bóng có thể có để đáp ứng điều kiện trên. Tìm công thức cho số lượng các phân phối này.

▷ **1.184.** *Chuỗi hạt*
Một chiếc vòng cổ bao gồm các hạt trắng và đen sao cho không có hai hạt đen liền nhau (xem Hình1.22.).



Hình 1.22. Vòng cổ

Nếu n là số hạt đen và k là số hạt trắng thì số vòng cổ khác nhau có thể được tạo thành từ các hạt là bao nhiêu? Viết chương trình tạo t dây chuyền khác nhau, cho trước một số tự nhiên t.

▷ **1.185.** *Chiếc cầu*
Viết chương trình chia bài soi cầu: 4 người chơi, mỗi người 13 lá. Trừu tượng thực tế, hãy viết chương trình của bạn theo cách sao cho mỗi lần chơi liên tiếp, một ván bài duy nhất được tạo ra - không có người chơi nào nhận được hai lá bài giống nhau.

▷ **1.186.** *Hình vuông trong lưới*
Một lưới hình vuông với kích thước $n \times n$ được đưa ra. Số ô vuông trong lưới là một hàm của n: tức là tổng số các ô vuông khác nhau có kích thước $1 \times 1, 2 \times 2, ..., n \times n$?

▷ **1.187.** *Các biến trong C*
Số tên biến hợp lệ trong C có dưới k ký tự, k là số tự nhiên là bao nhiêu?

▷ **1.188.** *Số có n chữ số*
Tìm công thức số các số có n chữ số ($n < 11$) chia hết cho k nếu không được phép lặp lại các chữ số. Để đề xuất một thuật toán để tạo thứ tự từ vựng của tất cả các số như vậy.

▷ **1.189.** *Khoảng cách giữa các quân cờ*
Tìm cách xếp 9 quân trên bàn cờ 8×8 sao cho khoảng cách giữa hai quân là khác nhau, hoặc chứng tỏ rằng điều này là không thể. "Khoảng cách giữa hai hình" đề cập đến số lượng trường trên đường đi ngắn nhất giữa chúng (không có chuyển đổi đường chéo).

▷ **1.190.** *MJ-hộp đêm*
n đôi vợ chồng được đưa ra. Có bao nhiêu cách có thể chia người thành từng nhóm k (k chẵn, k chia hết n) sao cho trong mỗi nhóm có đúng $k/2$ nữ và $k/2$ nam và không có ai ở cùng nhóm với vợ/chồng của bạn?

▷ **1.191.** *Một hacker trẻ*
Viết chương trình tạo số thẻ tín dụng hợp lệ (từ một loại thẻ tín dụng - VISA hoặc MasterCard). Sử dụng mô tả sau để biết số thẻ tín dụng hợp lệ:

Số thẻ tín dụng có 16 chữ số trong nhóm bốn chữ số. Hãy kiểm tra xem thẻ tín dụng sau có hợp lệ không:
4 2 0 4 - 5 8 7 6 - 9 0 1 2 - 5 2 3 4

1) Chúng ta chuyển các chữ số của số và nhận được:
4 3 2 5 2 1 0 9 6 7 8 5 4 0 2 4

2) Nhân đôi các số ở vị trí chẵn và nhận được:
4 6 2 10 2 2 0 18 6 14 8 10 4 0 2 8

3) Nếu lấy số sưu tầm lớn hơn 9, ta chia chúng thành hai chữ số riêng biệt. Chúng ta thu thập các số liệu thu được như sau:

$$4+6+2+1+0+2+2+0+1+8+6 \\ +1+4+8+1+0+4+0+2+8 = 60. \quad (1.28)$$

4) Nếu tổng chia cho 10 mà không có số dư, thẻ tín dụng vẫn hợp lệ. Nếu số bắt đầu bằng 4, thì loại của nó là $VISA$ và nếu nó có tiền tố 51-55, thì đó là *MasterCard*.

▷ **1.192.** *Số PIN*
Để viết một chương trình tạo số PIN hợp lệ cho phụ nữ.

Gợi ý: Việc xác nhận mã PIN được thực hiện theo sơ đồ sau: các trọng số sau được so sánh trên từng chữ số của chín đầu tiên: 2, 4, 8, 5, 10, 9, 7, 3, 6. Mỗi chữ số của mã PIN được nhân với trọng lượng tương ứng và tính tổng các sản phẩm thu được. Phần còn lại cho phép chia số nguyên của tổng cho 11 là một số kiểm soát và phải trùng với chữ số thứ mười của mã PIN. Đối với phần dư của 0 hoặc 10, chữ số điều khiển phải là 0.

Ngoài ra, vì 6 chữ số đầu tiên đại diện cho ngày sinh nên chúng phải là ngày hợp lệ trong năm. Ví dụ: mã PIN bắt đầu bằng 810229 không hợp lệ vì năm 1981 không phải là năm cao.

Giới tính được xác định bằng chữ số thứ chín: nếu là số chẵn thì mã PIN dành cho nam giới.

▷ **1.193.** *Số Palindromes*
Số từ đối xứng (palindromes) có độ dài n ký tự, bao gồm các chữ cái Latinh viết thường là bao nhiêu? Viết chương trình tạo tất cả các palindromes có độ dài n.