第3章

递归与分治策略


视频讲解

任何一个可以用计算机求解的问题所需的计算时间都与其规模 n 有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于 n 个元素的排序问题，当 $n=1$ 时，不需要任何计算；$n=2$ 时，只要作一次比较即可排好序；$n=3$ 时，只要作 3 次比较即可……当 n 较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。分治策略的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

如果原问题可分割成 k 个子问题（$1<k\leqslant n$），且这些子问题都可解，并可利用这些子问题的解求出原问题的解，这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题的类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。由此自然导致递归算法的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生了许多高效算法。

3.1 递归算法

程序直接或间接调用自身的编程技巧称为递归算法。

递归是一个过程或函数在其定义或说明中又直接或间接调用自身的一种方法，它通常把一个大型复杂的问题层层转换为一个与原问题相似的规模较小的问题来求解。递归策略只需少量的程序就可以描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。

递归的能力在于用有限的语句来定义对象的无限集合。用递归思想写出的程序往往十分简洁易懂。

一般来说，递归需要有边界条件、递归前进段和递归返回段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回。注意：在使用递归策略时，必须有一个明确的递归

结束条件，称为递归出口，否则将无限进行下去（死锁）。

递归算法一般用于解决三类问题：

（1）数据的定义是按递归定义的。例如 Fibonacci（斐波那契）函数。

（2）问题的解法按递归算法实现，例如回溯算法。

（3）数据的结构形式是按递归定义的，例如树的遍历、图的搜索。

递归算法的缺点：递归算法解题的运行效率较低。在递归调用过程中，系统为每一层的返回点、局部变量等开辟了栈来存储。递归次数过多容易造成栈溢出等。

递归算法是解决问题的一种最自然且合乎逻辑的方式，利用递归算法不需花费太多的精力就能够解决问题，但是程序的执行效率可能会变差。在这种情况下，通常把递归算法转换为非递归算法，例如模拟算法或者递推算法。

3.1.1　斐波那契数列

斐波那契数列是意大利数学家列昂纳多·斐波那契（Leonardo Fibonacci，1170—1240）首先研究的一种递归数列，它的每一项都等于前两项之和。此数列的前几项为 1，1，2，3，5。在生物数学中，许多生物现象都会呈现出斐波那契数列的规律。斐波那契数列相邻两项的比值趋近于黄金分割数。斐波那契数也以密码的方式出现在诸如《达·芬奇密码》等影视作品中。其递归定义为：

$$F(n)=\begin{cases}1 & (n=0,1)\\ F(n-1)+F(n-2) & (n>1)\end{cases}$$

这是一个递归关系式。当 $n>1$ 时，这个数列的第 n 项的值是它前面两项之和。它用两个较小的自变量函数值定义一个较大的自变量函数值，所以需要两个初始值 $F(0)$ 和 $F(1)$。

算法 3.1　斐波那契数列的递归算法。

```
int fib(int n)
{
    if (n<=1) return 1;
    return fib(n-1)+fib(n-2);
}
```

显然，该算法的效率非常低，因为重复递归的次数太多。

通常采用递推算法进行改进。

算法 3.2　斐波那契数列的递推算法。

```
int fib[50];                    //采用数组保存中间结果
void fibonacci(int n)
{
    fib[0] = 1;
    fib[1] = 1;
    for (int i=2; i<=n; i++)
        fib[i] = fib[i-1]+fib[i-2];
}
```

在整数(int)范围内,可以计算的最大数 $n=46$;在长整数(long long)范围内,可以计算的最大数 $n=92$。

斐波那契数列的非递归定义:

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

类似地,勒让德多项式的递推关系式如下:

$$P_n(x)=\begin{cases}1 & (n=0)\\ x & (n=1)\\ ((2n-1)xP_{n-1}(x)-(n-1)P_{n-2}(x))/n & (n>1)\end{cases}$$


视频讲解

3.1.2 集合的全排列问题

设 $R=\{r_1,r_2,\cdots,r_n\}$ 是要进行排列的 n 个元素,显然一共有 $n!$ 种排列。令 $R_i=R-\{r_i\}$。集合 X 中元素的全排列记为 perm(X),则(r_i)perm(X)表示在全排列 perm(X)的每一个排列前加上前缀 r_i 得到的排列。R 的全排列可归纳定义如下:

当 $n=1$ 时,perm(R)=(r),其中 r 是集合 R 中唯一的元素;

当 $n>1$ 时,perm(R)由(r_1)perm(R_1),(r_2)perm(R_2),…,(r_n)perm(R_n)构成。

依此递归定义,可设计产生 perm(R)的递归算法 3.3。

算法 3.3 全排列问题的递归算法(回溯算法)。

```
//产生元素 k～m 的全排列,作为前 k-1 个元素的后缀
void Perm(int list[],int k,int m)
{
    //构成了一次全排列,输出结果
    if(k==m)
    {
        for(int i=0;i<=m;i++)
            cout<<list[i]<<" ";
        cout<<endl;
    }
    else
        //在数组 list 中,产生元素 k～m 的全排列
        for(int j=k;j<=m;j++)
        {
            swap(list[k],list[j]);
            Perm(list,k+1,m);
            swap(list[k],list[j]);
        }
}
```

例如,数组 list[]={1, 2, 3, 4, 5, 6},则调用 Perm(list,0,3)就是产生元素 1～4 的全排列。全排列的结果如表 3-1 所示。算法实现源代码:perm.cpp。

一般情况下,$k<m$。该算法将 list[k:m]中的每一个元素分别与 list[k]中的元素交换,然后递归地计算元素 list[$k+1$:m]的全排列,并将计算结果作为 list[0:k]的后缀。

算法 3.3 中，函数 swap()是标准库函数，用于交换两个元素的值。

表 3-1　Perm(list,0,3)的排列结果

$r_1=1$ 的排列	$r_2=2$ 的排列	$r_3=3$ 的排列	$r_4=4$ 的排列
1 2 3 4	2 1 3 4	3 2 1 4	4 2 3 1
1 2 4 3	2 1 4 3	3 2 4 1	4 2 1 3
1 3 2 4	2 3 1 4	3 1 2 4	4 3 2 1
1 3 4 2	2 3 4 1	3 1 4 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 1 2	4 1 3 2
1 4 2 3	2 4 1 3	3 4 2 1	4 1 2 3

3.1.3　整数划分问题


视频讲解

整数划分问题是算法中的经典命题之一。所谓整数划分，是指把一个正整数 n 表示成一系列正整数之和：

$$n=n_1+n_2+\cdots+n_k \quad (\text{其中},n_1\geqslant n_2\geqslant\cdots\geqslant n_k\geqslant 1,k\geqslant 1)$$

正整数 n 的这种表示称为正整数 n 的划分。正整数 n 的不同划分个数称为正整数 n 的划分数，记作 $p(n)$。例如，正整数 6 有如下 11 种不同的划分，所以 $p(6)=11$。

```
6
5 + 1
4 + 2, 4 + 1 + 1
3 + 3, 3 + 2 + 1, 3 + 1 + 1 + 1
2 + 2 + 2, 2 + 2 + 1 + 1, 2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```

如果$\{n_1,n_2,\cdots,n_i\}$中的最大加数 s 不超过 m，即 $s=\max(n_1,n_2,\cdots,n_i)\leqslant m$，则称它为属于 n 的一个 m 划分。我们记 n 的 m 划分的个数为 $f(n,m)$。该问题就转化为求 n 的所有划分个数 $f(n,n)$。可以建立 $f(n,m)$的递归关系如下：

(1) $f(1,m)=1,m\geqslant 1$。当 $n=1$ 时，无论 m 的值为多少($m>0$)，只有一种划分，即 1 个 1。

(2) $f(n,1)=1,n\geqslant 1$。当 $m=1$ 时，无论 n 的值为多少($n>0$)，只有一种划分，即 n 个 1：

$$n=\overbrace{1+1+\cdots+1}^{n\text{个}}$$

(3) $f(n,m)=f(n,n),m\geqslant n$。最大加数 s 实际上不能超过 n。例如，$f(3,5)=f(3,3)$。

(4) $f(n,n)=1+f(n,n-1)$。正整数 n 的划分是由 $s=n$ 的划分和 $s\leqslant n-1$ 的划分构成的。例如，$f(6,6)=1+f(6,5)$。

(5) $f(n,m)=f(n,m-1)+f(n-m,m),n>m>1$。正整数 n 的最大加数 s 不大于 m 的划分，是由 $s=m$ 的划分和 $s\leqslant m-1$ 的划分组成的。

例如，$f(6,4)=f(6,3)+f(2,4)=f(6,3)+f(2,2)$，如表 3-2 所示：

表 3-2 $f(6,4)$分解示例

$f(6,4)=9$	$f(6,3)=7$	$f(2,2)=2$
4+2,4+1+1 3+3,3+2+1,3+1+1+1 2+2+2,2+2+1+1,2+1+1+1+1 1+1+1+1+1+1	3+3,3+2+1,3+1+1+1 2+2+2,2+2+1+1,2+1+1+1+1 1+1+1+1+1+1	4+2,4+1+1 (实际上是 2 的划分)

综合以上递归关系,给出计算 $f(n,m)$的递归公式如下:

$$f(n,m)=\begin{cases}1 & (n=1,m=1)\\ f(n,n) & (n<m)\\ 1+f(n,n-1) & (n=m)\\ f(n,m-1)+f(n-m,m) & (n>m>1)\end{cases}$$

计算 $f(n,m)$的递归函数代码如算法 3.4 所示。

正整数 n 的划分数 $p(n)=f(n,n)$。

算法 3.4(1) 正整数 n 的划分算法。

```
int split(int n,int m)
{
    if(n==1||m==1) return 1;
    else if (n<m) return split(n,n);
    else if(n==m) return split(n,n-1)+1;
    else return split(n,m-1)+split(n-m,m);
}
```

算法实现源代码:split.cpp。

由于重复搜索太多,简单的递归算法非常耗时。采用记忆式搜索,就是使用数组保存中间结果,以空间换时间,效率就会提高很多,如算法 3.4(2)所示。

算法 3.4(2) 正整数 n 的划分算法——采用记忆式搜索。

```
int s[110][110];
int split(int n,int m)
{
    if (s[n][m]) return s[n][m];              //查找已有结果
    int x=0;
    if(n==1||m==1) x=1;
    else if (n<m) x=split(n,n);
    else if(n==m) x=split(n,n-1)+1;
    else x=split(n,m-1)+split(n-m,m);
    s[n][m]=x;                                //保存当前结果
}
```

算法实现源代码:split-记忆式搜索.cpp。


视频讲解

3.2 分治策略

分治策略是对于一个规模为 n 的问题,若该问题可以很容易地解决(比如说规模 n 较小)则直接解决,否则将其分解为 k 个规模较小的子问题,这些子问题互相独立且与原问题

的形式相同。递归地求解这些子问题，然后将各子问题的解合并来得到原问题的解。

3.2.1　分治策略的基本步骤

分治策略在每一层递归上都有三个步骤。

(1) 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题。

(2) 解决：若子问题规模较小而容易被解决则直接求解，否则递归地求解各个子问题。

(3) 合并：将各个子问题的解合并为原问题的解。

分治策略的算法设计模式如算法 3.5 所示。

算法 3.5　分治策略的算法设计模式。

```
Divide_and_Conquer(P)
{
    if (|P|<=n0 ) return adhoc(P);
    divide P into smaller substances P1,P2,…,Pk;
    for (i=1; i<=k; k++)
        yi=Divide-and-Conquer(Pi)              //递归解决 Pi
    Return merge(y1,y2,…,yk)                   //合并子问题
}
```

其中，$|P|$表示问题 P 的规模；n_0 为一阈值，表示当问题 P 的规模不超过 n_0 时，问题已容易直接解出，不必再继续分解。adhoc(P)是该分治策略中的基本子算法，用于直接解小规模的问题 P。当 P 的规模不超过 n_0 时，直接用算法 adhoc(P)求解。算法 merge(y_1, y_2,…,y_k)是该分治策略中的合并子算法，用于将 P 的子问题 P_1,P_2,…,P_k 的解 y_1,y_2,…,y_k 合并为 P 的解。

分治策略的合并步骤是算法的关键所在。有些问题的合并方法比较明显，有些问题的合并方法比较复杂，或者是有多种合并方案，或者是合并方案不明显。究竟应该怎样合并，没有统一的模式，需要具体问题具体分析。

根据分治策略的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才最为适当？这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治策略设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。许多问题可以取 $k=2$。这种使子问题规模大致相等的做法是出自一种平衡(Balancing)子问题的思想，它几乎总是比子问题规模不等的做法要好。

3.2.2　分治策略的适用条件

分治策略能解决的问题一般具有以下几个特征：

(1) 该问题的规模缩小到一定的程度就可以很容易地解决。

(2) 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构的性质。

(3) 利用该问题分解出的子问题的解可以合并为该问题的解。

(4) 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

上述第 1 个特征是绝大多数问题都可以满足的，因为问题的计算复杂度一般是随着问

题规模的增加而增加；第 2 个特征是应用分治策略的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用；第 3 个特征是关键，能否利用分治策略完全取决于问题是否具有第 3 个特征，如果具备了第 1 个和第 2 个特征，而不具备第 3 个特征，则可以考虑贪心算法或动态规划算法；第 4 个特征涉及分治策略的效率，如果各子问题是不独立的，则分治策略要做许多不必要的工作，重复地求解公共的子问题，此时虽然可以使用分治策略，但一般用动态规划法较好。

3.2.3 二分搜索算法

二分搜索算法是运用分治策略的典型例子。

给定 n 个元素的数组 $a[0:n-1]$，需要在这 n 个元素中找出一个特定元素 x。

首先对 n 个元素进行排序，可以使用 C++标准模板库函数 sort()。

比较容易想到的是用顺序搜索方法，逐个比较 $a[0:n-1]$中的元素，直至找到元素 x 或搜索遍整个数组后确定 x 不在其中。因此在最坏的情况下，顺序搜索方法需要 $O(n)$次比较。二分搜索技术充分利用了 n 个元素已排好序的条件，采用分治策略的思想，在最坏情况下用 $O(\log_2 n)$时间完成搜索任务。

二分搜索算法的基本思想是将 n 个元素分成个数大致相同的两半，取 $a[n/2]$与 x 比较。如果 $x=a[n/2]$，则找到 x，算法终止。如果 $x<a[n/2]$，则只要在数组 a 的左半部分继续搜索 x。如果 $x>a[n/2]$，则我们只要在数组 a 的右半部分继续搜索 x。

由此得到利用分治策略在有序表中查找元素的算法，如算法 3.6 所示。

算法 3.6 二分搜索算法。

```
//数组 a[]中有 n 个元素,假定下标从 0 开始,已经按升序排序,待查找的元素为 x
template < class Type >
int BinarySearch(Type a[ ],const Type& x,int n)
{
    int left = 0;                                  //左边界
    int right = n - 1;                             //右边界
    while(left < = right)
    {
        int middle = (left + right)/2;             //中点
        if (x == a[middle]) return middle;         //找到 x,返回数组中的位置
        if (x > a[middle]) left = middle + 1;
          else right = middle - 1;
    }
    return  - 1;                                   //未找到 x
}
```

每执行一次算法的 while 循环，待搜索数组的大小减小一半。在最坏情况下，while 循环被执行了 $O(\log_2 n)$次。循环体内运算需要 $O(1)$时间，因此整个算法在最坏情况下的时间复杂度为 $O(\log_2 n)$。

例如，在有序表{7,14,17,21,27,31,38,42,46,53,75}中查找值为 21 时，初始状态如图 3-1 所示。此时 $n=11$，right$=n-1=10$，注意下标是从 0 开始的。

第一次查找时，middle$=5$，$a[5]=31\neq x$，而且 $a[5]>x$。显然，待查找的 x 在数组的左半

下标	0	1	2	3	4	5	6	7	8	9	10
数值	7	14	17	21	27	31	38	42	46	53	75
	↑					↑					↑
	left					middle					right

图 3-1　二分搜索算法的初始状态

部分。此时，改变区间的右边界 right＝middle－1＝5－1＝4，然后在 $a[0..4]$ 中查找即可。

3.2.4　循环赛日程表

设有 $n=2^k$ 个运动员要进行网球循环赛。现要设计一个满足以下要求的比赛日程表：

(1) 每个选手必须与其他 $n-1$ 个选手各赛一次；

(2) 每个选手一天只能参赛一次；

(3) 循环赛在 $n-1$ 天内结束。

请按此要求将比赛日程表设计成有 n 行和 $n-1$ 列的一个表。在表中的第 i 行第 j 列处填入第 i 个选手在第 j 天所遇到的选手，其中 $1\leqslant i\leqslant n$，$1\leqslant j\leqslant n-1$。

按分治策略，可以将所有的选手分为两半，则 n 个选手的比赛日程表可以通过 $n/2$ 个选手的比赛日程表来决定。递归地用这种一分为二的策略对选手进行划分，直到只剩下两个选手时，比赛日程表的制定就变得很简单。这时只要让这两个选手进行比赛就可以了。

图 3-2(c)是 8 个选手的比赛日程表。其中左上角与左下角的两小块分别为选手 1～4 和选手 5～8 前 3 天的比赛日程。据此，将左上角小块中的所有数字按其相对位置抄到右下角，又将左下角小块中的所有数字按其相对位置抄到右上角，这样就分别安排好了选手 1～4 和选手 5～8 在后 4 天的比赛日程。依此思想容易将这个比赛日程表推广到具有任意多个选手的情形。

1	2
2	1

(a) 2个选手

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

(b) 4个选手

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

(c) 8个选手

图 3-2　比赛日程表

我们得到利用分治策略安排循环赛日程表的算法，如算法 3.7 所示。

算法 3.7　循环赛日程表的算法。

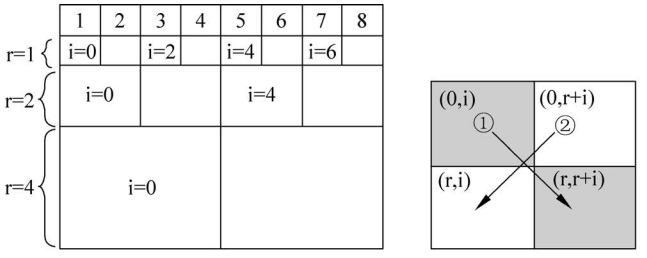
```
//当 k = 6 时,2^6 = 64,矩阵元素的输出宽度定义为 3;
//当 k>6 时,数组 a[]的大小 MAX 和矩阵元素的输出宽度都需要调整
#define MAX 100
int a[MAX][MAX];
```

```
//实现方阵的备份
//源方阵的左上角顶点坐标(fromx,fromy),行列数为 r
//目标方阵的左上角顶点坐标(tox,toy),行列数为 r
void Copy(int tox, int toy, int fromx, int fromy, int r)
{
    for (int i = 0; i < r; i++)
        for (int j = 0; j < r; j++)
            a[tox + i][toy + j] = a[fromx + i][fromy + j];
}

//构造循环赛日程表,选手的数量 n = 2^k
void Table(int k)
{
    int i, r;
    int n = 1 << k;
    //构造正方形表格的第一行数据
    for (i = 0; i < n; i++)
        a[0][i] = i + 1;
    //采用分治策略,构造整个循环赛日程表
    for (r = 1; r < n; r << = 1)
        for (i = 0; i < n; i += 2 * r)
        {
            Copy(r, r + i, 0, i, r);          //图 3-3(b)中的①
            Copy(r, i, 0, r + i, r);          //图 3-3(b)中的②
        }
}
```

比赛日程表的分治策略实现如图 3-3 所示。

算法实现源代码：round-robin.cpp。



(a) 8个选手时,分治策略划分的方阵　　(b) 一次循环的复制结构

图 3-3　比赛日程表的分治策略实现

3.2.5　半数集问题

给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数，如下所示：

(1) $n \in$ set(n)。

(2) 在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半。

(3) 按此规则进行处理，直到不能再添加自然数为止。

例如，set(6)＝{6,16,26,126,36,136}。半数集 set(6)中有 6 个元素。

注意，半数集是多重集，里面有相同的元素。范围 n 越大，重复的元素越多。如果 $n \leqslant 200$，请参考百度文库：半数单集问题。

对于给定的自然数 n，编程计算半数集 set(n)中的元素个数。

输入

数据有多行，给出整数 $n(0<n<1000)$。

输出

每个数据都输出 1 行，给出半数集 set(n)中的元素个数。

输入样例

```
6
23
```

输出样例

```
6
74
```

【算法分析】

设 set(n)中的元素个数为 $f(n)$，则显然有：

$$f(n)=1+\sum_{i=1}^{n/2} f(i)$$

以数字 12 为例，如表 3-3 所示：

表 3-3　数字 12 的半数集示例

第一次半数集	112	212	312	412	512	612
忽略原始数字 12	1	2	3	4	5	6
第二次及其后的分解		12	13	14 24,124	15 25,125	16 26,126 36,136

递归算法的设计，如算法 3.8 所示。

算法 3.8　计算半数集问题的递归算法。

```
int comp(int n)
{
    int ans = 1;
    if (n > 1) for(int i = 1;i < = n/2;i++)
        ans += comp(i);
    return ans;
}
```

上述算法中显然有很多重复的子问题计算。使用数组存储已计算过的结果，避免重复计算，可以明显改进算法的效率。改进后的算法如算法 3.9 所示。

算法 3.9　计算半数集问题的递归算法——记忆式搜索。

```
int a[1001];
```

```
int comp(int n)
{
    int ans = 1;
    if(a[n]> 0)return a[n];                    //已经计算
    for(int i = 1;i < = n/2;i++)
        ans += comp(i);
    a[n] = ans;                                //保存结果
    return ans;
}

//主函数 main()中数据的读取与调用
int n;
while(cin >> n)
{
    memset(a,0,sizeof(a));
    a[1] = 1;
    cout << comp(n)<< endl;
}
```

算法实现源代码：Halfnumber.cpp。

半数单集问题，请参考代码：半数单集问题.cpp。


视频讲解

3.2.6 整数因子分解

大于 1 的正整数 n 可以分解为：$n=x_1\times x_2\times\cdots\times x_m$。

例如，当 $n=12$ 时，共有 8 种不同的分解式：

$$12=12$$
$$12=6\times 2$$
$$12=4\times 3$$
$$12=3\times 4$$
$$12=3\times 2\times 2$$
$$12=2\times 6$$
$$12=2\times 3\times 2$$
$$12=2\times 2\times 3$$

对于给定的正整数 n，编程计算 n 共有多少种不同的分解式。

输入

数据有多行，给出正整数 $n(1\leqslant n\leqslant 2\ 000\ 000\ 000)$。

输出

每个数据输出 1 行，是正整数 n 的不同的分解式数量。

输入样例

```
12
35
```

输出样例

```
8
3
```

【算法分析】

对 n 的每个因子递归搜索，如算法 3.10 所示。

算法 3.10　整数因子分解的算法。

```
int total;                                          //定义为全局变量
void solve(int n)
{
    if (n == 1) total++;                            //获得一个分解
    else for (int i = 2; i <= n; i++)
        if (n % i == 0) solve(n/i);
}

//主函数 main()中数据的读取与调用
int n;
while(scanf("%d",&n)!= EOF)
{
    total = 0;
    solve(n);
    printf("%d\n",total);
}
```

算法实现源代码：factorization.cpp。

当 $n=12$ 时，整数因子分解的递归过程如图 3-4 所示。

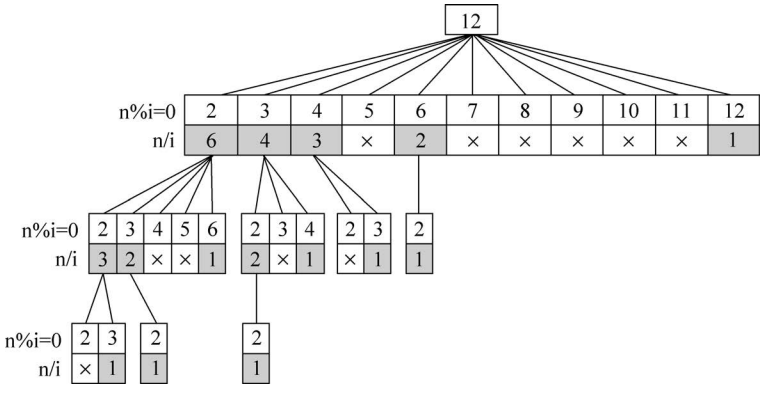


图 3-4　当 $n=12$ 时，整数因子分解的递归过程

3.2.7　取余运算

【问题描述】

输入三个正整数 a，p，k，求 $a^p \% k$ 的值。

输入

输入有多组测试例。对每组测试例，都有三个正整数 a，p，k（$0<a,p,k\times k<2^{32}$）。

输出

对每组测试例都输出 1 行，是 $a^p \% k$ 的值。

输入样例

```
2 10 9
3 18132 17
```

输出样例

```
7
13
```

【算法分析】

由于数据的规模很大,如果直接计算,不仅需要采用高精度,而且时间复杂度很大。例如,$10^{25}\%7=3$,但 10^{25} 超出了整数的表示范围,不能直接计算。此问题的特点是 k 在整数(int)范围内,因此结果必然在整数(int)范围内。

模运算有如下运算规则:

$$(a\times b)\%n=(a\%n\times b\%n)\%n \tag{3.1}$$

$$a^b\%n=((a\%n)^b)\%n \tag{3.2}$$

根据公式(3.2),$10^{25}\%7=(10\%7)^{25}\%7=3^{25}\%7$,显著降低了 a 的值。

根据公式(3.1),$3^{25}\%7=(3\times3^{12}\times3^{12})\%7=(3\times3^{12}\%7\times3^{12}\%7)\%7$,显著降低了 p 的值。

因此,得到如下递推公式:

$$a^p\%k=\begin{cases}a\%k & (p=1)\\ (a\times a^{p-1}\%k)\%k & (p\text{ 是奇数})\\ ((a\times a)\%k)^{p/2}\%k & (p\text{ 是偶数})\end{cases}$$

该递推公式体现了分治策略的应用,将一个大的指数 p 逐渐减小,同时对运算过程的中间结果不断进行模运算,降低中间结果的数字大小,避免了使用高精度运算。

通常将该算法称为快速幂算法。

利用递推公式实现取余运算,如算法 3.11 所示。

算法 3.11 利用递推公式实现取余运算的算法。

```
//计算 a^p % k 的值
//为了防止运算过程中的溢出,采用 64 位整数,在 C++环境中是 long long
int mod(__int64 a, __int64 p, __int64 k)
{
    if (p == 1) return a % k;
    if (p % 2) return mod(a % k, p - 1, k) * a % k;   //p 是奇数
    else return mod((a * a) % k, p/2, k);             //p 是偶数
}

//主函数 main()中数据的读取与调用
unsigned a, p, k;
while (scanf("%u%u%u",&a,&p,&k)!= EOF)
    printf("%d\n", mod(a, p, k));
```

算法实现源代码:moduloArithmetic.cpp。

3.3　ZOJ1633-Big String

【问题描述】

设 A＝“^__^”(4 个字符)，B＝“T. T”(3 个字符)，然后以 AB 为基础，构造无限长的字符串。重复规则如下：

(1) 把 A 接在 B 的后面构成新的字符串 C。例如，A＝“^__^”，B＝“T. T”，则 C＝BA＝“T. T^__^”。

(2) 令 A＝B，B＝C，如上例所示，则 A＝“T. T”，B＝“T. T^__^”。

编程任务：给出此无限长字符串中的第 n 个字符。

输入

输入有多组测试例。每个测试例只有一个整数 $n(1\leqslant n\leqslant 2^{63}-1)$。

输出

对每个测试例输出一行，是此无限长字符串中的第 n 个字符(序号从 1 开始)。

输入样例

```
1
2
4
8
```

输出样例

```
T
.
^
T
```

题目来源

Zhejiang University Local Contest 2003

【算法分析】

本题看起来很简单，字符串的组合也很有规律，有的同学就试图研究叠加后的字符串规律。结果发现，叠加后的字符串虽然有规律，但是与输入的数据 n 之间没有直接的联系。

1. 从字符串的长度研究字符串生成规律

```
a = strlen("^__^")     → a = 4
b = strlen("T.T")      → b = 3
c = strlen("T.T^__^") → c = 7
```

再按照题目给定的步骤重复，很容易发现，这正是以 a，b 为基数的斐波那契数列。

对于输入的正整数 n，它位于经过若干次按斐波那契数列的规律叠加后的字符串中。无论它如何叠加，该位置的字符总是在字符串 C 中。本题就变成给定一个正整数 n，求出小于 n 的最大斐波那契数，n 与该斐波那契数的差正是该字符在字符串 C 中的位置。

输出时要注意，字符串的位置是从 0 开始编号的，所以用这个差值当下标时需要减去 1。

2. 算法优化

由于 n 最大可达 $2^{63}-1$,对于输入的每个 n,都去计算小于 n 的最大斐波那契数显然是非常浪费时间的。解决的办法是预先把在 $2^{63}-1$ 范围内的所有斐波那契数求出来,放到一个数组中。经过测算,该斐波那契数列最多为 86 项,第 86 项的斐波那契数约为 6.02×10^{18},而 $2^{63}-1$ 约为 9.22×10^{18},如算法 3.12 所示。

算法 3.12 利用斐波那契数列计算无限长字符串第 n 个字符的算法

```
#define LEN 88
string base = "T.T^__^";
//将斐波那契数列在 2^63 - 1 之内的数全部计算出来
long long int f[LEN];
f[0] = 7; f[1] = 10;
for(int i = 2; i < LEN; i++)
    f[i] = f[i - 1] + f[i - 2];
long long int n;
while(cin >> n)
{
    //对于每一个 n,减去小于 n 的最大斐波那契数
    while(n > 7)
    {
        int i = 0;
        while (i < LEN && f[i]< n)
            i++;
        n -= f[i - 1];
    }
    //n 中剩下的值,就是该字符在 base 中的位置
    cout << base[n - 1]<< endl;
}
```

算法实现源代码:zju1633-Big String.cpp。

3.4 洛谷 P1182 数列分段 Section Ⅱ

对于给定的一个长度为 N 的正整数数列 $A_{1\sim N}$,现要将其分成 $M(M\leqslant N)$段,并要求每段连续,且每段和的最大值最小。

关于最大值最小:例如数列 4 2 4 5 1 要分成 3 段。

将其如下分段:[4 2][4 5][1]

第一段和为 6,第 2 段和为 9,第 3 段和为 1,和最大值为 9。

将其如下分段:[4][2 4][5 1]

第一段和为 4,第 2 段和为 6,第 3 段和为 6,和最大值为 6。

并且无论如何分段,最大值不会小于 6。所以要分成 3 段,每段和的最大值最小为 6。

输入格式

第 1 行包含两个正整数 N,M。

第 2 行包含 N 个非负整数 A_i,含义如题目所述。

输出格式

一个正整数，即每段和最大值最小为多少。

输入样例

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

输出样例

```
6
```

【算法分析】

要将一个长度为 N 的正整数数列分成 M 段，并要求每段连续，同时使得每段和的最大值最小，我们可以使用二分搜索结合贪心策略来解决。

目标是找到一个分割点，使得分割后的每段和尽可能均衡。为此，我们可以使用二分搜索来找到一个合适的阈值 x，使得所有段的和都不超过 x。

贪心策略的部分在于，我们总是尝试将尽可能多的数放入当前段，直到和超过 x 为止，然后开始一个新的段。这样可以确保我们尽可能减少段的数量，同时保持每段和不超过 x。

下面是一个使用二分搜索和贪心策略来解决这个问题的算法，如算法 3.13 所示。

算法 3.13　使用二分搜索和贪心策略解决数列分段的算法。

```
int n,m,a[100005],ans;
//采用贪心策略,验证阈值 x 是否合适
bool check(int x)
{
    int cnt = 0,num = 0;
    for(int i = 1; i <= n; i++)
    {
        if(cnt + a[i]<= x) cnt += a[i];          //尽可能多
        else cnt = a[i],num++;                   //新增加一段
    }
    return num >= m;
}
//在主函数 int main()中
scanf("%d%d",&n,&m);
int left = 0,r;
for(int i = 1; i <= n; i++)
{
    scanf("%d",&a[i]);
    left = max(left,a[i]);                       //左边界是数列的最大值
    r += a[i];                                   //右边界是数列的和
}
while(left <= r)                                 //二分算法找阈值
{
    int mid = left + r >> 1;
    if(check(mid)) left = mid + 1;
    else r = mid - 1;
}
cout << left;
```

check 函数检查是否可以将数组 a 分成 M 段，每段和不超过 x。对于每个二分搜索中的阈值 x，我们需要遍历整个数列来确定是否可以将数列分割成 M 段，每段和不超过 x。这个遍历过程的时间复杂度是 $O(N)$，因为我们只需要一次遍历就可以确定是否满足条件。

如果考虑二分搜索中的每次迭代都进行这样的一次遍历，那么总的时间复杂度将是 $O(N\log(\text{sum}))$，其中 sum 是数列所有元素的总和。这是因为二分搜索的每次迭代都可能触发一次 $O(N)$的遍历。

算法实现源代码：P1182 数列分段 Section II. cpp。

3.5 洛谷 P1824 进击的奶牛

农夫 John 建造了一个有 $N(2\leqslant N\leqslant 10^5)$个隔间的牛棚，这些隔间分布在一条直线上，坐标是 $x_1,\cdots,x_N(0\leqslant x_i\leqslant 10^9)$。

他的 $C(2\leqslant C\leqslant N)$头奶牛不满于隔间的位置分布，它们为牛棚里其他奶牛的存在而愤怒。为了防止奶牛之间互相打斗，农夫 John 想把这些奶牛安置在指定的隔间，所有奶牛中相邻两头的最近距离越大越好。那么，这个最大的最近距离是多少呢？

输入格式

第 1 行：两个用空格隔开的数字 N 和 C。

第 $2\sim N+1$ 行：每行一个整数，表示每个隔间的坐标。

输出格式

输出只有一行，即相邻两头奶牛最大的最近距离。

输入样例

```
5 3
1 2 8 4 9
```

输出样例

```
3
```

题目给的坐标是无序的，我们需要排序，如样例数据排序后是 1 2 4 8 9。只有 3 头奶牛，分别放在 1 4 9，则相邻两头奶牛最大的最近距离是 3。

【算法分析】

使用二分搜索算法寻找相邻两头奶牛之间的最大最近距离是一种有效的方法，特别是当知道最大最近距离的可能范围时，二分搜索可以在对数时间内获得最优解。二分搜索需要数据是有序的，就需要先对坐标排序。

在二分搜索中，需要定义一个搜索范围，比如最小可能距离 l 为 0 和最大可能距离 r 为所有隔间的跨度。然后，不断将搜索范围一分为二，计算中间值 mid，并检查是否存在一种奶牛的排列方式，使得所有相邻两头奶牛之间的距离都不小于 mid。

在 check()函数中，因为想向右收缩，所以应该是给定的 x 可以装下更多的奶牛，这就要用到贪心策略。贪心策略是指在左手边第一个，必须安排一头奶牛，这样才能使得隔间利用最大化！其他的奶牛，看看两个隔间之间的距离是不是大于或等于 x，满足条件就意味着能够安排一头奶牛。变量 sum 是当前阈值 x 时安排奶牛的数量，每次安排完一头奶牛就计

数一次，最终判断 sum 是否大于或等于奶牛数 C。

计算相邻两头奶牛之间的最大最近距离的二分＋贪心的算法，如算法 3.14 所示。

算法 3.14　计算相邻两头奶牛之间的最大最近距离的二分＋贪心的算法。

```
int n,c;                                    //n－隔间的数量,c－奶牛的数量
int cow[1000001];                           //隔间坐标
bool check(int x)
{
    int right = cow[1] + x;                 //贪心策略,必须使用第 1 隔间
    int sum = 1;                            //已经使用 1 个隔间
    for(int i = 2; i <= n; i++)
        if(cow[i]>= right)                  //必须使用新的隔间
        {
            sum++;
            right = cow[i] + x;
        }
    return sum >= c;
}
//在主函数 int main()中
cin >> n >> c;
for(int i = 1; i <= n; i++)
    cin >> cow[i];                          //隔间坐标
sort(cow + 1,cow + 1 + n);                  //对坐标排序
//应用二分搜索,快速获取最优值
int l = 0,r = cow[n] - cow[1];
while(l <= r)
{
    int mid = (l + r)/2;
    if(check(mid)) l = mid + 1;
    else r = mid - 1;
}
cout << r << endl;
```

二分搜索的搜索空间就是隔间的跨度 D，二分搜索将搜索空间逐渐缩小，每次迭代都将搜索空间减半。因此，二分搜索的时间复杂度是 $O(\log D)$。但是每一次搜索，都需要检查阈值 x 是否合适。检查函数 check()采用贪心策略，每一步都尝试放置奶牛以最大化与已放置的奶牛之间的距离。贪心算法需要遍历所有隔间，时间复杂度是 $O(N)$，其中 N 是隔间的数量。当我们将二分搜索和贪心算法结合起来时，总体时间复杂度是 $O(N\log D)$。这是因为对于每次二分搜索中的不同距离，我们都运行一次贪心算法来检查该距离是否可行。

算法实现源代码：P1824 进击的奶牛.cpp。

3.6　洛谷 P1873-砍树

伐木工人 Mirko 需要砍 Mm 长的木材。他有一个漂亮的新伐木机，可以快速地砍伐树木。不过，他只被允许砍伐一排树。Mirko 设置一个高度参数 H(m)，伐木机升起一个巨大的锯片到高度 H，并锯掉所有树比 Hm 高的部分(当然，树木不高于 Hm 的部分保持不变)，Mirko 就得到树木被锯下的部分。例如，如果一排树的高度分别为 20、15、10 和 17，Mirko 把锯片升到 15m 的高度，切割后树木剩下的高度将是 15m、15m、10m 和 15m，而他

将从第 1 棵树得到 5m,从第 4 棵树得到 2m,共得到 7m 木材。

Mirko 非常关注生态保护,所以他不会砍掉过多的木材。这也是他尽可能高地设定伐木机锯片的原因。请帮助他找到伐木机锯片的最大的整数高度 H,使得他能得到的木材至少为 Mm。换句话说,如果再升高 1m,他将得不到 Mm 木材。

输入格式

第 1 行 2 个整数 N 和 M,N 表示树木的数量,M 表示需要的木材总长度。

第 2 行 N 个整数表示每棵树的高度。

输出格式

1 个整数,表示锯片的最高高度。

输入样例

```
4 7
20 15 10 17
```

输出样例

```
15
```

【算法分析】

Mirko 想要通过设置一个高度参数 H 来锯掉所有比 H 高的树的部分,从而得到树木被锯下的部分。我们可以考虑二分搜索来确定最佳的 H 值,使得锯下的部分长度在满足 M 的情况下最少。这里的关键是确定一个搜索范围,即最小高度 l 和最大高度 r,在这个范围内进行二分搜索。

每次迭代中,我们计算阈值 x,然后模拟锯木过程,计算锯下部分的总长度 sum。根据这个结果,我们调整搜索范围,继续搜索。如果提高砍树的高度 x,逐渐增加时,砍下的木材数量逐渐下降。这样,当 x 上升到某个确定的高度时,砍下的木材数量将少于需要的值 M,此时高度 H 减 1 的位置就是所求答案。本题 M 的数据范围比较大,求和时 sum 需要用到 long long 类型。

利用二分搜索求解的算法,如算法 3.15 所示。

算法 3.15 利用二分搜索求解的算法。

```
int n,m;                                    //n-树木的数量,m-需要的木材总长度
int h[1000010];
bool check(int x)
{
    long long sum = 0;                      //树木被锯下部分的长度和
    for(int i = 1; i <= n; i++)
        if(x < h[i]) sum += h[i] - x;
    return sum >= m;
}
//在主函数 int main()中
cin >> n >> m;
int r = 0,l = 1;                            //初始边界
for(int i = 1; i <= n; i++)
{
```

```
        cin >> h[i];
        r = max(r,h[i]);                    //r 为树木的最大高度
    }
    while(l <= r)
    {
        int mid = (l + r)/2;
        if(check(mid)) l = mid + 1;
        else r = mid - 1;
    }
    cout << l - 1;
```

二分搜索算法的时间复杂度是 $O(\log H_{\max})$，其中 $H_{\max}$ 是所有树中的最大高度。在这个问题中，搜索范围由树的最小高度和最大高度决定。因此，二分搜索可以在对数时间内快速逼近最佳的 H 值。但是每一次搜索，都需要检测，而检测的时间复杂度是 $O(n)$，其中 n 是树木的数量。因此该算法的时间复杂度是 $O(n\log H_{\max})$。

算法实现源代码：P1873-砍树.cpp。

3.7　洛谷 P1908 逆序对

在 vjudge.net 中搜索“逆序对”，会找到很多资源。

最近，TOM 老猫查阅到一个人类称为“逆序对”的东西，这东西是这样定义的：对于给定的一段正整数序列，逆序对就是序列中 $a_i > a_j$ 且 $i < j$ 的有序对。知道这个概念后，他们就比赛谁先算出给定的一段正整数序列中逆序对的数目。注意序列中可能有重复数字。

输入格式

第一行，一个数 n，表示序列中有 n 个数。

第二行 n 个数，表示给定的序列。序列中每个数字不超过 10^9。

输出格式

输出序列中逆序对的数目。

输入样例

```
6
5 4 2 6 3 1
```

输出样例

```
11
```

【算法分析】

使用归并排序来计算逆序对的数量是一个很好的方法，因为归并排序是稳定排序。在归并排序的过程中，我们不断将数组分为两半，分别排序，然后再合并两个有序数组。在归并排序的过程中，当合并两个有序子数组时，我们可以计算出跨越这两个子数组的逆序对数量。

为了计算逆序对，我们可以在合并两个已排序子数组时做如下操作。

(1) 初始化一个计数器 ans 为 0，用于记录逆序对的数量。

定义两个指针 i 和 j，分别指向两个子数组的第一个元素。

(2) 比较两个指针所指向的元素：

如果第一个子数组的元素 $a[i]$ 大于第二个子数组的元素 $a[j]$，那么 $a[i]$ 与 $a[j]$ 及其后面的所有元素都会构成逆序对。因此，将 $j-k$（其中 k 是归并数组的最后位置，或者是 mid$-i+1$）加到 ans 上，并将 j 向后移动一位。

如果 $a[i]$ 小于或等于 $a[j]$，将 i 向后移动一位。

(3) 重复上述步骤，直到其中一个子数组的所有元素都被处理完。

(4) 将剩余的子数组中的元素直接复制到结果数组中。

使用归并排序计算逆序对的数量的算法如算法 3.16 所示。

算法 3.16 使用归并排序计算逆序对的数量的算法。

```
long long ans = 0;                              //逆序对数
int a[500010],b[500010];                        //数组 a 是原始数据
//使用归并排序计算逆序对的数量
void merge(int l,int r)
{
    if (l == r) return;
    int mid  = l + r >> 1;
    merge(l,mid), merge(mid + 1,r);
    int i = l, j = mid + 1, k = l;
    while(i <= mid &&j <= r)
    {
        if (a[i]> a[j])                         //产生逆序对
        {
            b[k++] = a[j++];
            //或者 ans += mid - i + 1
            ans += j - k;                       //累加逆序对的距离
        }
        else b[k++] = a[i++];
    }
    while (i <= mid) b[k++] = a[i++];
    while (j <= r) b[k++] = a[j++];
    for (int i = l; i <= r; i++)
        a[i] = b[i];
}
//在主函数 int main()中
int n;
cin >> n;
for (int i = 0; i < n; i++) cin >> a[i];
merge(0,n - 1);
cout << ans << endl;
```

这个算法的时间复杂度是 $O(n\log n)$，其中 n 是序列的长度。这是因为归并排序本身的时间复杂度就是 $O(n\log n)$，而在合并过程中计算逆序对的操作也是线性的。

算法实现源代码：P1908 逆序对.cpp。

上机练习题

汉诺塔类的题目

浙江大学 ZOJ	1239-Hanoi Tower Troubles Again! 2338-The Towers of Hanoi Revisited 2954-Hanoi Tower
北京大学 POJ	1920-Towers of Hanoi 3572-Hanoi Towers 3601-Tower of Hanoi 1958-Strange Towers of Hanoi

斐波那契数列类的题目

浙江大学 ZOJ	1828-Fibonacci Numbers 2060-Fibonacci Again 2672-Fibonacci Subsequence
杭州电子科技大学 HDOJ	1021-Fibonacci Again 1250-Hat's Fibonacci 1316-How Many Fibs? 1568-Fibonacci 1588-Gauss Fibonacci 1708-Fibonacci String 1848-Fibonacci Again and Again 2018-母牛的故事 2070-Fibonacci Number 2814-Interesting Fibonacci 2855-Fibonacci Check-up 3054-Fibonacci 3117-Fibonacci Numbers 3306-Another Kind of Fibonacci 3509-Buge's Fibonacci Number Problem 4099-Revenge of Fibonacci 4786-Fibonacci Tree

由于在线题目很多，这里只列出了部分题目，仅供参考。