算法与复杂性

SHEN Jiamin

2020 年 6 月 13 日

本合集并非本人上交作业的原始版本。

目　录

1 0302

1. 求解一元二次方程 $ax^2 + bx + c = 0$

> 首先使用根的判别式判断方程根的情况，然后使用求根公式求解。
>
> 有观点认为应当先判断 a 是否为零，但我认为一元二次方程已经隐含了 $a \neq 0$ 的条件，无需额外判断。
>
> **伪代码见算法1**

算法 1 求实系数一元二次方程的根

输入: 实系数一元二次方程的系数 a, b, c

输出: 方程的根 x_1, x_2

$\Delta \leftarrow b^2 - 4ac$ ▷ 根的判别式

if $\Delta > 0$ **then** ▷ 方程有两个不同的实数根

  $x_1 \leftarrow \frac{-b+\sqrt{\Delta}}{2a}$

  $x_2 \leftarrow \frac{-b-\sqrt{\Delta}}{2a}$

else if $\Delta = 0$ **then** ▷ 方程有两个相等的实数根

  $x_{1,2} \leftarrow -\frac{b}{2a}$

else if $\Delta < 0$ **then** ▷ 方程有两个不同的复数根

  $x_1 \leftarrow \frac{-b+\boldsymbol{i}\sqrt{-\Delta}}{2a}$ ▷ 其中 $\boldsymbol{i}^2 = -1$

  $x_2 \leftarrow \frac{-b-\boldsymbol{i}\sqrt{-\Delta}}{2a}$

end if

return x_1, x_2

2. 有一堆棋子，A 和 B 两人轮流从中拿 1-3 个，A 第一个拿，那么 A 如何确保自己不拿到最后一个棋子

> 若每次 B 拿 b_i 个棋子（$b_i \in [1,3], b_i \in \mathbb{N}$）之后，A 都拿 $a_{i+1} = 4 - b_i$ 个棋子，使 $b_i + a_{i+1} = 4$，则最后剩余棋子的个数可以认为与 B 的决策无关。要保证 A 不拿到最后一个棋子，则 A 最后一次拿取之后必须只剩 1 个棋子。否则，B 可以拿取若干棋子使剩余棋子个数为 1，A 将不得不拿到最后一个棋子。由此可知，只要 A 在每一步拿取棋子时，都尽力保证剩余棋子的个数 N 满足 $N \equiv 1 \bmod 4$ 即可。
>
> 注：A 的策略只依赖于其决策时的状态。既不依赖于历史状态，也不依赖于 B 的决策细节。
>
> **伪代码见算法2**

算法 2 巴什博弈

输入: 轮到 A 拿棋子时，剩余棋子的个数 n

输出: A 本次拿取棋子的个数

```
take ← [(n − 1) mod 4]
if take > 0 then
    return take                          ▷ A 此次拿 take 个棋子，可满足前述条件
else
    return 1                             ▷ A 不可能通过此次拿取保证前述条件，可随意拿若干棋子
end if
```

2 0305

1. 数列 $1, 2, 3, 4, 5, 10, 20, 40, \ldots$，该数列开始是等差数列，第 5 项以后为等比数列，证明任意一个正整数都可表示为这个数列中的不同数之和。

(a)

引理 1 对任意的 $n \in \mathbb{N}^*, n < 2^k$，$n$ 可以表示为集合 $\{2^i \mid 0 \le i < k\}$ 中任意个不同数之和。

证明 定义集合 S_k:

1. $\{2^i \mid 0 \le i < k\} \subset S_k$
2. 若 $a \in \mathbb{N}^*$ 可表示为集合 $\{2^i \mid 0 \le i < k\}$ 中任意多个不同数之和，则 $a \in S_k$

考察集合 S_k:

1. 当 $k = 1$ 时，$S_1 = \{1\}$
2. 当 $k = 2$ 时，$S_2 = \{1, 2, 3\}$
3. 假设当 $k = n$ 时 $S_n = \{a \in \mathbb{N} \mid 1 \le a < 2^n\}$
   则当 $k = n + 1$ 时，

$$\begin{aligned} S_{n+1} &= S_n \cup \{2^n\} \cup \{2^n + a \mid a \in S_n\} \\ &= \{a \in \mathbb{N} \mid 1 \le a < 2^n\} \cup \{2^n\} \cup \left\{a \in \mathbb{N} \mid 2^n + 1 \le a < 2^{n+1}\right\} \\ &= \left\{a \in \mathbb{N} \mid 1 \le a < 2^{n+1}\right\} \end{aligned}$$

归纳可知 $S_k = \left\{a \in \mathbb{N} \mid 1 \le a < 2^k\right\}$，即引理1成立。 □

(b) 原数列通项公式可表示为

$$a_i = \begin{cases} i, & 1 \le i < 5 \\ 5 \cdot 2^{i-5} & i \ge 5 \end{cases}$$

对于任意正整数 $n \in \mathbb{N}^*$,

1. 若 $n \leq 5$，易知 $n \in \{a_i\}$，即由数列中的单个数即可表示。
2. 由引理1和整数加法的线性性可推知，$\forall n \in \mathbb{N}^*, 5|n, n < 5 \cdot 2^k$，$n$ 可以表示为集合 $\{5 \cdot 2^i \mid 0 \leq i < k\}$ 中任意个不同数之和。所以若 $5|n$，则 n 可以表示为 $\{a_i \mid i \geq 5\}$ 中任意个不同数之和。

$$\mathbb{N}^* = \{5x + y \mid x \in \mathbb{N}, y \in \{0, 1, 2, 3, 4\}\}$$

其中 $5x$ 可以表示为 $\{a_i \mid i \geq 5\}$ 中任意个不同数之和；又 $n \in \{a_i \mid 1 \leq i < 5\}$。归纳可知，任意正整数可以表示为 $\{a_i\}$ 中任意个不同数之和。

证明 该数列通项公式可表示为

$$a_i = \begin{cases} i, & 1 \leq i < 5 \\ 5 \cdot 2^{i-5} & i \geq 5 \end{cases}$$

对任意的 $n \in \mathbb{N}^*$，存在唯一一组 (t, b, q) 满足 $t \in \mathbb{N}, b \in \{0, 1\}, q \in \{1, 2, 3, 4\}$，使得

$$n = 5t + bq$$

t 有唯一的二进制表示，使 $t = \sum_k b_k \cdot 2^k$，其中 $b_k \in \{0, 1\}$

即

$$n = 5 \cdot \sum_k b_k \cdot 2^k + bq = \sum_k b_k \cdot (5 \cdot 2^k) + bq$$

因为 $b_k, b \in \{0, 1\}$，不可能有重复的加数。又由题可知,

$$\left\{5 \cdot 2^k\right\} = \{a_i \mid i \geq 5\}$$
$$\{q\} = \{a_i \mid 1 \leq i < 5\}$$

所以任意的 $n \in \mathbb{N}^*$ 可以表示为 $\{a_i\}$ 中任意多不同数之和。 □

2. 广场上站着 99 个间谍，间谍与间谍之间的距离互不相等，每个间谍都盯着离自己最近的那个间谍看，证明总存在一个没被人盯着的间谍。

考虑距离最近的两个间谍，显然他们俩正互相盯着。

- 如果还有别人盯着他们俩中的任何一个，就表明有人同时被两个人盯着，因此必然存在另一个人没被人盯着；
- 如果没有别人在盯这两个人，那么我们就可以去掉这两个人，这对其他人不会产生任何影响。

注意到广场上的总人数是个奇数，因此如此继续下去，要么我们能在某一步找到一个没被盯着的人，要么最终就只剩下一个人，而他显然没有被任何人盯着。

3. 有 10 个海盗抢得了 100 枚金币，每个海盗都能够很理智地判断自己的得失，他们决定这样分配金币：

   1. 按照强壮与否排序，其中最强壮的人为 10 号，以此类推，最瘦小的人为 1 号。
   2. 先由 10 号提出分配方案，然后由所有人表决，当且仅当**等于或多于半数人**（包括自己）同意时，方案才算被通过，否则他将被扔入大海喂鲨鱼；
   3. 如果 10 号死了，将由 9 号提方案，其余的人表决，当且仅当**超过半数**（包括自己）同意时，方案才算通过，否则 9 号同样将被扔入大海喂鲨鱼；
   4. 往下以此类推……

   海盗们都很精明，他们首先会尽量保住自己的命，其次在保住命的前提下都想分到尽可能多的金币，而且他们也很希望自己的同伴喂鲨鱼。

   (a) 假如你是那个 1 号海盗，你将怎样分配，才能既保住命，又能分到最多的金币？最多能分到多少呢？

   (b) 如果还是 100 枚金币，但海盗的数量是 20，50，100，200，400 又该怎么样呢？

设有 n 个海盗时，n 号海盗提出的方案中，j 号海盗分得的金币数量为 $c_n(j)$。

当仅剩两人（1、2 号）时，无论 2 号提出怎样的方案，1 号都会选择不同意从而把 2 号扔进大海并分得全部金币。所以，当剩余三人时（1、2、3 号），无论 3 号提出怎样的方案，2 号都会同意 3 号的方案以避免出现上述情况。因此，3 号会提出一个

$$c_3(j) = \begin{cases} 100 & , j = 3 \\ 0 & , j = 1, 2 \end{cases}$$

的方案，2、3 号海盗会同意这个方案。

在上述方案中，1、2 号海盗都没有得到金币。因此，当由 4 号海盗提出方案时，只要他分给 1 号或 2 号海盗 1 个金币，即可赢得其支持。如

$$c_4(j)=\begin{cases}99 & ,j=4\\ 1 & ,j=2\\ 0 & ,j=1,3\end{cases}$$

当由 5 号海盗提出方案时，需要 3 人同意方可通过。代价最小的方案为在 c_4 的基础上多分给 1、3 号各一个金币。即

$$c_5(j)=\begin{cases}98 & ,j=5\\ 1 & ,j=1,3\\ 0 & ,j=2,4\end{cases}$$

当由 6 号海盗提出方案时，需要 3 人同意即可通过。代价最小的方案为 c_5 基础上分给 2、4 号一个金币。如

$$c_6(j)=\begin{cases}98 & ,j=6\\ 1 & ,j=2,4\\ 0 & ,j=1,3,5\end{cases}$$

已知　当有 n 个海盗时，须有 $\lceil\frac{n}{2}\rceil=\begin{cases}\frac{n+1}{2} & ,n\text{为奇数}\\ \frac{n}{2} & ,n\text{为偶数}\end{cases}$ 个海盗支持方可通过方案。

猜想　当 $3<n\leq 200$ 时，n 号海盗提出如下方案可保住自己的命：

$$c_n(j)=\begin{cases}101-\lceil\frac{n}{2}\rceil & ,j=n\\ 1 & ,j<n,j+n\text{为偶数}\\ 0 & ,j<n,j+n\text{为奇数}\end{cases}$$

证明　下面使用数学归纳法证明上述猜想。

1. 前已论述 $n\leq 6$ 时的情况
2. 假设 $n=k$ 时猜想成立
   - 若 k 为奇数，则 k 号海盗可分得 $101-\frac{k+1}{2}$ 个金币。且有 $\frac{k+1}{2}-1$ 个奇数号海盗获得了 1 个金币，$\frac{k-1}{2}$ 个偶数号海盗没有获得金币。
   - 若 k 为偶数，则 k 号海盗可分得 $101-\frac{k}{2}$ 个金币。且有 $\frac{k}{2}-1$ 个偶数号海盗获得了 1 个金币，$\frac{k}{2}$ 个奇数号海盗没有获得金币。

则当 $n=k+1$ 时

- 若 k 为奇数，$k+1$ 为偶数。$k+1$ 号海盗须得到另外 $\frac{k-1}{2}$ 个海盗的支持方可通过方案。因此，只需给 $n=k$ 时没有分得金币的 $\frac{k-1}{2}$ 个偶数号海盗分 1 个金币即可赢得他们的支持。即 $k+1$ 为偶数时，

$$c_{k+1}(j)=\begin{cases}100-\frac{k-1}{2}=101-\lceil\frac{k+1}{2}\rceil & ,j=k+1\\ 1 & ,j<k+1,j\text{为偶数}\\ 0 & ,j<k+1,j\text{为奇数}\end{cases}$$

- 若 k 为偶数，$k+1$ 为奇数。$k+1$ 号海盗须得到另外 $\frac{k}{2}$ 个海盗的支持方可通过方案。因此，只需给 $n=k$ 时没有分得金币的 $\frac{k}{2}$ 个奇数号海盗分 1 个金币即可赢得他们的支持。即 $k+1$ 为奇数时，

$$c_{k+1}(j)=\begin{cases}100-\frac{k}{2}=101-\lceil\frac{k+1}{2}\rceil & ,j=k+1\\ 1 & ,j<k+1,j\text{为奇数}\\ 0 & ,j<k+1,j\text{为偶数}\end{cases}$$

综合可知上述猜想成立。 □

由此可知，

- 当 $n=200$ 时，$c_{200}(j)=\begin{cases}1 & ,j<200,j\text{为偶数}\\ 0 & ,j<200,j\text{为奇数}\end{cases}$
- 当 $n=201$ 时，201 号海盗必须获得除他自己以外的另外 100 名海盗的支持。因此，他须分给 1-200 号海盗中 100 名奇数号海盗各 1 枚金币，而他本人无法分得金币。
- 当 $n=202$ 时，202 号海盗必须获得除他自己以外的另外 100 名海盗的支持。因此，他须分给 1-200 号海盗中 100 名偶数号海盗各 1 枚金币，而他本人无法分得金币。
- 当 $n=203$ 时，203 号海盗必须获得除他自己以外的另外 101 名海盗的支持。但他只有 100 枚金币，无法获得 101 名海盗的支持。所以他的方案不可能通过。
- 当 $n=204$ 时，204 号海盗必须获得除他自己以外的另外 101 名海盗的支持。由上述可知，203 号海盗一定会支持 204 号海盗以保护自己。因此他只需在其他人中选 100 人，分给他们每人 1 个金币即可。
- 当 $n=205$ 时，205 号海盗必须获得除他自己、100 个金币能收买的 100 个海盗以外，另外 2 名海盗的支持。同理，当 $n=206$、$n=207$ 时，方案都无法被通过。

- 当 $n = 208$ 时，208 号海盗必须获得除他自己、100 个金币能收买的 100 个海盗以外，另外 3 名海盗的支持。205-207 号海盗为保命也会支持 208 号海盗，因此 208 号海盗的方案会被通过。

猜想 当 $n > 200$ 时，当且仅当 $n = 200 + 2^k, k \in \mathbb{N}$ 时，存在可以通过的方案。

证明

1. 由上述论述可知，当 $k = 0, 1, 2, 3$，$n = 201, 202, 204, 208$ 时，存在可以通过的方案，且本人均无法得到金币。

2. 设 $k \in \mathbb{N}^*$。

   - 若 $n = 200 + 2k$ 时，存在一个可以通过的方案，即 $100 + k$ 个海盗支持了方案。则当 $n = 201 + 2k$ 时，需要 $101 + k$ 个海盗支持。除他自己和能用金币收买的 100 个海盗（共 101 人）以外，剩余 k 个人均会选择拒绝方案从而将他扔进大海。
   - 若 $n = 200+2k$ 时, 不存在一个可以通过的方案, 即表示支持的海盗人数 $\delta < 100+k$。则当 $n = 201 + 2k$ 时，表示支持的海盗人数为 $\delta + 1 \leq 100 + k$，不足 $101 + k$ 个。方案仍然不会被通过。

   所以当 $n > 201$ 且 n 为奇数时，不存在能通过的方案。

3. 假设 $n = 200 + 2^k$ 时，存在可以通过的方案。若 $n = n' > 200 + 2^k$（由上述可知，n' 定为偶数）时，也存在可以通过的方案，且 $200 + 2^k < n < n'$ 时不存在可以通过的方案。则 $200 + 2^k$ 号至 $n' - 1$ 号共 $n' - 1 - (200 + 2^k)$ 名海盗都会支持 n' 号海盗。因此若有 n' 名海盗时存在可以通过的方案，定有

$$\left\lceil \frac{n'}{2} \right\rceil = 100 + 1 + (n' - 1 - (200 + 2^k)) = n' - 2^k - 100$$
$$n' = 2n' - 2^{k+1} - 200$$
$$n' = 200 + 2^{k+1}$$

归纳可得，上述猜想成立。 □

答

(a) 当 $n = 10$ 时，$c_{10}(j) = \begin{cases} 96 & , j = n \\ 1 & , j < 10, j\text{为偶数} \\ 0 & , j < 10, j\text{为奇数} \end{cases}$

(b) 当 $n = 20$ 时，$c_{20}(j) = \begin{cases} 91 & , j = n \\ 1 & , j < 20, j\text{为偶数} \\ 0 & , j < 20, j\text{为奇数} \end{cases}$

当 $n=50$ 时，$c_{50}(j)=\begin{cases}76 & ,j=n\\ 1 & ,j<50,j\text{为偶数}\\ 0 & ,j<50,j\text{为奇数}\end{cases}$

当 $n=100$ 时，$c_{100}(j)=\begin{cases}51 & ,j=n\\ 1 & ,j<100,j\text{为偶数}\\ 0 & ,j<100,j\text{为奇数}\end{cases}$

当 $n=200$ 时，$c_{200}(j)=\begin{cases}1 & ,j<200,j\text{为偶数}\\ 0 & ,j<200,j\text{为奇数}\end{cases}$

当 $n=400$ 时，不存在自然数 k 使 $200+2^k=400$，因此 400 号海盗一定会被扔进大海。

4. 以下利用数学归纳法证明“所有的马颜色相同”错在哪儿

    1. 只有一匹马时，命题成立
    2. 设有 n 匹马时命题成立。则当有 $n+1$ 匹马 $\{h_1,h_2,\ldots,h_n,h_{n+1}\}$ 时，
    由归纳假设，$\{h_1,h_2,\ldots,h_n\}$ 这 n 匹马颜色相同，$\{h_2,\ldots,h_n,h_{n+1}\}$ 这 n 匹马的颜色相同，
    即 h_1 和 h_{n+1} 这两匹马与 $\{h_2,\ldots,h_n\}$ 颜色相同，
    所以 $\{h_1,h_2,\ldots,h_n,h_{n+1}\}$ 这 $n+1$ 匹马的颜色是相同的

考察由 $n=1$ 推广至 $n=2$ 时的情况：

两匹马 $\{h_1,h_2\}$ 中，由归纳假设 $\{h_1\}$ 颜色相同，$\{h_2\}$ 颜色相同。但 $\{h_1\}\cap\{h_2\}=\Phi$，所以不能推出 $\{h_1,h_2\}$ 两匹马颜色相同。因此不能由此归纳“出所有的马颜色相同”。

3 0309

1. k 为正常数，证明 $\log n=o(n^k)$

证明 只须证 $\lim_{n\to+\infty}\frac{f(n)}{g(n)}=0$ □

2. 寻找单调递增函数 $f(n)$ 和 $g(n)$，使得 $f(n)=O(g(n))$ 和 $g(n)=O(f(n))$ 都不成立

即寻找单调递增函数 $f(n)$ 和 $g(n)$，使得 $\forall c>0, \forall N>0$

$$\exists n_1>N, f(n_1)>cg(n_1), \quad \exists n_2>N, g(n_2)>cf(n_2)$$

令 $k=\left\lfloor \frac{\lfloor \log_2 n \rfloor}{2} \right\rfloor \in \mathbb{N}$

$$f(n)=\begin{cases} 2^n & ,2^{2k}\le n<2^{2k+1} \\ 2^{2^{2k+1}} & ,2^{2k+1}\le n<2^{k+2} \end{cases}$$

$$g(n)=\begin{cases} 2^{2^{2k}} & ,2^{2k}\le n<2^{k+1} \\ 2^n & ,2^{2k+1}\le n<2^{2k+2} \end{cases}$$

则，当 $2^{2k}\le n<2^{2k+1}$ 时，

$$\frac{f(n)}{g(n)}=\frac{2^n}{2^{2^{2k}}}=2^{(n-2^{2k})} \tag{1}$$

当 $2^{2k+1}\le n<2^{2k+2}$ 时，

$$\frac{g(n)}{f(n)}=\frac{2^n}{2^{2^{2k+1}}}=2^{(n-2^{2k+1})} \tag{2}$$

$\forall c>0, N>0$，一定存在一个足够大的 $m>0$，使得 $c<2^{2^{2m}}$ 且 $N<2^{2m}$

- 由 (1) 式可知，当 $n=2^{2m+3}-1>N$ 时，$2^{2m+2}<n<2^{2m+3}$，

$$\begin{aligned} f(n)&=2^{(2^{2^{2m+3}}-1-2^{2^{m+2}})}g(n)=2^{(2^{2^{m+2}}-1)}g(n) \\ &>2^{2^{2^m}}g(n)>cg(n) \end{aligned}$$

- 由 (2) 式可知，当 $n=2^{2m+4}-1>N$ 时，$2^{2m+3}<n<2^{2m+4}$，

$$\begin{aligned} g(n)&=2^{(2^{2^{2m+4}}-1-2^{2^{m+3}})}f(n)=2^{(2^{2^{m+3}}-1)}f(n) \\ &>2^{2^{2^m}}f(n)>cf(n) \end{aligned}$$

综合可知，对于上述 $f(n), g(n)$，$f(n)=O(g(n))$ 和 $g(n)=O(f(n))$ 都不成立

3. 假设解决同一个问题的两个算法 A1 和 A2 的时间复杂性分别为 $O(n^3)$ 和 $O(n)$，如果为这两个算法分别编写程序并在同样的环境下运行，算法 A2 的程序一定比算法 A1 的程序运行得快吗？为什么？

不一定。程序的运行速度除了取决于时间复杂度，还取决于输入的规模以及程序的质量。在时间复杂度中，除了主项，还包括被忽略的常数以及其他项可能影响实际的运行时间。

如算法 A1 的程序可能运行的时间为 n^3，而算法 A2 的程序可能运行时间为 $256n$。此时，若输

入规模 $n < 16$，则算法 A1 比算法 A2 运行得更快。

4. 考虑以下冒泡排序算法。

算法 3 BubbleSort

输入: n 个元素的数组 $A[1 \dots n]$

输出: 按非降序排列的数组 $A[1 \dots n]$

```
1: i ← 1, sorted ← false
2: while i ≤ n − 1 ∧ sorted ≠ true do
3:     sorted ← true
4:     for j ← n downto i + 1 do
5:         if A[j] < A[j-1] then
6:             SWAP(A[j], A[j − 1])
7:             sorted ← false
8:         end if
9:     end for
10:    i ← i + 1
11: end while
```

(a) 元素比较的最少次数是多少？何时达到最小值?

当数组本身为非降序时可达到最小值。此时外层循环只执行 1 次，内层循环只执行 $n-1$ 次。所以元素比较次数最少为 $n-1$ 次。

(b) 元素比较的最多次数是多少？何时达到最大值?

当数组本身为严格升序时可达到最大值。此时外层循环执行 $n-1$ 次，内层循环共执行次数为

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

所以元素比较的最多次数为 $\frac{n(n-1)}{2}$ 次。

(c) 可以用 O,Ω,Θ 表示算法的运行时间吗?

- 最好情况:

$$n-1 = O(n) = \Omega(n) = \Theta(n)$$

- 最坏情况:

$$\frac{1}{2}(n^2-n)=O(n^2)=\Omega(n^2)=\Theta(n^2)$$

4 0312

1. 证明

$$\sum_{k=1}^{n}\frac{1}{k}=\Theta(\log n)$$

证明 由 $\frac{1}{k}$ 单调递减

$$\int_1^{n+1}\frac{1}{x}\mathrm{d}x\le\sum_{k=1}^{n}\frac{1}{k}\le 1+\int_1^{n}\frac{1}{x}\mathrm{d}x$$

$$\ln(n+1)-\ln 1\le\sum_{k=1}^{n}\frac{1}{k}\le 1+\ln n-\ln 1$$

$$\ln(n)<\ln(n+1)\le\sum_{k=1}^{n}\frac{1}{k}\le 1+\ln n$$

所以有

$$\Omega(\log n)=\sum_{k=1}^{n}\frac{1}{k}=O(\log n)$$

即

$$\sum_{k=1}^{n}\frac{1}{k}=\Theta(\log n)$$

□

2. 设有如下递推关系

$$T(n)=\begin{cases}T(\frac{n}{2})+1 & ,n\text{为偶数}\\ 2T(\frac{n-1}{2}) & ,n\text{为奇数}\end{cases}$$

其中 $T(1)=1$

(a) 证明当 $n=2^k$ 时，$T(n)=O(\log n)$

证明

$$\begin{aligned} T(n) = T(2^k) = T(\frac{n}{2}) + 1 = T(2^{k-1}) + 1 \\ = T(2^{k-2}) + 2 = \dots \\ = T(1) + k - 1 = k = \log n \end{aligned}$$

所以

$$T(n) = O(\log n)$$

□

(b) 证明存在无穷集合 X，当 $n \in X$ 时，$T(n) = \Omega(n)$

令 $a_1 = 1$, $a_n = 2a_{n-1} + 1 \Rightarrow a_n = 2^n - 1$，则无穷集合 $X = \left\{2^k - 1 \mid k \in \mathbb{N}^*\right\}$。

证明

$$\begin{aligned} T(n) = T(2^k - 1) = 2 \cdot T(\frac{n-1}{2}) + 1 = 2 \cdot T(2^{k-1} - 1) \\ = 4 \cdot T(2^{k-2} - 1) = \dots \\ = 2^{k-1} \cdot T(2 - 1) = 2^{k-1} \end{aligned}$$

$\exists c = \frac{1}{2}, N = 1$ 使得 $\forall n > N, n \in X$

$$T(n) = T(2^k - 1) = 2^{k-1} > 2^{k-1} - \frac{1}{2} = \frac{2^k - 1}{2} = \frac{n}{2}$$

所以

$$T(n) = \Omega(n)$$

□

(c) 以上两个结论说明了什么?

一个算法在输入具有不同特征时，可能具有不同的时间复杂度。
最佳状况和最差状况可能有不同的时间复杂度。

5 0316

1. 求解递推关系

$$T(n) = n + \sum_{i=1}^{n-1} T(i)$$

其中 $T(1)=1$

$$T(n+1)=(n+1)+\sum_{i=1}^{n}T(i)$$

$$T(n+1)-T(n)=\left[(n+1)+\sum_{i=1}^{n}T(i)\right]-\left[n+\sum_{i=1}^{n-1}T(i)\right]=1+T(n)$$

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

所以当 $n>2$ 时,

$$T(n)+1=T(1)\times 2^n\Rightarrow T(n)=2^n-1$$

又当 $n=1$ 时，$T(1)=1$ 恰好符合上式，所以对任意的 $n\in\mathbb{N}^*$,

$$T(n)=2^n-1$$

2. 在寻找一对一映射问题中，算法结束时集合 S 是否可能为空集？请证明你的结论。

给定一个集合 A 和一个从 A 到自身的映射 f，寻找一个元素个数最多的子集 $S\subseteq A$，满足 f 在 S 上是一一映射

不可能为空集。

证明 设集合 A 中有 n 个元素。

从集合中任取一个元素记作 x_1，记 $i\geq 2$ 时 $x_i=f(x_{i-1})$。

由于对任意的 $x\in A$，都有且仅有一个 $x'\in A, x'=f(x)$，所以序列 $\{x_i\}$ 的长度是无限的。但是集合 A 中元素个数是有限的，所以必然存在至少一对 $m<n$ 使得 $x_m=x_n$。

取一对使 $x_m,\ldots,x_{n-1}$ 各不相同的 m,n

- 若存在 $k:m<k<n, x_m=x_k$，则令 $n=k$
- 若存在 $p,q:m<p<q<n, x_p=x_q$，则令 $m=p, n=q$

由 $\{x_i\}$ 的定义可知，$x_m,\ldots,x_{n-1}$ 构成了一个循环置换

$$\begin{pmatrix} x_m & \cdots & x_{n-1}\end{pmatrix}=\begin{pmatrix} x_m & x_{m+1} & \cdots & x_{n-2}\\ x_{m+1} & \cdots & \cdots & x_{n-1}\end{pmatrix}$$

由置换的定义可知，f 在 $\{x_m,\ldots,x_{n-1}\}$ 上是一一映射，即 $\{x_m,\ldots,x_{n-1}\}\subseteq S$，所以 $S\neq\Phi$。

□

6 0319

1. 有 A、B、C 三个桩子，A 上放有 n 个不同大小的圆盘，按大小递减顺序自下而上。

   (a) 设计算法，用最少的移动步数将 A 上的圆盘都移到 C，每次只能移一个，且任一桩子上的圆盘都必须满足圆盘大小自下而上递减的顺序。

   (b) 给定数组 `p`，其元素取值只有 1，2，3 三种，代表了 n 个圆盘目前的位置，1 代表 A，2 代表 B，3 代表 C，`p[i]` 的值为第 i 个圆盘的位置。例如 `p=[3,3,2,1]`[1]表示共有 4 个圆盘，第一个和第二个都在 C，第三个在 B，第 4 个在 A。如果 `p` 代表了 (a) 中的最优移动过程中某一步的状态，则求出这是第几步。注意诸如 `p=[2,2]` 是不可能出现的，此时算法得到 -1。

(a) 解决汉诺塔问题首要的是注意到当只有两个圆盘存在时问题的解决是十分简单的。因此当解决有多个圆盘的汉诺塔问题时，可以将其中一部分圆盘看作是一个圆盘，另一部分圆盘看作是另一个圆盘，然后就可以应用两个圆盘的解法解决该问题，此时移动的每一步都是一个规模更小的汉诺塔问题。最后要注意到分组时必须将最大的圆盘单独放在一组，其他的圆盘单独放在一组，方可保证在移动的过程中任一桩子上的圆盘都满足大小自下而上递减。

算法 4 汉诺塔

输入: 三个栈 a, b, c，其中 a 中从栈顶到栈底为 0 到 $n-1$ 共 n 个元素，b, c 为空栈

```
1: procedure MAIN
2:     HANOI(a, b, c, n)
3: end procedure

4: procedure HANOI(src, via, dst, n)
                          ▷ 解决以 via 为中介，从 src 的顶端移 n 个圆盘到 dst 上的汉诺塔问题
5:     if n = 1 then
6:         dst.push(src.pop())                ▷ 从 src 上取出最顶端的圆盘，放到 dst 上
7:     else
8:         HANOI(src, dst, via, n − 1)
9:         HANOI(src, via, dst, 1)
10:        HANOI(via, src, dst, n − 1)
11:    end if
12: end procedure
```

[1] `p[0]` 是最小的圆盘

(b) 由上述算法描述可知，解决 n 个圆盘的汉诺塔问题需要移动圆盘的次数 $T(n)$ 符合表达式

$$T(n) = \begin{cases} T(n-1) + T(1) + T(n-1) & , n \geq 2 \\ 1 & , n = 1 \end{cases}$$

可易解得 $T(n) = 2^n - 1$。

分析该算法可知，在解决规模为 n 的问题 $H(n)$ 时，首先要将 $n-1$ 个圆盘从 src 经过 $2^{n-1}-1$ 次操作移至 via 上；然后经 1 次操作将第 n 个圆盘从 src 移至 dst 上；最后经 $2^{n-1}-1$ 次操作将 $n-1$ 个圆盘从 via 经过 $2^{n-1}-1$ 次操作移至 dst 上。因此在这个子问题中，第 n 个圆盘（即该问题中最大的圆盘）只有在 src 和在 dst 上两种状态，不可能存在某一步操作使其出现在 via 上。

因此给出一个解决问题 $H(n)$ 的状态时，考察最大的圆盘的位置：

- 如果该圆盘在问题 $H(n)$ 的 src 上则记 b_n 为 0，此时正在解决的子问题 $H(n-1)$ 是要将 $n-1$ 个节点从 src 移至 via 上
- 如果该圆盘在问题 $H(n)$ 的 dst 上则记 b_n 为 1，此时正在解决的子问题 $H(n-1)$ 是要将 $n-1$ 个节点从 via 移至 dst 上

迭代地进行上述过程，则可得到序列 $\{b_i\}_{i\in[1,n]}$。当 b_i 从 0 变为 1 时，恰好是第 i 个盘子从其 src 移到 dst 上的结果，此时问题 $H(i)$ 恰好执行了 2^i 步。因此 $N = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$ 可表示到此状态时已经经过的步数。

算法 5 检查汉诺塔状态

输入: 数组 $p[0..n-1]$ 表示用上述算法解决 n 个圆盘的汉诺塔问题的一个中间状态

输出: 到达这一状态经过的步数 N，如果该状态非法则为 -1

```
1:  src ← 1, via ← 2, dst ← 3
2:  N ← 0
3:  for i ← n − 1 downto 0 do
4:      if p[i] = src then
5:          N ← 2N + 0
6:          Swap(dst, via)
7:      else if p[i] = dst then
8:          N ← 2N + 1
9:          Swap(src, via)
10:     else
11:         return −1
12:     end if
13: end for
14: return N
```

2. 课上的最大连续子序列是指和最大，如果要求是积最大呢?

给定实数序列 $x_1, x_2, \ldots, x_n$，寻找连续子序列 $x_i, x_{i+1}, \ldots, x_j$ 使得其数值之积在所有连续子序列数值之积中为最大。

设空序列的积为 1，只需求出最大值即可，不需要知道是哪个序列。

考虑乘积，在一个连续子序列后面加上一个元素时，既要考虑乘积绝对值的变化，也要考虑符号的变化。若符号在某一次运算后为负数，且后面还有一个负的元素，则后缀的乘积仍有机会变正并大于现有的最大值。考虑到这一因素，除了 $MaxSuffix$ 以外，还需要一个 $MinSuffix$ 以记录出现负元素时后缀乘积的情况，并按如下规则更新：

- $MaxSuffix = \max\{MaxSuffix(i) \cdot x_{i+1}, MinSuffix(i) \cdot x_{i+1}, 1\}$
- 当 $MinSuffix(i) < 0$ 时，$MinSuffix(i+1) = MinSuffix(i) \cdot x_{i+1}$
- 当 $MinSuffix > 0$ 时，$MinSuffix$ 应当总等于 $MaxSuffix$

算法 6 最大乘积连续子序列

输入: 序列 $\{x_i\}_{1 \le i \le n}$

输出: 乘积最大的连续子序列的乘积 $MaxProd$

$MaxProd \leftarrow 1$ ▷ 最大子序列的乘积
$MaxSuffix \leftarrow 1, MinSuffix \leftarrow 1$ ▷ 最大正后缀乘积、最小负后缀乘积
for $i \leftarrow 1$ to n **do**
    $MaxSuffix \leftarrow MaxSuffix \cdot x_i$
    $MinSuffix \leftarrow MinSuffix \cdot x_i$
    if $MaxSuffix < 1$ **then**
        $MaxSuffix \leftarrow 1$
    end if
    if $MaxSuffix < MinSuffix$ **then**
        $MaxSuffix \leftarrow MinSuffix$
    end if
    if $MinSuffix \ge 0$ **then**
        $MinSuffix \leftarrow MaxSuffix$
    end if
    if $MaxSuffix > MaxProd$ **then**
        $MaxProd \leftarrow MaxSuffix$
    end if
end for

3. 背包问题中如果各种大小的物品的数量不限，那么如何知道背包是否能够恰好放满

给定一个整数 K 和 n 种不同大小的物品，第 i 中物品的大小为整数 s_i，每种物品的数量不限。寻找一个物品的组合，它们的大小之和正好为 K，或者确定不存在这样的组合。

记 $s_0 = 0$，$P(K, S = \{s_1, s_2, \ldots, s_n, s_0\})$ 表示从 S 中选取物品是否能恰好装满一个大小为 K 的背包的问题。

则有

$$P(K, \{s_1, s_2, \ldots, s_n, s_0\}) = \bigvee_{\substack{K - ms_n \geq 0, \\ m \geq 0}} P(K - ms_n, \{s_1, \ldots, s_{n-1}, s_0\})$$

且

$$\forall S \supset \{s_0\}, P(0, S) = \mathsf{true} \qquad \forall K > 0, P(K, \{s_0\}) = \mathsf{false}$$

算法 7 物品数量不限的背包问题

输入: $K, S[1..n]$

输出: 解是否存在

```
1:  P[0..n][0..K] ← {false}
2:  P[0][0] ← true
3:  for k = 0 to K do
4:      for i = 1 to n do
5:          if P[i − 1][k] = true then
6:              P[i][k] ← true
7:          else
8:              r ← [k mod s[i]]
9:              while r < k do
10:                 if P[i][r] = true then
11:                     P[i][k] ← true
12:                     break
13:                 end if
14:                 r ← r + k
15:             end while
16:         end if
17:     end for
18: end for
```

7 0323

1. 设计算法将 1 到 n^2 按顺时针方向由内而外填入一个 $n \times n$ 的矩阵

记该矩阵为 $A[1..n][1..n]$。

算法 8 螺旋方阵

输入: 空间 $A[1..n][1..n]$

```
1: SpiralMatrix(A, 0, 0, n)

2: procedure SpiralMatrix(M, top, left, size)
                                  ▷ 向以 M[top][left] 为 A[0][0] 的 size 维方阵中填入元素
3:     if n = 1 then
4:         M[top][left] ← 1
5:     else if [n mod 2] = 1 then ▷ 去掉第一列和最后一行后，剩余的元素为 n − 1 维的方阵
6:         SpiralMatrix(A, top, left + 1, n − 1)        ▷ 以左上角右侧的元素为基准
7:         for i ← 1 to n do
8:             M[n][i] ← (n − 1)^2 + n − (i − 1)                          ▷ 填充下边
9:             M[i][1] ← n^2 − (i − 1)                                    ▷ 填充左边
10:        end for
11:    else                       ▷ 去掉第一行和最后一行后，剩余的元素为 n − 1 维的方阵
12:        SpiralMatrix(A, top + 1, left, n − 1)        ▷ 以左上角下侧的元素为基准
13:        for i ← 1 to n do
14:            M[1][i] ← (n − 1)^2 + i                                    ▷ 填充上边
15:            M[i][n] ← (n − 1)^2 + n + (i − 1)                          ▷ 填充右边
16:        end for
17:    end if
18: end procedure
```

2. 给定整数数组 $A[1..n]$，相邻两个元素的值最多相差 1。设 $A[1] = x$，$A[n] = y$，并且 $x < y$，输入 z，$x \le z \le y$，判断 z 在数组 A 中出现的位置。给出算法及时间复杂性（不得穷举）

相邻两个元素的值最多相差 1，即有 $A[i+1] - A[i] \in \{-1, 0, 1\}$。可以据此类比连续函数上的零点存在性定理，即

若 $A[i] \le z \le A[j]$ 且 $i < j$，则一定存在一个 $m : i \le m \le j$ 使得 $A[m] = z$

因此可用二分法，有如下算法：

算法 9

输入: 整数数组 $A[1..n]$, A 中的一个整数 z

输出: 元素 z 在数组 A 中的位置

```
1: lidx ← 1, ridx ← n
2: mid ← ⌊(lidx+ridx)/2⌋
3: while A[mid] ≠ z do
4:     if A[mid] < z then
5:         left ← mid                          ▷ 更新后仍有 A[left] < z
6:     else                                    ▷ A[mid] > z
7:         right ← mid                         ▷ 更新后仍有 A[right] > z
8:     end if
9:     mid ← ⌊(lidx+ridx)/2⌋
10: end while
11: return mid
```

该算法的时间复杂度为 $O(\log n)$

3. 假设有 k 个长度为 n 的有序序列，采用下述方法将这些序列合并成一个具有 kn 个元素的有序序列：将前两个序列合并，再并入第三个序列，然后是第四个……直至所有序列合并。

   (a) 用 k 和 n 表示该方法的时间复杂性

当合并一个长度为 m 和一个长度为 n 的序列时

- 赋值操作执行了 $m+n$ 次
- 比较操作至少执行 $\min\{m,n\}$ 次，最多执行 $2\min\{m,n\}$ 次

所以此算法合并 k 个长度为 n 的有序序列所需时间为

$$\begin{aligned} T(0,k,n) = T(n,k-1,n) &= 2n + cn + T(2n,k-2,n) \\ &= (2+3)n + 2cn + T(3n,k-3,n) \\ &= \dots \\ &= \sum_{i=2}^{k} i\cdot n + (k-1)cn \\ &= \frac{(k+2)(k-1)}{2}n + (k-1)cn = O(k^2 n) \end{aligned}$$

(b) 能不能得到一个效率更高的方法来合并这 k 个序列?

首先将第奇数个序列和第偶数个序列合并，形成 $\lceil \frac{k}{2} \rceil$ 个最大长度为 $2n$ 的序列；然后重复上一步骤，直至最终只剩 1 个序列。

不妨设 $k = 2^m$

$$\begin{aligned} T(k,n) &= \frac{k}{2}T(2,n) + T\left(\frac{k}{2}, 2n\right) = 2^{m-1}(2+c)n + T\left(2^{m-1}, 2n\right) \\ &= 2^{m-1}(2+c)n + 2^{m-2}2(2+c)n + T\left(2^{m-2}, 4n\right) \\ &= 2 \cdot 2^{m-1}(2+c)n + T\left(2^{m-2}, 2^2 n\right) \\ &= \dots \\ &= (m-1) \cdot 2^{m-1}(2+c)n + T\left(2, 2^{m-1}n\right) \\ &= m \cdot 2^{m-1}(2+c)n = O(kn \log k) \end{aligned}$$

4. 设 $A[1..n]$ 是一个包含 n 个不同数的数组。如果在 $i < j$ 的情况下有 $A[i] > A[j]$，则 (i, j) 为 A 中的一个逆序对。

(a) 若 $A = \{2, 3, 8, 6, 1\}$，列出其中所有的逆序对

$$(2,1) \quad (3,1) \quad (8,6) \quad (8,1) \quad (6,1)$$

(b) 若数组元素取自 $\{1, 2, \dots, n\}$，那么怎样的数组含有最多的逆序对？它包含多少个逆序对?

当数组为递减序列时包含有最多的逆序对，此时序列中任取一对数，都能构成逆序对。

逆序对数为 $\frac{n(n-1)}{2}$。

(c) 求任意数组 A 中逆序对的数目，并证明算法的时间复杂性为 $\Theta(n \log n)$

应用分治法，结合归并排序。首先将数组等分成左右两个子序列，分别求出两边的逆序对数（并同时进行排序)。然后再求出分属于两个子序列的逆序对，即对每一个左序列中的元素，找出右序列中大于该元素的所有元素个数。

$$T(n) = 2T(n/2) + (2+c)n = O(n \log n)$$

8 0326

1. 证明二分查找是有序序列查找的最优算法

算法 10 求逆序对数

输入: 数组 $A[1..n]$

输出: A 中的逆序对数 m

```
(m, _) ← MergeInversions(A[1..n])

procedure MergeInversions(A[1..n])
                ▷ 输入一个数组 A，输出一个二元组，包含该数组中的逆序对数及一个有序的该数组
    if A.size = 1 then
        return (0, A)
    else if A.size = 2 then
        if A[1] > A[2] then
            return (1, [A[2], A[1]])
        else
            return (0, A)
        end if
    else                                                          ▷ 分治
        (nL, L) ← MergeInversions(A[1.. ⌊n/2⌋])
        (nR, R) ← MergeInversions(A[⌊n/2⌋ + 1..n])
        k ← 1, cnt ← 0                                            ▷ 归并
        while ¬L.empty ∧ ¬R.empty do
            if L.front ≤ R.front then
                A[k] = L.PopFront()
            else                                          ▷ L[1] 大于 R 的所有剩余元素
                A[k] = R.PopFront()
                cnt ← cnt + R.size                ▷ L[1] 与 R 中所有剩余元素都构成逆序对
            end if
            k ← k + 1
        end while
        if ¬L.empty then                                          ▷ 填充剩余元素
            A[k..n] = L
        else if ¬R.empty then
            A[k..n] = R
        end if
        return (cnt, A)
    end if
end procedure
```

在一个长度为 n 的有序数组中查找一个元素，该元素的位置为 i 的概率为 $p_i = \frac{1}{n}$。因此，**在没有附加信息的前提下**，事件“确定一个元素的位置”能提供的信息至少为 $I_e = -\log p_i = \log n$。而一次比较只能给出真、假两种结果，即一次比较能提供的信息为 $I_c = \log 2$。所以至少需要经过

$$\frac{I_e}{I_c} = \frac{\log n}{\log 2} = \log_2 n$$

次比较方能确定长度为 n 的数组中一个元素的位置。即该问题的信息论下界为 $\Omega(\log n)$。

二分查找法的效率恰好为 $\Omega(\log n)$，因此二分查找法是最优的基于比较的有序序列查找算法。

但如果有附加信息存在（例如给定数组的分布情况）使该元素的位置为 i 的概率 $p_i' > \frac{1}{n}$，则其信息量 $I_e' < \log n$。此时利用这些附加信息可能不需要 $\log_2 n$ 次比较即可找到它的位置。

2. 写出两种建堆方法的伪代码

算法 11 自顶向下建堆

输入: $A[1..n]$

输出: 重排 A，使其每个非叶子节点的值不小于其子节点的值

```
1: for i ← 2 to n do
2:     j ← i
3:     while j > 1 ∧ A[j] > A[⌊j/2⌋] do
4:         SWAP(A[j], A[⌊j/2⌋])
5:         j ← ⌊j/2⌋
6:     end while
7: end for
```

算法 12 自底向上建堆

输入: $A[1..n]$

输出: 重排 A，使其每个非叶子节点的值不小于其子节点的值

```
1: for i ← ⌊n/2⌋ to 1 do
2:     j ← 2i
3:     while j ≤ n do
4:         if j + 1 ≤ n ∧ A[j + 1] > A[j] then
5:             j ← j + 1
6:         end if
7:         if A[j] > A[⌊j/2⌋] then
8:             Swap(A[j], A[⌊j/2⌋])
9:         else
10:            break
11:        end if
12:        j ← 2j
13:    end while
14: end for
```

3. 对玻璃瓶做强度测试。设地面高度为 0，从 0 往上有刻度 1 到 n，相邻两个刻度间距离都相等。如果一个玻璃瓶从刻度 i 落到地面没有破碎，而在高度 $i+1$ 处落到地面时碎了，则此类玻璃瓶的强度为 i。

   (a) 若只有一个样品供测试，如何得到该类玻璃瓶的强度

   > 从刻度 1 开始逐个高度进行测试，找到使其摔碎的最低高度。

   (b) 如果样品数量充足，如何用尽量少的测试次数得到强度

   > 使用二分法。首先测试高度 $\frac{n}{2}$，若摔碎了则测试 $\frac{n}{4}$，若没摔碎则测试 $\frac{3}{4}n$，依此类推。

   (c) 如果有两个样品，如何用尽量少的测试次数得到强度

   > 第一个瓶子测试高度 $k\sqrt{n}$（k 从 1 到 $\sqrt{n}$），可以确定一个高度为 $\sqrt{n}$ 的区间。
   > 第二个瓶子在上述区间内从最低高度开始逐渐提升高度，最多测试 $\sqrt{n}$ 次。

9 0330

1. 有 n 个数存放在两个有序数组中，如何找到这 n 个数的第 k 小的数

比较两个数组的头部，将最小的一个提取出来。重复该步骤 k 次，第 k 个数即为第 k 小的数。该算法的时间复杂度为 $O(k)$

算法 13 归并取第 k 小的数

输入: 数组 $A[1\dots p], B[1\dots q]$；目标 k ▷ $p+q=n\geq k$

输出: A, B 中第 k 小的数 m

```
cnt ← 0, i ← 1, j ← 1
while cnt < k ∧ i ≤ p ∧ j ≤ q do
    if A[i] < B[j] then
        m ← A[i], i ← i + 1
    else
        m ← B[j], j ← j + 1
    end if
    cnt ← cnt + 1
end while
while cnt < k ∧ i < p do
    m ← A[i], i ← i + 1
    cnt ← cnt + 1
end while
while cnt < k ∧ j < q do
    m ← B[j], j ← j + 1
    cnt ← cnt + 1
end while
```

2. 如何修改 KMP 算法，使之能够获得字符串 B 在字符串 A 中出现的次数

给定原有算法，已经能找到一个 s，使得 $\forall 1\leq i\leq m, A[s+i-1]=B[i]$。

如果存在一个 $s'=s+\delta, 1\leq\delta<m$，使得 $\forall i\in[1,m], A[s'+i-1]=B[i]$ （即 A 中有两个重叠的 B，重叠长度为 $m-\delta$），那么必然有即 $\forall 1\leq i\leq m-\delta, B[i]=B[i+\delta]$ （即 B 长度为 $m-\delta$ 的前后缀是相等的）。

伪代码见算法14

3. 设计高效算法求序列 T 和 P 的最长公共子序列和最短公共超序列的长度。

算法 14 计数 KMP

输入: $Text[1 \ldots n]$（被查找字符串）, $Pattern[1 \ldots m]$（查找目标）

输出: cnt（$Pattern$ 在 $Text$ 中出现的次数）

```
1: procedure StringMatch(Text, n, Pattern, m)
2:     next ← ComputeNext(Pattern, m)
3:     i ← 1, j ← 1
4:     cnt ← 0
5:     while i ≤ n do
6:         if Pattern[j] = Text[i] then
7:             j ← j + 1, i ← i + 1
8:             if j = m + 1 then
9:                 j ← next[j] + 1
10:                cnt ← cnt + 1
11:            end if
12:        else
13:            j ← next[j] + 1
14:            if j = 0 then
15:                j ← 1, i ← i + 1
16:            end if
17:        end if
18:    end while
19:    return cnt
20: end procedure

21: procedure ComputeNext(Pattern, m)
22:    next[1 ... m + 1]
23:    next[1] ← −1, next[2] ← 0
24:    for i ← 3 to m + 1 do
25:        j ← next[i − 1] + 1
26:        while j > 0 ∧ Pattern[i − 1] ≠ Pattern[j] do
27:            j ← next[j] + 1
28:        end while
29:        next[j] ← j
30:    end for
31:    return next
32: end procedure
```

(a) 最长公共子序列（LCS）L 定义为 T 和 P 的共同子序列中最长的一个

记 $T[:i]$ 为序列 T 的前 i 个元素组成的序列。记 $l_{LCS}(T,P)$ 为 T 和 P 的最长公共子序列的长度。不妨设序列 T 和 P 的长度分别为 m,n。

则有归纳关系

$$l_{LCS}(T[:i],P[:j]) = \begin{cases} l_{LCS}(T[:i-1],P[:j-1])+1 & ,T[i]=P[j] \\ \max\{l_{LCS}(T[:i],P[:j-1]),l_{LCS}(T[:i-1],P[:j])\} & ,T[i]\neq P[j] \end{cases}$$

可用动态规划的方法解决此问题。该算法的时间复杂度为 $O(|T\times P|)$，空间复杂度为 $O(\min\{|T|,|P|\})$。

算法 15 最长公共子序列的长度

输入: $P[1\ldots m],T[1\ldots n]$（两个序列） ▷ 不妨设 $m\geq n$

输出: l（T,P 的最长公共子序列的长度）

```
1:  dp[0...1][0...n] ← {0}
2:  for i ← 1 to m do
3:      for j ← 1 to n do
4:          if T[i] = P[j] then
5:              dp[i mod 2][j] ← dp[(i − 1) mod 2][j − 1] + 1
6:          else if dp[i mod 2][j − 1] ≥ dp[(i − 1) mod 2][j] then
7:              dp[i mod 2][j] ← dp[i mod 2][j − 1]
8:          else
9:              dp[i mod 2][j] ← dp[(i − 1) mod 2][j]
10:         end if
11:     end for
12: end for
13: return dp[m mod 2][n]
```

(b) 最短公共超序列（SCS）S 定义为所有以 T 和 P 为子序列的序列中最短的一个

$T+P$ 一定是 T 和 P 的一个公共超序列，所以 $|S|\leq|T|+|P|$。

当 T,P 存在一个公共子序列 Q 时，只需从 P 中移除一个 Q，然后将 P 中其他元素按照与 T 中属于 Q 的元素相对顺序不变的顺序插入 T 中，使 P 成为该新序列的子序列。此时该新序列为 T 和 P 的公共超序列，其长度为 $|T|+|P|-|Q|$。

所以

$$|S| = \min(|T|+|P|-|Q|) = |T|+|P|-\max(|Q|) = |T|+|P|-|L|$$

求 $|L|$ 的算法见算法15。

4. 给定规模为 n 的整数数组，如何知道其中是否有**两个数之和**恰好等于 x。给出算法及相应的时间复杂性。

记该数组为 A

解 1 建立一个可以容纳 n 个整数的哈希表 H。扫描整个数组，到第 i 个数时，计算 $r := x - A[i]$。若 r 不在哈希表中，记 $H[r] = i$；否则有 $A[i] + A[H[r]] = x$。哈希表查找和插入的时间复杂度为 $O(1)$，所以遍历整个数组的时间复杂度为 $O(n)$。

算法 16 求补 1

输入: $A[1 \dots n]$（被查找的数组）
输出: 是否存在两个数之和恰好等于 x

```
1:  H ← Hashmap(n)
2:  for i ← 1 to n do
3:      r ← x − A[i]
4:      if H.contains(r) then
5:          return true                          ▷ 此时 A[i] + A[H[r]] = x
6:      else
7:          H[r] ← i
8:      end if
9:  end for
10: return false
```

解 2 将数组中的所有数分成两个堆，超过 $x/2$ 的所有数插入最大化堆 S，不足 $x/2$ 的所有数插入最小化堆 I。去两个堆最顶端的数 s, i 的和并与 x 比较:

- 若 $s + i = x$，则恰好符合条件
- 若 $s + i > x$，则移除 S 最顶端的元素并重新调整堆。因为 I 中所有的元素都比 i 大，不可能有一个 i' 使得 $i' + s \le i$
- 若 $s + i < x$，则移除 I 最顶端的元素并重新调整堆。因为 S 中所有的元素都比 s 大，不可能有一个 s' 使得 $i + s' \ge i$

建堆的时间复杂度为 $O(n \log n)$，出堆的时间复杂度最大为

$$a \log a + (n-a) \log(n-a) \le a \log n + (n-a) \log n = n \log n$$

所以该算法的时间复杂度为 $O(n \log n)$。

算法 17 求补 2

输入: $A[1 \dots n]$（被查找的数组）

输出: 是否存在两个数之和恰好等于 x

```
1: S ← MaximizeHeap(), I ← MinimizeHeap(), halfcnt ← 0
2: for i ← 1 to n do                                              ▷ 建堆
3:     if 2A[i] > x then
4:         S.Push(A[i])
5:     else if 2A[i] < x then
6:         I.Push(A[i])
7:     else                                                       ▷ 2A[i] = x
8:         halfcnt ← halfcnt + 1
9:     end if
10: end for
11: if halfcnt ≥ 2 then                                           ▷ 数组中恰好存在 2 个或更多 x/2
12:     return true
13: end if
14: while ¬S.empty ∧ ¬I.empty do                                  ▷ 出堆
15:     s ← S.top, i ← I.top                                      ▷ 取堆顶的元素
16:     if s + i > x then
17:         S.Pop()
18:     else if s + i < x then
19:         I.Pop()
20:     else                                                      ▷ s + i = x
21:         return true
22:     end if
23: end while
24: return false
```

5. 每个螺母需要一个螺栓配套使用，现有 n 个不同尺寸的螺母和相应的 n 个螺栓，如何快速地为为每一个螺母找到对应的螺栓？只能将一个螺母与一个螺栓进行匹配尝试，从而知道相互之间的大小关系，不能够比较两个螺母的大小，也不能比较两个螺栓的大小。给出算法及相应的时间复杂性。

（类似于快排）

分组方法：随机取一个螺栓（记其大小为 x_0），与每个螺母都进行一次比较。在找到配对的螺

母的同时，把比这个螺栓小的螺母和比它大的螺母分成两部分 Y_0, Y_1。有

$$\forall y_i \in Y_0, y_j \in Y_1, y_i < x_0 < y_j$$

然后从 Y_0, Y_1 两堆螺母里各选一个螺母（y_0, y_1），和螺栓匹配，同时将螺栓按小、中、大分成三堆 X_0, X_1, X_2。且有

$$\forall x_i \in X_0, x_i < y_0 \qquad \forall x_j \in X_2, x_j > y_0$$

此时从 X_0 中任选一个螺栓，其配对的螺母必然在 Y_0 中，候选螺母的数量大约减半。

归纳假设：因为螺栓与螺母是一一配对的，即

$$\forall x_i \in X, \exists y_i \in Y : x_i = y_j$$

所以则若有 $\forall x_i \in \hat{X}, x_j \in \complement_{\hat{X}}^{\hat{X}} : x_i > x_j$，$\forall y_i \in \hat{Y}, y_j \in \complement_{\hat{Y}}^{\hat{Y}} : y_i > y_j$ 则必有

$$\left|\hat{X}\right| \leq \left|\hat{Y}\right| \Rightarrow \hat{X} \subseteq \hat{Y}$$

即与 $\hat{X}$ 中的螺栓匹配的螺母全部都在 $\hat{Y}$ 中。所以一定可以用 $\hat{X}$ 中的一个螺栓，将 $\hat{Y}$ 分为两组（除非选中的螺栓匹配的螺母是 $\hat{Y}$ 中最小或最大的）。

算法描述：见算法18

复杂性分析：若只考虑拆分一个集合，不考虑拆分另一个集合时从本集合中抽取的元素。

将一个规模为 n 的集合分为两部分需要进行 n 次比较，且每分一次元素的总个数少 1 个。若要将集合分为 2^k 个部分，需要比较的次数为 $\sum_{i=0}^{k-1}(n-i)$，元素总数减少了 $\sum_{i=1}^{k} i$。此时，剩余的元素数量有 $n - \frac{(1+k)k}{2}$。使拆分结束，有

$$n - \frac{(1+k)k}{2} = 2^k \Rightarrow 2n = 2^{k+1} + k(1+k) \geq 2^{k+1}$$
$$k \leq \log n$$

所以总的比较次数为

$$\begin{aligned}\sum_{i=0}^{k-1}(n-i) = \frac{k\left[n+n-(k-1)\right]}{2} &= -\frac{1}{2}k^2 + (n+\frac{1}{2})k = -\frac{1}{2}k\left[k-(2n+1)\right] \\ &\leq -\frac{1}{2}(\log n)^2 + (n+\frac{1}{2})(\log n) \\ &= n\log n - \frac{1}{2}(\log n)^2 + \frac{1}{2}(\log n) = O(n\log n)\end{aligned}$$

算法 18 螺栓螺母配对

输入: $X[1 \dots n]$（螺栓），$Y[1 \dots n]$（螺母）

输出: $P = \{(x_i, y_j)\}$（配对的螺栓螺母）

$\mathcal{X} \leftarrow \{X\}, \mathcal{Y} \leftarrow \{Y\}$ ▷ 初始时，螺栓、螺母都只有一组

repeat

  $\hat{X} \leftarrow \mathcal{X}.back, \hat{Y} \leftarrow \mathcal{Y}.back$ ▷ 取最后面的一组（尺寸最大的）

  if $\hat{X}.length \leq \hat{Y}.length$ **then**

    $x \leftarrow \textsc{Select}(\hat{X})$ ▷ 从 $\hat{X}$ 中随机选一个 x，并从 $\hat{X}$ 中删除 x

    $Y_1, Y_2, y \leftarrow \textsc{Split}(x, \hat{Y})$ ▷ Y_1 中的螺母更小一些，Y_2 中的螺母更大一些

    $\mathcal{Y}.PopBack()$

    $\mathcal{Y}.PushBack(Y_1)$ if $\neg Y_1.empty$ ▷ 确保 $\mathcal{Y}$ 中靠前的分组中任何一个螺母的尺寸

    $\mathcal{Y}.PushBack(Y_2)$ if $\neg Y_2.empty$ ▷ 严格小于靠后的分组中所有螺母的尺寸

    $P.PushBack((x, y))$ ▷ 配对的

  else

    $y \leftarrow \textsc{Select}(\hat{Y})$ ▷ 从 $\hat{Y}$ 中随机选一个 y，并从 $\hat{Y}$ 中删除 y

    $X_1, X_2, x \leftarrow \textsc{Split}(y, \hat{X})$

    $\mathcal{X}.PopBack()$

    $\mathcal{X}.PushBack(X_1)$ if $\neg X_1.empty$

    $\mathcal{X}.PushBack(X_2)$ if $\neg X_2.empty$

    $P.PushBack((x, y))$

  end if

until $\neg \mathcal{X}.empty \wedge \neg \mathcal{Y}.empty$ ▷ $\mathcal{X}, \mathcal{Y}$ 应当同时为空

procedure $\textsc{Split}(pivot, C)$

  $S \leftarrow \Phi, I \leftarrow \Phi, peer$

  while $\neg C.empty$ **do**

    $c \leftarrow C.PopBack()$

    if $c > pivot$ **then**

      $S.PushBack(c)$

    else if $c < pivot$ **then**

      $I.PushBack(c)$

    else

      $peer \leftarrow c$

    end if

  end while

  return $I, S, peer$

end procedure

10 0402

1. 用分治法找到数组中的最大数和最小数，若数组规模为 2 的幂，证明需要的比较次数为 $\frac{3}{2}n-2$

易判断初始条件 $T(1)=0, T(2)=1$，且有递推关系

$$T(n)=T(2^m)=2T(2^{m-1})+2 \Rightarrow T(2^m)+2=2T(2^{m-1})+4$$

即

$$\frac{T(2^m)+2}{T(2^{m-1})+2}=2$$

所以当 $m\geq 1$ 时，$T(2^m)+2=3\cdot 2^{m-1}$，可得 $T(2^m)=\begin{cases}3\cdot 2^{m-1}-2 & ,m\geq 1\\ 1 & ,m=0\end{cases}$

即

$$T(n)=\begin{cases}\frac{3}{2}n-2 & ,n\geq 2\\ 1 & ,n=1\end{cases}$$

2. 对于任意给定的 4 个 1-10 之间的整数（可以相同），判断是否可以通过整数四则运算得到 24

我们将运算中不具有交换律的情况算作两种运算，并规定 $x_1\leq x_2$，使两个数构成有序数对。则任一个有序数对的两个数之间可能有 6 种运算，即

$$x_1+x_2 \quad x_1\cdot x_2 \quad x_1-x_2 \quad x_2-x_1 \quad x_1/x_2 \quad x_2/x_1$$

记 $Q(a,b)$ 为 a 与 b 进行这 6 种计算得到的所有结果的集合，$|Q(a,b)|\leq 6$。

记 $J(A,B,O,n)$ 为判断是否存在 A 中的某个数能和 B 中的某个数通过 O 中的运算直接得到 n。对于每一种运算，使用类似于算法16中的算法，首先扫描一遍较小的数组并在哈希表中记录下互补的值，然后扫描另一个数组看是否存在于哈希表中。这一部分需要进行 $\min\{|A|,|B|\}$ 次计算和 $\max\{|A|,|B|\}$ 次哈希表查找。所以 $J(A,B)$ 的时间复杂度为 $|O|(|A|+|B|)$。

4 个数的结合次序有两种，可表示为后缀表达式

- $(x_1,x_2,op_1,x_3,x_4,op_2,op_3)$：4 个数两两一组分为 2 组共有 $\frac{\binom{4}{2}+\binom{2}{2}}{2}=3$ 种分组方式。所以讨论全部结果需要执行 3 次

  $$J\left(Q(x_1,x_2),Q(x_3,x_4),\left\{+,\times,-,\div,\hat{-},\hat{\div}\right\},24\right)$$

  总的运算次数为 $3\times 6\times 12=216$

- $(x_1, x_2, x_3, x_4, op_1, op_2, op_3)$：此时若 op_2 与 op_3 的运算顺序是可交换的，则与上一种情况相同。
  - op_2, op_3 同为加、减法（可通过增减括号后等价）
  - op_2, op_3 同为乘、除法

  因此，给定 op_2 后，op_3 只有 3 种选择排列方式共有 $\binom{4}{2}\binom{2}{1} = 12$ 种。

  $$J\left(Q(Q(x_3, x_4), x_2), \{x_1\}, \{+, -, \hat{-}\}, 24\right)$$

  $Q(x_3, x_4)$ 的运算结果可由上一种情况的运算结果查表得到。运算次数为 $12 \times 6^2 \times 3 = 1296$ 次

总运算次数为 1512。

3. 有 $n = 2^k$ 位选手参加一项单循环比赛，即每位选手都要与其他 $n-1$ 位选手比赛，且在 $n-1$ 天内每人每天进行一场比赛

   (a) 设计算法以得到赛程安排

赛程安排可表示为表格 M 如下（以 $k = 2$ 为例）：

	1	2	3	4
1	×	0	1	2
2	−	×	2	0
3	−	−	×	1
4	−	−	−	×

上表中，$m_{1,2} = 0$ 表示 1 号选手和 2 号选手在第 0 天进行一场比赛。由于自己不能和自己比赛，所以对角线上的值是无意义的。同时表格中的值应关于对角线对称，所以只需考虑对角线一侧的值（这里只考虑上方）。因此只要以某种算法，使用 $[0, n-1)$ 中的整数填写上表，使每行、每列都不存在重复的数即可。

填写表格方法为：

- 第一个数：

  $$m_{1,2} = 0$$

- 第一行其他的数：

  $$\forall j : 2 < j \leq n, m_{1,j} = [1 + m_{1,j-1} \bmod (n-1)]$$

- 表中其他的数：

  $$\forall i, j : i > 1, j > i, m_{i,j} = [1 + m_{i-1,j-1} \bmod (n-1)]$$

算法 19 比赛安排

输入: n（参加比赛的人数）

输出: $M[1\ldots n][1\ldots n]$（比赛的安排，i 号选手与 j 号选手在第 $M[i][j]$ 天比赛）

```
1: M[1][2] ← 0
2: for j ← 3 to n do
3:     M[1][j] ← M[1][j − 1] mod (n − 1)
4: end for
5: for i ← 2 to n do
6:     for j ← i + 1 to n do
7:         M[i][j] ← M[i − 1][j] mod (n − 1)
8:     end for
9: end for
```

由于第一行中只有 $n-1$ 个数，所以恰能使用 $[0, n-1)$ 中所有的数填满且不重复。由于每一列中最多只有 $n-1$ 个数，且有循环递增关系，所以一定能使用 $[0, n-1)$ 中的数填满而不产生重复。由于第一行从左到右有循环递增关系，第二行的值为第一行对应的值 $+1$，所以行中的递增关系保持，即每一行中也没有重复的数。因此由此方法填写的表格满足要求。

该表格中安排了 $\frac{n^2-n}{2} = 2^{k-1}(n-1)$，满足每天 k 场比赛，$n-1$ 天恰好比完。算法的时间复杂度为 $O(n^2)$

(b) 若比赛结果存放在一矩阵中，针对该矩阵设计 $O(n \log n)$ 的算法，为各个选手赋予次序 P_i，使得 P_1 打败 P_2，P_2 打败 P_3，依此类推。

对于任意两个选手 i, j，定义若 i 赢了 j 则 $i < j$。由于每两个人之间都有一场比赛，所以任意两个人之间都可以按上述定义比较大小，该大小关系可以通过查询比赛结果的数组得到。使用一种基于比较的排序算法（如快速排序），即可在 $O(n \log n)$ 的时间内排除顺序。

4. 数组 A 中包含 n 个互不相同的整数，用 $O(n)$ 的时间找出 A 中最大的 i 个数，并按从大到小的次序输出（$i \leq n^{1/2}$）

首先使用自底向上的方法建最大堆，然后出堆 i 次。

- 自底向上建堆过程中，每一步比较的次数最多是该节点高度的 2 倍。对于高度为 i 的节点，记其所有子节点的高度和为 $H(i)$，则有

$$H(i) = 2H(i-1) + 1 \xrightarrow{H(0)=0} H(i) = 2^{i+1} - i - 2$$

而 n 个节点的最大高度为 $\log n$，所以这一部分的时间复杂度为

$$T_1(n) = 4n - 2\lceil \log n \rceil - 4$$

- 出堆 i 次所需的时间 $T_2(i) \leq i \log n$。

所以整体的时间复杂度

$$\begin{aligned} T(n,i) = T_1(n) + T_2(i) &\leq (4n - 2\lceil \log n \rceil - 4) + i \log n \\ &\leq 4n - 2\lceil \log n \rceil - 4 + \sqrt{n} \log n \\ &\leq 4n - 2\lceil \log n \rceil - 4 + \sqrt{n} \cdot n^{1/2} \\ &= 5n - 2\lceil \log n \rceil - 4 = O(n) \end{aligned}$$

11 0409

1. $G = (V, E)$ 是一个无向图，每个顶点的度数都为偶数。设计线性时间算法，给 G 中每条边一个方向，使每个顶点的入度等于出度。（请先简单说明算法思想，再给出伪代码，然后证明其时间复杂性符合要求）

因为无向图每个顶点的度数都为偶数，所以该图是欧拉图，即一定存在欧拉回路能经过每一条边且每条边仅经过一次。沿欧拉回路标记每一条边的方向，即可保证每个顶点的入度等于出度。

2. 连连看游戏中用户可以把两个相同的图用线连到一起，如果连线拐的弯小于等于两个则表示可以消去。设计算法，判断指定的两个图形能否消去。

分以下三种情况讨论：两个图案通过一条直线、两段折线、三段折线相连。

伪代码见算法20、算法21。

3. 证明任意连通无向图中必然存在一个点，删除该点不影响图的连通性。用线性时间找到这个点。

以图中任意一节点为根节点做深度优先搜索，一定存在搜索到某个节点时，该节点的所有相邻节点都已经被标记了。即该节点的所有相邻节点都可以从根节点通过不经过该节点的路径到达。所以删除该节点一定不影响图的连通性。

算法 20 判别两图案是否可以消除 (3)

输入: 矩阵 $M[0..m+1][0..n+1]$，其中 $M[1..m][1..n]$ 为游戏图案；矩阵上两个元素 $M[r_1][c_1], C[r_2][c_2]$
输出: 两元素是否可以消除

```
1: if OneSegment(M, r1, c1, r2, c2) then
2:     return True
3: else if TwoSegment(M, r1, c1, r2, c2) then
4:     return True
5: else if ThreeSegment(M, r1, c1, r2, c2) then
6:     return True
7: else
8:     return False
9: end if

10: procedure Expands(M[0...m+1][0...n+1], row, col)
11:     找到 d ≤ row ≤ u，使得 M[d...u][col] 范围内除了 M[row][col] 全为空
12:     找到 l ≤ col ≤ r，使得 M[row][l...r] 范围内除了 M[row][col] 全为空
13:     return (u, d, l, r)
14: end procedure

15: procedure ThreeSegment(M[0...m+1][0...n+1], r1, c1, r2, c2)
16:     (u1, d1, l1, r1) ← Expands(M, r1, c1), (u2, d2, l2, r2) ← Expands(M, r2, c2)
17:     d ← max(d1, d2), u ← min(u1, u2), l ← max(l1, l2), r ← min(r1, r2)
18:     for r ← d to u (r ≠ r1 ∧ r ≠ r2) do
19:         if OneSegment(M, r, c1, r, c2) then
20:             return True
21:         end if
22:     end for
23:     for c ← l to r (c ≠ c1 ∧ c ≠ c2) do
24:         if OneSegment(M, r1, c, r2, c) then
25:             return True
26:         end if
27:     end for
28:     return False
29: end procedure
```

算法 21 判别两图案是否可以消除 (1)(2)

```
1: procedure OneSegment(M[0...m+1][0...n+1], r_1, c_1, r_2, c_2)
2:     if r_1 = r_2 then                                  ▷ 判断是否能通过一条水平线消除
3:         for c ← min(c_1, c_2) + 1 to max(c_1, c_2) − 1 do
4:             if ¬M[r_1][c].isEmpty then
5:                 return False
6:             end if
7:         end for
8:         return True
9:     else if c_1 = c_2 then                             ▷ 判断是否能通过一条竖直线消除
10:        for r ← min(r_1, r_2) + 1 to max(r_1, r_2) − 1 do
11:            if ¬M[r][c_1].isEmpty then
12:                return False
13:            end if
14:        end for
15:        return True
16:    else                                               ▷ 两图案不在同一条直线上
17:        return False
18:    end if
19: end procedure

20: procedure TwoSegment(M[0...m+1][0...n+1], r_1, c_1, r_2, c_2)
21:    if OneSegment(M, r_1, c_1, r_1, c_2) ∧ OneSegment(M, r_2, c_2, r_1, c_2) then
22:        return True
23:    else if OneSegment(M, r_1, c_1, r_2, c_1) ∧ OneSegment(M, r_2, c_2, r_2, c_1) then
24:        return True
25:    end if
26:    return False
27: end procedure
```

进行深度优先搜索的时间复杂度为 $O(|V|+|E|)$。因为该算法不需要完成对整个图的遍历，所以该算法的时间复杂度不超过 $O(|V|+|E|)$。

伪代码见算法22。

算法 22 寻找割点

输入: 图 $G=(V,E)$

输出: 割点 $v \in V$，使图中删除点 v 时不影响图的连通性

```
1: v ← SELECT(V)                                  ▷ 任选一个节点作为根节点
2: v.marked ← True                                ▷ 标记已经搜索过 v_0
3: repeat
4:     continue ← False
5:     for (v, w) ∈ E do
6:         if ¬w.marked then                      ▷ 如果还存在一个与 v 相邻的节点 w 未被标记
7:             w.marked ← True                    ▷ 标记该点
8:             v ← w
9:             continue ← True                    ▷ 从该点开始继续搜索
10:            break
11:        end if
12:    end for
13: until ¬continue                               ▷ continue 为 False 时，节点 v 不存在未被标记的邻点
14: return v
```

12 0413

1. 对于给定的二叉树，求其最小深度，即从根节点到最近的叶子的距离。

广度优先搜索最先到达的叶子节点有最小的深度。

最差情况下遍历了整棵树，时间复杂度为 $O(N)$，其中 N 为节点的数量。

算法 23 求二叉树的最小深度

输入: $root$（给定二叉树的根节点）

输出: $mindepth$（二叉树中的最小深度）

```
1:  q ← Queue.Init
2:  Queue.Push(q, ⟨root, 0⟩)
3:  while ¬Queue.isEmpty(q) do
4:      ⟨current, depth⟩ ← Queue.Pop(q)
5:      if current.isLeaf then                          ▷ 当前节点是叶子节点
6:          return depth                                ▷ BFS 最先到达的叶子节点有最小的深度
7:      end if
8:      if current.left then                            ▷ 左子树入队
9:          Queue.Push(q, ⟨current.left, depth + 1⟩)
10:     end if
11:     if current.right then                           ▷ 右子树入队
12:         Queue.Push(q, ⟨current.right, depth + 1⟩)
13:     end if
14: end while
```

2. 设 G 是有向非循环图，其所有路径最多含 k 条边。设计线性时间算法，将所有顶点分为 $k+1$ 组，每一组中任意两个点之间不存在路径。

DAG 中一定存在入度为 0 的节点，且一定存在出度为 0 的节点，最长路径出现在入度为 0 的节点和出度为 0 的节点之间（否则可以沿两端点继续扩展路径，得到更长的路径）。

按照拓扑排序的顺序，删除 DAG 中入度为 0 的节点后，上述最长路径的长度一定减小 1。且在拓扑排序中同一批被移除的节点之间一定不存在路径（否则删除起点时，终点的入度一定不为 0）。因此可按照拓扑排序中删除节点的批次将所有节点分为 $k+1$ 组，且每一组中任意两个节点之间不存在路径。

当图以出边邻接表给出时，算法的时间复杂度为 $O(|V|+|E|)$。

算法 24 拓扑分组

输入: $G = (V, E)$（给定 DAG 的出边表）

输出: $vtop\,[1 \ldots |V|]\,, bnd$（$vtop\,[bnd[i] \ldots bnd[i+1]]$ 中的节点不存在路径）

```
1: vtop ← Vector.Init, bnd ← Vector.Init
2: indeg ← Array.Init(|V|, 0)                                  ▷ 构造入度表
3: for (v, w) in E do
4:     indeg[w] ← indeg[w] + 1
5: end for
6: for v in V where indeg[v] = 0 do                            ▷ 找到入度为 0 的节点
7:     Vector.PushBack(vtop, v)
8: end for
                                                               ▷ 拓扑排序
9: inf ← 1, Vector.PushBack(bnd, inf)
10: sup ← Vector.size(vtop), Vector.PushBack(bnd, sup)
11: while Vector.size(vtop) < |V| do
12:     for idx ← inf to sup do          ▷ 对于每一个上一阶段找到的入度为 0 的节点
13:         for (v, w) in E where v = vtop[idx] do              ▷ 找到其后继节点
14:             indeg[w] ← indeg[w] − 1                        ▷ 减后继节点的入度
15:             if indeg[w] = 0 then                           ▷ 并检查是否减到了 0
16:                 Vector.PushBack(vtop, w)
17:             end if
18:         end for
19:     end for
20:     inf ← sup + 1
21:     sup ← Vector.size(vtop), Vector.PushBack(bnd, sup)
22: end while
```

3. 给定连通无向图 G 以及 3 条边 a, b, c，在线性时间内判断 G 中是否存在一个包含 a 和 b 但不含 c 的闭链。

删除边 c，在剩余的图中判断是否存在包含 a, b 的回路。

在剩余的无向图中划分双连通分支（线性时间），若 a, b 在同一个双连通分支里，则存在一条包含 a, b 但不包含 c 的回路。

13 0416

1. 求 $n \times m$ 棋盘上任意两点之间马能够走的最短路径长度

构造图 $G = (E, V)$

- E 为棋盘上所有格点
- 若“马”能从棋盘上的点 v 经过一步走到 w，则 $(v, w) \in V$

对上图进行广度优先搜索即可找出最短路径长度。进一步地，构图和广度优先搜索可以同时进行。

2. 设计线性时间算法求树的最大匹配

树中一个非叶子节点连向其子节点的若干条边及连向其父节点的边中，至多只有一条边属于匹配。

记函数 $f(v, b)$ 为以 v 为根节点的子树中最大匹配的边数，其中 b 为布尔值

- 当 $b = 1$ 时，存在一条与 v 直接相连的边在最大匹配中
- 当 $b = 0$ 时，与 v 直接相连的边都不在最大匹配中

对于树的根节点 v，假设其有 3 个子节点 w_1, w_2, w_3 并有相应地与之相连的三条边 e_1, e_2, e_3，则

$$f(v) = \max_b f(v, b) = \max \begin{cases} f(w_1, 1) + f(w_2, 1) + f(w_3, 1) & (b = 0) \\ \max \begin{cases} f(w_1, 0) + f(w_2, 1) + f(w_3, 1) + 1 \\ f(w_1, 1) + f(w_2, 0) + f(w_3, 1) + 1 \\ f(w_1, 1) + f(w_2, 1) + f(w_3, 0) + 1 \end{cases} & (b = 1) \end{cases}$$

观察可知对于每一棵子树，只需考察 $b = 1$ 和 $b = 0$ 两种情况；且 $b = 0$ 当且仅当对于其所有子树 $b = 1$。当 v 是叶子节点时，$f(v) = f(v) = 0$；所以当 v 的所有子节点都是叶子节点时，$f(v) = f(v, 1) = 1$。对树做一次后序遍历，并按照上述规则计算出所有节点的 $f(v, 0)$ $f(v, 1)$ 即可。

每个节点被访问两次（下行入栈一次，上行出栈一次）；计算 f 时比较的情况数等于该节点的度数，所有节点的度数和为 $2|E|$。所以算法的时间复杂度为 $O(|V| + |E|)$

3. 无向图 G 的顶点覆盖是指顶点集合 U，G 中每条边都至少有一个顶点在此集合中。设计线性时间算法为树寻找一个顶点覆盖，并且使该点集的规模尽量小。

初始化所有结点的度。对于所有度数为 1 的节点，首先标记其相邻的点都在 U 中，然后删除与 U 中的点直接相邻的所有边。重复上述步骤，直至所有边都被删除。

14 0420

1. 设计算法判定平面上 n 个点是否在一条直线上

任选一点 $p_0(x_0, y_0)$ 作为基准点。对其他 $n-1$ 个点，计算其相对于点 p_0 的斜率。若所有的斜率都相等，则这 n 个点在同一条直线上。

该算法具有线性时间复杂度。

2. 设 P 是包围在给定矩形 R 中的一个简单多边形，q 为 R 中任意一点。设计高效算法寻找连接 q 和 R 外部一点的线段，使得该线段与 P 相交的边的数量最少。

将该多边形表示为用邻接表存储的图 $G = (V, E)$，设有 n 个顶点 n 条边，图中每个顶点的度数均为 2。以 q 为坐标原点建立极坐标系，求出每个顶点的极坐标 (ρ, θ)，并按极角 θ 的大小排序。应用扫描线算法，事件点进度表按极角排序的所有顶点，扫描线状态为与射线 $\theta = \alpha$ 相交的边集。

见算法25

15 0423

1. 给定平面上一组点，已知每个点的坐标，求最远点对之间的距离，即点集的直径。（不得穷举，文献查阅，然后用自己的语言进行算法思想的描述，包括时间复杂性分析）

Shamos M I. Computational geometry[Ph. D. Thesis][J]. 1978.

引理 最远点对一定在这组点的凸包上。

证明 反证法。假设最远点对为 (p, q)，其中 q 不在这组点的凸包上，则根据凸包的定义 q 一定在凸包的内部。对凸包多边形进行三角形分划，一定能找到

- 凸包上一点 x，使点 q 在线段 px 上。此时一定有 $d_{px} > d_{pq}$，即 pq 不是最远点对。

算法 25 最小相交边数量

输入: $G = (V, E)$（给定多边形 P），q（给定矩形 R 内的一点）

输出: γ （射线 $\theta = \gamma$ 与 P 相交的边数量最少）　▷ 在该射线上任取 R 外的一点即满足题设

以 q 为坐标原点，x 轴正方向为极轴，建立极坐标系，并求各顶点的极坐标　▷ $O(n)$

$Q \leftarrow \textsc{MinHeap}(V, \theta)$　▷ 对顶点按极角构建最小化堆，$O(n)$

$S \leftarrow \Phi, count \leftarrow 0$

for (v, w) in E **do**　▷ 找出多边形 P 与射线 $\theta = 0$ 相交的所有边，$O(n)$

  if $\sin(v.\theta)\sin(w.\theta) < 0 \wedge q.x < \frac{q.y - w.y}{v.y - w.y}(v.x - w.x) + w.x$ **then**

  ▷ 两点在极轴上下两侧，且与直线 $y = q.y$ 的交点在 $q.x$ 的右侧

    $S \leftarrow S \cup \{(v, w)\}, count \leftarrow count + 1$

  end if

end for

$minCount \leftarrow count, \alpha \leftarrow 0, \beta \leftarrow 0$

while $p \leftarrow \textsc{Heap.Pop}(Q)$ **do**　▷ 扫描线：按极角升序遍历所有顶点，$O(n \log n)$

  $s, t \leftarrow \textsc{Adjacency}(G, p)$　▷ 从邻接矩阵中找 p 的两个相邻顶点，$O(1)$

  if $(p, s) \in S \wedge (p, t) \in S$ **then**　▷ 扫描线越过顶点 p 后，两条边都不与扫描线相交

    $S \leftarrow S - \{(p, s), (p, t)\}$

    $count \leftarrow count - 2$

    if $count < minCount$ **then**

      $minCount \leftarrow count$

      $\alpha \leftarrow p.\theta, \beta \leftarrow p.\theta$

    end if

  else if $(p, s) \in S \wedge (p, t) \notin S$ **then**　▷ 扫描线越过顶点 p 后，与另一条边相交

    $S \leftarrow S \cup \{(p, t)\} - \{(p, s)\}$

  else if $(p, s) \notin S \wedge (p, t) \in S$ **then**

    $S \leftarrow S \cup \{(p, s)\} - \{(p, t)\}$

  else if $(p, s) \notin S \wedge (p, t) \notin S$ **then**　▷ 扫描线越过顶点 p 后，两条边都与扫描线相交

    $S \leftarrow S \cup \{(p, s), (p, t)\}$

    $count \leftarrow count + 2$

    $\beta \leftarrow p.\theta$ if $\alpha = \beta$ else β

  end if

end while

$\gamma \leftarrow \frac{\alpha + \beta}{2}$ if $\alpha \neq \beta$ else $\frac{\alpha + 2\pi}{2}$

- 或凸包上的两点 x, y，使点 q 在 $\triangle pxy$ 的内部。

  因为三角形内角和为 $180°$，所以在 $\triangle qxy$ 中，$\angle xqy < 180°$，所以 $\angle pqx + \angle yqp > 180°$。

不妨设 $\angle pqx \le \angle yqp$，则有 $2\angle yqp > 180°$，即 $\angle yqp > 90°$。

所以在 $\triangle yqp$ 中，$\angle yqp$ 是最大的角，$\angle yqp > \angle pyq$。由正弦定理，$d_{yp} > d_{pq}$，所以 pq 不是最远点对。

因此最远点对一定都在这组点的凸包上。 □

应用 Graham 扫描算法可以找到这组点的凸包，时间复杂度为 $O(n \log n)$

作凸多边形两条平行的支撑线，并沿逆时针方向同时旋转两条平行支撑线。则若凸包上两点是最远点对，一定存在某一时刻，使两点均在平行线上。因此在旋转的过程中求出能同时出现在两平行线上的点对之间的距离，并找到最大值即可。

因为两条支撑线将共同遍历全部的点一次，所以算法的时间复杂度为 $O(n)$。

总的时间复杂度为 $O(n \log n)$

2. 给定测度空间中位于同一平面的 n 个点，已知任意两点之间的距离 d_{ij}，存储在矩阵 D 中，求这组点的直径。

该问题的直观解法就是把 D 扫描一遍，选择其中最大的元素即可。由于是在一个测度空间中，因此 d_{ij} 满足距离的基本要求，即非负性、对称性和三角不等式。我们就可以给出一种时间亚线性的近似算法。算法很简单，由原来确定性算法的检查整个矩阵改为只随机检查 D 的某一行，这样时间复杂性就由原来的 $O(n^2)$ 减少为 $O(n)$。相对于输入规模 n^2 而言，这是一个时间亚线性的算法。那么时间代价减小的同时，证明解不会小于最优值的一半。

证明 记 (p, q) 为平面上最远点对，即 d_{pq} 为该点集的直径。

在矩阵 D 中任取一行，即在平面上任取一点 x，考察 x 与平面上其他点的距离。

- 若 $x = p \vee x = q$，则 d_{pq} 一定在选定的这一行中，所得解即为最优值。
- 若 $x \neq p \wedge x \neq q$，则 d_{xp}, d_{xq} 一定在选定的这一行中。
  - 若 x, p, q 共线，则有 $d_{xp} + d_{xq} = d_{pq}$
  - 若 x, p, q 不共线，则三点构成平面上的一个三角形，有 $d_{xp} + d_{xq} > d_{pq}$

即 $d_{xp} + d_{xq} \ge d_{pq}$。不妨设 $d_{xp} \le d_{xq} \le d_{pq}$，

$$d_{pq} \le d_{xp} + d_{xq} \le 2d_{xq} \Longrightarrow d_{xq} \ge \frac{1}{2} d_{pq}$$

设选定的这一行中最大的距离（即算法的输出）为 d_{xy}，则

$$d_{xy} \ge d_{xq} \ge \frac{1}{2} d_{pq}$$

3. 在平面上给定一个有 n 个点的集合 S，求 S 的极大点。

   极大点的定义：设 $p_1 = (x_1, y_1)$ 和 $p_2 = (x_2, y_2)$ 是平面上的两个点，如果 $x_1 \le x_2$ 并且 $y_1 \le y_2$，则称 p_2 支配 p_1，记为 $p_1 \prec p_2$。点集 S 中的点 p 为极大点，意味着在 S 中找不到一个点 q，$q \ne p$ 并且 $p \prec q$，即 p 不被 S 中其它点支配。

算法 26 求平面的极大点

输入: $S = \{(x_i, y_i)\}$（平面上的 n 个点）

输出: $P = \{(x_i, y_i)\}$（平面上所有的极大点）

```
1: S ← Sort(S, x, descending)        ▷ 对横坐标降序排序，O(n log n)
2: maxY ← −∞
3: for (x, y) in S do                ▷ 按横坐标降序遍历
4:     if y > maxY then              ▷ 横坐标大于 x 的点纵坐标都小于 y，没有其他点可以支配该点
5:         P ← P ∪ {(x, y)}
6:         maxY ← y
7:     else
                                     ▷ 该点被之前遍历过的某个纵坐标为 maxY 的点支配
8:     end if
9: end for
```

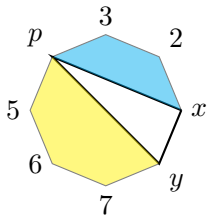
16 0426

1. 对凸多边形

   (a) 有多少种三角划分的方法?

任意一个三角形可以由一条边和不在这条边所在直线上的一个点构成。

凸多边形的任意一条边都一定属于划分后的一个三角形，任意一个顶点都一定是划分后三角形的一个顶点。在凸多边形中，确定一条边后，其他的顶点一定都不在这条边所在的直线上（否则该多边形一定不是凸多边形）。



所以凸 n 边形的一条边可以与其他的 $n-2$ 个点构成三角形，且该三角形将原凸多边形分成两个凸多边形，这两个凸多边形可以以相同的算法分划，直到其边数不超过 3。

设 $T(n)$ 为一个 n 边形的三角形划分方法。则

$$T(n)=\begin{cases}0 & ,n=2\\ 1 & ,n=3\\ \sum_{i=2}^{n-1}T(i)T(n-i+1) & ,n>3\end{cases}$$

(b) 如何使对角线长度之和最小?

使用与上题相同的归纳方式。

在 $\triangle xpy$ 是三角形划分后结果中的一个三角形的条件下，当多边形对角线长度之和最小时，黄色和蓝色两个凸多边形一定达到了使其对角线长度之和分别最小的分划。

记凸 n 边形为 $\langle p_1,p_2,\ldots,p_n\rangle$，其对角线长度和最小为 $Q(\langle p_1,p_2,\ldots,p_n\rangle)$。

$$Q(\langle p_1,p_2,\ldots,p_n\rangle)=\begin{cases}0 & ,n\le 3\\ \min_{1<i<n}\left\{\begin{array}{r}Q(\langle p_1,\ldots,p_i\rangle)+Q(\langle p_i,\ldots,p_n\rangle)\\ +D(p_1,p_i)+D(p_n,p_i)\end{array}\right\} & ,n>3\end{cases}$$

其中，$D(p_i,p_j)=\begin{cases}0 & ,|i-j|>1\\ ||p_i-p_j|| & ,\text{otherwise}\end{cases}$

由于子问题重叠，考虑使用动态规划求解。共有 $O(n^2)$ 个子问题需要求解，求解每个子问题所需的时间复杂度为 $O(n)$。总的时间复杂度为 $O(n^3)$

伪代码见算法27

2. 给定平面上 n 条线段，设计算法用 $O(n\log n)$ 时间确定其中是否有两条线段相交。

使用扫描线算法，事件列表为线段的所有端点及所有交点按横坐标升序，当遇到第一个交点时算法结束。交点出现时，相交的两线段一定是相邻的。

该算法的时间复杂度为 $O((2n)\log(2n))=O(n\log n)$。

伪代码见算法28

3. 用扫描线算法求解最近邻点对问题

事件为所有的点，按横坐标升序遍历。扫描线状态为已经扫描过且到扫描线的距离小于某个值所有点。

伪代码见算法29

4. 有 n 种液体 $S_1, S_2, \ldots, S_n$, 都含有 A,B 两种成分, 含量分别为 $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$, $a_i + b_i < 100\%$。现欲利用这 n 种液体配制目标液体 T，使之 A 和 B 的含量分别为 x 和 y。设计算法**判别**能否成功配制，并给出算法时间复杂性。

即找到一组数 $r_i : 0 \le r_i \le 1$ 满足 $\begin{cases} \sum_{i=1}^{n} r_i = 1 \\ \sum_{i=1}^{n} r_i a_i = x \\ \sum_{i=1}^{n} r_i b_i = y \end{cases}$

即求解非齐次线性方程组

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \end{bmatrix} \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{pmatrix} = \begin{bmatrix} 1 \\ x \\ y \end{bmatrix}$$

有增广矩阵

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b} \end{bmatrix} = \left[\begin{array}{cccc|c} 1 & 1 & \cdots & 1 & 1 \\ a_1 & a_2 & \cdots & a_n & x \\ b_1 & b_2 & \cdots & b_n & y \end{array}\right]_{3 \times n+1} = \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \boldsymbol{r}_3 \end{bmatrix}$$

对 $\begin{bmatrix} \boldsymbol{A} & \boldsymbol{b} \end{bmatrix}$ 进行行变换。行变换 $\boldsymbol{r}_i \leftarrow \alpha \boldsymbol{r}_i + \beta \boldsymbol{r}_j$ 的时间复杂度为 $O(n)$，经过最多 3 次行变换，然遍历三行即可求出 $\boldsymbol{A}$ 的秩 $r(A)$。

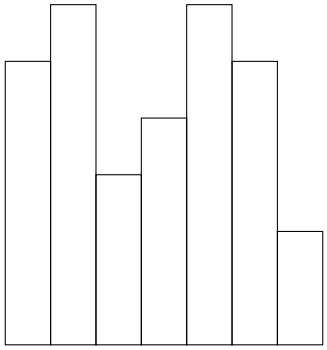
设变换后的增广矩阵为 $\begin{bmatrix} \boldsymbol{R} & \boldsymbol{d} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_1 & \boldsymbol{r}_2 & \boldsymbol{r}_3 \end{bmatrix}^T$

- 若 $r(A) = 3$，则再经过最多三次行变换即可使 $\boldsymbol{A}$ 中的三列构成三阶单位阵，进而容易求得线性无关的通解 $\boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{s}_3$ 和特解 $\boldsymbol{\eta}$。因为三组通解是线性无关的，可以作为 $\mathbb{R}^3$ 的一组基，所以一定存在 $\boldsymbol{r} = \boldsymbol{\eta} + \sum_{i=1}^{3} \boldsymbol{\alpha}_i s_i$ 满足要求。
- 若 $r(A) = 2$，则方程有解当且仅当 $d_{3,1} = 0$
- 若 $r(A) = 1$，则方程有解当且仅当 $d_{3,1} = d_{2,1} = 0$

因此判别过程的时间复杂度不超过 $6n$ 即为 $O(n)$。

17 0427

1. 海报墙由 n 块宽度相同高度不同的木板组成，那么在此海报墙上能够张贴的最大海报面积是多少？设木板宽度为 1，高度为 $h_1, h_2, \ldots, h_n$，海报必须整体都粘贴在墙上，并且不能斜贴。



应用扫描线算法，事件点序列为木板高度升序。

$$S(m,n)=\begin{cases}\max\begin{cases}(n-m+1)\cdot \min_{m\le i\le n} h_i \\ S(m, \operatorname{argmin}_{m\le i\le n} h_i - 1) & m<n \\ S(\operatorname{argmin}_{m\le i\le n} h_i + 1, n)\end{cases} \\ h_m & m=n \\ 0 & m>n\end{cases}$$

子问题互不重叠，不需要使用动态规划。

伪代码见算法30

2. 平面有两组点，如何证明存在直线可以将这两组点分开?

1. 对两组点分别求凸包，$O(\log n)$
2. 判断两个凸包是否相交，$O(n)$

18 0430

1. 已知 n 个矩形，这些矩形的边都平行于坐标轴
   (a) 求出这些矩形的交集

求两个矩形的交集所需时间为 $O(1)$。任取两个求交集，求得的交集再与下一个矩形求交集即可。该交集若存在则仍然是一个矩形。

伪代码见算法31

(b) 求出这些矩形能够覆盖的面积

使用扫描线算法。

事件列表为所有矩形的竖直边。扫描线状态为当前扫描线穿过的所有矩形。

当扫描线状态发生变化时，计算当前事件点到上一个事件点之间的有效面积，然后更新当前扫描线上的有效高度。

伪代码见算法32

2. 求 $n!$ 包含质因子 p 的数量，例如 6 含有 4 个 2，2 个 3 和 1 个 5。并给出算法的时间复杂性

伪代码见算法33

时间复杂度为 $O(n)$

3. 设 Fibonacci 数列的定义为：

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

证明每个大于 2 的整数 n 都可以写成至多 $\log n$ 个 Fibonacci 数之和，并设计算法对于给定的 n 寻找这样的表示方式

首先设计用若干 Fibonacci 数之和表示给定大于 2 的整数 n 的算法，并由此证明每个大于 2 的整数 n 都可以写成 Fibonacci 数之和。然后证明在这样的表示中，使用的 Fibonacci 数至多有 $\log_2 n$ 个。

对于一个大于 2 的整数 n，若 n 是 Fibonacci 数，则命题显然成立。若 n 不是 Fibonacci 数，则一定存在一个 $m(m > 2)$，使得

$$F(m) < n < F(m+1)$$

又

$$F(m+1) = F(m) + F(m-1)$$

所以

$$0 < n - F(m) < F(m-1)$$

不妨设 $n' = n - F(m)$。若 n' 是 Fibonacci 数，则 $n = F(m) + n'$ 表示成了两个 Fibonacci 数之和。否则若 $n' > 2$，则令 $n = n'$，重复上述步骤，直到 $0 < n' \leq 2$。此时 $n' = 1$ 或 2 仍然是 Fibonacci 数。

所以每个大于 2 的整数 n 都可以用这种算法写成若干个 Fibonacci 数之和。

伪代码见算法34

注意到

$$F(m') < n' < F(m'+1) \leq F(m-1) < F(m) < n < F(m-1)$$

所以每个 Fibonacci 数至多使用一次，且不会有连续两个 Fibonacci 出现在结果中。

又不超过 n 的最大 Fibonacci 数为 $F(m-2)$，即

$$\frac{F(m)}{F(m')} \geq \frac{F(m)}{F(m-2)} = \frac{F(m-1)+F(m-2)}{F(m-2)} = 2 + \frac{F(m-3)}{F(m-2)} \geq 2 \quad (m \geq 3)$$

即结果中相邻两个 Fibonacci 数之间的倍数一定超过 2。所以一定不超过 $\log_2 n$ 个。

19 0507

1. 设有复数 $x = a + b\boldsymbol{i}$ 和 $y = c + d\boldsymbol{i}$，设计算法，只用 3 次乘法计算乘积 xy

$$xy = (a + b\boldsymbol{i})(c + d\boldsymbol{i}) = (ac - bd) + (ad + cb)\boldsymbol{i}$$

1. $A = ad$
2. $B = bc$
3. $C = (a+b)(c-d) = ac - ad + bc - bd$

$$ac - bd = C + A - B$$
$$ad + cb = A + B$$

2. 设 P 是一个 n 位十进制正整数。如果将 P 划分为 k 段，则可得到 k 个正整数，这 k 个正整数的乘积称为 P 的一个 k 乘积。

(a) 求出 1234 的所有 2 乘积

$$1 \times 234 = 234$$
$$12 \times 34 = 408$$
$$123 \times 4 = 492$$

(b) 对于给定的 P 和 k，求出 P 的最大 k 乘积的值

使用动态规划求解。设 $Num(n,i)$ 是整数 n 的最低 i 位构成的数。设 $Prod(n,k)$ 是整数 n 的最大 k 乘积。则

$$Prod(n,k) = \begin{cases} n & ,k = 1 \\ Num(n,i) \cdot \max_i Prod(Num(n,i), k-1) & ,k > 1 \end{cases}$$

3. 分析在一般微机上如何计算二项式系数 C_n^k

输入: 正整数 n, k
输出: C_n^k

```
result ← 1
k ← min(k, n − k)
for i ← 1 to k do
    result ← result × (n − i + 1)
    result ← result ÷ i
end for
return result
```

20 0511

1. 设计算法求出 n 个矩阵 $M_1, M_2, \ldots, M_n$ 相乘最多需要多少次乘法，请给出详细的算法描述和时间复杂性

给定 $n+1$ 个正整数 $c_0, c_1, \ldots, c_n$，其中 c_{i-1} 和 c_i 为矩阵 M_i 的行数和列数，$1 \le i \le n$。

记 M_{ij} 为 $M_iM_{i+1}\dots M_j$ 的乘积，$Q(i,j)$ 为计算 M_{ij} 所需要的最多乘法数量，则

$$Q(i,j)=\begin{cases}\max_{i\le k<j}\{Q(i,k)+Q(k+1,j)+c_{i-1}c_kc_j\} & ,i<j\\ 0 & ,i=j\end{cases}$$

伪代码见算法35，算法的时间复杂度为 $O(n^3)$。

2. 设有算法 A 能够在 $O(i)$ 时间内计算一个 i 次多项式和一个 1 次多项式的乘积，算法 B 能够在 $O(i\log i)$ 时间内计算两个 i 次多项式的乘积。现给定 d 个整数 $n_1,n_2,\dots,n_d$，设计算法求出满足 $P(n_1)=P(n_2)=\cdots=P(n_d)=0$ 且最高次项系数为 1 的 d 次多项式 $P(x)$，并给出算法的时间复杂性。

满足 $P(n_1)=P(n_2)=\cdots=P(n_d)=0$ 且最高次项系数为 1 的 d 次多项式 $P(x)$ 为

$$P(x)=(x-n_1)(x-n_2)\cdots(x-n_d)$$

记 $Q(i,j)=(x-n_i)\cdots(x-n_j)$，则 $P(x)=Q(1,d)$

$$Q(i,j)=\begin{cases}A(Q(i,j-1),(x-n_j)) & ,j-i+1\equiv 1 \bmod 2\\ B(Q(i,(i+j-1)/2),Q((i+j-1)/2+1,j)) & ,j-i+1\equiv 0 \bmod 2\end{cases}$$

伪代码见算法36

设该算法运行所需时间为 $T(d)$，则

$$T(d)=\begin{cases}1 & ,d=2\\ T(d-1)+O(d-1) & ,d\equiv 1 \bmod 2\\ 2T\left(\frac{d}{2}\right)+O\left(\frac{d}{2}\log\frac{d}{2}\right) & ,d\equiv 0 \bmod 2\end{cases}$$

- 当 $d=2^k$ 时，

$$\begin{aligned}T\left(2^k\right)=&2T\left(2^{k-1}\right)+O\left((k-1)2^{k-1}\right)\\ =&2^2T\left(2^{k-2}\right)+O\left(2(k-2)2^{k-2}+(k-1)2^{k-1}\right)\\ =&2^2T\left(2^{k-2}\right)+O\left(\left[(k-2)+(k-1)\right]2^{k-1}\right)\\ =&2^{k-1}T(2)+O\left(\sum_{i=1}^{k-1}(k-i)2^{k-1}\right)\\ =&2^{k-1}+O\left(k^22^{k-1}\right)=O\left(d\cdot\log^2 d\right)\end{aligned}$$

- 当 $d=2^k-1$ 时，记 $u_k=2^k-1$，且有 $u_k-1=2u_{k-1}$

$$\begin{aligned}
T\left(2^k-1\right)=T\left(u_k\right)=&T\left(u_k-1\right)+O\left(u_k-1\right)=T\left(2u_{k-1}\right)+O\left(2u_{k-1}\right)\\
=&2T\left(u_{k-1}\right)+O\left(u_{k-1}\log\left(u_{k-1}\right)\right)+O\left(2u_{k-1}\right)\\
=&2^2T\left(u_{k-2}\right)+O\left(2u_{k-2}\log\left(u_{k-2}\right)+u_{k-1}\log\left(u_{k-1}\right)\right)+O\left(2^2u_{k-2}+2u_{k-1}\right)\\
=&2^{k-2}T\left(u_2\right)+O\left(\sum_{i=2}^{k-1}2^{k-i-1}u_i\log\left(u_i\right)\right)+O\left(\sum_{i=2}^{k-1}2^{k-i}u_i\right)\\
=&O\left(3\cdot2^{k-2}\right)+O\left(\sum_{i=2}^{k-1}2^{k-i-1}\left(2^i-1\right)i+\sum_{i=2}^{k-1}2^{k-i}\left(2^i-1\right)\right)\\
=&O\left(2^{k-2}k^2+3\times2^{k-2}k+k-3\times2^k+3\right)=O\left(2^kk^2\right)=O\left(d\log^2d\right)
\end{aligned}$$

3. 将正整数 n 表示成一系列正整数之和：$n=n_1+n_2+\cdots+n_k$，其中 $n_1\geq n_2\geq\cdots\geq n_k\geq1$，$k\geq1$。正整数 n 的这种表示称为正整数 n 的划分，例如正整数 6 有如下 11 种不同的划分：

$$\begin{array}{ccccc}
6 & & & & \\
5+1 & & & & \\
4+2 & & 4+1+1 & & \\
3+3 & & 3+2+1 & & 3+1+1+1 \\
2+2+2 & & 2+2+1+1 & & 2+1+1+1+1 \\
1+1+1+1+1+1 & & & &
\end{array}$$

设计算法求正整数 n 的不同划分个数并证明其时间复杂性为 $\Theta(n^2)$。

记 $Q(n,m)$ 为最大数为 m 的 n 的所有划分，即 $m=n_1\geq n_2\geq\cdots\geq n_k\geq1$。当一个划分中最大的数为 k 时，划分中剩余的部分为 $Q(n-k,k)$，使用动态规划求解。

$$P(n)=\bigcup_{1\leq m\leq n}Q(n,m)$$

其中

$$Q(n,m)=\begin{cases}\bigcup_{1\leq k\leq m,k<n}\{k\}\times Q(n-k,k) & n>m\geq1\\ \{\{n\}\} & n=m\end{cases}$$

代码见算法37

求解 $P(n)$，需要求解 $\frac{n(n-1)}{2}$ 个子问题 Q，所以时间复杂度为 $\Theta(n^2)$

21 0514

1. 输入是由数轴上的区间所组成的集合，这些区间由它们的两个端点表示。设计 $O(n \log n)$ 算法识别所有包含在集合中其它某个区间的区间。这个问题与二维平面极大点问题有什么关系

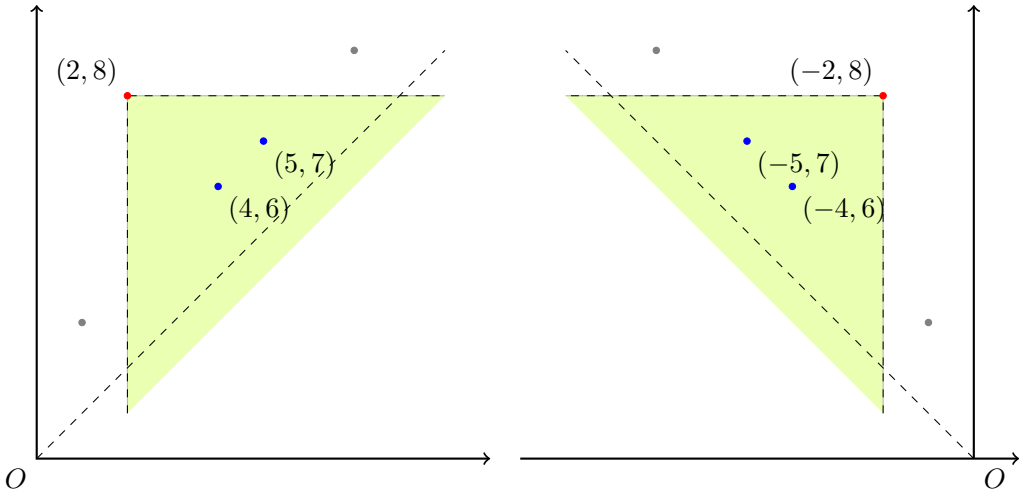
   例如输入：$(1,3),(2,8),(4,6),(5,7),(7,9)$，则输出为 $(4,6)$ 和 $(5,7)$

极大点的定义 设 $p_1 = (x_1, y_1)$ 和 $p_2 = (x_2, y_2)$ 是平面上的两个点，如果 $x_1 \leq x_2$ 并且 $y_1 \leq y_2$，则称 p_2 支配 p_1，记为 $p_1 \prec p_2$。点集 S 中的点 p 为极大点，意味着在 S 中找不到一个点 q，$q \neq p$ 并且 $p \prec q$，即 p 不被 S 中其它点支配。

区间 (x_1, y_1) 包含区间 (x_2, y_2) 当且仅当

$$x_1 \leq x_2 \wedge y_1 \geq y_2$$

将区间用平面上的点表示，所有的点都在直线 $y = x$ 上方。



(a) $(x, y) \longmapsto (x, y)$ (b) $(x, y) \longmapsto (-x, y)$

图 1: 将区间映射到二维平面上的点

为了与二维平面上极大点的定义相匹配，定义区间 (x, y) 映射到平面上的一点 $(-x, y)$。则上述区间包含的等价条件相应地变为

$$(x_2, y_2) \subseteq (x_1, y_1) \iff -x_2 \leq -x_1 \wedge y_2 \leq y_1 \iff (-x_2, y_2) \prec (-x_1, y_1)$$

所以该问题等价于在平面点集中，找出被其他任何一个点支配的所有点。即将区间集合映射到二维平面上的点集之后，找出二维平面中所有极大点的补集即可。

2. 证明 Graham 算法是求凸包问题的一种最优算法

在 n 个点中选取 k 个点按一定顺序构成凸包，可能有 $P(n,k)$ 种结果。由于 k 是未知的，所以共有

$$\sum_{k=3}^{n} P(n,k)$$

种可能性。任取其中一种，选中正确的凸包的概率为

$$p = \frac{1}{\sum_{k=3}^{n} P(n,k)}$$

其信息量为

$$I = -\log p = \log\left(\sum_{k=3}^{n} P(n,k)\right) = \log\left(\sum_{k=3}^{n} \frac{n!}{(n-k)!}\right) = \log\left(n!\sum_{k=3}^{n} \frac{1}{(n-k)!}\right)$$
$$= \log n! + \log\left(\sum_{k=3}^{n} \frac{1}{(n-k)!}\right) = O(n\log n)$$

一次比较操作能提供的信息量为 $\log 2$，所以凸包问题的信息论下界为 $O(n\log n)$。

Graham 算法的时间复杂度为 $O(n\log n)$，与信息论下界同阶，是最优算法。

3. 证明如果存在时间复杂度为 $O(T(n))$ 的两个 $n\times n$ 下三角矩阵的乘法，则存在时间复杂度为 $O(T(n)+n^2)$ 的任意两个 $n\times n$ 矩阵相乘的算法。

令 E 为 n 阶单位阵。设 A,B,C 均为 n 阶方阵，且 $C=AB$。则矩阵

$$Q = \begin{pmatrix} E & O & O & O \\ B & E & O & O \\ O & A & E & O \\ O & O & B & E \end{pmatrix}$$

是 $4n$ 阶的下三角方阵。则有

$$Q^2 = \begin{pmatrix} E & O & O & O \\ 2B & E & O & O \\ AB & 2A & E & O \\ O & BA & 2B & E \end{pmatrix}$$

构造矩阵 Q 需要时间 $(4n)^2$，作矩阵乘法需要时间 $T(4n)$，取出 AB 的值需要时间 n^2。

因此该算法的时间复杂度为 $O(T(4n)+17n^2)$。在假设 $T(cn)=T(n)$ 的条件下，该复杂度即为 $O(T(n)+n^2)$。

4. 如果在序列 $x_1, x_2, \ldots, x_n$ 中，存在某个 i 使 x_i 是序列中的最小者，且序列

$$x_i, x_{i+1}, \ldots, x_n, x_1, \ldots, x_{i-1}$$

是递增的，则称序列 $x_1, x_2, \ldots, x_n$ 是循环序列。设计算法找出循环序列中最小元素的位置。为简单起见，假设该位置是唯一的。证明你的算法是最优的。

遍历整个序列以寻找最小值的时间复杂度为 $O(n)$。

在 n 个数组中随机选取一个数，该数是序列中最小的值的概率为 $p = 1/n$，信息量为 $\log n$。下面给出一种时间复杂度为 $O(\log n)$ 的算法，该算法的时间复杂度达到了问题的信息论下界，是一个最优算法。

若序列 $\{x_i\}$ 是递增的，则 $\forall i, j : i < j \to x_i < x_j$。所以，若 $\exists i < j : x_i > x_j$，则该序列是非增的；又因为该序列是循环递增的，所以序列的极值点一定在 i, j 之间。

伪代码见算法38

该算法类似于二分查找，每次搜索范围减半直到找到最小值，其时间复杂度为 $O(\log n)$。

22 0518

23 0521

算法 27 最小对角线和

输入: $P = \langle p_i(x_i, y_i) \rangle_{1 \le i \le n}$（要求解的多边形的顶点序列）

输出: $S = \{\langle p_i, p_j \rangle\}$（长度和最小的对角线集合）

```
Dist ← {Null}_{n×n}, Q ← {Null}_{n×n}
(minsum, S) ← MinimizeDiagonal(1, n)

procedure Distance(i, j)                          ▷ 获取对角线 p_i − p_j 的长度
    if Dist[i][j] is Null then
        distance ← ‖(x_i, y_i) − (x_j, y_j)‖ if |i − j| > 1 else 0
        Dist[i][j] ← distance, Dist[j][i] ← distance
    end if
    return Dist[i][j]
end procedure

procedure MinimizeDiagonal(m, n)
                          ▷ 求使多边形 ⟨p_m, ..., p_n⟩ 对角线长度和最短的对角线集合及其长度和
    if Q[m][n] is Null then
        if n − m + 1 ≤ 3 then
            Q[m][n] ← (0, Φ)
        else
            min ← +∞, Diag ← Φ
            for i ← m + 1 to n − 1 do
                (sum_1, D_1) ← MinimizeDiagonal(m, i)
                (sum_2, D_2) ← MinimizeDiagonal(i, n)
                d_1 ← Distance(m, i), d_2 ← Distance(i, n)
                sum ← sum_1 + sum_2 + d_1 + d_2
                if sum < min then
                    min ← sum, Diag ← D_1 ∪ D_2
                    Diag ← Diag ∪ {(p_1, p_i)} if d_1 > 0
                    Diag ← Diag ∪ {(p_i, p_n)} if d_2 > 0
                end if
            end for
            Q[m][n] ← (min, Diag)
        end if
    end if
    return Q[m][n]
end procedure
```

算法 28 线段交点存在性判断

输入: $L = \{l_i\}$（线段集合）

输出: 是否存在交点

对所有线段的两个端点以横坐标为键，(x, y, l) 为值建立最小化堆 H ▷ $O(n)$
$Segments \leftarrow$ Vector( )
$result \leftarrow$ Null
for (x, y, l) in H **do** ▷ 按横坐标升序遍历，$O(n \log n$
  if $l \notin Segments$ **then**
    $index \leftarrow$ Push(x, y, l) ▷ 将 l 插入 $Segments$ 中，使在 x 处纵坐标升序，$O(n)$
    return True if Intersect$(l, Segments[index - 1])$ ▷ 判断两线段是否相交，$O(1)$
    return True if Intersect$(l, Segments[index + 1])$
  else
    $index \leftarrow$ Vector.search$(Segments, l)$ ▷ 找到 l 在 $Segments$ 的索引，$O(n)$ 或 $O(\log n)$
    return True if Intersect$(Segments[index - 1], Segments[index + 1])$
    Remove(l) ▷ 从 $Segments$ 中移除 l，$O(n)$
  end if
end for

算法 29 扫描线算法求最近点对

输入: $P = \{(x_i, y_i)\}$（点集合）

输出: p_1, p_2（距离最近的两个点）

对所有点以横坐标为键，建立最小化堆 H_x
$H_y \leftarrow$ RedBlackTree(key $=y$), $Q \leftarrow$ Queue( )
$d \leftarrow +\infty, p_1 \leftarrow$ Null, $p_2 \leftarrow$ Null ▷ d 为扫面线左侧所有点对之间的最小距离
for (x, y) in H_x **do**
  while $(x', y') \leftarrow$ Queue.Top$(Q) \wedge |x' - x| > d$ **do** ▷ 最左侧的点到扫描线的距离超过了 d
    RBTree.Remove$((x', y'))$, Queue.Pop(Q) ▷ 从当前状态中移除该元素
  end while
  for (x', y') in H_y where $|y' - y| < d$ **do** ▷ 最多只有 6 个满足该条件的点
    if $\|(x, y) - (x', y')\| < d$ **then**
      $d \leftarrow \|(x, y) - (x', y')\|$
      $p_1 \leftarrow (x, y), p_2 \leftarrow (x', y')$
    end if
  end for
  RBTree.Insert$((x, y))$, Queue.Push$((x, y))$
end for

算法 30 最大内接矩形

输入: $H[1 \ldots n]$（n 块木板的高度）

输出: $MaxArea$（最大面积）

```
I ← SortedList(H, key = H[i], value = i)  ▷ 存储的是板子在 H 中的索引，只有顺序查找，用链表
MaxArea ← −1
Q ← Queue( )
Queue.Push(Q, (1, n))
while (x, y) ← Queue.Pop(Q) do
    if y > x then
        idx ← List.findFirst(I, [x, y])          ▷ 找到 [x, y] 区间内最低的板子 idx
        area ← H[idx] · (y − x + 1)              ▷ 算面积
        MaxArea ← area if MaxArea < area
        Queue.Push(Q, (x, idx − 1))              ▷ 添加两个字问题
        Queue.Push(Q, (idx + 1, y))
        List.Remove(I, idx)                      ▷ 移除这块板子
    else if y = x then
        area ← H[x]                              ▷ 算面积
        MaxArea ← area if MaxArea < area
        List.Remove(I, x)                        ▷ 移除这块板子
    end if
end while
```

算法 31 矩形交集

输入: 一组矩形 $R = \{\langle y_u, x_l, y_b, x_r\rangle\}$ ▷ 矩形的四条边，按上左下右的顺序

输出: 这些矩形的交集 $\langle Y_u, X_l, Y_b, X_r\rangle$ ▷ 若不存在交集，则这四条边不构成矩形

```
⟨Y_u, X_l, Y_b, X_r⟩ ← Select(R)   ▷ 从集合中任选一个矩形作为当前的交集
for ⟨y_u, x_l, y_b, x_r⟩ in R do
    Y_u ← min(Y_u, y_u), Y_d ← max(Y_d, y_d)
    X_l ← max(X_l, x_l), X_r ← min(X_r, x_r)
    if Y_u ≤ Y_d ∨ X_l ≥ X_r then
        break
    end if
end for
```

算法 32 矩形并集面积

输入: 一组矩形 $R = \{\langle y_u, x_l, y_b, x_r\rangle\}$ ▷ 矩形的四条边，按上左下右的顺序

输出: 这些矩形的覆盖面积

对所有矩形按两条竖直边的横坐标建最小化堆 $events$，有元素 $2n$ 个 ▷ $O(n)$
$state \leftarrow$ RedBlackTree.New
$xlast \leftarrow \min(x), ylast \leftarrow 0$
$space \leftarrow 0$
for e in $events$ **do**
    $x, y_u, y_d \leftarrow$ Extract(e)
    $space \leftarrow space + (x - xlast) \times ylast$
    if x_l in e **then** ▷ 遇到左边，出现一个新的矩形
        RBTree.Insert($state$, key $=y_u$, value $=(y_u,$ up$)$)
        RBTree.Insert($state$, key $=y_b$, value $=(y_b,$ bottom$)$)
    else if x_r in e **then** ▷ 遇到右边，这个矩形消失
        RBTree.Remove($state$, key $=y_u$)
        RBTree.Remove($state$, key $=y_b$)
    end if
    $xlast \leftarrow x, ylast \leftarrow 0$ ▷ 更新当前的有效高度
    $current \leftarrow$ Stack.New
    for $(y, type)$ in $state$ **do**
        if $type =$ up **then**
            Stack.Push($current, y$)
        else if $type =$ bottom **then**
            $ytop \leftarrow$ Stack.Pop($current$)
            if Stack.empty **then**
                $ylast \leftarrow ylast + (y - ytop)$
            end if
        end if
    end for
end for

算法 33 质因子数量

输入: 正整数 n，素数 p

输出: $n!$ 包含质因子 p 的数量 C

```
1: cnt[1...n] ← {0}
2: C ← 0
3: for k ← p up to n do
4:     if k ≡ 0 mod p then
5:         cnt[k] ← cnt[k/p] + 1
6:         C ← C + cnt[k]
7:     end if
8: end for
```

算法 34 Fibonacci 表示

输入: 正整数 n

输出: 若干个 Fibonacci 数 $repr$，其和为 n

```
1: repr ← Vector.New
2: Fib ← Stack.New
3: Stack.Push(Fib, 1)
4: last ← 1
5: repeat
6:     next ← last + Stack.top(Fib)
7:     Stack.Push(Fib, last)
8:     last ← next
9: until last > n
10: repeat
11:     next ← Stack.Pop(Fib)
12:     if n ≥ next then
13:         Vector.PushBack(repr, next)
14:         n ← n − next
15:     end if
16: until n = 0
```

算法 35 矩阵最多乘法次数

输入: $n+1$ 个正整数 $c[0\ldots n]$

输出: 最大乘法次数

```
1: Q[1...n][1...n] ← {0}
2: for j ← 2 to n do
3:     for i ← j − 1 down to 1 do
4:         precompute ← c[i − 1]c[j]
5:         for k ← i to j − 1 do
6:             t ← Q[i][k] + Q[k + 1][j] + c[k] · precompute
7:             Q[i][j] ← t if Q[i][j] < t
8:         end for
9:     end for
10: end for
11: return Q[1][n]
```

算法 36 零点多项式

输入: d 个整数 $n[1\ldots d]$

输出: 多项式

```
1: procedure Q(n[1...d])
2:     if d = 1 then
3:         return (−n[d], 1)                                  ▷ x − n_d
4:     else if d ≡ 0 mod 2 then
5:         P_1 ← Q(n[1...d/2])
6:         P_2 ← Q(n[d/2 + 1...d])
7:         return GivenAlgoB(P_1, P_2)
8:     else
9:         P_1 ← Q(n[1...d − 1])
10:        P_2 ← Q(n[d...d])
11:        return GivenAlgoA(P_1, P_2)
12:    end if
13: end procedure
14: return Q(n)
```

算法 37 正整数分划

```
Q_state = {}


def Q(n, m):
    if(n, m) in Q_state:        # 如果已经求解过该问题，不重复求解
        return Q_state[(n, m)]

    assert (n >= m)

    result = []
    if n == m:
        result = [[m]]
    else:
        r = n - m
        for k in range(1, min(r, m)+1):
            for q in Q(r, k):
                assert(max(q) <= m)
                result.append([m] + q)

    Q_state[(n, m)] = result
    return result


def solve(n):
    result = []
    for i in range(1, n+1):
        result += Q(n, i)

    return result
```

算法 38 循环递增序列的极值点

输入: 序列 $x[1 \ldots n]$

输出: $\operatorname{argmin}_i x_i$

$left \leftarrow 1, right \leftarrow n$
if $x[left] < x[right]$ **then** ▷ 序列是有序的
  return $left$
else
  repeat
    $mid \leftarrow \frac{left+right}{2}$
    if $x[mid] < x[right]$ **then** ▷ $(mid, right)$ 是有序的
      $right \leftarrow mid$
    else ▷ $(left, mid)$ 是有序的
      $left \leftarrow mid$
    end if
  until $right - left \leq 1$
  return $right$
end if