第一章

1. 结合实际, 简述算法设计和分析的目的和意义。
   解：
   算法就是一个非确定的计算过程，是一个求解良说明的计算问题的工具。简单的说，算法描述了一个特定的计算过程来实现输入到输出的关系。
   算法最重要的品质是效率。如果现实社会中，计算机的存储的免费的，计算机的速度是无限快的，那么算法也就没有研究的意义了。但是这两点在现实生活中都不存在，所以研究更有效率的算法是有意义的。
   举一个很简单的例子，对于矩阵连乘问题，不同的计算顺序可能效率之间相差几十甚至几百倍。对于实际运用中，比如计算天气问题，我们需要一种有效的方法，使得能在可以接受的效率下解决这个问题。这就是研究算法的目的和意义。

2. 欧几里德算法利用算术基本定理 (任何一个正整数去除另一个正整数, 必然产生商和余数, 且余数大于等于 0 而严格小于商) 求解正整数 m, n 的最大公因数 $gcd(m, n)$。试完成下列各题:
   (1) 写出欧几里德算法;
   (2) 用循环不变量方法证明欧几里德算法的正确性;
   (3) 最大公因数问题的输入大小为 $\log_2 a$, $\log_2 b$, 分别将算法中进行的求余操作和赋值操作的次数表示成 $\log_2 a$, $\log_2 b$ 的函数, 并由此得出欧几里德算法的渐近复杂性。
   解：
   (1) 欧几里德算法实现

```
GCD(A,B)
1  r = a%b
2  while r ≠ 0
3        do
4           a = b
5           b = r
6           r = a%b
7  return b
```

   (2) 取循环不变量为 $\gcd(a, b) = m$ 是我们所要求的最大公约数
   初始化： 首先在我们迭代之前，$a = a$,$b = b$，所以我们的循环不变式成立。
   保持： 在每次迭代一次之后，$a = b$,$b = a\%b$，现在我们要证明 $\gcd(a, b) = \gcd(b, a\%b)$。
   设 $\gcd(a, b) = m$，可以得到 $a = m\times n_1$,$b = m\times n_2$ 且 $\gcd(n_1, n_2) = 1$ 又可以得到 $a = b\times k+a\%b$，所以 $a\%b = m\times n_1-m\times n_2\times k \Rightarrow a\%b = m\times(n_1 - n_2 \times k)$，所以 $\gcd(b, a\%b) = m \times \gcd(n_2, (n_1 - n_2 \times k)) = m$。
   终止： r 是一个非负整数，每一次循环一定会使得 r 减小，也就是说最多经过 r 次循环后，r 会减小到 0，使得 $r = 0$，然后退出循环。
   (3) 欧几里德算法的渐进复杂性：
   引理 1：设 $m > n$，则 $r = m\%n \leq m/2$。
   证：当 $\lfloor m/n \rfloor = 1$ 时，$r = m - n < n$；

当 $\lfloor m/n \rfloor > 1$ 时，$r = m - \lfloor m/n \rfloor n < n \leq m - n$；
综上所述：$r = m\%n \leq \min(m-n, n)$
当 $n \leq m/2$ 时，$n \leq m - n$；
当 $n > m/2$ 时，$m - n < n$ 且 $m - n < m/2$。
所以可得：$r = m\%n \leq \min(m-n, n) \leq m/2$，引理 1 得证。
对于欧几里德算法，设输入为 a，b 且 $a > b$，连续进行两次循环迭代后。
由引理 1 得：$r_2 = b\%(a\%b) \leq b/2$。
迭代 $2k$ 次之后：$r_{2k} \leq b/2^k$。
最后当 $r_{2k} = 1$ 时算法结束，这时 $k = \log_2 b$，总迭代次数最多为 $2k = 2\log_2 b$。
所以欧几里德算法的时间复杂度为 $O(\log(\min(a, b)))$。

3. 理解下面的插入排序算法, 并完成后面的分析。

插入排序算法 InsertSort

```
1  for i ← 2 to n
2        do
3           key ← A[i];
4           j ← i − 1;
5           while j > 0 且 A[j] > key
6                 do
7                    A[j + 1] ← A[j];
8                    j ← j − 1;
9           A[j + 1] ← key;
```

(a) 证明算法必然停止;
(b) 利用循环不变量方法, 证明算法的正确性。
(c) 分别分析最坏情况下、最好情况下、平均情况下算法执行的比较操作次数和赋值操作次数, 将分析结果表示成 n 的函数。分析平均复杂度是, 假设所有输入服从均匀分布。
解：
(a) 证明：内层 while 循环从 $j = i - 1$ 开始，每次循环 $j \leftarrow j - 1$，最终到 $j = 0$ 内层循环会退出。外层 for 循环终止的条件是 $i > n$。因为每次循环迭代 i 增加 1，那么必然有 $i = n + 1$，因此算法必然停止。

(b) 证明：循环不变式：在每次循环开始前，$A[1, \cdots, i-1]$ 包含来原来的 $A[1, \cdots, i-1]$ 的元素，并且已经排好序。

初始化： 当 $i = 2$ 时，$A[1, \cdots, i-1] = A[1]$，已经排好序了，循环不变式成立。
保持： 已知在循环迭代开始前，$A[1, \cdots, i-1]$ 已排好序，而循环体的目的是将 $A[1, \cdots, i-1]$ 中大于 $A[i]$ 的数向后移，再把 $A[i]$ 插入 $A[1, \cdots, i-1]$ 中，使得 $A[1, \cdots, i]$ 排好序，这时 $A[1, \cdots, i]$ 中的元素由原来 $A[1, \cdots, i-1]$ 中元素组成，并且已经排好序了，循环不变式成立。
终止： 最后 $i = n + 1$，并且 $A[1, \cdots, n]$ 已排好序，而 $A[1, \cdots, n]$ 就是整个数组，循环不变式成立，因此证毕。

(c) 该算法在第 5 行执行比较操作，第 1、3、4、7、8、9 行执行赋值操作。
最坏情况下： 输入数组反向排序时导致最坏情况，必须将每个元素 $A[i]$ 与整个已排序子数组 $A[1:i-1]$ 中的每个元素进行比较。
算法第 5 行执行的比较次数为：$\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$。
算法执行的赋值次数为: 第 1 行为 n，第 3 行为 n-1，第 4 行 n-1，第 7 行为 $\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$，第 8 行为 $\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$，第 9 行为 n-1。
因此算法总的赋值次数为 n^2+3n-3。

最好情况下： 输入数组已经排好序时导致最好的情况，无需移动；算法执行的比较次数为 n-1；赋值次数为 $n+(n-1)+(n-1)+(n-1) = 4n-3$。

平均情况下：
(a) 比较次数：假设插入第 k 个元素时，其插入的位置为 j，$1 \le j \le k$。则需要比较 k-j+1 次，所以其平均比较次数为：

$$\frac{1}{k}\sum_{j=1}^{k}(k-j+1) = \frac{1}{k}[k^2 - \frac{k(k+1)}{2} + k] = \frac{k+1}{2}。$$

插入 n 个元素时，算法总平均比较次数为：

$$\sum_{k=2}^{n}(\frac{k+1}{2}) = \frac{1}{2}[\frac{(n+1)(n+2)}{2}] = \frac{1}{4}n^2 + \frac{3}{4}n + 1。$$

(b) 赋值次数：假设插入第 k 个元素时，其插入位置为 j，$1 \le j \le k$，第 7 行执行的赋值次数为 k-j，因此平均赋值次数为：

$$\frac{1}{k}\sum_{j=1}^{k}(k-j) = \frac{1}{k}[k^2 - \frac{k(k+1)}{2}] = \frac{k-1}{2}。$$

因此，插入 n 个元素时，第 7 行总的赋值次数为：

$$\sum_{k=2}^{n}(\frac{k-1}{2}) = \frac{1}{2}[\frac{n(n-1)}{2}] = \frac{1}{4}n^2 - \frac{1}{4}n。$$

则算法总的赋值次数为：

$$n + (n-1) + (n-1) + 2(\frac{1}{4}n^2 - \frac{1}{4}n) + (n-1) = \frac{1}{2}n^2 + \frac{7}{2}n - 3。$$

4. 结合你曾进行过的程序设计过程或算法设计过程, 说明算法基本技术、基本算法和问题特征分析三者间的关系。
解：
对于一个计算问题的解决，我们可以通过采用算法的基本技术对问题进行分析求解，不同的技术分析方法不同，要分析问题可计算否，能行可计算否，然后根据分析的结果，采用基本的算法用计算机语言进行算法实现，最终解决这个计算问题。

第二章

1. 证明：$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$
   解：
   由已知条件可得：
   存在正常量 c_1，c_2，n_1，使得 $n \geq n_1$，有 $0 \leq c_1 f(n) \leq \Theta(f(n)) \leq c_2 f(n)$。
   存在正常量 c_3，c_4，n_2，使得 $n \geq n_2$，有 $0 \leq c_3 g(n) \leq \Theta(g(n)) \leq c_4 g(n)$。
   取 $c_5 = \min(c_1, c_3)$，$c_6 = \max(c_2, c_4)$，$n_3 = \max(n_1, n_2)$ 由上面的式子可得：
   存在 c_5，c_6，n_3，使得 $n \geq n_3$，有 $0 \leq c_5 f(n) \leq c_1 f(n) \leq \Theta(f(n)) \leq c_2 f(n) \leq c_6 f(n)$。
   存在 c_5，c_6，n_3，使得 $n \geq n_3$，有 $0 \leq c_5 g(n) \leq c_3 g(n) \leq \Theta(g(n)) \leq c_4 g(n) \leq c_6 g(n)$。
   所以，存在 c_5，c_6，n_3，使得 $n \geq n_3$，有 $0 \leq c_5 f(n) + C_5 g(n) \leq \Theta(f(n)) + \Theta(g(n)) \leq c_6 f(n) + c_6 g(n)$。
   综上所述：$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$。

2. 证明：$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
   解：
   当 $f(n) = O(g(n))$ 时，得到存在正常量 c_1，n_1，使得 $n \geq n_1$，有 $0 \leq f(n) \leq c_1 g(n)$。
   取 $c_2 = \frac{1}{c_1}$，则存在正常量 c_2，n_1，使得 $n \geq n_1$，有 $0 \leq c_2 f(n) \leq g(n)$。就可以得到 $g(n) = \Omega(f(n))$。

   当 $g(n) = \Omega(f(n))$ 时，得到存在正常量 c_1，n_1，使得 $n \geq n_1$，有 $0 \leq c_1 f(n) \leq g(n)$。
   取 $c_2 = \frac{1}{c_1}$，则存在正常量 c_2，n_1，使得 $n \geq n_1$，有 $0 \leq f(n) \leq c_2 g(n)$。就可以得到 $f(n) = O(g(n))$。

3. 证明课件中列出的复杂性函数阶的各种性质 (传递性、自反性、反对称性)
   解：
   传递性：$f(n) = \theta(g(n)) \wedge g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$。
   由已知得：存在正常量 c_1，c_2，n_1，使得 $n \geq n_1$，有 $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$。
   存在正常量 c_3，c_4，n_2，使得 $n \geq n_2$，有 $0 \leq c_3 h(n) \leq g(n) \leq c_4 h(n)$。
   我们取 $c_5 = c_1 \times c_3$，$c_6 = c_2 \times c_4$，$n_3 = \max(n_1, n_2)$
   由上面的式子可得：存在 c_5，c_6，n_3，使得 $n \geq n_3$，$0 \leq c_1 \times c_3 h(n) \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \leq c_2 \times c_4 h(n)$
   就可以得到：$f(n) = \theta(h(n))$，传递性得证。

   自反性：$f(n) = \theta(f(n))$。
   容易找到正常量 $c_1 = 1$，$c_2 = 1$，$n_1 = 1$，使得 $n \geq n_1$ 时，有 $0 \leq c_1 f(n) \leq f(n) \leq c_2 f(n)$。
   所以可以得到 $f(n) = \theta(f(n))$。

   反对称性：$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$。

我们先证明 $f(n)=O(g(n))\Rightarrow g(n)=\Omega(f(n))$。
由已知得：存在正常量 c_1，n_1，使得 $n\geq n_1$，有 $0\leq f(n)\leq c_1g(n)$。
我们取 $c_2=\frac{1}{c_1}$，则存在正常量 c_2，n_1，使得 $n\geq n_1$，有 $0\leq c_2f(n)\leq g(n)$。
就可以得到 $g(n)=\Omega(f(n))$。
接下来我们来证明 $g(n)=\Omega(f(n))\Rightarrow f(n)=O(g(n))$。
由已知得：存在正常量 c_1，n_1，使得 $n\geq n_1$，有 $0\leq c_1f(n)\leq g(n)$。
我们取 $c_2=\frac{1}{c_1}$，则存在正常量 c_2，n_1，使得 $n\geq n_1$，有 $0\leq f(n)\leq c_2g(n)$。
就可以得到 $f(n)=O(g(n))$。

4. 证明: 对任意正整数常数 k，$\log^k(n)=o(n)$。
解：
我们先给出原问题的一个等价命题：对任意正整数常数 k，$\lim_{n\to\infty}\frac{\log^k(n)}{n}=0$。
如果该命题成立，则可以得到 $\forall c>0,\exists n_0>0$ 使得 $n>n_0$ 时，$\frac{\log^k(n)}{n}-0<c$ 成立。由这个可以得到 $\log^k(n)<cn$，即 $\log^k(n)=o(n)$。
利用洛比达法则，对分子分母同时求导，$\lim_{n\to\infty}\frac{\log^k(n)}{n}=\lim_{n\to\infty}\frac{k\log^{k-1}(n)}{n}$
连续利用洛比达法则 k 次，可以得到 $\lim_{n\to\infty}\frac{\log^k(n)}{n}=\lim_{n\to\infty}\frac{k!}{n}=0$，命题得证。
所以原命题 $\log^k(n)=o(n)$ 成立。

5. 证明：$\log n!=\Theta(n\log n)$
解：
我们先证明 $\log n!=O(n\log n)$。

$$\log n!=\sum_{i=1}^{n}\log i\leq\sum_{i=1}^{n}\log n=n\log n=O(n\log n)。$$

再来证明 $\log n!=\Omega(n\log n)$。

$$\log n!=\sum_{i=1}^{n}\log i\geq\sum_{i=n/2}^{n}\log n\geq\sum_{i=n/2}^{n}\log\frac{n}{2}=\frac{n}{2}\log\frac{n}{2}=\frac{n}{2}\log n-\frac{n}{2}\log 2。$$

当 $n\geq 4$ 时，$\frac{n}{2}\log n-\frac{n}{2}\log 2>\frac{1}{4}\times n\log n$ 恒成立。
所以 $\log n!=\Omega(n\log n)$ 成立。
综上所述，$\log n!=\Theta(n\log n)$。

6. 对于任意实数 $r>1$，令 $H_r(n)=\frac{1}{1^r}+\frac{1}{2^r}+\frac{1}{3^r}+\cdots+\frac{1}{n^r}$。证明：$H_r(n)=\Theta(1)$。
解：
我们先来证明 $H_r(n)=\Omega(1)$。
显然，$H_r(n)>1$, 所以 $H_r(n)=\Omega(1)$ 成立。
接下来证明 $H_r(n)=O(1)$。
$H_r(n)=\frac{1}{1^r}+\frac{1}{2^r}+\frac{1}{3^r}+\cdots+\frac{1}{n^r}=1+\sum_{i=2}^{n}\frac{1}{i^r}\leq 1+\int_1^n\frac{1}{x^r}dx=1+\frac{1}{r-1}x^{1-r}|_1^n=1+\frac{1}{r-1}(n^{1-r}-1)$
又因为 $r>1\Rightarrow 1-r<0$ 并且 $n\geq 2$，所以 $n^{1-r}-1\leq 0$。

所以可以得到 $H_r(n) \leq 1$，所以 $H_r(n) = O(1)$。
综上所述，$H_r(n) = \Theta(1)$，命题得证。

7. 证明：$\lceil \log(n+1) \rceil = \lfloor \log n \rfloor + 1$ 对任意正整数 n 成立。
解：
设 $\lfloor \log n \rfloor = k \geq 0$。
则 $k \leq \log n < k+1 \Rightarrow 2^k \leq n < 2^{k+1}$。
又因为 n 是一个正整数，则 $2^k \leq n \leq 2^{k+1} - 1$
可以得到 $2^k + 1 \leq n + 1 \leq 2^{k+1} \Rightarrow \log(2^k+1) \leq \log(n+1) \leq \log(2^{k+1})$。
又因为 $\log(2^k+1) > \log 2^k = k$
所以可以得到 $k < \log(n+1) \leq \log(2^{k+1}) = k+1$。
所以 $\lceil \log(n+1) \rceil = k+1 = \lfloor \log n \rfloor + 1$，命题得证。

8. 证明: 对于任意正整数 a,b 均有
(a)$\lfloor \frac{a}{b} \rfloor \leq \frac{a+(b-1)}{b}$
(b)$\lceil \frac{a}{b} \rceil \geq \frac{a-(b-1)}{b}$
解：
(这题觉得是老师出错了，上整和下整写反了)
$\lfloor \frac{a}{b} \rfloor \leq \frac{a}{b} \leq \frac{a}{b} + \frac{b-1}{b} = \frac{a+(b-1)}{b}$，命题 (a) 得证。
$\lceil \frac{a}{b} \rceil \geq \frac{a}{b} \geq \frac{a}{b} - \frac{b-1}{b} = \frac{a-(b-1)}{b}$，命题 (b) 得证。
现在来证一下新的命题：
$(\alpha)\lceil \frac{a}{b} \rceil \leq \frac{a+(b-1)}{b}$
$(\beta)\lfloor \frac{a}{b} \rfloor \geq \frac{a-(b-1)}{b}$
当 $1 \leq a < b$ 时，$\lceil \frac{a}{b} \rceil = 1 \leq 1 + \frac{a-1}{b} = \frac{a+(b-1)}{b}$。
当 $1 \leq b \leq a$ 时，设 $a = b \times k + r$，其中 k,r 为整数并且 $k > 0, 0 \leq r \leq b-1$
所以 $\lceil \frac{a}{b} \rceil = k + \lceil \frac{r}{b} \rceil$。
当 $r = 0$ 时，$\lceil \frac{a}{b} \rceil = k \leq k + \frac{b-1}{b} = \frac{b \times k + r + b - 1}{b} = \frac{a+b-1}{b}$。
当 $r \neq 0$ 时，$\lceil \frac{a}{b} \rceil = k + 1 \leq k + 1 + \frac{r-1}{b} = \frac{b \times k + b + r - 1}{b} = \frac{a+b-1}{b}$。
综上所述，命题 (α) 得证。

当 $1 \leq a < b$ 时，$\lfloor \frac{a}{b} \rfloor = 0 \geq 0 - \frac{(b-1)-a}{b} = \frac{a-(b-1)}{b}$。
当 $1 \leq b \leq a$ 时，设 $a = b \times k + r$，其中 k,r 为整数并且 $k > 0, 0 \leq r \leq b-1$
所以 $\lfloor \frac{a}{b} \rfloor = k + \lfloor \frac{r}{b} \rfloor$。
$\lceil \frac{a}{b} \rceil = k \geq k - \frac{(b-1)-r}{b} = \frac{b \times k - (b-1) + r}{b} = \frac{a-(b-1)}{b}$。
综上所述，命题 (β) 得证。

9. 用迭代法解递归方程 $T(n) = 2T(n/2) + n \log n$，$T(1) = 1$。
解：

$$
\begin{aligned}
T(n) &= 2T(n/2) + n\log n \\
&= 2^2T(n/4) + 2\times\frac{n}{2}\log\frac{n}{2} + n\log n \\
&= 2^3T(n/8) + 4\times\frac{n}{4}\log\frac{n}{4} + 2\times\frac{n}{2}\log\frac{n}{2} + n\log n \\
&= \cdots \\
&= 2^kT(1) + 2^{k-1}\times\frac{n}{2^{k-1}}\log\frac{n}{2^{k-1}} + 2^{k-2}\times\frac{n}{2^{k-2}}\log\frac{n}{2^{k-2}} + \cdots + n\log n \\
&= 2^kT(1) + \sum_{i=0}^{k-1} 2^i\times\frac{n}{2^i}\log\frac{n}{2^i}
\end{aligned}
$$

上式中，k 表示的是迭代的次数，易知 $k=\log_2 n$，代入上式中可得：

$$
\begin{aligned}
T(n) &= 2^{\log_2 n}T(1) + \sum_{i=0}^{\log_2 n-1} 2^i\times\frac{n}{2^i}\log\frac{n}{2^i} \\
&= n\times 1 + \sum_{i=0}^{\log_2 n-1} n\log\frac{n}{2^i} \\
&= n + n\sum_{i=0}^{\log_2 n-1}\log\frac{n}{2^i} \\
&= n + n\log\left(\frac{n^{\log_2 n}}{2^{(\sum_{i=0}^{\log_2 n-1} i)}}\right) \\
&= n + n\log\left(\frac{n^{\log_2 n}}{2^{\frac{(\log_2 n-1)\log_2 n}{2}}}\right) \\
&= n + n\log\left(\frac{n^{\log_2 n}}{n^{\frac{(\log_2 n-1)}{2}}}\right) \\
&= n + n\log\left(n^{\frac{\log_2 n+1}{2}}\right) \\
&= n + n\times\left(\frac{\log_2 n+1}{2}\right)\times\log n \\
&= \frac{1}{2}\log_2 n\times n\times\log n + \frac{1}{2}\times n\log n + n \\
&= O(\log_2 n\times n\times\log n)
\end{aligned}
$$

10. 求解下列递归方程。
(1)$T(n)=5T(n/3)+n$，$T(1)=1$;
(2)$T(n)=4T(n/2)+n$，$T(1)=1$;
(3)$T(n)=2T(n/2)+n^{1/2}$，$T(n)=1$ 对 $n<4$ 成立;
(4)$T(n)=T(\lfloor n/2\rfloor)+T(\lfloor 3n/4\rfloor)+n$，$T(n)=4$ 对 $n<4$ 成立;
(5)$T(n)=2T(n/2)+n^2$，$T(1)=1$;
(6)$T(n)=T(n/2)+n^{1/2}$，$T(n)=2$ 对 $n<4$ 成立;
解：

(1) 等式符合 master 定理，其中 $a = 5, b = 3, f(n) = n$。
并且存在某个常数 $\varepsilon > 0$ 有 $n = O(n^{\log_3 5-\varepsilon})$。
用 master 定理可得：$T(n) = \Theta(n^{\log_3 5})$。

(2) 等式符合 master 定理，其中 $a = 4, b = 2, f(n) = n$。
并且存在某个常数 $\varepsilon > 0$ 有 $n = O(n^{\log_2 4-\varepsilon})$。
用 master 定理可得：$T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$。

(3) 等式符合 master 定理，其中 $a = 2, b = 2, f(n) = n^{1/2}$。
并且存在某个常数 $\varepsilon > 0$ 有 $n^{1/2} = O(n^{\log_2 2-\varepsilon})$。
用 master 定理可得：$T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$。

(4) 用迭代法估计 $T(n)$ 的上界，再用代入法验证。

$$\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lfloor 3n/4 \rfloor) + n \\
&= T(\lfloor n/4 \rfloor) + T(\lfloor 3n/8 \rfloor) + T(\lfloor 3n/8 \rfloor) + T(\lfloor 9n/16 \rfloor) + \frac{5}{4}n + n \\
&= T(\lfloor n/8 \rfloor) + 3T(\lfloor 3n/16 \rfloor) + 3T(\lfloor 9n/32 \rfloor) + T(\lfloor 27n/64 \rfloor) + (\frac{5}{4})^2 n + \frac{5}{4}n + n \\
&= \cdots
\end{aligned}$$

由上式可以看出当进行到第 i 次迭代，会增加一项 $(\frac{5}{4})^i n$。
而对于迭代次数而言，最长的迭代路径为 $n \to \frac{3}{4}n \to (\frac{3}{4})^2 n \to \cdots \to 1$，所以最大的 i 为 $\log_{4/3} n$。
而当进行到第 $\log_2 n$ 次迭代的时候，增加的项就不足 $(\frac{5}{4})^i n$，会逐渐减小。
所以代价和要小于 $n + n\sum_{i=1}^{\log_{4/3} n}(\frac{5}{4})^i = n + 5n\left(n^{\log_{\frac{4}{3}} \frac{5}{4}} - 1\right)$。
并且代价和要大于 $n + n\sum_{i=1}^{\log_2 n}(\frac{5}{4})^i = n + 5n\left(n^{\log_2 \frac{5}{4}} - 1\right)$。
所以我们猜测 $T(n)$ 的上界为 $O(n^2)$。

$$\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lfloor 3n/4 \rfloor) + n \\
&\le c \times \frac{n^2}{4} + c \times \frac{9n^2}{16} + n \\
&\le c \times \frac{13n^2}{16} + n \\
&\le cn^2 + n - c \times \frac{3n^2}{16} \\
&\le cn^2
\end{aligned}$$

只要 $c \times \frac{3n^2}{16} \ge n \Rightarrow c \ge \frac{16}{3n}$ 就能成立，而 n 为正整数，所以 c 只要取大于 $\frac{16}{3}$ 的数就行。
由上面可得：$T(n) = O(n^2)$。

(5) 等式符合 master 定理，其中 $a = 2, b = 2, f(n) = n^2$。
并且存在某个常数 $\varepsilon > 0$ 有 $n^2 = \Omega(n^{\log_2 2+\varepsilon})$。
用 master 定理可得：$T(n) = \Theta(n^2)$。

(6) 等式符合 master 定理，其中 $a = 1, b = 2, f(n) = n^{1/2}$。
并且存在某个常数 $\varepsilon > 0$ 有 $n^{1/2} = \Omega(n^{\log_2 1+\varepsilon})$。
用 master 定理可得：$T(n) = \Theta(n^{1/2})$。

第三章

1. 根据表达式 $a^{2k} = a^k \cdot a^k$ 和 $a^{2k+1} = a^k \cdot a^k \cdot a$ 设计分治 (或递归) 算法求解下列问题, 并分析算法的时间复杂度。
(a) 输入实数 a 和自然数 n, 输出实数 a^n；
(b) 输入实数矩阵 A 和自然数 n, 输出实数矩阵 A^n。
解：
(a) 对于这个问题，采取分治算法将原问题分解为两个规模相同的子问题进行求解，算法设计如下：

```
CALCULATE-POWER(a, n)
1  if n == 1
2     then return a
3  if n%2 == 0
4     then c = CALCULATE-POWER(a, n/2)
5          return c × c
6     else
7          c = CALCULATE-POWER(a, (n−1)/2)
8          return c × c × a
```

算法分析： 我们可以列出算法的递归表达式:

$$T(n) = T(\lfloor n/2 \rfloor) + O(1)$$

由 master 主定理可以很容易得到解：$T(n) = \Theta(\log n)$。

(b) 我们可以采用相同的方法来分析这个问题，只是在分治算法的 merge 步骤的时候会不同。

```
CALCULATE-MATPOWER(A, n)
1  if n == 1
2     then return A
3  if n%2 == 0
4     then c = CALCULATE-POWER(A, n/2)
5          return MATRIX-MULT(c, c)
6     else
7          c = CALCULATE-POWER(A, (n−1)/2)
8          m = MATRIX-MULT(c, c)
9          return MATRIX-MULT(m, A)
```

其中，MATRIX-MULT(c, c) 这个函数表示两个矩阵相乘，由题目可知，问题中的矩阵都是方阵，设输入的矩阵 A 为 $m \times m$ 的矩阵，我们可以列出递归表达式：

$$T(n) = T(\lfloor n/2 \rfloor) + m^3$$

采用迭代法求解上面的递归表达式可得：$T(n) = \Theta(m^3 \times \log n)$。

2. 斐波那契数列满足递归方程 $F(n+2)=F(n+1)+F(n)$，其中 $F(0)=F(1)=1$。

(a) 分别用数学归纳法和第 2 章例 19 的结论, 证明:

$(1)F(n+2)=1+\sum_{i=0}^{n}F(n)$; $(2)F(n+2)>\left(\frac{1+\sqrt{5}}{2}\right)^n$;

(b) 设计算法根据递归方程计算 $F(n)$, 将时间复杂度表达成 n 的函数;

(c) 根据第 2 章例 19 中 $F(n)$ 的解析表达式, 设计算法计算 $F(n)$, 将时间复杂度复杂性表达成 n 的函数, 并指明计算机运行该算法时可能遇到的问题。

(d) 根据表达式 $\begin{pmatrix}1&1\\1&0\end{pmatrix}\begin{pmatrix}F(n+1)&F(n)\\F(n)&F(n-1)\end{pmatrix}=\begin{pmatrix}F(n+2)&F(n+1)\\F(n+1)&F(n)\end{pmatrix}$ 和习题 3.1(b) 的算法, 设计算法计算 $F(n)$, 将时间复杂度复杂性表达成 n 的函数。

(e) 比较 (b),(c),(d) 得到的算法。

解：

(a)

1. 当 $n=0$ 时，$F(0+2)=1+F(0)=2$，满足表达式。

当 $n=1$ 时，$F(1+2)=1+F(0)+F(1)=3$，满足表达式。

假设当 $n<N$ 时，$F(n+2)=1+\sum_{i=0}^{n}F(n)$ 成立；

当 $n=N$ 时，$F(N+2)=F(N+1)+F(N)=1+\sum_{i=0}^{N-1}F(n)+F(N)=1+\sum_{i=0}^{N}F(n)$，表达式成立。

综上所述，$F(n+2)=1+\sum_{i=0}^{n}F(n)$。

2. 当 $n=0$ 时，$F(0+2)=2>0$，满足表达式。

当 $n=1$ 时，$F(1+2)=3>\left(\frac{1+\sqrt{5}}{2}\right)$，满足表达式。

假设当 $n<N$ 时，$F(n+2)>\left(\frac{1+\sqrt{5}}{2}\right)^n$ 成立；

当 $n=N$ 时，$F(N+2)=F(N+1)+F(N)>\left(\frac{1+\sqrt{5}}{2}\right)^{N-1}+\left(\frac{1+\sqrt{5}}{2}\right)^{N-2}=\left(\frac{1+\sqrt{5}}{2}\right)^{N}$，表达式成立。

综上所述，$F(n+2)>\left(\frac{1+\sqrt{5}}{2}\right)^n$。

(b) 我们根据递推方程来求解 $F(n)$，列出时间复杂度的递推方程：

$$T(n)=T(n-1)+T(n-2)+O(1)$$

用代入法可以求解：$T(n)=T(n-1)++T(n-2)+O(1)=T(n-2)+T(n-3)+T(n-3)+T(n-4)+O(2)+O(1)=\cdots=\sum_{i=0}^{n-1}2^i=\Theta(2^n)$

(c) 我们可以得到 $F(n)$ 的表达式：$F(n)=\frac{1}{\sqrt{5}}\cdot\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$

利用公式求解就能将原问题转换为求解 $\left(\frac{1+\sqrt{5}}{2}\right)^n$ 和 $\left(\frac{1-\sqrt{5}}{2}\right)^n$ 两个问题。

我们可以用习题 3.1(a) 的算法来解决这两个问题，由第一题的结论可知，该问题的时间复杂度为：$T(n)=\Theta(2\log n)$。

但该算法在实际执行的时候会出现一些问题，因为 $\sqrt{5}$ 是无理数，而计算机存储位数有限，在计算的过程中，会丢弃掉无理数的一部分，而误差会随着 n 的增加而增加，最后会影响我们对斐波那契数列的求解。

(d) 我们通过对表达式 $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}F(n+1) & F(n)\\ F(n) & F(n-1)\end{pmatrix}=\begin{pmatrix}F(n+2) & F(n+1)\\ F(n+1) & F(n)\end{pmatrix}$ 进行迭代，可以得到一个递推表达式：

$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n\begin{pmatrix}F(2) & F(1)\\ F(1) & F(0)\end{pmatrix}=\begin{pmatrix}F(n+2) & F(n+1)\\ F(n+1) & F(n)\end{pmatrix}$$

我们就将原问题转换为了求解 $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n$，我们采用习题 3.1(b) 的算法，对问题进行求解，由习题 1 的分析可得，时间复杂度为：$T(n)=\Theta(8\log n)$。

(e) 下面对以上的三种方法进行总结：

(b) 中的算法简单，容易实现，但是时间复杂度太高，可以用动态规划的方法，将时间复杂度降到线性时间。

(c) 中的算法时间复杂度最低，但是因为要对无理数进行计算，会产生让人不愉快的误差。

(d) 中的算法时间复杂度和 (c) 中算法的时间复杂度时同阶的，且避免了对无理数的操作，计算很快，我认为时最理想的方法。

3. 给定平面上 n 个点构成的集合 $S=\{p_1,\cdots,p_n\}$。如果存在边平行于坐标轴的矩形仅包含 S 中的两个点 p_i 和 p_j $(1\le i,j\le n)$, 则称 p_i 和 p_j 为“友谊点对”。试设计一个分治算法统计 S 中友谊点对的个数。

解：

使用分治算法来解决这道题。首先将点的坐标离散化，将所有 x 坐标相同的点视为一个整体来进行分治。

Preprocessing:

1. 如果 $n\le 2$ 时，则算法结束。
2. 将所有的位于 S 上的点按照 x-坐标和 y-坐标进行排序。

Divide:

1. 假设 S 中最小的 x-坐标为 l，最大的 x-坐标为 r，我们要处理的问题为 $\textsc{Solve}(l,r)$ 我们令 $mid=\lfloor(l+r)/2\rfloor$。
2. 将点集分成左右两个点集 S_L和S_R，分别递归处理 $n_L=\textsc{Solve}(l,mid)$ 和 $n_R=\textsc{Solve}(mid+1,r)$。

Merge:

1. 对于位于左边区域的点 $a\in S_L$，我们现在需要找出 a 的可视区域 $[y_l,y_r]$，可视区间的意思是指在区域 $[y_l,y_r]$ 中，左边区域不存在点位于这个可视区域中。
2. 对每个位于左边区域的点 $a\in S_L$，只需在右边区域和可视区域的交集区域进行搜索就行了，记录每个点的友谊点数 n_a。
3. 返回求解的总的数目：$n_L+n_R+\sum_{a\in S_L}n_a$。

说明：

(1) 对于 Merge 的第一步，计算方法如下：对于左边区域的点，从右往左处理，每次处理一批 x-坐标相等的点。首先将这些点的 y-坐标全部插入平衡树。插入完毕之后一个一个的检查，如果一个点的 y-坐标仅在平衡树中出现了一次，那么求出它的前驱的 y 值 y_{prev} 和后继的由值 y_{succ}，区间 $[y_{prev}-1,y_{succ}-1]$ 就是我们所求的可视区间，特别要说明的一点，如果在第一次处理的时候得到的点只有一个，则设它的 $y_{prev}=-\infty$ 和 $y_{succ}=\infty$。

(2) 对于 Merge 的第二步，计算方法如下：先对于右边区域中的点，查找在可视区域范围内是否存在点，如果没有点的话，则返回 0，否则对于在可视区域的点，从左往右处理，找到该 x-坐标下位于可视区域中点的最大 y-坐标值 y_{max} 和最小 y-坐标值 y_{min}，更新可视区域为 $[y_{min}-1, y_{max}-1]$，再反复执行上面操作。
下面给出伪代码：
先将所有的位于 S 上的点按照 x-坐标和 y-坐标进行排序。

CALCULATE-FRINDLYPOINT(n, S)

```
1  if n == 2
2     then return 1
3  if n ≤ 1
4     then return 0
5  l ← min(x)，r ← max(y)
6  mid ← ⌊(l + r)/2⌋
7  n_L ← CALCULATE-FRINDLYPOINT(l, mid, S_L)
8  n_R ← CALCULATE-FRINDLYPOINT(mid + 1, r, S_R)
9  [y_prev, y_succ] ← CALCULATE-VIEWAREA(S_L)
10 for a ∈ S_L
11       do prev ← y_prev[a], succ ← y_succ[a], i ← 1, n_a ← 0
12          while FIND-NUM(prev, succ) ≠ 0
13                do x_now ← FIND-RIGHTX(i)
14                   [y_min, y_max] ← FIND-MAXMIN(x_now)
15                   [prev, succ] ← [y_min, y_max]
16                   if prev ≠ y_min
17                      then n_a ← n_a + 1
18                   if succ ≠ y_max
19                      then n_a ← n_a + 1
20                   i ← i + 1
21 return n_L + n_R + Σ_{a∈S_L} n_a
```

最后，我们来分析算法的时间复杂性：
(1) 在 Preprocessing 阶段，排序所需要的时间复杂性为 $O(n\log n)$。
(2) 在 Divide 阶段，分割集合所需要的时间复杂性为 $O(n)$。
(3) 在 Merge 阶段的第一步，维护和搜索平衡树的时间为 $n\log n$。
(4) 在 Merge 阶段的第二步，看似时二层循环，但是因为第二层循环每进行一次必会缩小可视区域，而可视区域是常数数量级，所以第二层循环能在常数级别的时间内结束，而在有序的数组中进行查找操作的时间复杂度为 $\log n$，所以总的时间复杂度为 $n\log n$。
可以给出分治算法的迭代公式：

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

我们可以很容易解出：$T(n) = O(n\log^2 n)$。

4. 给定平面上 n 个点构成的集合 S, 试设计一个分治算法输出 S 的三个点, 使得以这三个点为顶点的三角形的周长达到最小值。(提示: 模仿最邻近点的分

治过程)。
解：
我们采用分治算法的思想来求解：
Preprocessing:
1. 如果 $n \leq 2$，则算法结束。
2. 把 S 中的点分别按 x-坐标值和 y-坐标值排序。
Divide:
1. 计算 S 中各点 x-坐标的中位数 m。
2. 用垂线 $L : x = m$ 把 S 划分为两个大小相等的子集 S_L 和 S_R，集合 S_L 中的点在直线 L 左边集合 S_R 中的点在直线 L 右边。
3. 递归的在子集 S_L 和 S_R 上找出能构成三角形并且周长最小的三个点：$(p_1, p_2, p_3) \in S_L$ $(q_1, q_2, q_3) \in S_R$。
4. 记 $d = min\{Dis(p_1, p_2, p_3), Dis(q_1, q_2, q_3)\}$。
Merge:
1. 在临界区查找距离小于 d 的三个点，并且这三个点不在同一个子集中。
2. 如果找到，则新找到的三个点为所求的点，否则，就是 (p_1, p_2, p_3) 和 (q_1, q_2, q_3) 中距离最小者为周长最小的三角形顶点。
说明：
(1) 对于上述问题的描述，判断三点 $(p_1, p_2, p_3) \in S$ 是否能构成三角形，也就是判断这三点是否在一条直线上，即判断三点坐标满不满足条件 $\left|\frac{x_{p_1}-x_{p_2}}{y_{p_1}-y_{p_2}}\right| = \left|\frac{x_{p_2}-x_{p_3}}{y_{p_2}-y_{p_3}}\right|$，如果不满足上式的话，就说明可以构成三角形。
(2) 算法的关键在于 Merge 中第一步的求解，我们将搜索的区域限定在垂线 $L : x = m$ 两侧，垂线 $L_1 : x = m - \frac{d}{2}$ 和垂线 $L_2 : x = m + \frac{d}{2}$ 之间。
因为当三角形中一条边长度大于 $\frac{d}{2}$ 时，由三角形定义，两边之和大于第三边可以得到，三角形的周长将大于 d。
对于 S_L 集合中位于搜索区域的点，搜寻另外两点都位于 S_R 中，使得所构成三角形周长小于 d，同理，对于 S_R 集合中位于搜索区域的点，搜寻另外两点都位于 S_L 中，使得所构成三角形周长小于 d。
而对于位于搜索区域的点 $P \in S_L$，我们只需要在它的 P-右邻域中搜寻两点，它的右邻域为 $d \times \frac{d}{2}$ 大小的矩形。
我们将证明，这个邻域中最多存在 16 个点。将 $d \times \frac{d}{2}$ 的矩形分成 8 个 $\frac{d}{4} \times \frac{d}{4}$ 的矩形，每个矩形中最多有两个点，如果存在 3 个点在 $\frac{d}{4} \times \frac{d}{4}$ 矩形中，则三个点构成的三角形的周长将小于矩形的周长 $L < 4 \times \frac{d}{4} d = d$，与条件矛盾。
给出算法的伪代码:
先将 S 中的点分别按 x-坐标值和 y-坐标值排序。

```
CALCULATE-MINPERIMETER(S, n)
1  if n ≤ 2
2     then return 0
3  计算 S 中各点 x-坐标的中位数 m。
4  用垂线 L : x = m 把 S 划分为两个大小相等的子集 S_L 和 S_R。
5  a ← CALCULATE-MINPERIMETER(S_L, n/2)
6  b ← CALCULATE-MINPERIMETER(S_R, n/2)
7  d = min(a, b)
8  return MERGE(S_L, S_R, d)
```

最后分析算法的时间复杂性：

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

用 Master 定理求解 $T(n) = O(n \log n)$。

5. 证明求解凸包问题的蛮力算法的正确性。
解：先给出求解凸包问题的蛮力算法的伪代码：

BruteForceCH(Q)

```
1  for ∀A, B, C, D ∈ Q
2        do if 其中一点位于其它三点构成的三角形内
3              then 从 Q 中删除该点
4  A ← Q 中横坐标最大的点
5  B ← Q 中横坐标最小的点
6  S_L ← {P|P ∈ Q且g(A, B, P) < 0}
7  S_U ← {P|P ∈ Q且g(A, B, P) > 0}
8  排序 S_L, S_U
9  输出 A, S_L, B, 逆序S_U
```

蛮力算法的思想就是对点集的任意 4 个点，判断是否有点位于其它三个构成的三角形内，有的话就删除该点，外层循环是对点集所有的点的 4 层循环遍历，所以蛮力算法的时间复杂性为 $O(n^4)$，我们将采用循环不变量来证明算法的正确性。
设置循环不变量为：当前点集的凸包和原始点集的凸包是一样的。
初始化： 在循环开始前点集 Q 中间的点和原始点集中的点一样，所以凸包也时一样的，循环不变式成立。
保持： 经过一次循环迭代之后，如果不存在其中一点位于其它三点构成的三角形内，则点集跟循环前的点集一样，所以凸包也是一样的，当存在其中一点位于其它三点构成的三角形内，则删除掉该点。
现在我们来证明引理 1：给定平面点集 S，如果 $P, P_i, P_j, P_k \in S$ 是四个不同的点，且 P 位于三角形 $\Delta P_i P_j P_k$ 的内部或边界上，则 P 不是 S 的凸包顶点。
我们再给出引理 2：给定平面点集 S，如果 P 点是点集凸包上的顶点，则其余的点都在 P 与其在凸包上相邻的顶点 P_l 构成的直线 PP_l 的一侧。
现在来证明引理 2 的正确性：
因为凸包会包含点集上所有的点，所以我们只需要证明凸包的顶点点都位于直线的一侧。
如果存在凸包上的顶点 P_1, P_2 分别位于直线 PP_l 的两侧，则 ΔPP_lP_1 是凸包内的一部分，并且形 ΔPP_lP_2 也是凸包内的一部分，而两个三角形共用一条边 PP_l，因此，构成的四边形 $PP_1P_lP_2$ 位于凸包的内部，所以直线 PP_l 位于凸包内部，与假设矛盾，所以引理 2 成立。
再来证明引理 1：
假设 P 是凸包的顶点，则它与其在凸包上相邻的顶点会构成一条直线，而 $P_i, P_j, P_k \in S$，所以 P_i, P_j, P_k 将位于直线的一侧。
当 P 位于三角形 $\Delta P_i P_j P_k$ 的内部时，不可能存在一条直线通过 P 使得

P_i, P_j, P_k 位于直线的一侧，与引理 2 矛盾。
P 位于三角形 $\Delta P_i P_j P_k$ 的边界上，不失一般性，设 P 位于边 $P_i P_j$ 上，则通过 P 的直线并且满足 P_i, P_j, P_k 位于直线一侧，该直线必经过 $P_i P_j$，所以 P_i，P_j 也为凸包的顶点，并且 P 不能称之为凸包的顶点，矛盾。
综上所述，定理 1 成立，循环不变式成立。
终止：因为点集中点的个数时有限的，最终循环会停止，只留下部分点，这些点就构成了凸包的顶点，循环不变式成立。

6. (选做) 给定平面上 n 个白点和 n 个黑点, 试设计一个分治算法将每个白点与一个黑点相连, 使得所有连线互不相交, 分析算法的时间复杂度。(提示: 划分时类似于 GrahamScan 算法考虑极角, 确保子问题比较均匀)
解：
下面给出一个分治算法：
Preprocessing:
1. 如果 $n == 1$，说明平面上只有一个白点和一个黑点，就将白点和黑点连接就可以满足题意。
2. 将所有的点，根据 y-坐标进行排序。
3. 对于所有的点，考虑 y-坐标最小的点 (最低点)。如果存在多个这样的点，就考虑 x-坐标最小的点 (最左边的点)，以这点为原点，对剩余的点建立极坐标，且所有的点的极角都在范围 $[\ 0, \pi\ )$ 内。
4. 这一点可能是白点，也可能是黑点，这个不影响我们接下来的说明，不妨设该点为白点，按照极角从小到大的顺序排序。
Divide:
1. 按照极角排序后的顺序搜索，搜索直到得到黑点和白点的数目一样多为止。
2. 根据搜索的结果将平面上的点集的集合分成两个集合，递归求解就行。
Merge:
1. 将分解的方案合并，就能完成对问题的求解。
说明：
1. 对于 Divide 阶段的第一步，我们已经假设原点为白点，如果搜索的第一点为黑点，则我们就可以让原点和该点配对，剩下的点继续搜索。如果搜索的第一点为白点，则必会存在搜索到 $2k$ 个点时，包括 k 个白点和 k 个黑点。因为搜索到第一个点时，白点数量比黑点数量多一个 (第一个点为白点)，而搜索到最后一个点时黑点的数量比白点数量多一个 (最后的时候是 n 个黑点，n-1 个白点)，而每次搜索黑点数目和白点数目最多只能加 1，所以必有一个时刻，黑点和白点的数目时一样的。
2. 分成两个集合之后在递归的重复上面的划分过程进行求解。
下面给出伪代码：
先将白点和黑点按照 y-坐标值排序。

```
CALCULATE-BLACKWHITE(n, B, W)
 1  if n == 1
 2     then return (B, W)
 3  [s_0, label_0] ← FIND-LOWPOINT(B, W)
 4  [S_polar, Label] ← CACULATE-POLAR(B, W, s_0)
 5  [S_polar, Label] ← SORT(S_polar, Label)
 6  num_b ← 0, num_w ← 0, B_1 ← Null, B_1 ← Null
 7  if Label[1] ≠ label_0
 8     then B_1 ← B − {s_0}, W_1 ← W − {s_1}
 9          num_b ← 1, num_w ← 1
10     else
11          while num_b == num_w&&num_b ≠ 0
12               do s_i ∈ S_polar
13                  if Label[i] == label_0
14                     then num_b ← num_b + 1
15                          B_1 ← B_1 ⋃ s_i
16                     else  num_w ← num_w + 1
17                          W_1 ← W_1 ⋃ s_i
18  CALCULATE-BLACKWHITE(num_b, B_1, W_1)
19  CALCULATE-BLACKWHITE(n − num_b, B − B_1, w − W_1)
```

最后进行时间复杂度分析：
该算法主要是划分并不稳定，但是平均时间复杂度为 $O(n\log^2 n)$。

7. 给定凸多边形 $p_1, p_2, \cdots, p_n$ (边界逆时针顺序) 和 n 个点 $q_1, q_2, \cdots, q_n$, 试设计一个分治算法计算 $q_1, q_2, \cdots, q_n$ 中位于凸多边形 $p_1, p_2, \cdots, p_n$ 内部的点的个数, 使其时间复杂度是 n^2 的严格低阶函数。

解：

我们设计一个分治算法来解决这个问题：

Preprocessing:

1. 找出 $p_1, p_2, \cdots, p_n$ 中最大的 x-坐标值 x_{max} 最小的 x-坐标值 x_{min}，最大的 y-坐标值 y_{max} 最小的 y-坐标值 y_{min}。

2. 在 $q_1, q_2, \cdots, q_n$ 中去掉满足 $x < x_{min}$，$x > x_{max}$，$y < y_{min}$，$y > y_{min}$ 条件的点，对剩下的点进行 x-坐标值排序。

3. 当 $n == 1$ 时，选择 $p_1, p_2, \cdots, p_n$ 中 x 坐标最小的点 p_0，如有多个，则选择其中 y 坐标最小的那一个点，从这点的左侧点开始，计算其它点到这一点的极角值，并且按照原来逆时针的顺序排列，将这一点与 S 中的点也连接起来算极角值，将该极角值插入到上面的序列中，得到前后两个点 p_l, p_r ，判断该点是否在三角形 $\Delta p_0 p_l p_r$ 中，如果在返回 1，否则返回 0。

Divide:

1. 计算 S 中各点 x-坐标的中位数 x_m 和 Y-坐标的中位数 y_m。

2. 用垂线 $L_1 : x = x_m$ 和水平线 $L_2 : y = y_m$ 把 S 划分为四个大小相等的子集 $S_{LT}, S_{RT}, S_{LB}, S_{RB}$。

3. 递归的在子集 $S_{LT}, S_{RT}, S_{LB}, S_{RB}$ 中寻找满足条件的点的个数 $n_{LT}, n_{RT}, n_{LB}, n_{RB}$。

Merge:

1. 返回 $n_{LT} + n_{RT} + n_{LB} + n_{RB}$。

接下来给出伪代码：

```
CALCULATE-INNERNUM(P, Q, n)
 1  x_max ← FIND-MAXX(P)
 2  x_min ← FIND-MINX(P)
 3  y_max ← FIND-MAXY(P)
 4  y_min ← FIND-MINY(P)
 5  WIPEOUT(Q, x_max, x_min, y_max, y_min)
 6  if n == 1
 7     then [p_0, P_polar, q_polar] ← POLAR-ANGLE(P, Q)
 8          [p_l, p_r] ← FIND-POINT(P_polar, q_polar)
 9          if IS-INTRIANGLE(Q, p_0, p_l, p_r)
10             then return 1
11             else  return 0
12  计算 S 中各点 x-坐标的中位数 x_m 和 Y-坐标的中位数 y_m
13  用垂线 L_1 : x = x_m 和水平线 L_2 : y = y_m 把 S 划分为四个大小相等的子集
14  a ← CALCULATE-INNERNUM(S_LT, Q, n/4)
15  b ← CALCULATE-INNERNUM(S_RT, Q, n/4)
16  c ← CALCULATE-INNERNUM(S_LB, Q, n/4)
17  d ← CALCULATE-INNERNUM(S_RB, Q, n/4)
18  return a + b + c + d
```

最后给出算法的时间复杂度：$O(n\log n)$。

8. 输入含有 n 个顶点的加权树 T 和实数 τ , 树 T 中每条边的权值均非负, 树中顶点 x, y 的距离 $dis(x, y)$ 定义为从 x 到 y 的路径上各边权值之和。试设计一个分治算法输出满足 $dis(x, y) \leq \tau$ 的顶点对的个数。

解：

我们将距离分为两部分计算，第一部分是两个节点在同一棵子树中，并且距离小于 τ，第二部分是两个节点在不同的子树中，并且距离小于 τ。设计一个分治算法：

Preprocessing:

1. 如果 $n \leq 2$ 时，则算法结束。

Divide:

1. 计算得到加权树中的一点 t_m，使得删除这个点后最大的子树是最小的。
2. 设这点的子节点个数为 k，删除这点，把 T 分成 k 个子树。
3. 递归在子树上找出满足条件的点对数目 $S_1, S_2, \cdots, S_k$。

Merge:

1. 求出每个节点到根节点 t_m 的距离 $A = \{a_1, a_2, \cdots, a_{n-1}\}$。
2. 将集合 A 中元素排序。
3. 求解满足条件 $a_i + a_j \leq \tau (1 \leq i < j \leq n-1)$ 的数目 n_a。
4. 求解满足条件 $a_i + a_j \leq \tau (i, j \in T_l 且 1 \leq l \leq k)$ 的数目 $n_1, n_2, \cdots, n_k$。
5. 返回总的数目：$\sum_{i=1}^{k} S_k + \left(n_a - \sum_{j=1}^{k} n_j\right)$。

说明：

(1) 在 Divide 阶段中，找到 t_m 的算法先可以采用动态规划的方法对树进行深度优先搜索，得到每个节点分割之后，最大子树的节点数，再通过在 n 个数中寻找最大的数的方法，得到 t_m。

(2) 在 Merge 阶段中，第一步采用深度遍历算法，得到距离；因为在在 Divide 阶段中已经对点集进行了划分，所以第三步和第四步所求解的问题时相同的，我们转换为求解满足条件 $a_i + a_j \leq \tau (1 \leq i < j \leq n)$ 的数目 k。而对于这类问题，我们可以先对距离序列排序，然后找头尾两个数，如果符合情况，算中间的个数，然后从头的下一个开始算，如果不符合情况说明太大，要从尾的前一个开始计算，直到头等于尾。
下面给出伪代码：

CALCULATE-TREE(n, T, τ)

if $n == 2$
  then return $dis(x, y) < \tau ? 1 : 0$
if $n \leq 1$
  then return 0
$(t_m, k) \leftarrow$ CALCULATE-TREEHEART(T)
for $i \leftarrow 1$ **to** k
  then CALCULATE-TREE(n_i, T_i, τ)
求出每个节点到根节点 t_m 的距离 $A = \{a_1, a_2, \cdots, a_{n-1}\}$
$A \leftarrow$ SORT(A)
$n_a \leftarrow$ CALCULATE-NUM(A, τ)
for $j \leftarrow 1$ **to** k
  then $n_j \leftarrow$ CALCULATE-NUM(A_j, τ)
return $\sum_{i=1}^{k} S_k + \left(n_a - \sum_{j=1}^{k} n_j\right)$

接下来分析算法的时间复杂度：
(1) 在 Divide 阶段中，寻找 t_m 的时间复杂度为 $O(n)$
(2) 在 Merge 阶段中，第一步的时间复杂度为 $O(n)$；第二步的时间复杂度为 $O(n \log n)$；第三步和第四步的时间复杂度均为 $O(n)$。
综上所述，每一次总的时间复杂度为 $O(n \log n)$，而因为我们每次迭代寻找的是最优的 t_m，总的次数为 $O(\log n)$，所以算法总的时间复杂度为 $O(n \log^2 n)$。

9. 设 $X[0 : n-1]$ 和 $Y[0 : n-1]$ 为两个数组, 每个数组中的 n 个均已经排好序, 试设计一个 $O(\log n)$ 的算法, 找出 X 和 Y 中 $2n$ 个数的中位数, 并进行复杂性分析。
解：
设计分治算法来实现：
Preprocessing:
1. 将 $X[0 : n-1]$ 和 $Y[0 : n-1]$ 两个数组按从小到大的顺序排序，记为 $A[0 : n-1]$ 和 $B[0 : n-1]$。
2. 如果 $n \leq 2$，问题很容易解决。
Divide:
1. 找到两个数组的中位数 mid_a, mid_b。
2. 将两个数组分别平均分成两部分 $A[0 : mid_a], A[mid_a : n-1]$ 和 $B[0 : mid_b], B[mid_b : n-1]$。
3. 将 $A[0 : mid_a]$ 和 $B[mid_b : n-1]$ 分为一组构成新问题的子问题，求出该子问题的中位数 ans_1；将 $A[mid_a : n-1]$ 和 $B[0 : mid_a]$ 分为一组构成新问题的另一个子问题，求出该子问题的中位数 ans_2。

Merge:
1. 比较两个中位数的大小，如果 $mid_a < mid_b$，则 ans_2 为原问题的解，如果 $mid_a > mid_b$，则 ans_1 为原问题的解，如果 $mid_a = mid_b$，则 mid_a 和 mid_b 为原问题的解。

说明：
(1) 对于 Divide 阶段的第一步，寻找两个数组的中位数，为了保证得到的子问题中 A 数组和 B 数组中数的个数一样，如果 n 为偶数，则对 A 数组取上中位数 $\lfloor n/2 \rfloor$，对 B 数组取下中位数 $\lceil n/2 \rceil$，如果 n 为奇数，则就取中位数 $n/2$。
(2) 对于 Merge 阶段的第一步，如果 $mid_a < mid_b$，则说明对于数组 A 中比 mid_a 小的数在数组 A 中小于 $n/2$ 个数，在数组 B 中小于 $n/2$ 个数，所以这些数肯定不是中位数。同理，对于数组 B 中比 mid_b 大的数也不是中位数；如果 $mid_a > mid_b$，则说明对于数组 A 中比 mid_a 大的数在数组 A 中大于 $n/2$ 个数，在数组 B 中大于 $n/2$ 个数，所以这些数肯定不是中位数。同理，对于数组 B 中比 mid_b 小的数也不是中位数；如果 $mid_a = mid_b$，则说明 mid_a 比 A 数组中 $n/2$ 个数大，比 B 数组中 $n/2$ 个数大，所以 mid_a 和 mid_b 为中位数。

FIND-MIDNUM$(A, l_1, r_1, B, l_2, r_2)$

if $(r_1 - l_1 + 1)\%2 = 0$
  then $mid_1 \leftarrow (l_1 + r_1)/2 + 1$; $//A$ 取上中位数
    $mid_2 \leftarrow (l_2 + r_2)/2$; $//B$ 取下中位数
  else
    $mid1 \leftarrow (l_1 + r_1)/2$;
    $mid2 \leftarrow (l_2 + r_2)/2$;
if $l_1 = r_1 \,\&\&\, l_2 = r_2$ //n=1 的情况
  then return $(double)(A[l_1] + B[l_2])/2$;
if $r_1 - l_1 = 1 \,\&\&\, r_2 - l_2 = 1$ //n=2 的情况
  then return $(\max\{A[r_1], B[r_1]\} + \min\{A[r_2], B[r_2]\})/2$;
if $A[mid_1] = B[mid_2]$
  then return $A[mid_1]$;
elseif $A[mid_1] > B[mid_2]$
  then return FIND-MIDNUM$(A, l_1, mid_1, B, mid_2, r_2)$;
else return FIND-MIDNUM$(A, mid_1, r_1, B, l_2, mid_2)$;

算法复杂性分析:
Preprocessing:$O(1)$；
Divide 阶段：寻找两个有序数组的中位数，并进行比较，需要 $O(1)$ 时间；
Conquer 阶段：需要 $T(n/2)$ 时间；
Merge 阶段：需要 $O(1)$ 时间。
递归方程：

$$T(n) = \begin{cases} O(1), & n = 1, 2 \\ T(n/2) + O(1), & n \geq 3 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(\log(n))$。

10. 设 $A[1:n]$ 是由不同实数组成的数组, 如果 $i<j$ 且 $A[i]>A[j]$, 则称实数对 $(A[i],A[j])$ 是该数组的一个反序。如, 若 $A=[3,5,2,4]$, 则该数组存在 3 个反序 (3,2)、(5,2) 和 (5,4)。反序的个数可以用来衡量一个数组的无序程度。设计一个分治算法 (要求时间复杂度严格低于 n^2), 计算给定数组的反序个数。

解：

在这个问题上采用类似于归并排序的算法, 只是每次在 merge 的时候每次组间比较都根据比较结果对反序对计数量进行加一操作即可。

Preprocessing: 如果数组 A 中仅有一个数，算法结束；

Divide:

(1) 用数组 A 的中间位置的数将 A 分成两个长度相当的数组 $A_1=A[1:\lfloor n/2\rfloor]$ 和 $A_2=A[\lfloor n/2\rfloor+1:n]$；

(2) 递归地计算 A_1 和 A_2 各自的反序数 a_1 和 a_2；

Merge: 将数组 A_1 和 A_2 进行归并排序，同时得到两个数组之间的反序数 m，则 a_1+a_2+m 为 A 的反序数。

FIND-INVERSIONCOUPLES(A)

```
1  n ← length(A);
2  if n = 1
3     then return 0;
   //递归地求子问题的反序数;
4  a1 ←FIND-INVERSIONCOUPLES(A[1 : ⌊n/2⌋])
5  a2 ←FIND-INVERSIONCOUPLES(A[⌊n/2⌋ + 1 : n]);
   //求两组元素之间的反序数;
6  num ← 0;
7  i1 ← 1; i2 ← ⌊n/2⌋ + 1;
8  k ← 1;
9  while i1 ≤ ⌊n/2⌋ && i2 ≤ n
10       do
11          if A[i1] > A[i2]
12             then num++;
13                  new_A[k++] ← A[i2++];
14             else  new_A[k++] ← A[i1++];
15 while i1 ≤ ⌊n/2⌋
16       do new_A[k++] = A[i++];
17 while i2 ≤ ⌊n/2⌋
18       do new_A[k++] = A[i++];
19 return a1 + a2 + m;
```

算法复杂性分析: 该算法和归并排序过程完全类似，很容易给出递归方程：

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

用 Master 定理求解 $T(n)$, 得 $T(n)=O(n\log(n))$。

11. 给定一个由 n 个实数构成的集合 S 和另一个实数 x, 判断 S 中是否有两个元素的和为 x。试设计一个分治算法求解上述问题, 并分析算法的时间复杂度。

解：
Preprocessing:
1. 将集合 S 按从小到大的顺序排序。
2. 将 S 中去掉相同的元素，如果存在多个等于 $x/2$ 的元素，则直接返回 True。
Divide:
1. 得到一个新的集合 $S_1 = \{y|y = x - a; a \in S\}$，并且容易得知，$S_1$ 是从大到小排列的。
2. 将 S_1 从小到大排列。
Merge:
1. 通过将 S 和 S_1 两个有序的数列合并，如果存在相同的数，则返回 True，否则返回 False。
下面给出伪代码：

IsEqual(S, x)
```
1  S ← Sort(S)
2  if Find(x/2) ≥ 2
3      then return True
4  S ← WarpOut(S)
5  S1 ← x − S
6  S1 ← Sort(S1)
7  i ← 0, j ← 0
8  n ← length(S)
9  while i ≤ n && j ≤ n
10       do
11           if S[i] > S1[j]
12              then j++
13           elseif S[i] < S1[j]
14              then i++
15           else return True
16 return False
```

时间复杂度为 $O(n \log n)$。

12. 设单调递增有序数组 A 中的元素被循环右移了 k 个位置, 如 $< 35; 42; 5; 15; 27; 29 >$ 被循环右移两个位置 $(k = 2)$ 得到 $< 27; 29; 35; 42; 5; 15 >$。
(1). 假设 k 已知, 给出一个时间复杂度为 $O(1)$ 的算法找出 A 中的最大元素。
(2). 假设 k 未知, 设计一个时间复杂度为 $O(logn)$ 的算法找出 A 中的最大元素。
解：
考虑数组元素严格单调递增的情况。
(1) 算法伪代码如下：

FIND-MAX(A, k)

```
1  n ← A.length;
2  index ← k%n;
3  if index == 0
4      then return A[n − 1];
5  else return A[index − 1];
```

算法的时间复杂度显然为 $O(1)$.
(2)
数组 A 在初始阶段是单调递增的，当 $k \geq n$ 时，与 $k \leftarrow k\%n$ 的情况一致，我们在这里只考虑 $k < n$ 的情况。
由第一问可知，左移 $k > 0$ 位之后，最大数在第 $k-1$ 位，这样就将数组 A 分为两部分，其中 $A[0 : K-1]$ 和 $A[k : n-1]$ 都是单调递增的并且 $A[0 : K-1]$ 中的元素都要比 $A[k : n-1]$ 中的元素大。这就给我们设计分治算法提供来办法。
算法伪代码如下

FIND-MAX(A)

```
1  n ← A.length;
2  if n ≤ 3
3      then return max(A)
4  mid ← n/2 − 1;
5  if A[0] < A[mid] && A[mid] < A[n − 1]
6      then return A[n − 1];
7  elseif A[0] > A[mid]
8      then return FIND-MAX(A[0 : mid]);
9  else return FIND-MAX(A[mid : n − 1]);
```

递归方程：

$$T(n) \begin{cases} = O(1), & n = 2, 3 \\ \leq T(n/2) + O(1), & n \geq 4 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(\log n)$.

13. 给定非负数组 $A[1 : n]$，$B[1 : m]$ 和整数 k。试设计一个算法计算集合 $\{A[i] \cdot B[j] | 1 \leq i \leq n, 1 \leq j \leq m\}$ 中的第 k 小的元素。
解：
先将数组 A 和数组 B 排好序。

```
FIND-KMIN(A, B, k)
 1  m ← A.Length, n ← B.length
 2  mid_a ← A[m/2], mid_b ← B[n/2]
 3  mid ← mid_a · mid_b
 4  cnt ← JUDGE(mid, A, B)
 5  if cnt < k
 6     then FIND-KMIN(A[mid_a : m − 1], B[mid_b : n − 1], k)
 7          FIND-KMIN(A[0 : mid_a], B[mid_b : n − 1], k)
 8          FIND-KMIN(A[mid_a : m − 1], B[0 : mid_b], k)
 9  elseif cnt > k
10     then FIND-KMIN(A[0 : mid_a], B[0 : mid_b], k)
11          FIND-KMIN(A[0 : mid_a], B[mid_b : n − 1], k)
12          FIND-KMIN(A[mid_a : m − 1], B[0 : mid_b], k)
13  else return mid
```

```
JUDGE(mid, A, B)
 1  m ← A.Length, n ← B.length
 2  num ← m/2 × n/2
 3  for i ← 0 to m/2
 4       do
 5          now ← mid/A[i]
 6          cnt ← BINARY-SEARCH(B[n/2 : n − 1], now)
 7          num ← num + cnt
 8  for j ← 0 to n/2
 9       do
10          now ← mid/B[j]
11          cnt ← BINARY-SEARCH(A[m/2 : m − 1], now)
12          num ← num + cnt
13  return num
```

上面出现的函数 BINARY-SEARCH(A, now) 表示的不是二分查找的算法，而是查找 now 在 A 中插入的位置，返回查找的位置前面有多少个数。
递归方程：

$$T(n) = \begin{cases} O(1), & m, n = 1 \\ 3T(n/2) + O(n \log n), & m, n \geq 2 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(n^{\log_2 3})$.

14. 设 M 是一个 $m * n$ 的矩阵，其中每行的元素从左到右单增有序，每列的元素从上到下单增有序。给出一个分治算法计算出给定元素 x 在 M 中的位置或者表明 x 不在 M 中。分析算法的时间复杂性。
解：

```
FIND-X(M, x, i, j)
    /* 返回给定元素 x 在 M 中的位置 (i, j),(−1, −1) 表明 x 不在 M 中。*/
 1  m ← M.Rownum, n ← Columnnum
 2  if M[[m/2] + 1, [n/2] + 1] = x
 3     then return ([m/2] + 1, [n/2] + 1);
 4  if m == 1&&n == 1
 5     then if M[0, 0] == x
 6             then return (i, j);
 7             else  return (−1, −1);
 8  if M[m/2, n/2] > x
 9     then if (a1, a2) ← Find − X(M[0 : m/2, 0 : n/2], x, i, j) ≠ (−1, −1)
10             then return (a1, a2);
11          elseif (a1, a2) ← Find − X(M[0 : m/2, n/2 : n − 1], x, i, j + n/2) ≠ (−1, −1)
12             then return (a1, a2);
13          elseif (a1, a2) ← Find − X(M[m/2 : m − 1, 0 : n/2], x, i + m/2, j) ≠ (−1, −1)
14             then return (a1, a2);
15          else return (−1, −1);
16  elseif M[m/2, n/2] < x
17     then if (a1, a2) ← Find − X(M[m/2 : m − 1, n/2 : n − 1], x, i + m/2, j + n/2) ≠ (−1,
18             then return (a1, a2);
19          elseif (a1, a2) ← Find − X(M[0 : m/2, n/2 : n − 1], x, i, j + n/2) ≠ (−1, −1)
20             then return (a1, a2);
21          elseif (a1, a2) ← Find − X(M[m/2 : m − 1, 0 : n/2], x, i + m/2, j) ≠ (−1, −1)
22             then return (a1, a2) + ([m/2] + 1, 0);
23          else return (−1, −1);
24  else return (i, j)
```

算法复杂性分析:
Preprocessing: $O(1)$;
Divide: $O(1)$;
Conquer: $3T(n/4)$;
Merge: $O(1)$.
递归方程:

$$T(n) = \begin{cases} O(1), & m, n = 1 \\ 3T(n/4) + O(1), & m, n \geq 2 \end{cases}$$

用 Master 定理求解 $T(n)$, 得 $T(n) = O(n^{\log_4 3})$.

第四章

1. 考虑三个字符串 X,Y,Z 的最长公共子序列 LCS(X,Y,Z)。
(1) 寻找反例 X,Y,Z 使得 LCS(X,Y,Z) LCS(X, LCS(Y,Z));
(2) 设计动态规划算法计算 X,Y,Z 的最长公共子序列, 分析算法的时间复杂度。
解：
(1) 给除一个反例：$X = "acbaced"$，$Y = "afbgecd"$，$Z = "afcgbhd"$。
容易得到，X,Y,Z 的最长公共子序列 $LCS(X,Y,Z) = "abcd"$；
而 $LCS(Y,Z) = "afgd"$，$LCS(X, LCS(Y,Z)) = "ad" \neq LCS(X,Y,Z)$。

(2)LCS 的最优子结构：
令 X=$< x_1, x_2, \cdots, x_m >$，Y=$y_1, y_2, \cdots, y_n$，Z=$z_1, z_2, \cdots, z_l$，U=$u_1, u_2, \ldots, u_k$ 为 X,Y,Z 的任意 LCS。
1. 如果 $x_m = y_n = z_l$，则 $u_k = x_m = y_n = z_l$，且 U_{k-1} 是 X_{m-1}，Y_{n-1} 和 Z_{l-1} 的一个最长公共序列。
2. 否则，如果 $u_k \neq x_m$，则意味着 U_k 是 X_{m-1}，Y_n 和 Z_l 的一个最长公共序列。
3. 否则，如果 $u_k \neq y_n$，则意味着 U_k 是 X_m，Y_{n-1} 和 Z_l 的一个最长公共序列。
4. 否则，如果 $u_k \neq z_l$，则意味着 U_k 是 X_m，Y_n 和 Z_{l-1} 的一个最长公共序列。
这就意味着，我们在求解最长公共子序列的时候会遇到求解 4 个子问题的情况，根据 LCS 的最优子结构，我们可以给出下面的公式：

$$c[i,j,k] = \begin{cases} 0, & i = 0或j = 0或k = 0 \\ c[i-1, j-1, k-1] + 1, & x_i = y_j = z_k \\ \max(c[i-1,j,k], c[i,j-1,k], c[i,j,k-1]), & 其它 \end{cases}$$

下面给出伪代码：

```
LCS-LENGTH(X, Y, Z)
 1  m ← X.length, n ← Y.length, l ← Z.length
 2  建立数组 c[1···m, 1···n, 1···l], b[1···m, 1···n, 1···l]
 3  for i = 1 to m
 4        do for j = 1 to n
 5              do c[i, j, 0] ← 0
 6  for i = 1 to m
 7        do for j = 1 to l
 8              do c[i, 0, j] ← 0
 9  for i = 1 to n
10        do for j = 1 to l
11              do c[0, i, j] ← 0
12  for i = 1 to m
13        do for j = 1 to n
14              do for k = 0 to l
15                    do if x[i] == y[j]&&x[i] == z[k]
16                       then c[i, j, k] ← c[i − 1, j − 1, k − 1] + 1
17                            b[i, j, k] ← 0
18                       elseif c[i − 1, j, k] > c[i, j − 1, k]&&c[i − 1, j, k] > c[i, j, k − 1]
19                       then c[i, j, k] ← c[i − 1, j, k]
20                            b[i, j, k] ← 1
21                       elseif c[i, j − 1, k] > c[i − 1, j, k]&&c[i, j − 1, k] > c[i, j, k − 1]
22                       then c[i, j, k] ← c[i, j − 1, k]
23                            b[i, j, k] ← 2
24                       else c[i, j, k] ← c[i, j, k − 1] ,b[i, j, k] ← 3
25  return c, b
```

```
PRINT-LCS(X, b, i, j, k)
 1  if i == 0orj == 0ork == 0
 2     then return
 3  if b[i, j, k] == 0
 4     then PRINT-LCS(X, b, i − 1, j − 1, k − 1)
 5          Print X[i]
 6  elseif b[i, j, k] == 1
 7     then PRINT-LCS(X, b, i − 1, j, k)
 8  elseif b[i, j, k] == 2
 9     then PRINT-LCS(X, b, i, j − 1, k)
10  else PRINT-LCS(X, b, i, j, k − 1)
```

我们可以通过调用 PRINT-LCS$(X, b, X.length, Y.length, Z.length)$ 来得到最长公共子序列。
时间复杂度为 $O(m \times n \times l)$。

2. 设计动态规划算法输出数组 A[0:n] 中的最长单调递增子序列。
解：
给定数列 $A[1:n]$，以 $A[i]$ 为单调递增子序列的最后一个元素时，所得最长

单调递增子序列的长度记作 $L[i]$，可得递推式：

$$L[i] = \max\{1, \max_{\substack{1\le k<i,\\ A[k]\le A[i]}} \{L[k]+1\}\},\ 1 \le i \le n$$

对上式稍作解释，当不存在 k 满足 $1 \le k < i$ && $A[k] \le A[i]$ 时，就将 $L[i]$ 置为 1。
以 $C[i]$ 记录上式中取得最值时的 k，即以 $A[i]$ 为最后一个元素的最长单调递增子序列的倒数第二个元素的下标，当 $L[i] = 1$ 时，记 $C[i] = -1$。
该问题具有优化子结构且子问题具有重叠性，可用动态规划解决，算法如下：

LONGEST-SUBSEQ(A)

```
    //返回最长单调递增子序列的长度 maxL 及最后一个元素在原数组中的下标 maxi，并返
 1  n = length(A);
 2  for i ← 1 to n
 3        do L[i] = 1; //初始化
 4           C[i] = −1;
 5           maxL = 1; //最长单调递增子序列的长度
 6           maxi = 1; //最长单调递增子序列的最后一个元素在原数组中的下标。
 7           for k ← 1 to i − 1
 8                do if A[k] ≤ A[i] & L[k] + 1 > L[i]
 9                      then L[i] ← L[k] + 1;
10                           C[i] = k;
11           if L[i] > maxL
12              then maxL = L[i]; maxi = i;
13  return maxL, maxi, C;
```

PRINT-OPTIMAL-SUBSEQ(A, C, i)

```
1  if C[i] ≠ −1
2     then PRINT-OPTIMAL-SUBSEQ(A, C, C[i])
3  Print A[i];
```

调用 PRINT-OPTIMAL-SUBSEQ($A, C, maxi$) 即可输出最长单调递增子序列。
算法复杂性：
时间复杂性：
(1) 计算代价的时间：两层循环，每层不超过 n 步，$O(n^2)$;
(2) 构造最优解的时间：$O(n)$;
总时间复杂性为：$O(n^2)$;
空间复杂性：使用数组 $L[1:n]$ 和 $C[1:n]$，需要空间 $O(n)$。

3. 输入数组 A[0:n] 和正实数 d, 试设计一个动态规划算法输出 A[0:n] 的一个最长子序列, 使得子序列中相继元素之差的绝对值不超过 d。分析算法的时间复杂度。
解：
给定数列 $A[1:n]$，以 $A[i]$ 为满足条件的最长子序列的最后一个元素时，所

得最长子序列的长度记作 $L[i]$，可得递推式：

$$L[i] = \max\{1, \max_{\substack{1 \le k < i, \\ |A[k]-A[i]| \le d}} \{L[k]+1\}\},\ 1 \le i \le n$$

对上式稍作解释，当不存在 k 满足 $1 \le k < i$ && $|A[k]-A[i]| \le d$ 时，就将 $L[i]$ 置为 1。
以 $C[i]$ 记录上式中取得最值时的 k，即以 $A[i]$ 为最后一个元素的最长子序列的倒数第二个元素的下标，当 $L[i]=1$ 时，记 $C[i]=-1$。
该问题具有优化子结构且子问题具有重叠性，可用动态规划解决，算法如下：

```
LONGEST-SUBSEQ(A, d)
    //返回最长单调递增子序列的长度 maxL 及最后一个元素在原数组中的下标 maxi，并返
 1  n = length(A);
 2  for i ← 1 to n
 3        do L[i] = 1;
 4           C[i] = −1;
 5           maxL = 1;
 6           maxi = 1;
 7           for k ← 1 to i − 1
 8                do if |A[k] − A[i]| ≤ d & L[k] + 1 > L[i]
 9                      then L[i] ← L[k] + 1;
10                           C[i] = k;
11           if L[i] > maxL
12              then maxL = L[i]; maxi = i;
13  return maxL, maxi, C;
```

```
PRINT-OPTIMAL-SUBSEQ(A, C, i)
1  if C[i] ≠ −1
2     then PRINT-OPTIMAL-SUBSEQ(A, C, C[i])
3  Print A[i];
```

调用 PRINT-OPTIMAL-SUBSEQ$(A, C, maxi)$ 即可输出满足条件的最长子序列。
算法复杂性：
时间复杂性：
(1) 计算代价的时间：两层循环，每层不超过 n 步，$O(n^2)$;
(2) 构造最优解的时间：$O(n)$;
总时间复杂性为：$O(n^2)$;
空间复杂性：使用数组 $L[1:n]$ 和 $C[1:n]$，需要空间 $O(n)$。

4. 给定一个整数序列 $a_1, ..., a_n$。相邻两个整数可以合并, 合并两个整数的代价是这两个整数之和。通过不断合并最终可以将整个序列合并成一个整数, 整个过程的总代价是每次合并操作代价之和。试设计一个动态规划算法给出 $a_1, ..., a_n$ 的一个合并方案使得该方案的总代价最大。你的答案应包括:
(1) 用简明的语言表述这个问题的优化子结构;

(2) 根据优化子结构写出代价方程;
(3) 根据代价方程写出动态规划算法 (伪代码) 并分析算法的时间复杂性;
解：
(1) 若合并 $a_1, ..., a_n$ 的最优解最后一次合并操作在 $a_1, \ldots, a_{k-1}$ 和 $a_k, \ldots, a_n$ 之间，即最后一次操作合并 $a_1 + \ldots + a_{k-1}$ 和 $a_k + \ldots + a_n$,$2 \le k \le n$，则在合并 $a_1, ..., a_n$ 的优化顺序中，对应子问题合并 $a_1, ..., a_{k-1}$ 的解必须是合并 $a_1, ..., a_{k-1}$ 的优化解，对应子问题合并 $a_k, ..., a_n$ 的解必须是合并 $a_k, ..., a_n$ 的优化解。因此问题具有优化子结构，问题的优化解包含子问题的优化解。
(2) 设 $S(i, j) = a_i + \cdots + a_j$;
$m[i, j]$ 表示合并 $a_i, ..., a_j$ 的最大代价，$1 \le i \le j \le n$，
那么 $m[1, n]$ 表示合并 $a_1, ..., a_n$ 的最大代价。
代价方程：
$m[i, j] = 0,\ i = j$;
$m[i, j] = \max\limits_{i+1 \le k \le j} \{m[i, k-1] + m[k, j]\} + S[i, j],\ i < j$;
为了简化运算，我们可以对 $S[i, j]$ 用动态规划来求解。
$S[i, j] = a_i,\ i = j$;
$S[i, j] = S[i, j-1] + a_j,\ i < j$
(3) 以 $D[i, j] = k$ 记录合并 $a_i, ..., a_j$ 的最后一次合并操作在 $a_i, \ldots, a_{k-1}$ 和 $a_k, \ldots, a_j$ 之间，动态规划算法如下：

```
INTEGER-MERGE-ORDER(a_1, ..., a_n)
 1  for i ← 1 to n //计算第 1 对角线
 2        do m[i,i] ← 0, S[i,i] ← a_i;
 3  for l ← 2 to n //计算第 l 对角线;
 4        do for i ← 1 to n − l + 1
 5                 do j ← i + l − 1;
 6                    S[i,j] ← S[i,j−1] + a_j;
 7                    m[i,j] ← −∞;
 8                    for k ← i + 1 to j
 9                         do q ← m[i,k−1] + m[k,j];
10                            if q > m[i,j]
11                               then m[i,j] ← q, D[i,j] ← k;
12                    m[i,j] ← m[i,j] + S[i,j];
13  return m, D;
```

```
PRINT-OPTIMAL-PARENS(D, i, j)
1  if i = j
2     then Print a_i;
3     else Print "(";
4          PRINT-OPTIMAL-PARENS(D, i, D(i,j) − 1);
5          Print "+";
6          PRINT-OPTIMAL-PARENS(D, D(i,j), j);
7          Print ")";
```

调用 PRINT-OPTIMAL-PARENS$(D, 1, n)$ 即可输出 $a_1, ..., a_n$ 的合并顺序。
时间复杂性：

计算时间的代价：(l,i,k) 三层循环，每层至多 $n-1$ 步，需要 $O(n^3)$ 时间。
构造最优解的时间：$O(n)$;
总时间复杂性为：$O(n^3)$。

5. 输入表达式 $a_1O_1a_2O_2\cdots O_{n-1}a_n$, 其中 a_i 是整数 $(1\le i\le n)$,$O_j\in\{+,-,\times\}(1\le i\le n-1)$。
(1) 试设计一个动态规划算法, 输出一个带括号的表达式 (不改变操作数和操作符的次序), 使得表达式的值达到最大, 分析算法的时间复杂性。
(2) 令 $O_j\in\{+,-,\times,\div\}$, 重新完成 (1) 规定各项任务。
解：
(1)
寻找子结构：当确定最后一个运算符 O_k 之后，可以把当前的表达式分为两部分：$a_1O_1a_2\cdots a_k$ 和 $a_{k+1}O_{k+1}\cdots a_n$，这两部分构成了原表达式的子结构。
优化子结构：当分解为两个子表达式之后，对于不同的运算符，子表达式的最优解有所不同。
代价方程为：
用 $m[i,j]$ 表示 $a_iO_i\cdots a_j$ 表达式的最大值；$b[i,j]$ 表示 $a_iO_i\cdots a_j$ 表达式的最小值。

$$m[i,j]=b[i,j]=a_i,\ \ i==j$$

$$m[i,j]=\max\begin{cases} m[i,k]O_km[k+1,j], & O_k==' +' \\ m[i,k]O_kb[k+1,j], & O_k==' -' \\ \max(m[i,k]O_km[k+1,j],m[i,k]O_kb[k+1,j] & \\ \quad b[i,k]O_km[k+1,j],b[i,k]O_kb[k+1,j]), & O_k==' \times' \end{cases}$$

$$b[i,j]=\min\begin{cases} b[i,k]O_kb[k+1,j], & O_k==' +' \\ b[i,k]O_km[k+1,j], & O_k==' -' \\ \min(m[i,k]O_km[k+1,j],m[i,k]O_kb[k+1,j] & \\ \quad b[i,k]O_km[k+1,j],b[i,k]O_kb[k+1,j]), & O_k==' \times' \end{cases}$$

给出伪代码：

```
CALCULATE-MAX(A, O)
 1  n ← A.length
 2  for i ← 1 to n
 3        do m[i,i] = b[i,i] = a[i]
 4  for l ← 2 to n
 5        do for i ← 1 to n − l + 1
 6                 do j = i + l − 1
 7                    m[i,j] ← −∞, b[i,j] ← ∞
 8                    for k ← i to j − 1
 9                         do if O_k == "+"
10                              then a ← m[i,k] + m[k+1,j]
11                                   b ← b[i,k] + b[k+1,j]
12                            elseif O_k == "-"
13                              then a ← m[i,k] − b[k+1,j]
14                                   b ← b[i,k] − m[k+1,j]
15                            else O_k == "×"
16                                   a ← max(m[i,k] × m[k+1,j], m[i,k] × b[k+1,j],
17                                   b[i,k] × m[k+1,j], b[i,k] × b[k+1,j])
18                                   b ← min(m[i,k] × m[k+1,j], m[i,k] × b[k+1,j],
19                                   b[i,k] × m[k+1,j], b[i,k] × b[k+1,j])
20                            if a > m[i,j]
21                              then m[i,j] ← a
22                                   S[i,j] ← k
23                            if b < b[i,j]
24                              then b[i,j] ← b
25  return m, S
```

```
PRINT-OPTIMAL-PARENS(S, i, j)
1  if i = j
2     then Print a_i;
3     else k = S[i,j]
4          Print "(";
5          PRINT-OPTIMAL-PARENS(S, i, k);
6          Print "O_k";
7          PRINT-OPTIMAL-PARENS(S, k+1, j);
8          Print ")";
```

时间复杂度：$T(n) = O(n^3)$。

6. 设平面上有一个 $m \times n$ 的网格, 将左下角的网格点标记为 $(0,0)$，而右上角的网格点标记为 (m,n)。某人想从 $(0,0)$ 出发沿网格线行进到达 (m,n), 但是在网格点 (i,j) 处他只能向上行进或者向右行进, 向上行进的代价为 $a_{ij}(a_{mj} = +\infty)$, 向右行进的代价是 $b_{ij}(b_{in} = +\infty)$。试设计一个动态规划算法, 在这个网格中为该旅行者寻找一条代价最小的旅行路线。
解：

(1) 从 $(0,0)$ 出发沿网格线行进到达 (m,n)，共需 $m+n$ 步。令

$$S[i,j]=\begin{cases}1, & \text{在 (i,j) 处选择向上行进}\\ 0, & \text{在 (i,j) 处选择向右行进}\end{cases}$$

$S[i,j]$ 表示在 (i,j) 处想要到达 $[m,n]$ 处，最优的旅行路径的第一步。

(2) 设 $C[i,j]$ 为从 (i,j) 出发沿网格线行进到达 (m,n) 的最优解的代价，则有递归方程：

$C[m,n]=0;$
$C[i,n]=C[i+1,n]+a_{ij},\ 0\le i<m;$
$C[m,j]=C[m,j+1]+b_{ij},\ 0\le j<n;$
$C[i,j]=\max\{C[i+1,j]+a_{ij},C[i,j]+b_{ij}\},\ 0\le i\le m-1;0\le j\le n-1;$

(3) 给出算法的伪代码：

FIND-SHORTEST-PATH(m,n,a,b)

```
1  C[m,n] ← 0, S[m,n] ← −1; //初始化
2  for j ← n − 1 to 0 //第 m 行只能向右行进；
3        do C[m,j] ← C[m,j+1] + b[m,j], S[m,j] ← 0;
4  for i ← m − 1 to 0 //第 n 列只能向上行进；
5        do C[i,n] ← C[i+1,n] + a[i,n], S[i,n] ← 1;
6  for i ← m − 1 to 0
7        do for j ← n − 1 to 0
8              do if C[i,j+1] + b[i,j] ≤ C[i+1,j] + a[i,j]
9                    then C[i,j] ← C[i,j+1] + b[i,j], S[i,j] ← 0;
10                   else  C[i,j] ← C[i+1,j] + a[i,j], S[i,j] ← 1;
11 return C, S;
```

构造最优解（经过的网格点集合）：

PRINT-SHORTEST-PATH(S,C,i,j)

```
1  Print (i,j);
2  if S[i,j] = 1
3     then PRINT-SHORTEST-PATH(S,C,i+1,j);
4  elseif S[i,j] = 0
5     then PRINT-SHORTEST-PATH(S,C,i,j+1);
```

调用 PRINT-SHORTEST-PATH$(S,C,0,0)$ 就可以打印出代价最小的旅行路线。

算法的时间复杂性：

计算代价的时间：(i,j) 两层循环，i 循环 m 步，j 循环 n 步，时间复杂性 $O(mn)$；

构造最优解的时间：$O(m+n)$;

总时间复杂性：$O(mn)$。

7. 给定一个 $n \times n$ 的矩阵 A, 矩阵中的元素只取 0 或者 1。设计一个动态规划算法, 求解得到 A 中元素全是 1 的子方阵使其阶数达到最大值。

解：

设 $B[i,j]$ 表示以元素 $A[i,j]$ 为右下角元素并且元素全为 1 的方阵的最大阶数。

当 $B[i,j]=k$ 的时候，则 $B[i-1,j] \geq k-1$，$B[i,j-1] \geq k-1$，$B[i-1,j-1] \geq k-1$ 会成立。

我们得到如下的递归方程：

$B[i,j]=0$, 当$A[i,j]==0$；

$B[i,j]=\min\{B[i-1,j],B[i,j-1],B[i-1,j-1]\}+1$, 当$A[i,j]==1$；

$B[0,j]=A[0,j], 0 \leq j \leq n-1$；

$B[i,0]=A[i,0], 0 \leq i \leq n-1$。

现在将对上面的递归方程做一定的解释：

(1) 当 $A[i,j]==0$ 的时候，以 0 为右下角的全 1 方阵很显然不存在，所以 $B[i,j]=0$。

(2) 当 $A[i,j]==1$ 的时候，设 $k=\min\{B[i-1,j],B[i,j-1],B[i-1,j-1]\}$，所以 $B[i-1,j] \geq k$，$B[i,j-1] \geq k$，$B[i-1,j-1] \geq k$，很容易得到 $B[i,j] \geq k+1$。假设 $B[i,j]=k+2$，由之前推论可以得到 $B[i-1,j] \geq k+1$，$B[i,j-1] \geq k+1$，$B[i-1,j-1] \geq k+1$，也就是说，$\min\{B[i-1,j],B[i,j-1],B[i-1,j-1]\} \geq k+1$，矛盾。

所以，$B[i,j]=k+1=\min\{B[i-1,j],B[i,j-1],B[i-1,j-1]\}+1$。

(3) 当 $i==0$或者$j==0$ 的时候，最多只能构成 1 阶方阵，所以 $B[i,j]=A[i,j]$。

下面给出伪代码：

MAX-ALLONES-SUBMAT(A,n)

```
1  for i ← 0 to n − 1
2        do B[i, 0] ← 0
3           B[0, i] ← 0
4  for i ← 1 to n − 1
5        do for j ← 1 to n − 1
6                 do if A[i, j] == 1
7                       then k ← B[i − 1, j]
8                            if k ≥ B[i, j − 1]
9                              then k ← B[i, j − 1]
10                           if k ≥ B[i − 1, j − 1]
11                             then k ← B[i − 1, j − 1]
12                           B[i, j] ← k + 1
13                      else  B[i, j] ← 0
14 max ← 0
15 for i ← 1 to n − 1
16       do for j ← 1 to n − 1
17                do if B[i, j] ≥ max
18                      then max ← B[i, j]
19 return max
```

时间复杂度为：$O(n^2)$。

8. 集合划分问题描述如下:
输入: 正整数集合 $S = \{a_1, a_2, a_3, ..., a_n\}$;
输出: 是否存在 $A \subseteq S$ 使得 $\sum_{a_i \in A} a_i = \sum_{a_i \in S-A} a_i$;
试设计一个动态规划算法, 求解集合划分问题。
解：
记 $C = \frac{1}{2} \times \sum_{a_i \in S} a_i$，如果 C 不为整数，则不能划分，当 C 为整数时，上述问题就等价于：对于正整数集合 $S = \{a_1, a_2, a_3, ..., a_n\}$，判断是否存在 $A \subseteq S$ 使得 $\sum_{a_i \in A} a_i = C$，这个问题其实跟 0-1 背包问题时等价的，我们转化为背包容量为 C，价值和代价都等于 a_i 的 0-1 背包问题，只需求解下述问题：

$$当 \sum_{i=1}^{n} a_i x_i \leq C \text{ 并且 } x_i \in \{0,1\}, i = 1, \ldots, n$$

$$求解 \max \sum_{i=1}^{n} a_i x_i$$

只有当 $\max \sum_{i=1}^{n} a_i x_i = C$ 时原问题才存在解。
递归方程：
以 $m(i,j)$ 记和的上限为 j，可选数为 $a_i, \ldots, a_n$ 时问题的最优解的代价，得递归方程：
$m(i,j) = m(i+1,j), 0 \leq j < a_i$;
$m(i,j) = \max\{m(i+1,j), m(i+1,j-a_i) + a_i\}, j \geq a_i$;
$m(n,j) = 0, 0 \leq j < a_n$;
$m(n,j) = a_n, 0 \geq a_n \leq j$。
算法如下：

EXIST-PARTITION(S)

```
1  n ← length(S);
2  sum ← Σ_{a_i∈S} a_i
3  if sum/2 ≠ 0
4     then return False;
5  C ← 1/2 × sum;
6  for j ← 0 to min(a_n − 1, C)
7        do m[n, j] ← 0;
8  for j ← a_n to C
9        do m[n, j] ← a_n;
10 for i ← n − 1 to 2
11       do for j ← 0 to min(a_i − 1, C)
12                do m[i, j] ← m[i + 1, j];
13             for j ← a_i to C
14                do m[i, j] ← max{m[i + 1, j], m[i + 1, j − a_i] + a_i};
15 if C < a_1
16    then m[1, C] ← m[2, C];
17    else m[1, C] ← max{m[2, C], m[2, C − a_1] + a_1};
18 if m(1, C) = C
19    then return True;
20    else return False;
```

算法的时间复杂性为 $O(nC)$，空间复杂性为 $O(nC)$。

9. 输入平面上 n 个点, 点与点之间的距离定义为欧几里得距离。试设计一个动态规划算法输出一条先从左到右再从右到左的一条最短路径, 使得每个输入点恰好被访问一次。

解：

预处理：首先将各点按照 x 坐标从小到大排列，$P = p_1, p_2, \cdots, p_n$。

寻找子结构：定义从 p_i 到 p_j 的路径为：从 p_i 开始，从左到右一直到 p_n，然后从右到左一直到 p_j。在这个路径上，会经过 $\min(p_i, p_j)$ 到 p_n 之间的所有点并且只经过一次。

在定义 $c(i, j)$ 为满足这一条件的最短路径。因为 $c[i, j] = c[j, i]$，所以我们只考虑 $i \leq j$ 的情况。

同时，定义 $dist(i, j)$ 为点 p_i 到 p_j 之间的直线距离。

最优解：关于子问题 c(i,j) 的求解，分三种情况：

A、当 $j > i + 1$ 时，$c(i, j) = c(i + 1, j) + dist(i, i + 1)$。

由定义可知，点 p_{i+1} 一定在路径 $p_i - p_j$ 上，而且又由于 $j > i + 1$, 因此 p_i 的右边的相邻点一定是 p_{i+1}. 因此可以得出上述等式。

B、当 $j = i + 1$ 时，与 p_i 右相邻的那个点可能是 p_{i+2} 到 p_n 中的任何一个。因此需要递归求出最小的那个路径：

$c(i, j) = c(i, i + 1) = \min\{c(j, k) + dist(i, k)\}$, 其中$j + 1 <= k <= n$。

C、我们最后想要得到解是：$\min\{c(1, k)\}, 2 \leq k \leq n$。

下面给出伪代码：

```
FIND-SHORT-DISTANCE(P, n)
 1  c[n-1, n] = DISTANCE(P, n-1, n)
 2  for j ← n to 3
 3        do for i ← j-2 to 1
 4              do c[i, j] ← c(i+1, j) + DISTANCE(P, i, i+1)
 5           c[j-1, j] ← ∞
 6           for k ← j+1 to n
 7                 do temp ← c[j, k] + DISTANCE(P, j-1, k)
 8                    if temp < c[j-1][j]
 9                       then c[j-1][j] ← temp
10  min ← c[1, 2]
11  for k ← 3 to n
12        do temp ← c[1, k]
13           if temp < min
14              then min ← temp
15  return min
```

10. 输入凸 n 边形 $p_1, p_2, ...p_n$, 其中顶点按凸多边形边界的逆时针序给出, 多边形中不相邻顶点间的连线称为弦。试设计一个动态规划算法, 将凸边形 $p_1, p_2, ...p_n$ 剖分成一些无公共区域的三角形, 使得所有三角形的周长之和最小。

解：

设 $t[i, j] =< v_i, v_{i+1}, ..., v_j >$ 的优化三角剖分周长之和，可以很容易得到以下的递推公式**:**

$t[i, i] = t[j, j] = 0$

$t[i, j] = \min_{i \le k < j}\{t[i, k] + t[k, j] + w(\Delta v_i v_k v_j)\}$

其中 $w(\Delta v_i v_k v_j)$ 表示的是计算三角形 $\Delta v_i v_k v_j$ 的周长，即 $w(\Delta v_i v_k v_j) = d(v_i v_k) + d(v_k v_j) + d(v_i v_j)$。

```
CONVEXPOLYGON-TRIANGULATION(w)
 1  for i ← 1 to n //计算第 1 对角线
 2        do m[i, i] ← 0;
 3  for l ← 2 to n //计算第 l 对角线;
 4        do for i ← 1 to n-l+1
 5              do j ← i+l-1;
 6                 m[i, j] ← ∞;
 7                 for k ← i+1 to j-1
 8                       do q ← m[i, k] + m[k, j] + w(Δv_i v_k v_j);
 9                          if q < m[i, j]
10                             then m[i, j] ← q, S[i, j] ← k;
11  return m, S;
```

FIND-OPTIMAL-STRINGS(S, i, j)

```
1  k = S[i, j];
2  if i = j
3      then return ∅;
4  A ← Find-Optimal-Strings(S, i, k);
5  A ← A ⋃ Find-Optimal-Strings(S, k, j);
6  if k > i + 1
7      then A ← A ⋃ {v_i v_k};
8  if k < j − 1
9      then A ← A ⋃ {v_k v_j};
10 return A;
```

调用 Find-Optimal-Strings$(S, 1, n)$ 即可返回凸多边形 $(v_0, v_1, \ldots, v_n)$ 的优化三角剖分。

11. 输入一棵加权树 T, 其中每条边的权值均是实数, 试设计一个动态规划算法输出权值最大的子树。

解：

我们采用邻接表的形式存储树的节点和边，第一个节点为树的根。

寻找子结构：设 $c[i, j]$ 表示第 i 层的第 j 个节点作为根的子树的最大权值。该节点作为根的子树想要得到权值最大，逐个扫描该节点的子节点 $son(i)$，转而求解子节点为根的子树的最大权值。

最优子结构: 可以得到如下递推公式：

$c[i, n] = 0$，当 i 节点为叶节点，n 表示树的层数；

$c[i, j] = \sum_k \max(0, c[k, j+1] + b[i, k]),\ \ k \in son(i)$，当 i 节点有子节点。

在上式中 $b[i, k]$ 表示连接第 i 个节点与第 k 个节点的边的权值。

给出伪代码：用 $T[i]$ 表示树 T 的第 i 层。

FIND-MAX-SUBTREE(T, b, n)

```
1  for i ∈ T[n]
2      do c[i, n] ← 0
3  for j ← n − 1 to 1
4      do for i ∈ T[j]
5             do c[i, j] ← 0, S[i, j] ← ∅
6                for k ∈ son(T[j][i])
7                    do if c[k, j + 1] + b[i, k] > 0
8                           then c[i, j] ← c[i, j] + c[k, j + 1] + b[i, k]
9                                S[i, j] ← S[i, j] ⋃ T[j + 1][k]
10 max ← 0
11 for j ← n − 1 to 1
12     do for i ∈ T[j]
13            do if c[j][i] > max
14                  then max ← c[j][i]
15                       maxi ← i, maxj ← j
16                       return max, S, maxi, maxj
```

PRINT-SUBTREE(T, S, i, j)

```
1  Print T[j][i]
2  for k ∈ S[i, j]
3        do PRINT-SUBTREE(T, S, k, j + 1)
```

调用 PRINT-SUBTREE$(T, S, maxi, maxj)$ 就可以输出结果。
看似三重循环，实际只是把树的每个节点遍历一遍，时间复杂度为 $O(n)$。

第五章

1. 现有一台计算机, 在某个时刻同时到达了 n 个任务。该计算机在同一时间只能处理一个任务, 每个任务都必须被不间断地得到处理。该计算机处理这 n 个任务需要的时间分别为 $a_1, a_2, ..., a_n$。将第 i 个任务在调度策略中的结束时间记为 e_i。请设计一个贪心算法输出这 n 个任务的一个调度使得用户的平均等待时间 $1/n \sum e_i$ 达到最小。

   解：

   贪心思想：为了使得用户的平均等待时间 $1/n \sum e_i$ 达到最小，每次选择所需处理时间最短的任务进行处理，一个任务处理完后立刻进行下一个任务的处理，直到全部任务处理完成。

   贪心选择性：设 $A = \{1, 2, \ldots, n\}$ 是 n 个任务的集合，假设所有任务都已经按照所需时间的从小到大进行排序，即 $a_1 \leq a_2 \leq \cdots \leq a_n$，则存在 A 的调度问题的一个最优解即平均等待时间最短的调度，将时间最短的任务 1 安排在第一个处理。

   证明：设 $1, 2, \ldots, n$ 的一个排列 $i_1 i_2 \ldots i_n$ 是问题的一个最优解，e_j 为第 j 个被处理的任务的等待时间。

   若 $i_1 = 1$，贪心选择性成立。

   若 $i_1 \neq 1$，假设任务 r 是第一个处理的任务 $(r \neq 1)$，而任务 1 排第 k 个处理，即调度为 $ri_2 \ldots i_{k-1} 1 i_{k+1} \ldots i_n$，那么将任务 1 与任务 r 调换次序，得到新调度 $1i_2 \ldots i_{k-1} r i_{k+1} \ldots i_n$，其中 e'_j 为新的调度中第 j 个被处理的任务的等待时间。

   显然有 $e_j = e'_j,\ j = k+1, ..., n$。

   由于 $a_1 \leq a_r$，则有

   $e'_1 = a_1 \leq a_r = e_1,$

   $e'_2 = a_1 + a_{i_2} \leq a_r + a_{i_2} = e_2,$

   $\ldots$

   $e'_{k-1} = a_1 + a_{i_2} + \ldots a_{k-1} \leq a_r + a_{i_2} + \ldots a_{k-1} = e_{k-1},$

   $e'_k = a_1 + a_{i_2} + \ldots a_{k-1} + a_r = a_r + a_{i_2} + \ldots a_{k-1} + a_1 = e_k,$

   因而，有 $\frac{1}{n} \sum_{j=1}^{n} e'_j \leq \frac{1}{n} \sum_{j=1}^{n} e_j$, 由于调度 $ri_2 \ldots i_{k-1} 1 i_{k+1} \ldots i_n$ 是一个最优解，进而有 $\frac{1}{n} \sum_{j=1}^{n} e'_j \geq \frac{1}{n} \sum_{j=1}^{n} e_j$，从而可以得到 $\frac{1}{n} \sum_{j=1}^{n} e'_j = \frac{1}{n} \sum_{j=1}^{n} e_j$。所以调度 $1i_2 \ldots i_{k-1} r i_{k+1} \ldots i_n$ 也是问题的一个最优解，并且首先处理任务 1。

   优化子结构：设 $A = \{1, 2, \ldots, n\}$ 是 n 个任务的集合，计算机处理这 n 个任务需要的时间分别为 $a_1, a_2, ..., a_n$，且 $1 \leq a_2 \leq \ldots a_n$。设 $1i_2 \ldots i_n$ 是调度问题 A 的一个最优解，那么 $A' = A - \{1\}$ 一定是 $S' = S - \{1\}$ 的优化解。

   证明：调度 S' 中的每一项任务都包含在 A' 中，所以 S' 是 A' 的一个解。

   假设调度 $S' = i_2 \ldots i_n$ 不是 A' 的调度问题的最优解, 则存在另一个调度 $i'_2 \cdots i'_n$ 为问题的最优解，

   $\frac{1}{n-1} \sum_{j=2}^{n} e_{i'_n} < \frac{1}{n-1} \sum_{j=2}^{n} e_{i_n}$。

   那么 $1i'_2 \ldots i'_n$ 也是 A 的调度问题的一个解，且

$\frac{1}{n}(e_1+\sum_{j=2}^{n} e_{i'_n}) = \frac{1}{n}(a_1+\sum_{j=2}^{n} e_{i'_n}) < \frac{1}{n}(a_1+\sum_{j=2}^{n} e_{i_n}) = \frac{1}{n}(e_1+\sum_{j=2}^{n} e_{i_n})$，与 $1i_2 \dots i_n$ 是调度问题 A 的最优解矛盾。所以，结论成立。

贪心选择性：若任务都已经按照所需任务的从小到大进行排序，即 $a_1 \le a_2 \le ... \le a_n$，那么 $A = \{1 \to 2 \to ... \to n\}$ 是问题的最优解。

证明：当 $|S| = 1$ 时，由贪心选择性可知，命题成立；
若 $|S| < k$ 时，命题成立；
则 $|S| = k$ 时，$A = 1 \bigcup \{2 \to ... \to n\}$ 由优化子结构可知，命题成立
所以，得证。
算法：

```
TASK-PLAN(A, a_1, a_2, ..., a_n)
   （设 a_1 ≤ a_2 ≤ ··· ≤ a_n 已排序）
1  n = length(A);
2  for i ← 1to n
3        do B[i] = i;
4  return B;
```

(4) 时间复杂性：
如果 $a_1, a_2 \dots a_n$ 已排序，$T(n) = \Theta(n)$；
如果 $a_1, a_2 \dots a_n$ 未排序，$T(n) = \Theta(n) + \Theta(n \log n) = \Theta(n \log n)$。

2. 现有面值为 1 角、5 分、2 分、1 分的硬币, 每种硬币的个数都是无限的。给出一个贪心算法, 使得对任意给定的面值为 $n(n > 18)$ 分的纸币能够将它兑换成币值相等的硬币且使用硬币个数最少。证明算法的正确性并分析其复杂度。

解：
贪心思想：每次选择不大于当前需要兑换的币值的面值最大的硬币。
贪心选择性：
证明：
当 $1 \le n < 2$ 时，$n = 1$，只能选择 1 分的硬币。
当 $2 \le n < 5$ 时，一定会选择了一个 2 分的硬币，否则，如果全是由 1 分构成的方案，用 2 分的硬币代替 2 个 1 分的硬币，硬币数将减少，得到一个更优的解。
当 $5 \le n < 10$ 时, 一定会选择一个 5 分的硬币，否则，如果全是由 1 分和 2 分硬币构成，如果 1 分的硬币数量大于 2，则用一个 2 分硬币代替，硬币数量会减少，这样就可以得到 1 分硬币的数量 $n_1 \le 1$，2 分硬币的数量 $n_2 \ge 2$，如果 $n_1 = 1$，则用 5 分硬币代替 2 个 2 分硬币和 1 个一分硬币，得到一个更优的解。如果 $n_1 = 0$，则用一个 5 分硬币和一个 1 分硬币代替 3 个 2 分硬币，也可以得到更优的解。
当 $n > 10$ 时，存在某个最优解是需要先选择一个 10 分的。如果某个最优解第一步不是选择 10 分的，那么其后面一定有一步是选择了 10 分的，否则，如果全由 5 分、2 分和 1 分构成，可以由前面的推导过程知道，1 分的硬币数量 $n_1 \le 1$，2 分的硬币数量 $n_2 \le 2$，并且不能同时取等号，因为当 $n_1 = 1$并且$n_2 = 2$ 的时候，可以用一个 5 分硬币来代替它们，得到一个最优

解，所以可以得到 $n_1 \times 1 + n_2 \times 2 < 5$，又因为 $n > 10$，所以必会存在 2 个 5 分硬币，我们就可以用一个 10 分的硬币代替这 2 个 5 分硬币，得到一个更优的解。那么，可以当作这个解也是第一个选择了 10 分的。所以，一定存在某个优化解是第一次选择可选的最大币值，贪心选择性成立。

优化子结构：
设所需硬币数为 C; 由于上面的结论，我们假设 A 是一个第一次拿 10 分钱的最优解。那么 $A' = A - 10$分 一定是 $n' = n - 10$ 的找零问题的优化解.
如果不是这样，那么设 n' 找零的优化解为 B'，则对于 n 的找零优化解为 B，则 $C_B = C_{B'} + 1$. 所以 $C_B = C_{B'} + 1 > C_{A'} + 1 = C_A$, 这于条件中 A 是最优解矛盾，所以假设不正确，原结论得证。

算法的正确性：综上所述，算法按照贪心选择性确定的规则进行局部优化选择，即每次总选择不超过当前需要兑换的币值的面值最大的硬币，即依次选择 1 角，5 分，2 分，1 分的硬币，所以算法产生一个使用硬币最少的策略。

GREEDY-COINS-EXCHANGE(n)

```
1  i = 0;
2  while n >= 10
3        do i ← i + 1;
4           A[i] ← 10;
5           n = n − 10;
6  while n >= 5
7        do i ← i + 1;
8           A[i] ← 5;
9           n = n − 5;
10 while n >= 2
11       do i ← i + 1;
12          A[i] ← 2;
13          n = n − 2;
14 while n >= 1
15       do i ← i + 1;
16          A[i] ← 1;
17          n = n − 1;
18 return A, i;
```

时间复杂度为：$T(n) = O(n)$。

3. 给定 k 个排好序的有序序列 $s_1, s_2, ..., s_k$ ，现在用 2 路归并排序算法对这些有序序列排序。假定用 2 路归并排序算法对长度分别为 m 和 n 的有序序列排序要用 $m + n - 1$ 次比较操作。设计一个贪心算法合并 $s_1, s_2, ..., s_k$ 使得所需的比较操作次数最少。
解：
贪心思想：每次都选择所有待排序列中长度最小的两个序列拿出待排序列，进行合并，然后将合并后形成的新序列加入到待排序列中，直到最后成为一整个序列。

贪心选择性：
设 $s_1, s_2, ..., s_k$ 是 k 个排好序的有序序列，其长度分别为 $d_1, d_2, ..., d_k$，用 2 路归并排序算法对长度分别为 m 和 n 的有序序列排序要用 $m+n-1$ 次比较操作。设 s_i 和 s_j 是这些序列中长度最短的两个，则存在一个合并 $s_1, s_2, ..., s_k$ 的比较操作次数最少的策略中，s_i 和 s_j 在合并之前没有和其它的序列合并。
证明：
当两个序列 a, b 被合并之后，合并的序列还将和其它的序列合并，我们可以认为 a, b 序列被合并来多次，用 c_i 表示序列 s_i 合并的次数，显然 $c_i \leq k-1$，所以合并之后的代价也可以这样表示：$\sum_{i=1}^{k}(c_i \times d_i) - k + 1$。
设 A 是一个合并 $s_1, s_2, ..., s_k$ 的优化解，合并序列的操作次数为为 h。
若在 A 中，s_i 和 s_j 在合并之前没有和其它的序列合并，贪心选择性成立。
若在 A 中，$s_{i'}$ 和 $s_{j'}$ 合并次数相同且合并次数最大。且 $i \neq i'$ 或者 $j \neq j'$，那么调换 s_i 和 $s_{i'}$ 在合并过程中的次序，调换 s_j 和 $s_{j'}$ 在合并过程中的次序，得到另一个 $s_1, s_2, ..., s_k$ 次序安排 A'，其操作次数为 h'。代价差为：

$$\begin{aligned} h' - h &= c_{i'}d_i + c_{j'}d_j + c_i d_{i'} + c_j d_{j'} - (c_i d_i + c_j d_j + c_{i'}d_{i'} + c_{j'}d_{j'}) \\ &= (c_{i'} - c_i)(d_i - d_{i'}) + (c_{j'} - c_j)(d_j - d_{j'}) \end{aligned}$$

由于 $d_i \leq d_{i'}, d_j \leq d_{j'}, c_i \geq c_{i'}, c_j \geq c_{j'}$，所以 $h \geq h'$，又因为 A 是优化解，所以 $h \leq h'$，即 $h = h'$，A' 也是优化解。
优化子结构：
设 A 是 $S = \{s_1, s_2, ..., s_k\}$ 的合并问题的优化解，其首先合并长度最短的 s_i 和 s_j 得到序列 s_0，长度记为 d_0，那么在合并次序 A 中去掉 s_i 和 s_j，将其合并后的序列替换为 s_0，得到 A'，它是子问题 $S' = S - \{s_i, s_j\} \bigcup \{s_0\}$ 的最优解。
证明：
若不然，存在合并问题 $S' = S - \{s_i, s_j\} \bigcup \{s_0\}$ 的优化解 B'，B' 中比较操作次数 $d(B') < d(A')$，那么首先合并 s_i 和 s_j 得到 s_0，按照 B' 中的合并顺序继续合并，得到原问题的解 B，那么 $d(B) = d(B') + d_i + d_j - 1 < d(A') + d_i + d_j - 1 = d(A)$，与 A 是优化解矛盾。
综合 (1)(2) 可知，算法按照 Greedy 选择性确定的规则进行局部优化选择即可得到问题的优化解。
给出算法的伪代码：

```
GREEDY-MERGE-SORT(S)
1  k ← length(S);
2  n_i ← length(S[i]);
3  for i ← 1 to k − 1
4        do m_1 ← min_i(n_i);
5           m_2 ← min_i({n_i} \ {m_1});
6           S[m_1] = MERGE-SORT (S[m_1], S[m_2]);
7           S = S − {S[m_2]};
8           n_{m_1} = n_{m_1} + n_{m_2};
```

时间复杂度：设序列中共有 n 个元素，则每次归并排序的时间复杂度为

$O(n)$。共进行 $k-1$ 次归并排序，每次归并排序前寻找最短的两个序列需要 $O(k)$ 时间，$k \leq n$。
因此，总的时间复杂度为 $(k-1)*(O(n)+O(k))=O(nk)$。

4. 设计分治算法求解如下问题, 并分析算法的复杂性。
输入: 字符表 $C=\{c_1,...,c_n\}$ 的前缀编码 $code(c_1),...,code(c_n)$
输出: 与前缀编码 $code(c_1),...,code(c_n)$ 对应的编码树 T
解：
前缀编码保证了，编码越长的字符，位于编码树中的层数越深。
Divide:
1. 如果 C 中只有一个字符，则将返回该字符构成的节点。
2. 通过扫描字符表中每个字符的前缀编码的第一个编码，如果为 0 则删除第一个编码并且放入新的集合 C_1 中，如果为 1 则删除第一个编码并且放入另外一个新的集合 C_2 中。
3. 对子集合 C_1 和 C_2 递归调用，产生编码树 T_1 和 T_2。
Merge:
1. 建立一个新的节点，用这个节点作为树的根，并且将 T_1 作为该节点的左子树，将 T_2 作为该节点的右子树，形成一棵新的树。
下面给出伪代码：

CREATE-CODETREE$(C, code(C))$
1 $n \leftarrow C.length$
2 **if** $n == 1$
3 **then return** CREATE-NODE$(C[1])$
4 $C_1 \leftarrow \varnothing$,$C_1 \leftarrow \varnothing$
5 **for** $i \leftarrow 1$ **to** n
6 **do if** $code(C[i])[1] == 0$
7 **then** $C_1 \leftarrow C_1 \bigcup C[i]$
8 **else** $C_2 \leftarrow C_2 \bigcup C[i]$
9 $code(C[i]) \leftarrow code(C[i])[2:k]$
10 $tree_1 \leftarrow$ CREATE-CODETREE$(C_1, code(C_1))$
11 $tree_2 \leftarrow$ CREATE-CODETREE$(C_2, code(C_2))$
12 $node \leftarrow$ CREATE-NODE(0)
13 CREATE-TREE$(node, tree_1, tree_2)$

算法的复杂度：最好的情况为 $T(n)=O(n\log n)$，最坏的情况为 $T(n)=O(n^2)$，平均的情况为 $T(n)=O(n\log n)$。

5. 给定两个大小为 n 的正整数集合 A 和 B。对于 A 到 B 的一个一一映射 f, 不妨设 $f(a_i)=b_i(i=1,...,n)$, 则 f 的代价为 $\sum_{i=1}^{n} a_i^{b_i}$。试设计一个贪心算法, 找出从 A 到 B 的代价最大的一一映射。
解：
贪心思想：每次选择 A 集合中的最大元素映射为 B 集合中最大的元素。
贪心选择性：假设集合 A 和 B 已经进行了升序排序。
即需要证明存在某个优化解，使得映射 $f(a_n)=b_n$
证明：假设 F_1 是该问题的一个优化解，如果 F_1 已经包含了 $f(a_n)=b_n$，则

命题得证。
否则，如果 F_1 不包含 $f(a_n) = b_n$，则假设 $f(a_n) = b_k$，$f(a_t) = b_n$, 其中 $b_k \leq b_n, a_t \leq a_n$。
又因为，$a_n^{b_n} + a_t^{b_k} - (a_n^{b_k} + a_t^{b_n}) = a_n^{b_k}(a_n^{b_n - b_k} - 1) - a_t^{b_k}(a_t^{b_n - b_k} - 1) \geq 0$，即 $a_n^{b_n} + a_t^{b_k} \geq a_n^{b_k} + a_t^{b_n}$，所以，如果将这个优化解中的 $f(a_n) = b_k$，$f(a_t) = b_n$ 转换为 $f(a_n) = b_n$，$f(a_t) = b_k$ 那它还将是一个优化解。所以不符合条件要求的优化解也可以转化程符合要求的解。
综上所述，命题得证。
优化子结构：即证明对于一个包含 $f(a_n) = b_n$ 的优化解 F_1,$F_1' = F_1 - \{(a_n, b_n)\}$ 也是 $(A - \{a_n\}, B - \{b_n\})$ 的一个优化解。
证明：假设对于问题 $(A - \{a_n\}, B - \{b_n\})$，$F_1'$ 不是优化解，那么设存在优化解 F_2'，并且假设优化解 F_2' 的代价为 W，那么对于问题 (A, B) 存在一个解 $F_2 = F_2' \bigcup \{f(a_n) = b_n\}$，$W_{F_2} = W_{F_2'} + a_n^{b_n} > W_{F_1'} + a_n^{b_n} = W_{F_1}$，这显然与 F_1 是优化解相矛盾，所以假设不正确，所以优化子结构得证。
算法正确性：综上所述，我们每次都取两个数列中最大的两个数进行映射，则每次都能拿到最优解，然后子问题的最优解又构成了大问题的最优解，所以利用数学归纳法显然得出算法正确无误。
给出算法的伪代码：

```
BestF(A,B)
1  MergeSort(A)
2  MergeSort(B)
3  for i ← n to 1
4       do
5            f(A[i]) = B[i]
6  return f
```

算法的复杂度：归并排序的时间复杂度为 $O(n \log n)$，建立映射所需要的时间为 $O(n)$。
所以总的时间复杂度为 $O(n \log n)$。

6. 一个 DNA 序列 X 是字符集 G, T, A, C 上的串, 其上有大量信息冗余。设 x 是 X 的子串,x 及其冗余形式在 X 内在出现的起、止位置构成了一系列等长区间 $[p_1, q_1], ..., [p_m, q_m]$。试设计一个贪心算法找出 $[p_1, q_1], ..., [p_m, q_m]$ 中互不相交的区间的最大个数, 即确定 x 的独立冗余度。
解：
贪心思想：将区间按照起始位置 q_i 排序，每次选择与找出的区间不相交的，并且 q_i 最小的区间，加入选择的区间。
贪心选择性：即证明存在某个优化解，会包含区间 $L_1 = [p_1, q_1]$
证明：假设 A 是该问题的一个优化解，设其第一个区间是 L_k, 第二个区间是 L_j;
如果 $k = 1$，则无需证明；
如果 $k \neq 1$，则设 $B = A - \{L_k\} \cup \{L_1\}$;
由于 A 中序列均不相交，并且 $q_1 \leq q_k \leq p_j$，所以 B 中序列均不相交。
又由于 $|A| = |B|$, 所以 B 也是一个优化解，且包括序列 L_1
所以，得证。

优化子结构：即证明对于一个包含 L_1 的优化解 A，设 $A' = A - \{L_1\}$，那么 A' 是 $X' = \{i \in X | p_i \geq q_1\}$ 的优化解。
证明：如果对于问题 X'，A' 不是优化解，则假设其优化解为 B'；那么对于问题 X，存在 $B = B' \cup \{L_1\}$, 显然 B 中所有区间不相交，它是 X 的一个解。那么，$|B| = |B'| + 1 > |A'| + 1 = |A|$, 这与 A 是最优解矛盾。所以假设不正确，所以原命题成立。

算法正确性证明：当 $|A| = 1$ 时，由贪心选择性，得出算法正确；
当 $|A| < k$ 时，假设算法正确；
当 $|A| = k$ 时，由优化子结构得到 $A = \{L_1\} \cup A_{|A|=k}$，显然，算法正确。

```
FIND-REDUNDANCY()
1  lastQ ← 0
2  for i ← 1 to m
3        do
4           if p[i] ≥ lastQ
5             then
6                    Count++
7                    lastQ ← q[i]
8  return Count
```

时间复杂度：$T(n) = O(n)$。

7. 背包问题定义如下, 输入背包容量 C 和 n 个物品, 其中第 i 个物品 $(1 \leq i \leq n)$ 的重量为 w_i 且其价值为 v_i , 试设计一个贪心算法输出向量 $< x_1, ..., x_n >$ 使得 $0 \leq x_i \leq 1 (1 \leq i \leq n)$ 且 $\sum_{i=1} x_i v_i$ 达到最大值。
解：
贪心思想：每次选择当前可选择的使得 v_i/W_i 最大的 i。
算法正确性分析：
贪心选择性：设 i_0 是使得 v_i/W_i 最大的 i，则存在问题的一个最优解，使得 $X_{i_0} = \min\{1, C/W_{i_0}\}$。
证明：设问题存在一个最优解 $< x_1, ..., x_n >$，并且 $S_m = \sum_{1 \leq i \leq n} v_i X_i$ ，若 $X_{i_0} = \min\{1, C/W_{i_0}\}$，得证。
若 $X_{i_0} < \min\{1, C/W_{i_0}\}$, 令 $X'_{i_0} = \min\{1, C/W_{i_0}\}$，假设 $W_{i_0}(X'_{i_0} - X_{i_0})$ 的空间分配给 k 了，在 $m = C - W_{i_0} X'_{i_0}$ 的部分保持上述最优解不变，只是将分配给 k 的空间重新划分给 i_0，将得到问题的一个新的解 S'_m。
则 $S'_m - S_m = W_{i_0}(X'_{i_0} - X_{i_0}) \times (v_{i_0}/W_{i_0} - v_k/W_k)$，又因为 $v_{i_0}/W_{i_0} \geq v_k/W_k$，那么在新的解下有 $S'_m \geq S_m$，所以新的解也是一个优化解。得证。
优化子结构：设 $X_{i_0} = \min\{1, C/W_{i_0}\}$，$\{X_1, X_2, ..., X_n\}$ 是问题的最优解且 $X_{i_0} = \min\{1, C/W_{i_0}\}$。若 $M > W_{i_0} X_{i_0}$，那么 $\{X_1, X_2, ..., X_n\} - \{X_{i_0}\}$ 是子问题 $\max \sum_{\substack{1 \leq i \leq n \\ i \neq i_0}} v_i X_i$，$\sum_{\substack{1 \leq i \leq n \\ i \neq i_0}} W_i X_i \leq C - W_{i_0} X_{i_0}$ 的最优解。
证明：
容易得到，$\{X_1, X_2, ..., X_n\} - \{X_{i_0}\}$ 是子问题的一个解。
假设 $\{X_1, X_2, ..., X_n\} - \{X_{i_0}\}$ 不是子问题的最优解，那么存在子问题的一个

最优解 $\{X'_1, X'_2, \ldots, X'_n\} - \{X_{i_0}\}$，使得

$$\max \sum_{\substack{1\le i\le n\\ i\neq i_0}} v_i X'_i > \max \sum_{\substack{1\le i\le n\\ i\neq i_0}} v_i X_i$$

因而 $\{X'_1, X'_2, ..., X_{i_0}, ..., X'_n\}$ 是原问题的一个解，且

$$v_{i_0} X_{i_0} + \max \sum_{\substack{1\le i\le n\\ i\neq i_0}} v_i X'_i > v_{i_0} X_{i_0} + \max \sum_{\substack{1\le i\le n\\ i\neq i_0}} v_i X_i$$

即得到原问题的一个更优的解，与 $\{X_1, X_2, ..., X_n\}$ 是问题的最优解矛盾。
算法正确性：设 $v_1/W_1 \le v_2/W_2 \le \cdots \le v_n/W_n$,$X_i = \min\{1, (C - \sum_{1\le j\le i-1} W_j X_j)/W_i\}$, 则 $\{X_1, X_2, ..., X_n\}$ 是背包问题的最优解。
证明：由 (1)(2)，由数学归纳法易证结论成立。
算法的伪代码：

BAG-GREEDY$(v_1, v_2, \ldots, v_n, W_1, W_2, \ldots W_n, M)$

```
1  S ← 0
2  for i ← 1 to n
3        do X_i ← 0
4  for i ← 1 to n
5        do
6           if M = 0
7              then return S;
8           i ← argmax(v_i/W_i);
9           if M > W_i
10             then X_i = 1;
11             else  X_i = M/W_i;
12          M = M − W_i X_i;
13          S = S + W_i v_i;
14 return S;
```

时间复杂性：$T(n) = O(n \log n)$。

8. 给定平面点集 $P = \{(x_i, y_i) | 1 \le i \le m\}$ 和 $Q = \{(x_j, y_j) | 1 \le j \le n\}$。$(x_i, y_i) \in P$ 支配 $(x_j, y_j) \in Q$ 当且仅当 $x_i \ge x_j$ 且 $y_i \ge y_j$。试设计一个贪心算法输出集合 $\{(p, q) | p \in P, q \in Q, p$支配$q\}$ 使得该集合中点对最多。
解：
贪心思想：在点集 P 中选择 x-坐标最小的点 p_i，如果存在多个，则选择 y-坐标最小的那个点，构造集合 $Q_i = \{q | q \in Q, p_i$支配$q\}$，如果集合为空，则在集合 P 中删除点 p_i，否则，在 Q_i 中选择 y-坐标最大的点 q_i，如果存在多个，则选择 x-坐标最大的那个点，选择点对 (p_i, q_i) 加入结果。
贪心选择性：在点集 P 中选择 x-坐标最小的点 p_i，如果存在多个，则选择 y-坐标最小的那个点，构造集合 $Q_i = \{q | q \in Q, p_i$支配$q\}$，如果集合为空，则在集合 P 中删除点 p_i，否则，在 Q_i 中选择 y-坐标最大的点 q_j，如果存在多个，则选择 x-坐标最大的那个点，则一定存在最优解包含点对 (p_i, q_j)。
证明：设存在一个最优解 $C^* = \{(p, q) | p \in P, q \in Q, p$支配$q\}$，并且 $|C^*|$ 最

大。
如果 $(p_i, q_j) \in C^*$，则结论成立。
否则的话，如果 C^* 中不存在 $\{(p_i, q_m)|q_m \in Q\}$ 和 $\{(p_k, q_j)|p_k \in P\}$ 这样的点对，得到 $C = C^* \bigcup (p_i, q_j)$，并且 $|C| > |C^*|$，矛盾。
如果 C^* 中存在 $\{(p_i, q_m)|q_m \in Q\}$ 这样的点对，但是不存在 $\{(p_k, q_j)|p_k \in P\}$ 这样的点对，得到 $C = (C^* - (p_i, q_m)) \bigcup (p_i, q_j)$，并且 $|C| = |C^*|$，则 C 也为最优解。
如果 C^* 中存在 $\{(p_k, q_j)|p_k \in P\}$ 这样的点对，但是不存在 $\{(p_i, q_m)|q_m \in Q\}$ 这样的点对，得到 $C = (C^* - (p_k, q_j)) \bigcup (p_i, q_j)$，并且 $|C| = |C^*|$，则 C 也为最优解。
如果 C^* 中存在 $\{(p_i, q_m)|q_m \in Q\}$ 和 $\{(p_k, q_j)|p_k \in P\}$ 这样的点对，我们可以得到 $y_j \leq y_k$，又因为 $y_m \leq y_j$，所以 $y_m \leq y_k$。还可以得到 $x_m \leq x_i$，又因为 $x_i \leq x_k$，所以 $x_m \leq x_k$。综上所述，可以知道，p_k 支配着 q_m。构造 $C = (C^* - (p_i, q_m) - (p_k, q_j)) \bigcup (p_i, q_j) \bigcup (p_k, q_m)$，并且 $|C| = |C^*|$，则 C 也为最优解。
优化子结构：$C^* - (p_i, q_j)$ 是子问题 $P - \{p_i\}$,$Q - \{q_j\}$ 的最优解。
证明：如果不是的话，设子问题的最优解为 C'，可以得知：$|C'| > |C^*| - 1$，则可以构造 $C'^* = C' \bigcup (p_i, q_j)$ 为原问题的解，并且 $|C'^*| = |C'| + 1 > |C^*|$。与 C^* 是最优解矛盾。
下面给出伪代码：

```
FIND-DOTPAIRS(P, Q)
1  count ← 0, result ← ∅
2  while FIND-FIRSTPAIRS(P,Q) ≠ 0
3        do (p_i, q_j) ← FIND-FIRSTPAIRS(P, Q)
4           count ← count + 1
5           result ← result ⋃ (p_i, q_j)
6           P ← P − {p_i}
7           Q ← Q − {q_j}
8  return count, result
```

时间复杂度为：$T(n) = O(n \log n)$。

9. 某工厂收到 n 个订单 (a_i, b_i), 其中 a_i 和 b_i 均是正整数 $(1 \leq i \leq n)$, 订单 (a_i, b_i) 希望在时间 b_i 之前获得 a_i 件产品。工厂的生产能力为每个时间单位生产 1 件产品。工厂希望拒绝最少数量的订单, 并恰当地排序剩下的订单使得剩下的订单均能够被满足。试设计一个贪心算法求解上述问题。
解：
贪心思想：选择当前满足条件的订单中 a_i 最小的订单 (a_i, b_i)。
用集合 $R = \{C|C$为订单的集合且C中订单都能满足$\}$，我们的目标就是找到 R 中长度最长的订单集合。
引理：假设 A 为最优解，A 中满足条件的订单调度中，第 k 个完成的订单为 (a_m, b_m)，第 k+1 个完成的订单为 (a_l, b_l)，如果 $b_m \geq b_l$，则调换订单 m 和订单 l 的顺序也能满足条件。
证明：用 t_i 表示在 A 中，第 i 个调度的任务。
由题意可知，$b_m \geq \sum_{i=1}^{k-1} a_{t_i} + a_m$ 和 $b_l \geq \sum_{i=1}^{k-1} a_{t_i} + a_m + a_l$。

将两个订单调换之后，我们可以得到 $b_m \geq b_l \geq \sum_{i=1}^{k-1} a_{t_i} + a_l + a_m$ 和 $b_l \geq \sum_{i=1}^{k-1} a_{t_i} + a_m + a_l > \sum_{i=1}^{k-1} a_{t_i} + a_l$。而在 A 中其余的订单都没有变化，所以命题得证。

贪心选择性：将收到的 n 个订单按照 a_i 排序，设 $a_1 \leq a_2 \leq \cdots \leq a_n$，订单选择问题的某个最优解包括订单 (a_1, b_1)。

证明：假设 A 是一个最优解，如果 $(a_1, b_1) \in A$，则命题得证。

否则的话，设满足的第一个订单为 (a_k, b_k)，必然有 $a_1 \leq a_k$，A 中排在 k 订单之后的任意的订单 (a_j, b_j)，满足 $b_j \geq a_k + \sum_{i在j之前} a_i$。用 (a_1, b_1) 代替 (a_k, b_k) 的位置，我们可以得到一个新的解 $A' = (A - (a_k, b_k)) \bigcup (a_1, b_1)$，其中对于 A' 中排在 1 订单之后的任意订单 (a_j, b_j)，满足 $b_j \geq a_k + \sum_{i在j之前} a_i \geq a_1 + \sum_{i在j之前} a_i$，所以 A' 也是满足条件的解，并且 $|A| == |A'|$，所以 A' 也是最优解。命题得证。

优化子结构：$A - (a_1, b_1)$ 是 $S = \{C \subseteq A - (a_1, b_1) : C \bigcup (a_1, b_1) \in R\}$ 的最优解。其中由引理可得，已知一个集合 S 能被满足，S 中的排序可以按照 b_i 的大小排序满足的。

证明：可以很容易得到 $\forall (a_i, b_i) \in (A - (a_1, b_1)) \Rightarrow (a_i, b_i) \in S$，所以 $A - (a_1, b_1)$ 是 S 的一个解。

如果不是最优解，那么存在一个最优解 $B' \in S$，使得 $|A - (a_1, b_1)| < |B'|$，那么我们可以得到一个集合 $A' = B' \bigcup (a_1, b_1)$，由已知条件很容易得到 A' 为原问题的解，并且 $|A'| = |B'| + 1 > |A - (a_1, b_1)| + 1 = |A|$，矛盾。

下面给出伪代码：假设订单按照 A 的顺序排好序了。

```
FIND-ORDER(A, B)
1  n ← A.length
2  order ← ∅
3  for i ← 1 to n
4        do if IS-SATISFY(order ⋃ (a_i, b_i))
5              then order ← order ⋃ (a_i, b_i)
6  return order
```

```
IS-SATISFY(S)
1  S ← SORT-B(S, 2)
2  n ← S.length, A ← S[1], B ← S[2]
3  for i ← 1 to n
4        do if B[i] < Σ_{j=1}^{i} A[j]
5              then return False
6  return True
```

在判断是否为满足的订单时，采用动态规划的方法，可以使得复杂度降为 $O(n \log n)$，所以总的时间复杂度为：$T(n) = O(n^2 \log n)$。

10. 输入 n 个区间 $[a_i, b_i]$, 其端点满足 $1 \le a_i \le i \le b_i \le n$, 试设计一个贪心算法选出最少区间覆盖 $[1, n]$。

解：

贪心思想：每次在能覆盖当前最左边端点的所有区间中，选择 b_i 最大的区间。

贪心选择性： 设能够覆盖 1 的所有区间中，$[a_k, b_k]$ 是 b_i 最大的区间，则一定存在一个最优解包含 $[a_k, b_k]$。

证明：在能够覆盖当前最左边端点的所有区间中，最优解必然会包含其中一个区间，否则的话，端点 1 将不会被覆盖，得不到最优解。

假设一个最优解 A 能包含 $[1, n]$，如果 $[a_k, b_k] \in A$，则命题成立。

否则，假设 $[a_m, b_m] \in A$，构造一个新的解 $A' = (A - [a_m, b_m]) \bigcup [a_k, b_k]$，其中 A' 也能包含 $[1, n]$，并且 $|A| == |A'|$。所以，A' 也是最优解。命题得证。

优化子结构： 设 A 是原问题的最优解，第一次选择了能覆盖端点 1 的 $[a_k, b_k]$，则 $A - [a_k, b_k]$ 能覆盖 $[b_k, n]$ 的最少区间。

证明：设能覆盖 $[b_k, n]$ 的最优解为 S，很容易可以得到，$\forall [a_i, b_i] \in S \Rightarrow [a_i, b_i] \in (A - [a_k, b_k])$。

如果 S 不是子问题的最优解，设 S' 为子问题的最优解，S' 能覆盖 $[b_k, n]$，且 $|S'| < |S|$。所以 $A' = S' \bigcup [a_k, b_k]$ 能覆盖区间 $[1, n]$，且 $|A'| = |S'| + 1 > |S| + 1 = |A|$。与 A 为最优解矛盾。

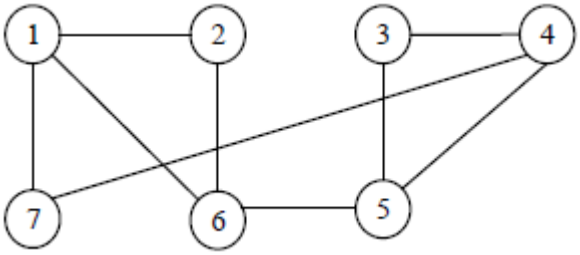
下面给出伪代码：

假设 B 已经按照递减的顺序排好序了。

```
FIND-REGION(A, B)
 1  n ← A.length
 2  left ← 1
 3  result ← ∅
 4  while left < n
 5        do for i ← 1 to n
 6              do if A[i] ≤ left
 7                    then left ← B[i]
 8                         result ← result ⋃ (A[i], B[i])
 9
10  return result
```

时间复杂度为排序的时间复杂度：$T(n) = O(n \log n)$。

8.1在下图中考虑哈密顿环问题。将问题的解空间表示成树，并分别利用深度优先搜索和广度优先搜索判定该图中是否存在哈密顿环。



算法思想如下：

利用邻接链表存储存储图结构，再分别利用深度优先和广度优先搜索哈密顿环。

算法描述如下：

CreateALGraph：创建邻接链表

输入：图$G(V,E)$,V表示顶点编号数组

输出：邻接链表$AdjList[1:|V|]$

1.$n \leftarrow G.|V|, e \leftarrow G.|E|$

2.*for* $i \leftarrow 0$ *to* n *Do*

3. $AdjList[i].vertex \leftarrow V[i]$ //读入顶点信息

4. $AdjList[i].firstedge \leftarrow NULL$ //边表置为空表

5.*for* $k \leftarrow 0$ *to* e *Do*

6. $cin >> e_i, cin >> e_j$ //读入边(e_i,e_j)的顶点对序号

7. $s.adjvex \leftarrow e_j$ //s是边表节点指针，$adjvex$存储边表节点编号，
//$next$存储指向下一节点的指针

8. $s.next \leftarrow AdjList[e_i].firstedge$

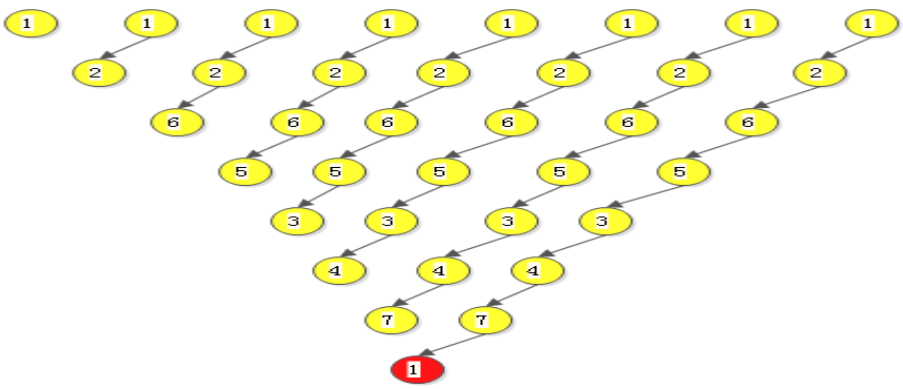
9. $AdjList[e_i].firstedge \leftarrow s$ //将新节点插入顶点e_i的边表头部

10. $s.adjvex \leftarrow e_i$

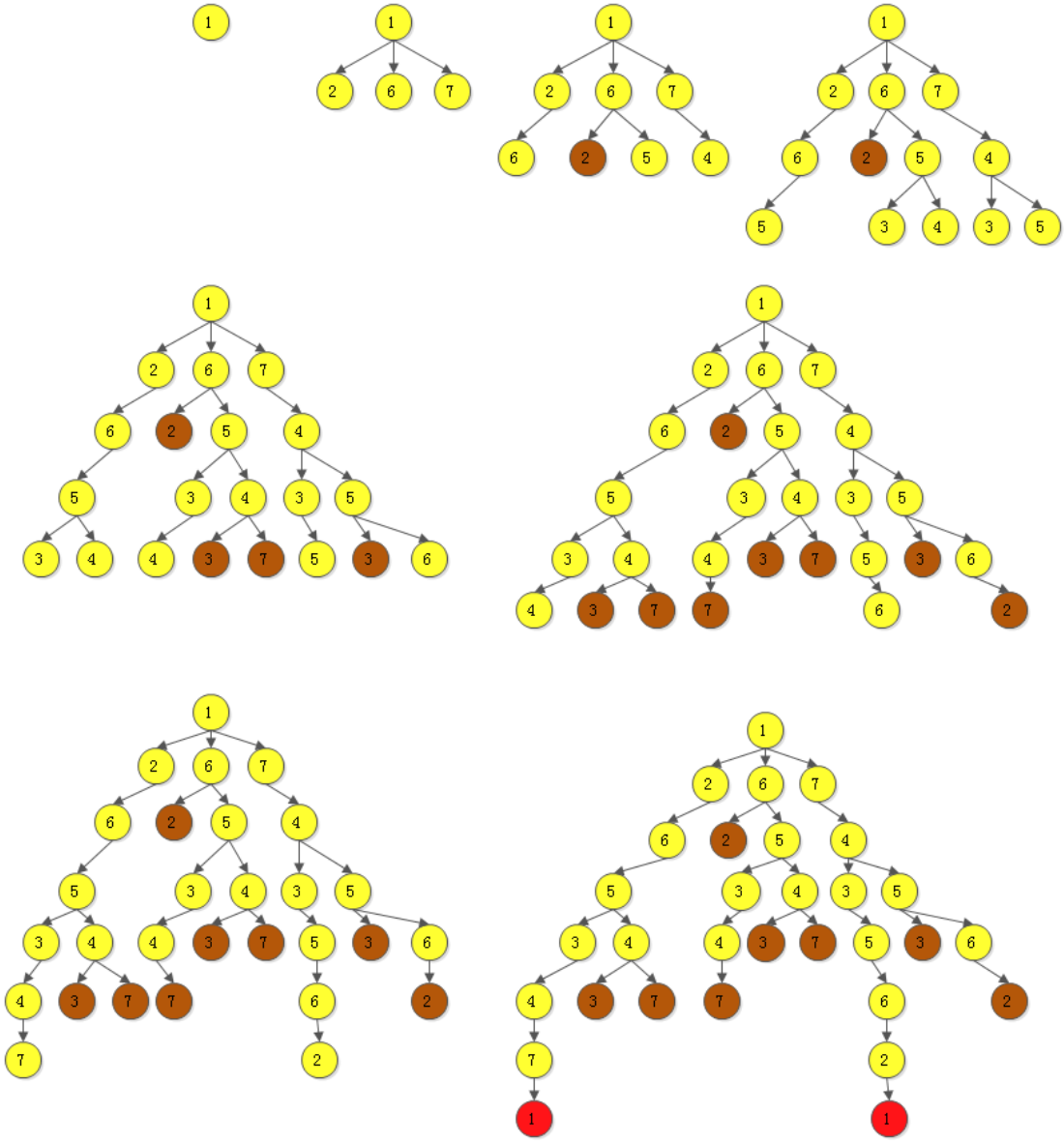
11. $s.next \leftarrow AdjList[e_j].firstedge$

12. $AdjList[e_j].firstedge \leftarrow s$ //将新节点插入顶点e_j的边表头部

深度优先搜索：



广度优先搜索：



2.考虑8－魔方问题。分别用深度优先方法，广度优先方法，爬山法，最佳优先方法判定上图所示的初始格局能够通过一系列操作转换为目标格局，将搜索过程的主要步骤书写清楚。

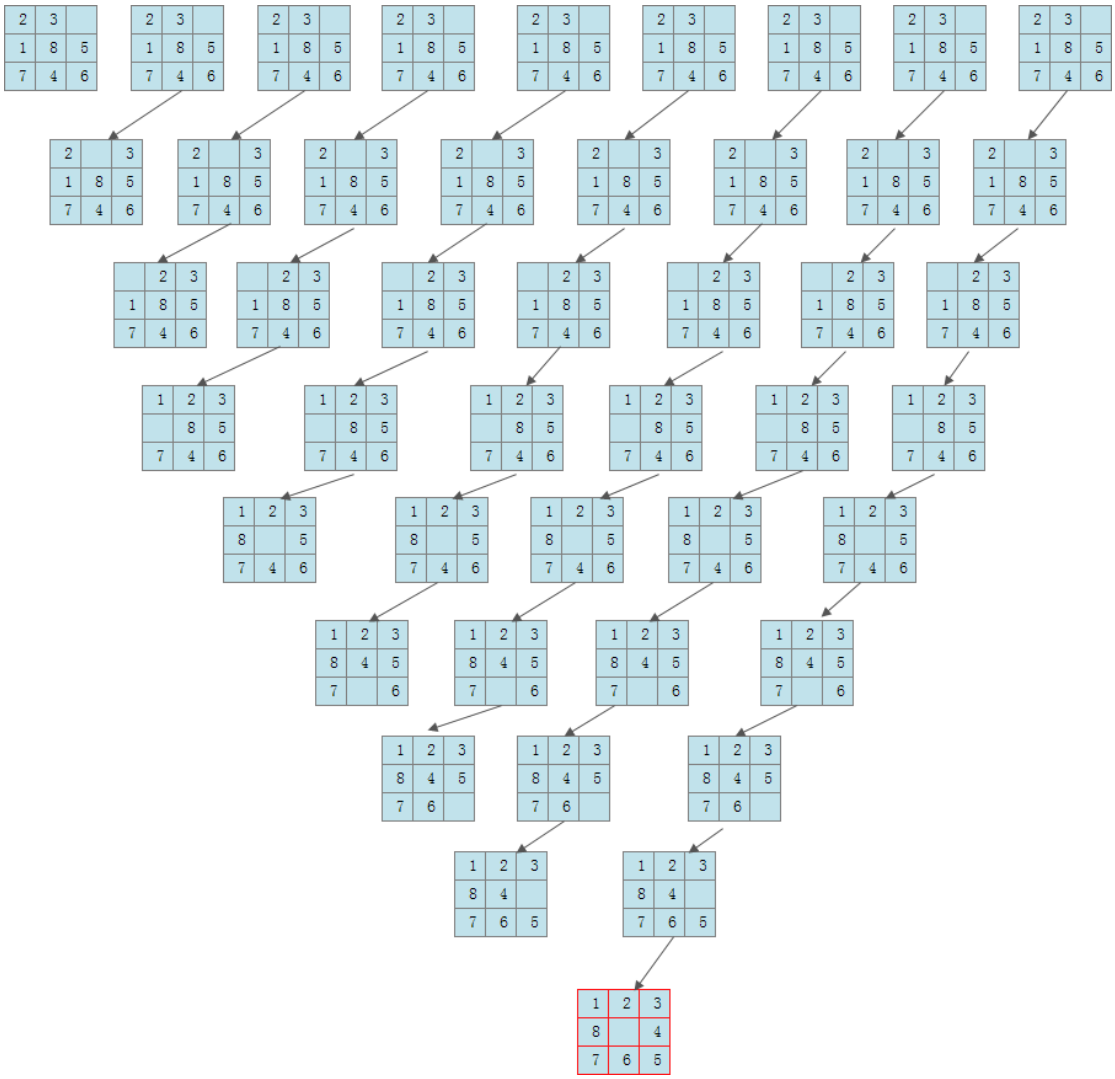
2	3	
1	8	5
7	4	6

起始格局

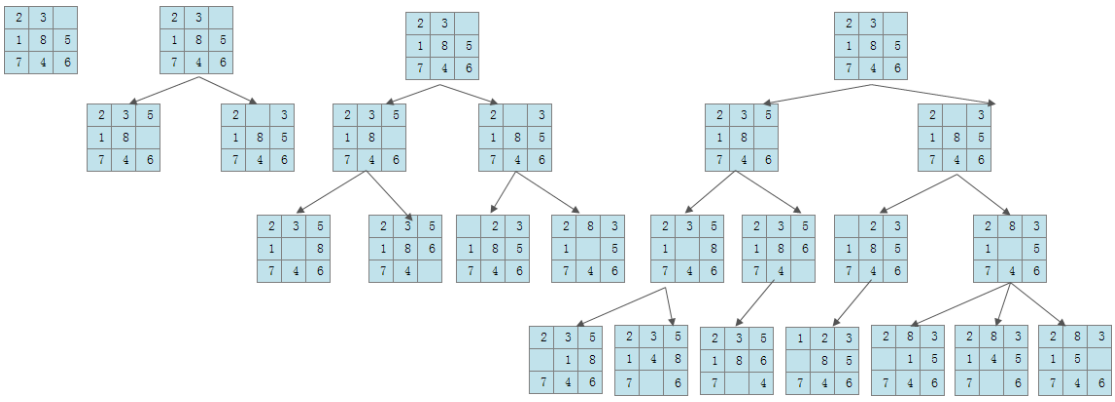
1	2	3
8		4
7	6	5

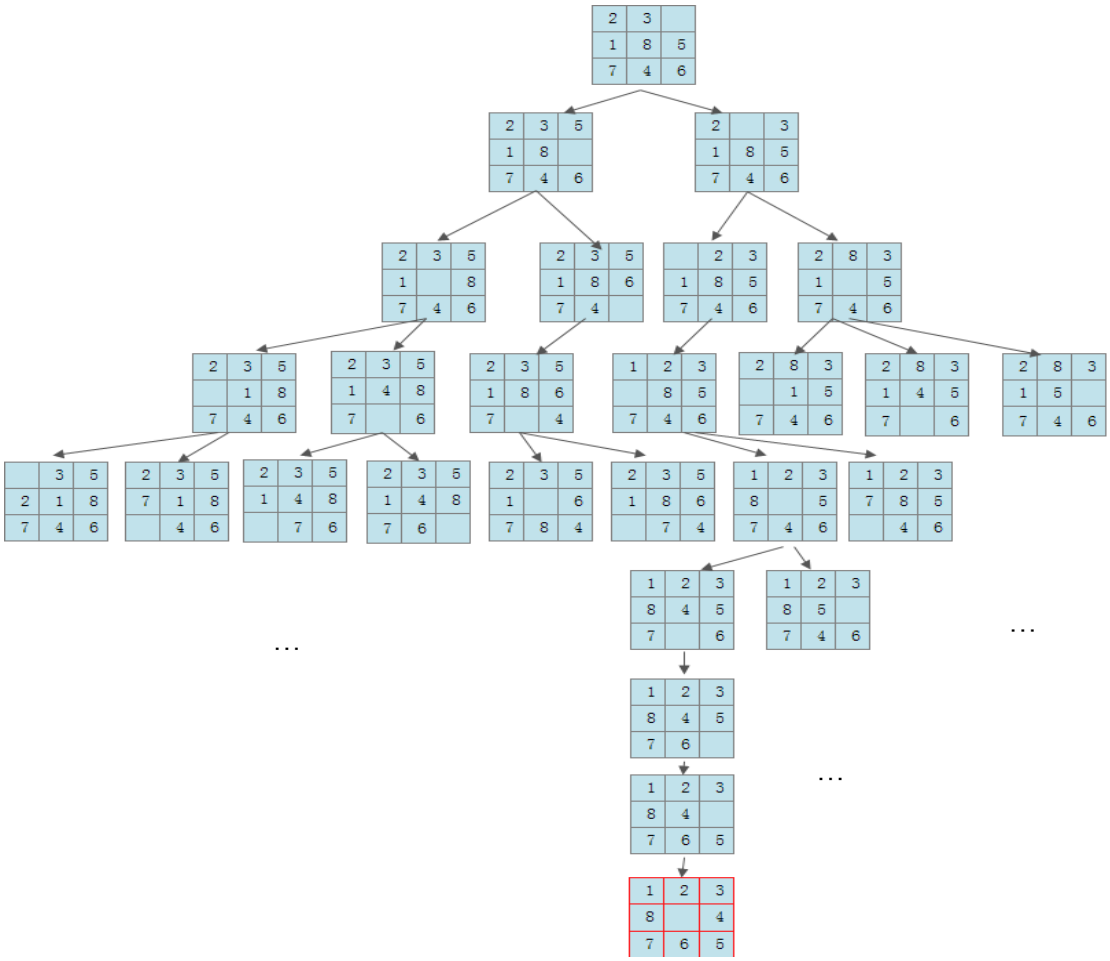
目标格局

深度优先：



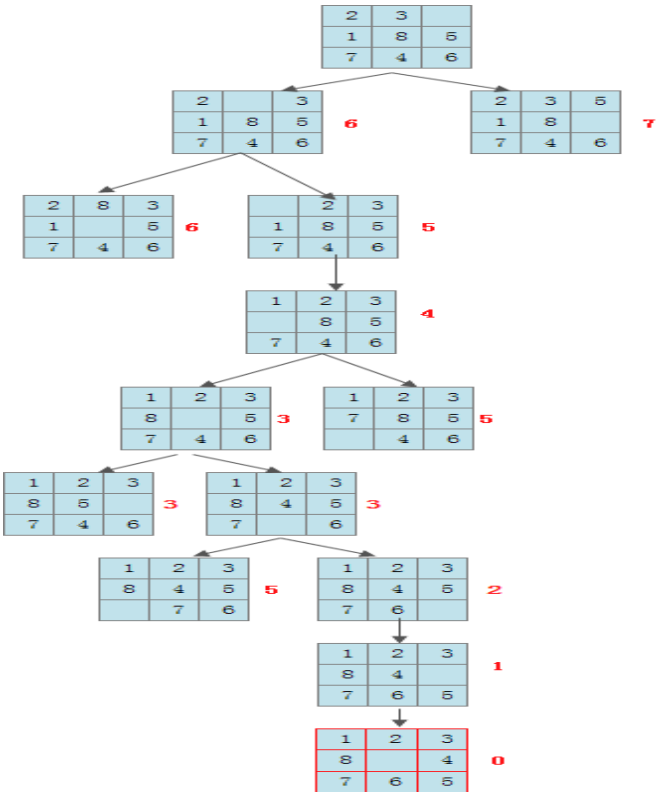
广度优先：



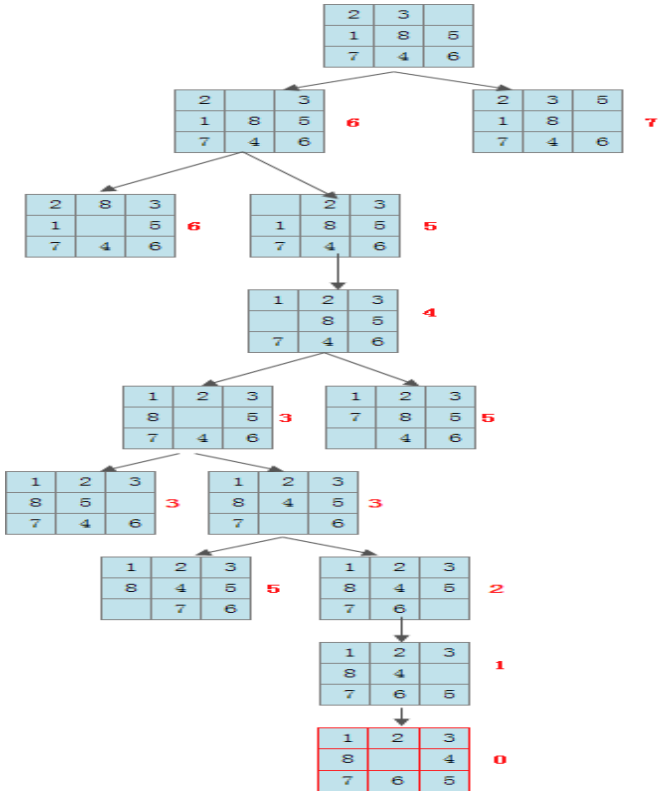


注意：（…）为省略的扩展步骤，因为步骤太多，只保留了关键搜索步骤。

爬山法：



最佳优先法：



对于深度和广度优先搜索而言，魔方的初始状态作为树的根节点，每次对节点的魔方中
变化一个数字的位置形成的新魔方作为当前节点的左右孩子，因此通过有限次转换数字位置
一定能得到具有目标格局的魔方。

广度优先搜索的步骤：

1. 构造由根组成的队列 Q；
2. If Q 的第一个元素 x 是目标节点 Then 停止;
3. 从 Q 中删除 x，把 x 的所有子节点加入 Q 的末尾;
4. If Q 空 Then 失败 Else goto 2.

深度优先搜索的步骤：

1. 构造一个由根构成的单元素栈 S;
2. If Top(S)是目标节点 Then 停止;
3. Pop(S)，把 Top(S)的所有子节点压入栈顶;
4. If S 空 Then 失败 Else goto 2.

对于爬山法而言，引入测度函数 f[n]=W[n]，W[n]表示节点 n 时位置错误的数字个数。在深度和广度优先搜索建立的求解问题树结构的基础上，每次只操作测度函数值小的那部分分支，沿着测度函数值下降的方向求解树结构，能减少一定不必要的求解操作，获得最终的目标结构。

爬山法的步骤：

1.构造由根组成的单元素栈 S;
2. If Top(S)是目标节点 Then 停止;
3. Pop(S);
4. S 的子节点按照其启发测度由大到小的顺序压入 S;

5. If S 空 Then 失败 Else goto 2.

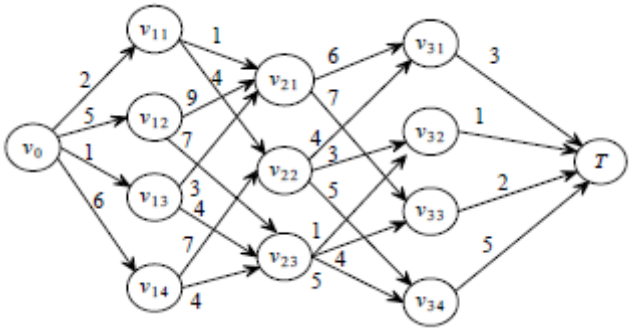
对于最佳优先法而言，是在爬山法的基础上用堆的形式实现，同样是沿着测度函数（评
价函数）下降的方向求解，只是在求解过程中，寻找每一层堆内最小的测度的子堆进行展开，
相比爬山法通过层内比较进一步减少了不必要的求解操作，获得最终的目标结构。

最佳优先搜索步骤：

1. 使用评价函数构造一个堆 H, 首先构造由根组成的单元素堆;
2. If H 的根 r 是目标节点 Then 停止;
3. 从 H 中删除 r, 把 r 的子节点插入 H;
4. If H 空 Then 失败 Else goto 2.

综上，利用深度优先搜索和广度优先搜索一定可以找到魔方问题的解，爬山法是在前两
者的基础上的一次优化，而最佳优先搜索则是在爬山法的基础上进一步优化，因此后两种方
法也一定能找到问题的解。

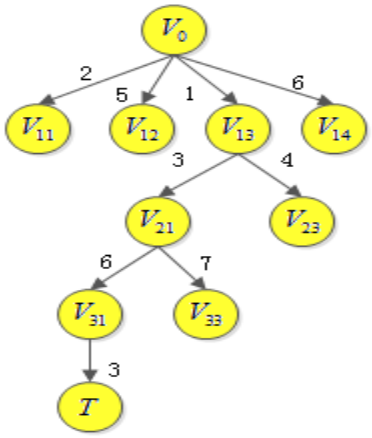
3.分别用分支限界法和A^*算法求出下图中从v_0到T的最短路径，写出算法执行的主要过程。



解：

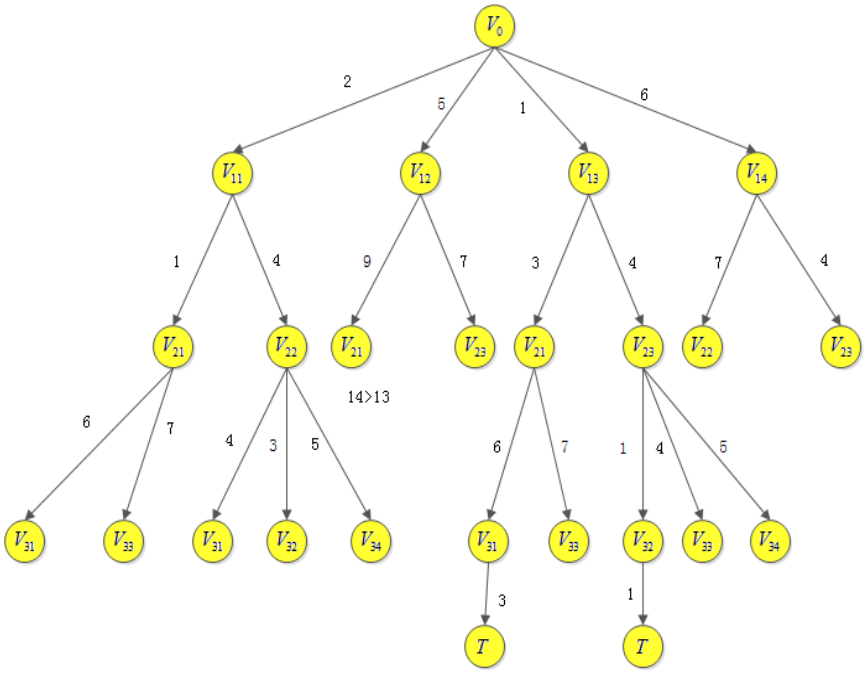
分支界限法：

（1）利用爬山法贪心的选择当前点所连接的最小边，并按此边进行展开。如下图所示，从v_0到T的最小路径为: $V_0-V_{13}-V_{21}-V_{31}-T$, 代价为13。



（2）以代价13为边界，对其他的分支按照爬山法进行展开。如果代价大于13则结束当前路径，如果找到一个最小的路径且代价小于13则对代价重新赋值，重复执行第二步骤，直到所有的点都被遍历,如下图所示，最小路径为：

$V_0-V_{13}-V_{23}-V_{32}-T$,代价为7



A^*算法：

*step*1：

$g(V_{11})=2 \quad h(V_{11})=Min\{1,4\}=1 \quad f(V_{11})=2+1=3$

$g(V_{12})=5 \quad h(V_{12})=Min\{9,7\}=7 \quad f(V_{12})=5+7=12$

$g(V_{13})=1 \quad h(V_{13})=Min\{3,4\}=3 \quad f(V_{13})=1+3=4$

$g(V_{14})=6 \quad h(V_{14})=Min\{7,4\}=4 \quad f(V_{14})=6+4=10$

*step*2：因为$f(V_{11})$最小，因此扩展V_{11}

$g(V_{21})=2+1=3 \quad h(V_{21})=Min\{6,7\}=6 \quad f(V_{11})=3+6=9$

$g(V_{11})=2+4=6 \quad h(V_{22})=Min\{4,3,5\}=3 \quad f(V_{22})=6+3=9$

*step*3：因为当前可扩展最小为$f(V_{13})$，因此扩展V_{13}

$g(V_{21})=1+3=4 \quad h(V_{21})=Min\{6,7\}=6 \quad f(V_{21})=4+6=10$

$g(V_{23})=1+4=5 \quad h(V_{23})=Min\{1,4,5\}=1 \quad f(V_{23})=5+1=6$

*step*4：因为当前可扩展最小为$f(V_{23})$，因此扩展V_{23}

$g(V_{32})=5+1=6 \quad h(V_{32})=Min\{1\}=1 \quad f(V_{32})=6+1=7$

$g(V_{33})=5+4=9 \quad h(V_{33})=Min\{2\}=2 \quad f(V_{33})=9+2=11$

$g(V_{34})=5+5=10 \quad h(V_{34})=Min\{5\}=5 \quad f(V_{34})=10+5=15$

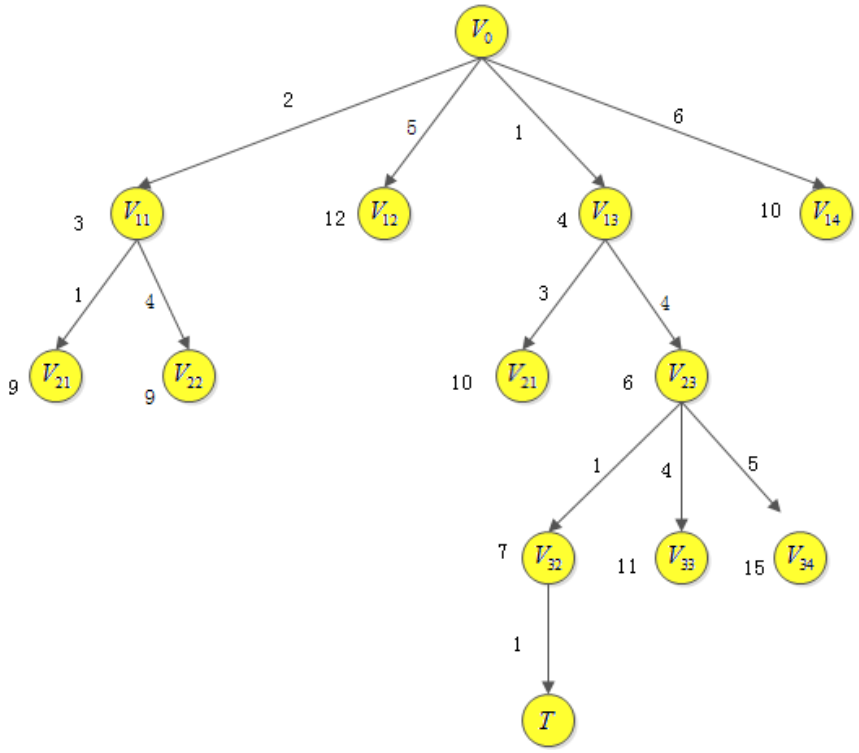
*step*5：因为当前可扩展最小为$f(V_{32})$，因此扩展V_{32}

$g(T)=6+1=7 \quad h(T)=0 \quad f(V_{11})=7+0=7$

*step*6.扩展T

因为T是目标节点，因此得到解为：$V_0-V_{13}-V_3-V_{32}-T$

如下图所示：



4.在上面邻接矩阵给出的有向图上，用分支限界法求出代价最小的哈密顿环。

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} \infty & 5 & 61 & 34 & 12 \\ 57 & \infty & 43 & 20 & 7 \\ 39 & 42 & \infty & 8 & 21 \\ 6 & 50 & 42 & \infty & 8 \\ 41 & 26 & 10 & 35 & \infty \end{pmatrix} \end{array}$$

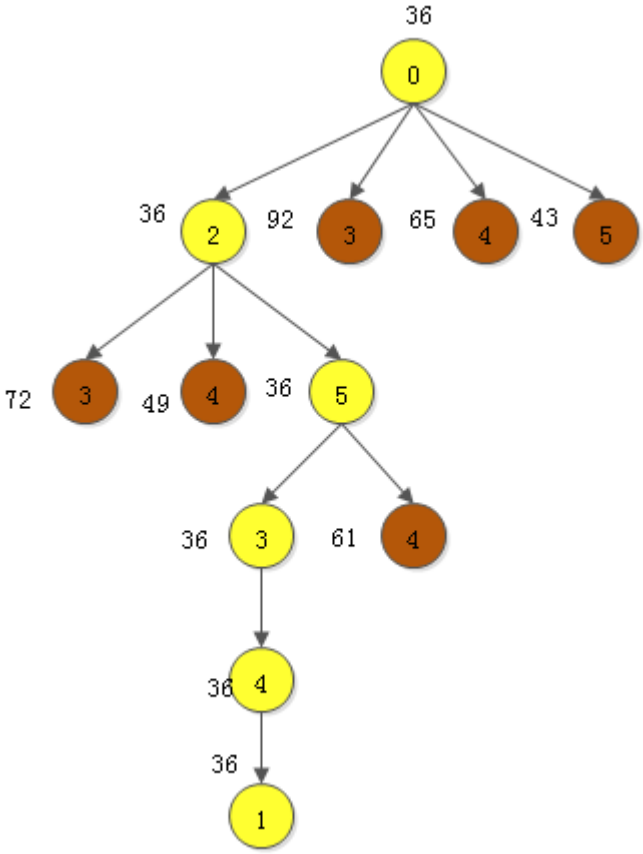
解：

(1)将代价矩阵的每行(列)减去同一个数(该行或列的最小数),使得每行和每列至少有一个零,其余各元素非负。

$$\begin{bmatrix} \infty & 5 & 61 & 34 & 12 \\ 57 & \infty & 43 & 20 & 7 \\ 39 & 42 & \infty & 8 & 21 \\ 6 & 50 & 42 & \infty & 8 \\ 41 & 26 & 10 & 35 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & 0 & 56 & 29 & 7 \\ 50 & \infty & 36 & 13 & 0 \\ 31 & 34 & \infty & 0 & 13 \\ 0 & 44 & 36 & \infty & 2 \\ 31 & 16 & 0 & 25 & \infty \end{bmatrix}$$

经过变换后解的下界为：5+7+8+6+10=36

(2)解空间的加权树：



代价最小的哈密顿环为:$v_1-v_2-v_5-v_3-v_4-v_1$

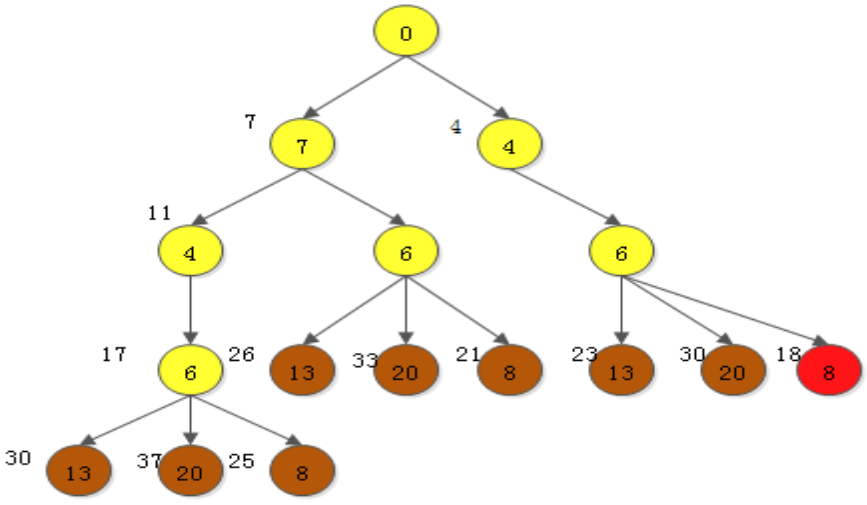
5.分别使用深度优先法和分支界限法求解子集和问题的如下实例。

输入：集合S={7, 4, 6, 13, 20, 8}和整数K=18

输出：$S^{'}\subseteq S$使得$S^{'}$中元素之和等于K

解：

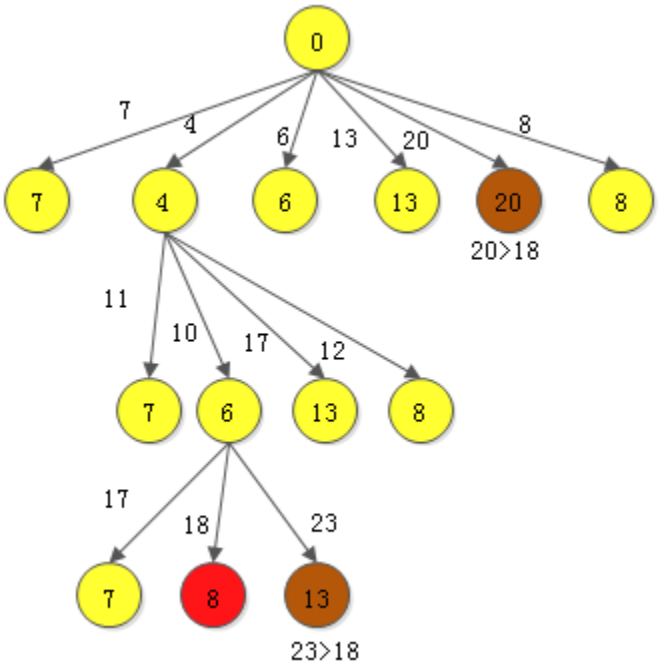
深度优先搜索：



分支界限法：

(1) 确定问题的上界为 18

(2) 利用爬山法进行问题搜索，每次选择当前可扩展的最小元素，并利用上界进行减枝。



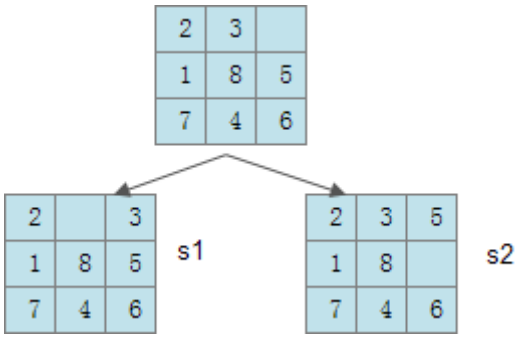
6.精确描述求解8-魔方问题的A^*算法，在习题2给出了起始格局和目标格局上给出A^*算法操作的主要步骤。

解:

(1)使用Best-first策略搜索树。

(2)节点n的代价函数为f(n)=g(n)+h(n),g(n)是初始魔方变换到当前魔方所产生的代价值，h(n)是当前魔方到终止魔方元素的差异个数。

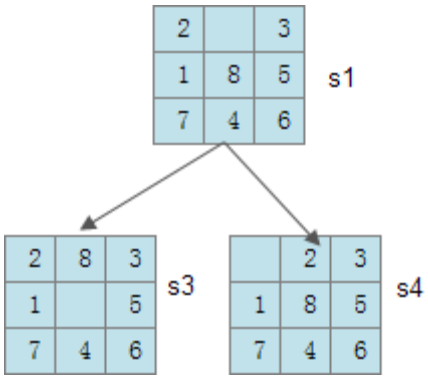
Step1:



$g(s1)=1 \quad h(s1)=6 \quad f(s1)=1+6=7$

$g(s2)=1 \quad h(s2)=7 \quad f(s2)=1+7=8$

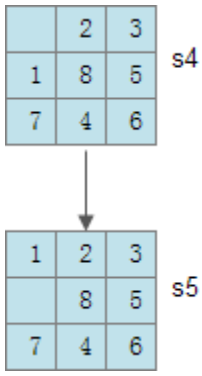
$step2$:因为$f(s1)$最小，扩展$s1$



$g(s3)=2 \quad h(s3)=6 \quad f(s3)=2+6=8$

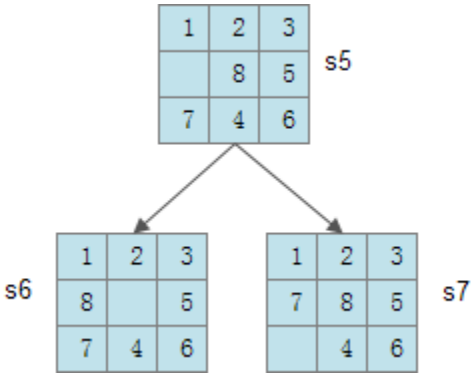
$g(s4)=2 \quad h(s4)=5 \quad f(s4)=2+5=7$

*step*3:因为$f(s4)$最小，扩展$s4$



$g(s5)=3 \quad h(s5)=4 \quad f(s5)=3+4=7$

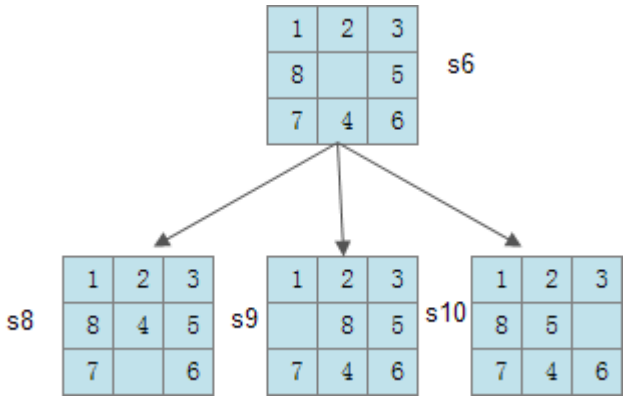
*step*4:因为$f(s5)$最小，扩展$s5$



$g(s6)=4 \quad h(s6)=3 \quad f(s11)=4+3=7$

$g(s7)=4 \quad h(s7)=5 \quad f(s7)=4+5=9$

*step*5:因为$f(s6)$最小，扩展$s6$

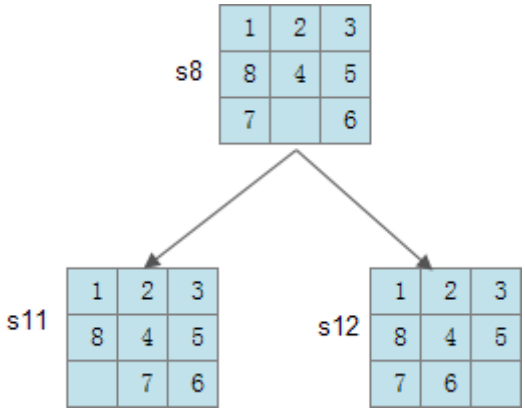


$g(s8)=5 \quad h(s8)=3 \quad f(s8)=5+3=8$

$g(s9)=5 \quad h(s9)=4 \quad f(s9)=4+4=9$

$g(s10)=5 \quad h(s10)=3 \quad f(s10)=5+3=8$

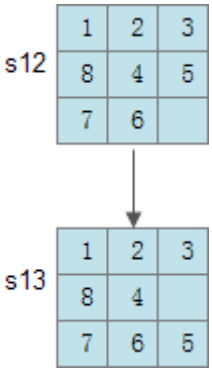
*step*6：根据最佳优先扩展，此时扩展$s8$



$g(s11)=6 \quad h(s11)=4 \quad f(s11)=6+4=10$

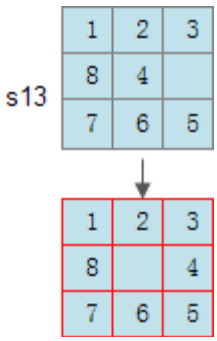
$g(s12)=6 \quad h(s12)=2 \quad f(s11)=6+2=8$

*step*7：根据最佳优先，此时扩展$s12$



$g(s13)=7 \quad h(s13)=1 \quad f(s13)=7+1=8$

*step*8：根据最佳优先，此时扩展$s13$



扩展完成，算法已经搜索到目标格局，算法结束。