

# 1 算法概述：

## 1.1 算法与程序

### 算法定义：

- 算法是指解决问题的一种方法或一个过程
- 算法是若干指令的有穷序列，其中每一条指令表示一个或多个操作
- 算法是求解一个问题类的无二义性的有穷过程

### 算法性质：

- (1) 输入：有0个或多个外部提供的量作为算法的输入
- (2) 输出：算法产生至少一个量作为输出
- (3) 确定性：组成算法的每条指令是清晰，无歧义的
- (4) 有限性：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的
- (5) 可行性：一个算法是能行的

### 算法描述方式：

自然语言、数学语言、伪代码、程序设计语言、流程图、表格、图示.....

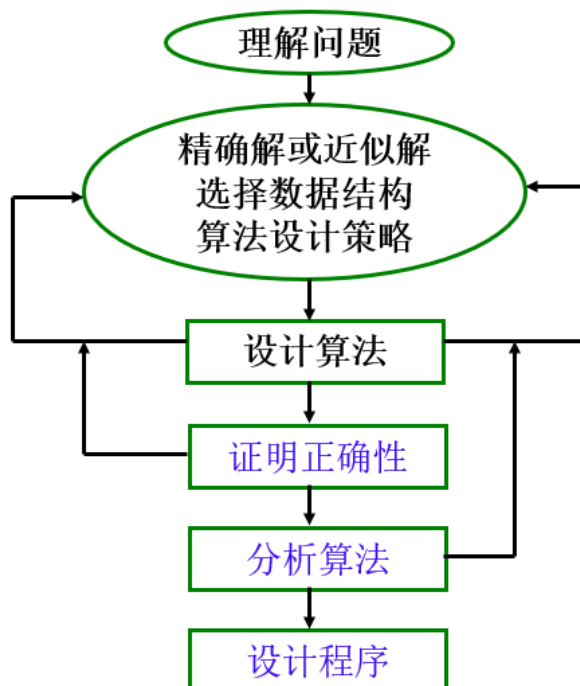
### 程序定义：

- 程序是算法用某种程序设计语言的具体实现
- 程序=算法+数据结构

### 算法与程序的联系与区别：

- 程序是算法用某种程序设计语言的具体实现
- 程序中的指令必须是机器可执行的，而算法中的指令则无此限制
- 程序可以不满足算法的性质（4），即有限性

### 算法求解过程：



## 1.2 算法复杂度分析

### 算法复杂性定义：

- 复杂性函数：  $C = F(N, I, A)$ 
  - C: 复杂性
  - N: 问题的规模
  - I: 算法的输入
  - A: 算法本身（通常让A隐含在复杂性函数名当中）
- 算法复杂性 = 算法所需要的计算机资源
  - 时间复杂性：需要时间资源的量，记为  $T = T(N, I)$
  - 空间复杂性：需要的空间资源的量，记为  $S = S(N, I)$

### 时间复杂性：

$$T(N, I) = \sum_{i=1}^k t_i e_i(N, I)$$

- $T(N, I)$  是时间复杂度，即所需的时间的量，是关于N和I的函数
- N、I：问题的规模、算法的输入
- $t_i$ ：第i个元运算所需的时间
- $e_i(N, I)$ ：第i个元运算的执行次数，是关于N和I的函数

利用某一算法处理一个问题规模为n的输入所需的时间，称为该算法的时间复杂性，记为  $T(n)$

### 渐进时间复杂性：

- 当问题的规模递增时，时间复杂性的极限称为渐进时间复杂性

$$\square T(n) \rightarrow \infty, \text{ as } n \rightarrow \infty;$$

$$\square (T(n) - t(n)) / T(n) \rightarrow 0, \text{ as } n \rightarrow \infty;$$

$\square t(n)$  是  $T(n)$  的 **渐近性态**，为算法的渐近复杂性。

$\square$  在数学上， $t(n)$  是  $T(n)$  当  $n \rightarrow \infty$  时的 **渐近表达式**。

- 当n趋近无穷时， $T(n)$  渐近于  $t(n)$ ，所以可以用  $t(n)$  替代  $T(n)$  作为算法A在n趋近于无穷时的时间复杂性的度量

### 渐进记号：

(1)  $O$ ：渐进上界记号

设函数  $f(n)$  和  $g(n)$  是定义在非负整数集合上的正函数， $f(n)$  的阶不高于  $g(n)$ ，如果存在两个非负常数  $c$  和  $n_0$ ，使得当  $n \geq n_0$  时，有  $f(n) \leq cg(n)$ ，则记做  $f(n) = O(g(n))$ ，称  $g(n)$  是  $f(n)$  的一个上界

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq \infty \text{ 蕴含着 } f(n) = O(g(n))$$

例：  $f(n) = 2n + 3$

当  $n \geq 3$  时，  $2n + 3 \leq 3n$

可选  $c = 3$ ，  $n_0 = 3$

对于  $n \geq n_0$ ，  $f(n) = 2n + 3 \leq 3n$

所以，  $f(n) = O(n)$

(2)  $\Omega$ : 渐进下界记号

设有函数 $f(n)$ 和 $g(n)$ 是定义在非负整数集合上的正函数,  $f(n)$ 的阶不低于 $g(n)$ , 如果存在两个非负常数  $c$ 和 $n_0$ , 使得当 $n \geq n_0$ 时, 有 $f(n) \geq c \cdot g(n)$ , 则记做 $f(n) = \Omega(g(n))$ , 称 $g(n)$ 是 $f(n)$ 的一个下界

例:  $f(n) = 2n + 3$

当 $n \geq 0$ 时,  $2n + 3 \geq 2n$

可选 $c = 2$ ,  $n_0 = 0$

对于 $n \geq n_0$ ,  $f(n) = 2n + 3 \geq 2n$

所以,  $f(n) = \Omega(n)$

(3)  $\Theta$ : 渐进确界记号

设有函数 $f(n)$ 和 $g(n)$ 是定义在非负整数集合上的正函数,  $f(n)$ 和 $g(n)$ 同阶, 如果存在非负常数 $c_1$ ,  $c_2$ 和 $n_0$ , 使得当 $n \geq n_0$ 时, 有 $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ , 则记做 $f(n) = \Theta(g(n))$

例:  $f(n) = 2n + 3$

当 $n \geq 3$ 时,  $3n \geq 2n + 3 \geq 2n$

可选 $c_1 = 3$ ,  $c_2 = 2$ ,  $n_0 = 3$

对于 $n \geq n_0$ ,  $3n \geq f(n) = 2n + 3 \geq 2n$

所以,  $f(n) = \Theta(n)$

**渐进阶的高低:**

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) \dots < O(2^n) < O(3^n) < \dots < O(n!)$

## 1.3 第一章测试

1. 解决问题的基本步骤是( )

(1) 算法设计(2)算法实现(3)数学建模(4)算法分析(5)正确性证明

答案: (3) (1) (5) (4) (2)

解释: 略

2. 算法的基本性质不包括( )

A. 外部提供的量作为输入

B. 算法产生的量作为输出

C. 算法指令确定、无歧义

D. 算法结构紧凑, 可读性强

答案: D

解释: 略

3. 下面关于程序和算法的说法正确的是 ( )

A. 算法是个过程, 计算机每次求解是针对问题的一个实例求解

B.算法的每一步骤必须要有确切的含义，必须是清楚的、无二义的

C.程序总是在有穷步的运算后终止

D.程序是算法用某种程序设计语的具体实现

答案：A B D

解释：略

4. 算法是一个语句集合，按照顺序执行语句，处理实例，得到正确答案( )

答案：对

解释：略

5. 若一个算法的计算时间为 $f(n) = 12n + 2n^2 \log n + 15n^3 + 3^n$ ，用O记号表示，则有 $f(n)$ 为( )

A.  $O(3^n)$

B.  $O(n^3)$

C.  $O(n^2 \log n)$

D.  $O(n)$

答案：A

解释：

$$\frac{12n + 2n^{2\log n} + 15n^3 + 3^n}{3^n} = 1 \neq \infty$$

6. 以下关于函数阶的关系中，哪几项是正确的 ( )

A.  $(\log n)^{\log n} = O(n^{\log \log n})$

B.  $\log(n!) = \theta(n \log n)$

C.  $2^{2n} = O(n^{2n})$

D.  $n \log n = \theta(n^2)$

答案：A B

解释：

$$A: f(n) = (\log n)^{\log n}, \quad g(n) = n^{\log \log n}$$

$$\text{令 } a = \log n, \text{ 则 } n = 2^a$$

$$f(n) = a^a, \quad g(n) = (2^a)^{\log a} = (2^{\log a})^a = a^a$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1 \neq \infty$$

$$B: f(n) = \log(n!), \quad g(n) = n \log n = \log n^n$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \dots (\text{泰勒展开}) \dots = 1$$

$$\Rightarrow f(n) = \theta(g(n))$$

$$C: f(n) = 2^{2n}, \quad g(n) = n^{2n}$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \left(\frac{2}{n}\right)^{2n} = 0$$

$$D: f(n) = n \log n, \quad g(n) = n^2$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\log n}{n} = 0$$

7. 一个问题的算法必须在有穷时间终止，并且对一切合法的输入都能得出满足要求的结果。（ ）

答案：对

解释：略

8. 输入:  $n = 2^t$ ,  $t$  为正整数, 输出:  $k$

(1)  $k \leftarrow 0$

(2) while  $n \geq 1$  do

(3) for  $j \leftarrow 1$  to  $n$  do

(4)  $k \leftarrow k + 1$

(5)  $n \leftarrow n/2$

(6) return  $k$

上述算法所执行的加法次数是: ( )

A.  $n-1$

B.  $n$

C.  $2n-1$

D.  $2n+1$

答案: C

解释:

$t$	$n$	$k$
1	2	$2+1=3$
2	4	$4+2+1=7$
3	8	$8+4+2+1=15$
...	...	...
	$n'$	$2n'-1$

9. 下列有关阶乘函数的表述错误的是 ( )

A.  $n! = w(2^n)$  ( $w$ :表示欧梅咖, 渐进下确界)

B.  $\log(n!) = O(n)$

C.  $n! = o(n^n)$

D.  $\log(n!) = O(n \log n)$

答案: B

解释: 略

10. 考虑下述选择排序算法:

算法ModSelectSort

输入:  $n$ 个整数的数组 $A[1..n]$  输出:按递增次序排序的 $A$

1. for  $i \leftarrow 1$  to  $n-1$  do

2. for  $j \leftarrow i+1$  to  $n$  do

3. if  $A[j] < A[i]$  then  $A[i] \leftrightarrow A[j]$

最坏情况下该算法做 $n(n-1)/2$ 次交换运算, 这种情况在下列哪种输入条件下发生? ( )

A. 数列元素各不相等且递增有序

B. 数列元素各不相等且递减有序

C. 数列元素各不相等且无序

D. 数列中有相同元素且递增(不减)有序

答案: B

解释: 略

11. 当输入规模为 $n$ 时, 算法增长率最大的是 ( )

A.  $200n$

B.  $1820 \log_2 n$

C.  $6n^2$

D.  $19n\log 3n$

答案: C

解释: 略

12. 算法与程序的区别是()

A. 输入

B. 确定性

C. 输出

D. 有穷性

答案: D

解释: 略

13. 同一算法只有一种形式描述。()

答案: 错

解释: 略

14. 证明算法不正确, 只需给出一个反例, 算法不能正确处理即可。()

答案: 对

解释: 略

15. 下表给出1,2,3,4,5组 $f(n)$ 和 $g(n)$ 函数,

	$f(n)$	$g(n)$
1	$2n^3+3n$	$100n^2+2n+100$
2	$50n+\log n$	$10n+\log \log n$
3	$50n \log n$	$10n \log \log n$
4	$\log n$	$\log^2 n$
5	$n!$	$5^n$

使得 $f(n)=O(g(n))$ 成立的组号(从小到大排列)是: (请直接填写数字序号,例如顺序为1-3-5,则填写"135")

答案: 2 4

解释:

$$1. \lim_{n \rightarrow \infty} \frac{2n^3+3n}{100n^2+2n+100} = \infty \quad \times$$

$$2. \lim_{n \rightarrow \infty} \frac{50n+\log n}{10n+\log \log n} = 5 \quad \checkmark$$

$$3. \lim_{n \rightarrow \infty} \frac{50n \log n}{10n \log \log n} = \lim_{n \rightarrow \infty} \frac{5 \log n}{\log \log n} \xrightarrow{a=\log n} \lim_{a \rightarrow \infty} \frac{5a}{\log a} = \infty \quad \times$$

$$4. \lim_{n \rightarrow \infty} \frac{\log n}{\log^2 n} \xrightarrow{a=\log n} \lim_{a \rightarrow \infty} \frac{a}{a^2} = 0 \quad \checkmark$$

$$5. \lim_{n \rightarrow \infty} \frac{n!}{5^n} = \infty \quad \times$$

## 2 递归与分治策略

### 2.1 递归

#### 2.1.0 递归概述

- 递归算法：直接或间接地调用自身的算法
- 递归函数：用函数自身给出定义的函数
- 递归应用；
  - 问题的定义是递归的：如求阶乘、Ackerman函数
  - 问题的求解过程是递归的：如汉诺塔问题、八皇后问题
  - 问题采用的数据结构是递归的：如二叉树的遍历
- 优点：
  - 结构清晰
  - 可读性强
  - 容易用数学归纳法来证明算法的正确性
  - 为设计算法、调试程序带来很大方便
- 缺点：运行效率较低（运行过程中调用自身），耗费的计算时间还是占用的存储空间都比非递归算法要多

#### 2.1.1 阶乘函数

递归式：

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

代码：

```
long fac(long n){
    if(n<=1){
        return 1;
    }
    return n*fac(n-1);
}
```



## 2.1.2 Fibonacci数列

递归式：

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

代码：

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

## 2.1.3 Ackerman函数

Ackerman函数定义：

$$\begin{cases} A(1,0) = 2 \\ A(0,m) = 1 & m \geq 0 \\ A(n,0) = n + 2 & n \geq 2 \\ A(n,m) = A(A(n-1,m),m-1) & n,m \geq 1 \end{cases}$$

Ackerman函数无法写出非递归定义式

## 2.1.4 排列问题

问题描述：

设计一个递归算法生成n个元素{r1,r2,...,rn}的全排列

例：

**例2：R={a,b,c,d}**

**abcd、 abdc、 acbd、 acdb、 adbc、 adcb**  
**bacd、 badc、 bcad、 bcda、 bdca、 bdac**  
**cbad、 cbda、 cabd、 cadb、 cdab、 cdba**  
**daca、 dbac、 dcba、 dcab、 dacb、 dabc**

思路：

- (1)  $R = \{r_1, r_2, \dots, r_n\}$ 是要进行排列的n个元素,  $R_i = R - \{r_i\}$
- (2) 集合R中元素的全排列记为 $perm(R)$
- (3)  $(r_i)perm(R_i)$ 表示在全排列 $perm(R_i)$ 的每一个排列前加上前缀 $r_i$ 得到的排列
- (4) R的全排列可归纳定义为：
  - 当n=1时,  $perm(R) = (r)$ , 其中r是集合R中唯一的元素

- 当 $n > 1$ 时,  $perm(R)$ 由 $(r_1)perm(R_1), (r_2)perm(R_2), \dots, (r_n)perm(R_n)$ 构成

代码:

```
#include "stdio.h"
void perm(int list[], int k, int m); //k: 前缀元素个数; m: 总元素个数-1
void swap(int* a, int* b);
int flag; // 14行到17行是为了使用flag控制输出格式
int main(){
    int list[] = {1, 2, 3, 4};
    flag = list[0];
    perm(list, 0, 3);
    return 0;
}

void perm(int list[], int k, int m){
    if(k==m){
        if(flag!=list[0]){
            flag = list[0];
            putchar('\n');
        }
        for (int i = 0; i <= m; ++i) {
            printf("%d", list[i]);
        }
        putchar(' ');
    } else{
        for (int i = k; i <= m; ++i) {
            swap(&list[k], &list[i]); //保证每个元素做一次前缀
            perm(list, k+1, m);
            swap(&list[k], &list[i]); // 还原
        }
    }
}

void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

## 2.1.5 正整数划分问题

问题描述:

将正整数 $n$ 表示成一系列正整数之和,  $n = n_1 + n_2 + \dots + n_k$ , 正整数 $n$ 的这种表示称为正整数 $n$ 的划分, 求正整数 $n$ 的不同划分个数

例:

## 正整数6有如下11种不同的划分：

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。

### 思路：

对正整数 $n$ 进行划分，将最大加数 $n_1$ 不大于 $m$ 的划分个数记作 $q(n,m)$

$q(n,m)$ 的递归关系：

- $q(n, 1) = 1, n \geq 1$ ：当最大加数 $n_1$ 不大于1时，任何正整数 $n$ 只有一种划分形式，即
$$n = \overbrace{1+1+\cdots+1}^n$$
- $q(n, m) = q(n, n), m > n$ ：最大加数 $n_1$ 实际上不能大于 $n$ 。因此 $q(1,m)=1$
- $q(n, n) = 1 + q(n, n-1)$ ：正整数 $n$ 的划分由 $n_1=n$ 的划分和 $n_1 \leq n-1$ 的划分组成
- $q(n, m) = q(n, m-1) + q(n-m, m), n > m > 1$ ：正整数 $n$ 的最大加数 $n_1$ 不大于 $m$ 的划分由 $n_1=m$ 的划分和 $n_1 \leq m-1$ 的划分组成

### 递归式：

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

## 2.1.6 Hanoi塔问题

### 问题描述：

有三个塔座,在其中一个塔座上从下到上穿好了由大到小的 $n$ 片圆盘,每次只能移动1个圆盘,任何时刻都不允许将较大的圆盘压在较小的圆盘之上,将所有的圆盘从原塔座移至另外两个中的一个塔座上

### 思路：

- (1) 将 $n-1$ 个较小盘子上设法移到辅助塔座（比原问题规模小1的问题）
- (2) 将最大的盘子从原塔座一步移至目标塔座
- (3) 将 $n-1$ 个较小的圆盘辅助塔座移动到目标塔座（比原问题规模小1的问题）

### 代码：

```
#include "stdio.h"
void hanoi(int n, char a, char b, char c); //a: 原塔座, b: 辅助塔座, c: 目标塔座
int main() {
    int n;
    char a, b, c;
    scanf("%d %c %c %c", &n, &a, &b, &c);
    hanoi(n, a, b, c);
    return 0;
}
```

```

}

void hanoi(int n,char a,char b,char c){
    if(n==1){
        printf("%d:%c->%c\n",n,a,c);
    }else{
        hanoi(n-1,a,c,b);
        printf("%d:%c->%c\n",n,a,c);
        hanoi(n-1,b,a,c);
    }
}
}

```

## 2.2 分治

### 2.2.0 分治概述

#### 分治思想：

- (1) 将要求解的较大规模的问题分割成k个更小规模的与原问题相同的子问题
- (2) 对这k个子问题分别求解
- (3) 如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止
- (4) 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解

#### 分治法适用条件：

- (1) 该问题的规模缩小到一定的程度就可以容易地解决
  - (2) 该问题可以分解为若干个规模较小的性质相同问题
  - (3) 利用该问题分解出的子问题的解可以合并为该问题的解
  - (4) 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题
- 第(4)条不成立时也可以使用分治法，但会做许多不必要的工作，这时一般使用动态规划较好
  - 如果具备了前两条特征，而不具备第三条特征，则可以考虑贪心算法或动态规划

#### 分治法求解步骤：

- (1) 分解：将原问题分解成一系列子问题
- (2) 解决：递归的解各个子问题，若子问题足够小，则直接求解
- (3) 合并：将子问题的结果合并成原问题的解

#### 问题规模的分割原则：

在用分治法设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的k个子问题

### 2.2.1 分治法的复杂性分析

- 将规模为n的问题分成k个规模为 $\frac{n}{m}$ 的子问题
- 设分解阈值 $n_0=1$ ，解规模为1的问题耗费1个单位时间
- 分解原问题和合并子问题的解一共耗费 $f(n)$ 个单位时间
- $T(n)$ 表示使用分治法求解规模为n的问题耗费的单位时间

有：

$$T(n) = \begin{cases} O(1) & n = 1 \\ kT(\frac{n}{m}) + f(n) & n > 1 \end{cases}$$

为什么“规模为n的问题分成k个规模为 $\frac{n}{m}$ 的子问题”：有m-k个子问题被舍弃（二分搜索），或不止一个规模为n的原问题（大整数乘法）

通过迭代法求得：

$$T(n) = n^{\log_m k} + \sum_{j=0}^{\log_m n - 1} k^j f(\frac{n}{m^j})$$

## 2.2.2 二分搜索

**问题描述：**

给定已按升序排好序的n个元素a[0:n-1]，现要使用二分搜索在这n个元素中找出一特定元素x

**分析：**

- 如果n=1即只有一个元素，则只要比较这个元素和x就可以确定x是否在表中
- 比较x和a的中间元素a[mid]
  - 若x=a[mid]，则x在L中的位置就是mid
  - 如果x<a[mid]，由于a是递增排序的，因此假如x在a中的话，x必然排在a[mid]的前面，所以我们只要在a[mid]的前面查找x即可
  - 如果x>a[mid]，同理我们只要在a[mid]的后面查找x即可
- 很显然此问题分解出的子问题相互独立，即在a[i]的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件

**代码：**

```
#include "stdio.h"
int binarySearch(int a[], int x, int n);
int main() {
    int n=10, x=7;
    int a[10]={1,3,4,7,9,12,14,16,18,19};
    int address = binarySearch(a, x, 10);
    if(address!=-1){
        printf("%d is at a[%d]", x, address);
    } else{
        printf("x not found!");
    }
    return 0;
}

int binarySearch(int a[], int x, int n){
    if(n==1&& a[0]==x){
        return 0;
    }
    int left=0, right=n-1, mid;
    while (left<=right){
        mid = (left+right)/2;
        if(a[mid]==x){
            return mid;
        }
        if(x<a[mid]){
            right = mid-1;
        } else{
            left = mid+1;
        }
    }
    return -1;
}
```

```

        left = mid+1;
    }
}
return -1;
}

```

## 2.2.3 大整数乘法

**问题描述：**

两个特别大的整数X和Y相乘

**思路：**

(1) 将X分为A和B两部分，Y分为C和D两部分

即：

$$\begin{array}{lcl}
 X = & \boxed{A} & \boxed{B} \\
 Y = & \boxed{C} & \boxed{D}
 \end{array}$$

例：X=1234，则A=12；B=34

(2) 通过2的幂指缩小问题规模：

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

## 2.2.4 Strassen矩阵乘法

**问题描述：**

两个n×n矩阵A和B的相乘，乘积为C，使时间复杂度最小

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

**思路：**

引入新矩阵：

$$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$$

$$\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$$

$$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$$

$$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$

$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$$

$$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$

$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$$

得：

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

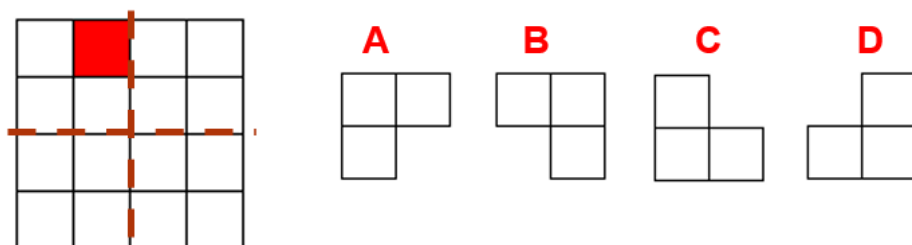
时间复杂度:

$$T(n) = O(n^{\log 7}) = O(n^{2.81})$$

## 2.2.5 棋盘覆盖

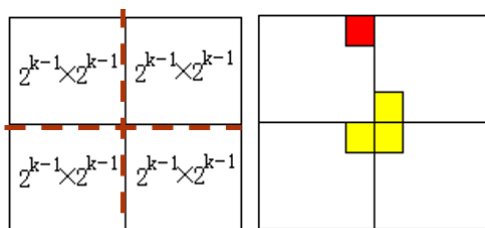
问题描述:

在一个 $2^k \times 2^k$ 个方格组成的棋盘中, 恰有一个方格与其它方格不同, 称该方格为一特殊方格, 且称该棋盘为一特殊棋盘。在棋盘覆盖问题中, 要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格, 且任何2个L型骨牌不得重叠覆盖



思路:

- 当 $k > 0$ 时, 将 $2^k \times 2^k$ 棋盘分割为4个 $2^{k-1} \times 2^{k-1}$ 子棋盘
- 特殊方格必位于4个较小子棋盘之一中, 其余3个子棋盘中无特殊方格
- 为了将这3个无特殊方格的子棋盘转化为特殊棋盘, 可以用一个L型骨牌覆盖这3个较小棋盘的会合处, 从而将原问题转化为4个较小规模的棋盘覆盖问题
- 递归地使用这种分割, 直至棋盘简化为棋盘 $1 \times 1$



代码:

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

int num = 0;
void chessBoard(int **Map, int tr, int tc, int dr, int dc, int size);

int main() {
    int k, row, col;

    //输入k值, 棋盘大小size=2^k
    printf("k:");
    scanf("%d", &k);
```

```

int size = pow(2, k);

//输入特殊方格所在坐标
printf("x and y:");
scanf("%d %d", &row, &col);

int **Map = (int **) malloc(sizeof (int *)*size);
for (int i = 0; i < size; i++) {
    Map[i] = (int *) malloc(sizeof (int) * size);
}

Map[row][col] = 0;
chessBoard(Map, 0, 0, row, col, size);
for (int i = 0; i < size; ++i) {
    for (int j = 0; j < size; ++j) {
        printf("%d ",Map[i][j]);
    }
    putchar('\n');
}
return 0;
}

void chessBoard(int **Map, int tr, int tc, int dr, int dc, int size) {

    int s, t;
    if (size == 1)
        return;
    s = size / 2; //边长每次减半, 分为4个子棋盘
    t = ++num; //编号
    if (dr < tr + s && dc < tc + s) { //左上
        chessBoard(Map, tr, tc, dr, dc, s);

    } else {
        Map[tr + s - 1][tc + s - 1] = t;
        chessBoard(Map, tr, tc, tr + s - 1, tc + s - 1, s);
    }

    if (dr < tr + s && dc >= tc + s) { //右上
        chessBoard(Map, tr, tc + s, dr, dc, s);

    } else {
        Map[tr + s - 1][tc + s] = t;
        chessBoard(Map, tr, tc + s, tr + s - 1, tc + s, s);
    }

    if (dr >= tr + s && dc < tc + s) { //左下
        chessBoard(Map, tr + s, tc, dr, dc, s);
    } else {
        Map[tr + s][tc + s - 1] = t;
        chessBoard(Map, tr + s, tc, tr + s, tc + s - 1, s);
    }

    if (dr >= tr + s && dc >= tc + s) { //右下
        chessBoard(Map, tr + s, tc + s, dr, dc, s);
    } else {
        Map[tr + s][tc + s] = t;
        chessBoard(Map, tr + s, tc + s, tr + s, tc + s, s);
    }
}

```



```
}
```

## 2.2.6 合并排序

### 基本思想:

将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并成为所要求的排好序的集合

### 求解步骤:

- (1) 分解: 将 $n$ 个元素分成各含 $n/2$ 个元素的子序列
- (2) 解决: 用合并排序方法对两个子序列递归的排序
- (3) 合并: 合并两个已排好序列的子序列以得到有序的完整序列

### 代码:

```
#include "stdio.h"
#include "stdlib.h"

void mergeSort(int* A, int lenA);
void merge(int* B, int lenB, int* C, int lenC, int* A);

int main(){
    int a[10]={9,7,5,3,1,2,4,6,8,0};
    mergeSort(a,10);
    for (int i = 0; i < 10; ++i) {
        printf("%d ",a[i]);
    }
    return 0;
}

void mergeSort(int* A, int lenA) {
    if (lenA > 1) {
        int n1 = lenA / 2;
        int n2 = lenA - n1;
        int* B = (int*)malloc(sizeof(int) * n1);
        int* C = (int*)malloc(sizeof(int) * n2);
        for (int i = 0; i < n1; i++) {
            B[i] = A[i];
        }
        for (int i = 0; i < n2; i++) {
            C[i] = A[n1 + i];
        }
        mergeSort(B, n1);
        mergeSort(C, n2);
        merge(B, n1, C, n2, A);
        free(B);
        free(C);
    }
}

void merge(int* B, int lenB, int* C, int lenC, int* A) {
    int i = 0, j = 0, k = 0;
    while (i < lenB && j < lenC){
        if(B[i] <= C[j]){
            A[k] = B[i];
            i++;
        } else {
            A[k] = C[j];
            j++;
        }
        k++;
    }
    while (i < lenB) {
        A[k] = B[i];
        i++;
        k++;
    }
    while (j < lenC) {
        A[k] = C[j];
        j++;
        k++;
    }
}
```

```

        A[k] = B[i];
        i++;
    } else{
        A[k] = C[j];
        j++;
    }
    k++;
}
if (i == lenB) {
    while (j < lenC){
        A[k] = C[j];
        k++;
        j++;
    }
} else {
    while (i < lenB){
        A[k] = B[i];
        k++;
        i++;
    }
}
}
}

```

## 2.2.7 快速排序

### 基本思想：

在快速排序中，记录的比较和交换是从两端向中间进行的,关键字较大的记录一次就能交换到后面单元，关键字较小的记录一次就能交换到前面单元，记录每次移动的距离较大，因而总的比较和移动次数较少

### 代码：

```

#include "stdio.h"

void quick_sort(int *a,int l,int n);

int main(){
    int a[10]={9,7,5,3,1,2,4,6,8,0};
    quick_sort(a,0,10);
    for (int i = 0; i < 10; ++i) {
        printf("%d ",a[i]);
    }
    return 0;
}

void quick_sort(int *a,int l,int n){
    if(l+1>=n){
        return ;
    }
    int first=l,last=n-1,key=a[first];
    while(first<last){
        while(first<last&& a[last]>=key){
            --last;
        }
        a[first]=a[last];
        while(first<last&& a[first]<key){
            ++first;
        }
    }
}

```

```

    }
    a[last]=a[first];
}
a[first]=key;
quick_sort(a,l,first);
quick_sort(a,first+1,n);
}

```

## 2.2.8 循环赛日程表

### 问题描述:

设计一个满足以下要求的比赛日程表 ( $n=2^k$ ) :

- 每个选手必须与其他 $n-1$ 个选手各赛一次
- 每个选手一天只能赛一次
- 循环赛一共进行 $n-1$ 天

### 思路:

- 按分治策略, 将所有的选手分为两半,  $n$ 个选手的比赛日程表就可以通过为 $n/2$ 个选手设计的比赛日程表来决定
- 递归地用对选手进行分割, 直到只剩下2个选手时, 比赛日程表的制定就变得很简单。此时只要让这2个选手进行比赛即可
- 问题的解大致为下图的样式:

1	2	3	4	5	6	7	8
2	1	A	3	6	5	B	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	D	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

其中A和D、B和C是相同的, 只要求出A和B就可以得到C和D的, 通过如下代码实现:

```

for(int i = k; i < k+n/2; i++){
    for(int j = n/2; j < n; j++){
        a[i][j] = a[i+n/2][j-n/2];
    }
}
for(int i = k+n/2; i < k+n; i++){
    for(int j = n/2; j < n; j++){
        a[i][j] = a[i-n/2][j-n/2];
    }
}

```

### 代码:

```

#include "stdio.h"
#include "math.h"
int a[10][10];

```

```

void fun(int n,int k);
int main(){
    int k,n;
    scanf("%d",&k);
    n = pow(2,k);
    fun(n,0);
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
    return 0;
}

void fun(int n,int k){
    if(n == 2){
        a[k][0] = k+1;
        a[k][1] = k+2;
        a[k+1][0] = k+2;
        a[k+1][1] = k+1;
    }
    else{
        fun(n/2,k);
        fun(n/2,k+n/2);
        for(int i = k; i < k+n/2; i++){
            for(int j = n/2; j < n; j++){
                a[i][j] = a[i+n/2][j-n/2];
            }
        }
        for(int i = k+n/2; i < k+n; i++){
            for(int j = n/2; j < n; j++){
                a[i][j] = a[i-n/2][j-n/2];
            }
        }
    }
}
}

```

## 2.3 第二章测试

1. 以下不可以使用分治法求解的是()

- A.棋盘覆盖问题
- B.排序问题
- C.归并排序
- D.0-1背包问题

答案: D

解释: 0-1背包问题应该使用动态规划

2. 使用分治法高效率求解不需要满足的条件是 ()

- A.子问题的解可以合并

$B_{n-1}$

C.  $n^{1/2}$

D.  $n/2$

答案: D

解释: 略

6. 设问题P的输入规模是n, 下述三个算法是求解P的不同的分治算法.

算法1: 在常数时间将原问题划分为规模减半的5个子问题, 递归求解每个子问题, 最多用线性时间将子问题的解综合而得到原问题的解.

算法2: 先递归求解2个规模为n-1的子问题, 最多用常量时间将子问题的解综合得到原问题的解.

算法3: 在常数时间将原问题划分为规模n/3的9个子问题, 递归求解每个子问题, 最多用线性时间将子问题的解综合得到原问题的解.

要求在上述三个算法中选择最坏情况下时间复杂度最低的算法, 需要选择哪个算法?

A. 2

B. 都不对

C. 3

D. 1

答案: C

解释:

- 算法1:

$$\begin{aligned}T(n) &= 5T\left(\frac{n}{2}\right) + O(n) \\k &= 5, m = 2, f(n) = n \\ \text{带入公式得: } T(n) &= n^{\log_2 5} + \sum_{j=0}^{\log_2 n - 1} 5^j f\left(\frac{n}{2^j}\right) \\ &= O(n^{\log_2 5})\end{aligned}$$

- 算法2:

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

由于这里的 $T(n)$ 不符合 $kT\left(\frac{n}{m}\right) + f(n)$ 的样式, 故不能使用公式, 而使用列举法

$$\begin{aligned}T(1) &= 1 \\T(2) &= 2T(1) + 1 = 2 + 1 \\T(3) &= 2T(2) + 1 = 2 * (2 + 1) + 1 = 2^2 + 2 + 1 \\T(4) &= 2T(3) + 1 = 2 * (2^2 + 2 + 1) + 1 = 2^3 + 2^2 + 2 + 1 \\&\dots \\T(n) &= 2T(n-1) + 1 = 2^{n-1} + 2^{n-2} + \dots + 2 + 1\end{aligned}$$

则

$$T(n) = O(2^n)$$

- 算法3:

$$T(n) = 9\left(\frac{n}{3}\right) + O(n)$$

$$k = 9, m = 3, f(n) = n$$

带入公式得：

$$T(n) = n^{\log_3 9} + \sum_{j=0}^{\log_3 n - 1} 9^j f\left(\frac{n}{3^j}\right)$$

$$= O(n^2)$$

由

$$O(2^n) > O(n^{\log_3 5}) > O(n^2)$$

得：算法3时间复杂度最低

7. 时间复杂度常指算法最坏情况下的运行时间。（）

答案：错

解释：看具体要求，最好情况、一般情况的时间复杂度也是时间复杂度

8.  $n! = O(2^n)$

答案：D

解释：n→∞时，阶乘比指数高

9. 给定n个数的数组L，其中 $n = 2^k$ ，k为非负整数，求L中的最大数。考虑下述算法A：

先把数组从中间划分成两个 $n/2$ 个数的数组 $L_1$ 和 $L_2$ ，在 $L_1$ 和 $L_2$ 中用同样的算法通过数之间的比较运算找最大数，如果 $L_1$ 的最大数是 $a_1$ ， $L_2$ 的最大数是 $a_2$ ，那么 $\max(a_1, a_2)$ 就是问题的解。假设对于n个数的数组L，在最坏情况下算法A的比较次数是W(n)，该算法在最坏情况下W(n)的递推方程是：

- A.  $W(n) = 2W(n/2) + n/2$
- B.  $W(n) = W(n/2) + n/2$
- C.  $W(n) = W(n/2) + 1$
- D.  $W(n) = 2W(n/2) + 1$

答案：D

解释： $k = 2, m = 2, f(n) = 1$

10. 题干同第9题，问：W(n)的精确值是？

- A. n
- B.  $2n-1$
- C.  $\log n-1$
- D.  $n+1$

答案：B

解释：

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + 1 \\
 k=2 \quad m=2 \quad f(n)=1 \\
 \text{代入公式} \\
 T(n) &= n^{\log_2 2} + \sum_{j=0}^{\log_2 n - 1} 2^j \cdot 1 \\
 &= n + \sum_{j=0}^{\log_2 n - 1} 2^j \\
 &= n + 2^{\log_2 n} - 1 \\
 &= 2n - 1
 \end{aligned}$$

11. 采用分治策略求解的问题必须具备其拆分的子问题不能重复的特征。 ( )

答案：错

解释：子问题可以重复

12. 递归算法指的是只直接调用自身的算法。 ( )

答案：错

解释：也可以间接调用

## 3 贪心算法

### 3.1 贪心概述

**贪心算法定义：**

从对问题的某一初始解出发,一步一步的攀登给定的目标,尽可能快地去逼近更好的解。当达到某一步,不能再攀登时,算法便终止

**贪心策略基本思想：**

从问题的初始状态出发,通过若干次的贪心选择得出问题的最优解或近似优解

**贪心算法特点：**

- 贪心算法总是做出在当前看来是最好的选择
- 它并不是从总体最优上加以考虑,它所作出的选择只是在某种意义上的局部最优选择
- 希望贪心算法得到的最终结果也是整体最优
- 虽然贪心算法不能对所有问题都得到整体最优解,但对许多问题它能产生整体最优解
- 在一些情况下,即使贪心算法不能得到整体最优解,其最终结果却是最优解的很好近似

### 3.2 贪心算法的两个基本要素



### (1) 贪心选择性质:

- 是指所求问题的整体最优解可以通过一系列局部最优的选择, 即贪心选择来达到
- 贪心算法通常以自顶向下的方式进行, 以迭代的方式作出相继的贪心选择, 每作一次贪心选择就将所求问题简化为规模更小的子问题
- 对于一个具体问题, 要确定它是否具有贪心选择性质, 做出贪心选择后, 原问题简化为规模更小的类似子问题吗, 必须证明每一步所作的贪心选择最终导致问题的整体最优解

### (2) 最优子结构性质:

- 当一个问题的最优解包含其子问题的最优解时, 称此问题具有最优子结构性质
- 问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征

## 3.3 活动安排问题

### 问题描述:

设有 $n$ 个活动的集合 $E=\{1,2,\dots,n\}$ , 其中每个活动都要求使用同一资源, 而在同一时间内只有一个活动能使用这一资源。每个活动 $i$ 都有一个要求使用该资源的起始时间 $s_i$ 和一个结束时间 $f_i$ , 且 $s_i < f_i$

### 例:

将待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下:

i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

例如, 当第1个活动结束后, 由于这个活动的结束时间 $f[1]=4$ , 对于下一个活动, 其开始时间必须大于或等于4, 所以第二个可以执行的活动按照顺序是第4个活动, 它的起始时间为 $s[4]=5$

### 代码:

```
//活动序列已按照结束时间升序排序
#include "stdio.h"
void fun(int n, int s[], int f[], bool a[]);
int main(){
    int s[11]={1,3,0,5,3,5,6,8,8,2,12};
    int f[11]={4,5,6,7,8,9,10,11,12,13,14};
    bool a[11];
    fun(11,s,f,a);
    for (int i = 0; i < 11; ++i) {
        printf("%d ",a[i]);
    }
    return 0;
}
void fun(int n, int s[], int f[], bool a[]){
    a[0] = true;
    for (int i = 1, j=0; i < n; ++i) {
        if(s[i]>f[j]){
            a[i] = true;
            j = i;
        } else{
            a[i] = false;
        }
    }
}
```

```
}  
}  
}
```

#### 分析:

- 由于输入的活动以其完成时间的非减序排列, 所以算法greedySelector每次总是选择具有最早完成时间的相容活动加入集合A中
- 按这种方法选择相容活动为未安排活动留下尽可能多的时间, 即该算法的贪心选择的意义是使剩余的可安排时间段极大化, 以便安排尽可能多的相容活动
- 算法greedySelector的效率极高。当输入的活动已按结束时间的非减序排列, 算法只需O(n)的时间安排n个活动, 使最多的活动能相容地使用公共资源

#### 贪心证明过程:

##### 1. 总存在以贪心选择开始的最优活动安排方案

- 设 $E = \{1, 2, \dots, n\}$ 为所给的活动集合, 且E中活动按照结束时间非减序排列, 故活动1具有最早完成时间
- 首先, 证明活动安排问题有一个最优解以贪心选择开始, 即该最优解中包含活动1:

设 $A \in E$ 是所给活动安排问题的一个最优解, 且A中活动也按结束时间非减序排列, A中的第一个活动是k

- 若 $k=1$ , 则A就是一个以贪心选择开始的最优解
- 若 $k>1$ , 则设 $B = A - \{k\} \cup \{1\}$

由于 $f_1 < f_k$ , 且A中活动都是相容的, 故B中的活动也是相容的, 且B和A中活动个数相等, 所以B也是最优的。即B是以贪心选择活动1开始的最优活动安排

##### 2. 证明活动安排问题具有最优子结构性质

- 在做了第一步贪心选择活动1后, 原问题简化为对E中所有与活动1相容的活动进行活动安排的子问题
- 若A是原问题的最优解, 则 $A' = A - \{1\}$ 是活动安排问题 $E' = \{i \in E | s_i \geq f_1\}$ 的最优解
- 反证法: 假设E'有一个解B', 包含比A'更多的活动,  $B = B' \cup \{1\}$ , 则B比A包含更多的活动这与A的最优性矛盾
- 结论: 每一步所做的贪心选择都将原问题简化为一个更小的与原问题具有相同形式的子问题

##### 3. 对贪心选择次数用数学归纳法证明

贪心选择次数用数学归纳法即知, 贪心算法最终产生原问题的最优解

## 3.4 最优装载问题

#### 问题描述:

有一批集装箱要装上一艘载重量为c的轮船。其中集装箱i的重量为 $W_i$ 。最优装载问题要求确定在装载体积不受限制的情况下, 将尽可能多的集装箱装上轮船

#### 分析:

- 用一个向量 $(x_1, x_2, x_3, \dots, x_n)$ , 表示装的个数多少
- 约束条件:

$$\begin{cases} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in 0, 1, i \leq n \end{cases}$$

- 目标函数:  $\max \sum_{i=1}^n x_i$

- 采用重量最轻者先装的贪心选择策略
- 最优装载问题可用贪心算法求解，可产生最优装载问题的最优解

代码：

```
// 集装箱序列已经按照重量排序
#include "stdio.h"
void load(int x[], int w[], int c, int n);
int main(){
    int x[10]={0};
    int w[5]={2,3,4,7,11};
    int n = 5;
    int c = 13;
    load(x,w,c,n);
    for (int i = 0; i < n; ++i) {
        printf("%d ",x[i]);
    }
    return 0;
}

void load(int x[], int w[], int c, int n){
    for (int i = 0; i < n&&w[i]<=c; ++i) {
        x[i] = 1;
        c -= w[i];
    }
}
```

贪心证明过程：

#### 1. 最优装载的贪心选择性质

- 设集装箱依其重量从小到大排序， $(x_1, x_2, \dots, x_n)$ 是最优装载问题的一个最优解
- 设 $k = \min_{1 \leq i \leq n} \{i | x_i = 1\}$ ,即k为第一个选择的集装箱序号
- 如果给定的最优装载问题有解，则 $1 \leq k \leq n$ 
  - 当 $k=1$ 时， $(x_1, x_2, \dots, x_n)$ 是满足贪心选择性质的最优解
  - 当 $k>1$ 时，取 $y_1 = 1, y_k = 0, y_i = x_i$ ，则 $1 \leq i \leq n, i \neq k$

$$\sum_{i=1}^n w_i y_i = w_1 - w_k + \sum_{i=1}^n w_i y_i \leq \sum_{i=1}^n w_i y_i \leq c$$

故， $(y_1, y_2, \dots, y_n)$ 是所给最优装载问题的可行解

由 $\sum_{i=1}^n y_i = \sum_{i=1}^n x_i$ 知： $(y_1, y_2, \dots, y_n)$ 是满足贪心选择性质的最优解

#### 1. 优装载的最优子结构性质

- 设 $(x_1, x_2, \dots, x_n)$ 是装载问题的满足贪心选择性质的最优解，则 $x_1 = 1$ ，且 $(x_2, \dots, x_n)$ 是轮船载重量为 $c - w_1$ 、待装船集装箱为 $\{2, 3, \dots, n\}$ 时相应最优装载问题的最优解。即最优装载问题具有最优子结构性质

## 3.5 背包问题

问题描述：

给定n种物品和一个背包。物品i的重量是 $W_i$ ，其价值为 $V_i$ ，背包的容量为c。应如何选择装入背包的物品，使得装入背包中物品的总价值最大？在选择物品i装入背包时，可以选择物品i的一部分，而不一定要全部装入背包， $1 \leq i \leq n$

背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解s

**分析：**

- 以单位质量价值做为量度标准进行选择时，相当于把每一个物品都分割成单位块，单位块的利益越大，显然,物品装入背包后，背包获取总效益越大
- 贪心策略以单位质量价值作为贪心标准求解(分数)背包问题能够得到一个最优解

## 3.6 哈夫曼编码

---

**前缀编码：**

对字符集进行编码时，要求字符集中任一字符的编码都不是其它字符的编码的前缀，这种编码称为前缀(编)码

**最优前缀编码：**

平均码长或文件总长最小的前缀编码称为最优的前缀编码

## 3.7 单源最短路径

---

**问题描述：**

用带权的有向图表示一个交通运输网，图中：

- 顶点——表示城市
- 边——表示城市间的交通联系
- 权——表示此线路的长度或沿此线路运输所花的时间或费用等

从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径——最短路径

给定带权有向图 $G=(V,E)$ ，其中每条边的权是非负实数。另外，还给定 $V$ 中的一个顶点，称为源。现在要计算从源到所有其它各顶点的最短路径长度。这里路的长度是指路上各边权之和。这个问题通常称为单源最短路径问题

**分析：**

Dijkstra算法是解单源最短路径问题的贪心算法

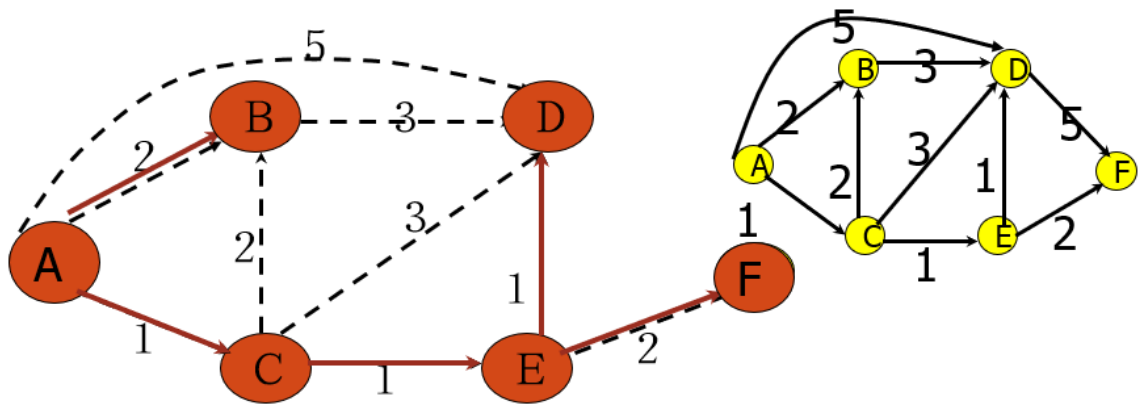
**基本思想：**

设置顶点集合 $S$ 并不断地作贪心选择来扩充这个集合。一个顶点属于集合 $S$ 当且仅当从源到该顶点的最短路径长度已知。

初始时， $S$ 中仅含有源。设 $u$ 是 $G$ 的某一个顶点，把从源到 $u$ 且中间只经过 $S$ 中顶点的路称为从源到 $u$ 的特殊路径，并用数组 $dist$ 记录当前每个顶点所对应的最短特殊路径长度。

Dijkstra算法每次从 $V-S$ 中取出具有最短特殊路长度的顶点 $u$ ，将 $u$ 添加到 $S$ 中，同时对数组 $dist$ 作必要的修改。一旦 $S$ 包含了所有 $V$ 中顶点， $dist$ 就记录了从源到所有其它顶点之间的最短路径长度

**例：**



步骤	s	D(B)	D(C)	D(D)	D(E)	D(F)
初始化	{ A }	2	1	5	$\infty$	$\infty$
1	{ A C }	2		4	2	$\infty$
2	{ A C B }			4	2	$\infty$
3	{ A C B E }			3		4
4	{ A C B E D }					4
5	{ A C B E D F }					

## 4 动态规划

### 4.1 概述

动态规划总体思想：

- 保存已解决的子问题的答案，在需要时再找出已求得的答案，避免大量重复计算，从而得到多项式时间算法。
- 动态规划法的关键就在于：对于重复出现的子问题，只在第一次遇到时加以求解并把答案保存起来，让以后再遇到时直接引用，不必重新求解。
- 动态规划算法对每个子问题只计算一次，不管该子问题是否以后会被用到，都将其结果保存到一张表中，从而避免每次遇到各个子问题时重新计算答案。
- 动态规划常用于求解最优解问题

### 4.2 动态规划基本步骤

- 找出最优解的性质，并刻画其结构特征。
- 递归地定义最优值
- 以自底向上的方式计算出最优值。
- 根据计算最优值时得到的信息，构造最优解。

### 4.3 矩阵连乘问题

问题描述：

- 给定n个矩阵 $\{A_1, A_2, \dots, A_n\}$ ，其中 $A_i$ 与 $A_{i+1}$ 是可乘的， $i=1, 2, \dots, n-1$ 。考察这n个矩阵的连乘积 $A_1 A_2 \dots A_n$
- 由于矩阵乘法满足结合律，所以计算矩阵的连乘可以有許多不同的计算次序。
- 计算次序可以用加括号的方式来确定。

- 若一个矩阵连乘积的计算次序完全确定，也就是说该连乘积已完全加括号。

#### 分析：

- 矩阵乘法满足结合律, 无论怎样加括号，改变计算次序，都会产生相同的计算结果
- 加括号的次序对求解矩阵连乘的代价有很大的影响，主要表现在矩阵元素的乘法次数上
- 完全加括号的矩阵连乘积可递归地定义为：
  - 单个矩阵是完全加括号的
  - 矩阵连乘积A已完全加括号的，则A可表示为2个完全加括号的矩阵连乘积B和C的乘积并加括号，即 $A=(BC)$ 。也就是A可以依此次序反复调用2个矩阵相乘的标准算法计算出矩阵连乘积次数
- 给定n个矩阵 $\{A_1, A_2, \dots, A_n\}$ ，其中 $A_i$ 与 $A_{i+1}$ 是可乘的， $i=1, 2, \dots, n-1$ 。如何确定计算矩阵连乘积的计算次序（完全加括号方式），使得依此次序计算矩阵连乘积需要的数乘次数最少。

#### 动态规划算法求矩阵连乘问题：

- 预定义
  - 将矩阵连乘积 $A_1 A_2 \dots A_n$ 简记为 $A[i:j]$ ，这里 $i \leq j$
  - 考察计算 $A[1:n]$ 的最优计算次序。设这个计算次序在矩阵 $A_k$ 和 $A_{k+1}$ 之间将矩阵链断开， $1 \leq k < n$ ，则其相应完全加括号方式为： $(A_1 \dots A_k)(A_{k+1} \dots A_n)$
  - 总计算量为： $A[1:k]$ 的计算量加上 $A[k+1:n]$ 的计算量，再加上 $A[1:k]$ 和 $A[k+1:n]$ 相乘的计算量
- 分析最优解的结构
  - 特征: 计算 $A[1:n]$ 的最优计算次序所包含的计算矩阵子链 $A[i:k]A[k+1:n]$ 的计算次序也是最优的。
  - 用反证法可以证明该假设。
  - 矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。
- 建立递归关系
  - 设计算 $A[i:j]$ ， $1 \leq i < j \leq n$ ，所需要的最少数乘次数 $m[i,j]$ ，则原问题的最优值为 $m[1,n]$

➤ 当 $i=j$ 时， $A[i:j]=A_i$ ，因此， $m[i,i]=0$ ， $i=1, 2, \dots, n$

◦ ➤ 当 $i < j$ 时， $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$

这里  $A_i$  的维数为  $p_{i-1} \times p_i$

$p_{i-1} * p_k * p_j$  是计算  $A_{i \dots k}$  和  $A_{k+1 \dots j}$  的代价

可以递归地定义  $m[i,j]$  为：

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & i < j \end{cases}$$

- 计算最优值：
  1. 用动态规划算法解此问题，可依据其递归式以自底向上的方式进行计算。
  2. 在计算过程中，保存已解决的子问题答案。
  3. 每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算，最终得到多项式时间的算法。
- 构造最优解：
  - $S[1..n, 1..n]$  记录了各个子矩阵链取最优值时分割位置 $k$ ，则可以由找到该矩阵连乘的最优解
  - $S[i,j]$  表示的含义:  $S[i,j]$  表示  $A[i:j]$  的最佳方式是  $(A[1:k])(A[k+1:j])$
  - 从  $s[1,n]$  记录的信息可以知道  $A[1:n]$  的最佳方式:  $(A[1:s[1,n]])(A[s[1,n]+1:n])$
  - 其中， $(A[1: s[1,n]])$  最佳方式可以递归的得到:  $(A[1: s[1, s[1,n]]])(A[s[1, s[1,n]] + 1: s[1,n]])$

- 由此递归，可得到最优完全加括号方式，即构造一个最优解

## 4.4 最长公共子序列问题

### 问题描述：

- 给定序列 $X=\{x_1, x_2, \dots, x_m\}$ ，另一序列 $Z=\{z_1, z_2, \dots, z_k\}$ ，若存在一个严格递增下标序列 $\{i_1, i_2, \dots, i_k\}$ 使得对于所有 $j=1, 2, \dots, k$ 有： $z_j=x_{i_j}$ ；则 $Z$ 是 $X$ 的子序列。
- 给定2个序列 $X$ 和 $Y$ ，当另一序列 $Z$ 既是 $X$ 的子序列又是 $Y$ 的子序列时，称 $Z$ 是序列 $X$ 和 $Y$ 的公共子序列。

### 分析：

- $X$ 的终止位置是 $i$ ， $0 \leq i < m$
- $Y$ 的终止位置是 $j$ ， $0 \leq j < n$
- $X_i = \{x_1, x_2, \dots, x_i\}$  和  $Y_j = \{y_1, y_2, \dots, y_j\}$



- 分析最长公共子序列的最优子结构：

### LCS最优子结构定理：

设序列 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ 的最长公共子序列为 $Z=\{z_1, z_2, \dots, z_k\}$ ，则：

- (1) 若 $x_m = y_n$ 
  - 则 $z_k = x_m = y_n$ ，且 $Z_{k-1}$ 是 $X_{m-1}$ 和 $Y_{n-1}$ 的LCS。
- (2) 若 $x_m \neq y_n$ 且 $z_k \neq x_m$ 
  - 则 $Z$ 是 $X_{m-1}$ 和 $Y$ 的最长公共子序列。
- (3) 若 $x_m \neq y_n$ 且 $z_k \neq y_n$ 
  - 则 $Z$ 是 $X$ 和 $Y_{n-1}$ 的最长公共子序列。

- 递归定义最优值：

求 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ 的一个LCS

- 当 $x_m = y_n$ 时，须找出 $LCS(X_{m-1}, Y_{n-1})$ ，然后将 $x_m = y_n$ 添加到LCS上，可以产生一个 $LCS(X_m, Y_n)$
- 当 $x_m \neq y_n$ 时，就必须解决两个问题：找出一个 $LCS(X_{m-1}, Y)$ 和一个 $LCS(X, Y_{n-1})$ ，这两个LCS中较长的就是最长公共子序列。

- 定义递归值：



□ 用  $c[i][j]$  记录序列  $X_i$  和  $Y_j$  的 **最长公共子序列的长度**。

● 其中,  $X_i = \{x_1, x_2, \dots, x_i\}$ ;  $Y_j = \{y_1, y_2, \dots, y_j\}$ 。

□ 由最优子结构性性质可建立递归关系如下:

$$c[i][j] = \begin{cases} 0 & i = 0, j = 0 \\ c[i-1][j-1] + 1 & i, j > 0; x_i = y_j \\ \max\{c[i][j-1], c[i-1][j]\} & i, j > 0; x_i \neq y_j \end{cases}$$

- 计算最优值:
  - 由于在所考虑的子问题空间中, 总共有  $O(mn)$  个不同的子问题
  - 因此, 用动态规划算法自底向上地计算最优值能提高算法的效率。

## 4.5 0-1背包问题

### 问题描述:

给定  $n$  种物品和一背包。物品  $i$  的重量是  $w_i$ , 其价值为  $v_i$ , 背包的容量为  $C$ 。问应如何选择装入背包的物品, 使得装入背包中物品的总价值最大?

### 分析:

- 在选择装入背包的物品时, 对每种物品  $i$  只有两种选择: 装入和不装入  $\{0, 1\}$
- 不能将物品  $i$  装入背包多次, 也不能只装入部分的物品
- 

■ **约束条件:** 
$$\begin{cases} \sum_{i=1}^n w_i x_i \leq C \\ x_i \in \{0, 1\}, 1 \leq i \leq n \end{cases}$$

■ **目标函数:** 
$$\max \sum_{i=1}^n v_i x_i$$

- 刻画最优值的递归关系:
  - 已选择完是否放入  $i-1$  个物品后, 从第  $i$  个物品开始, 对剩余的  $n-i+1$  个物品在背包剩余容量为  $j$  进行选择, 使得装入背包中的物品价值和最大。
  - 由 0-1 背包问题的最优子结构性性质, 可以建立计算  $m(i, j)$  的递归式如下:

$$m(i, j) = \begin{cases} \max\{m(i+1, j), m(i+1, j - w_i) + v_i\} & j \geq w_i \\ m(i+1, j) & 0 \leq j < w_i \end{cases}$$

### 算法复杂度分析:

从  $m(i, j)$  的递归式容易看出, 算法需要  $O(nc)$  计算时间。

## 4.6 投资问题



### 问题描述:

设有投资公司, 在3月份预计总投资额为m万元, 共有n个项目,  $G_i(x)$ 为向第i项工程投资费用为x万元时的预计收益, 如何分配资源才能获得最大利润?

### 分析:

- 设总投资额为m万元, 共有n个项目,  $F_n(m)$ 为向n个项目投资m万元所获最大收益,  $G_i(x_i)$ 为向第i项工程投资x时的收益, 则有:  $F_n(m) = \max \{G_1(x_1) + G_2(x_2) + \dots + G_{n-1}(x_{n-1}) + G_n(x_n)\}$

- □ 问题的解向量 (  $x_1, x_2, \dots, x_{n-1}, x_n$  )

  - $x_i$  是投给项目i的投资额,  $i=1, 2, 3, \dots, n$

□ 约束条件:  $x_1 + x_2 + \dots + x_{n-1} + x_n = m$

  - 即  $\sum_{i=1}^n x_i = m$ ,  $0 \leq x_i \leq m$

- 分析投资问题的最优解结构特征:
  - 首先, 向第n项工程投资, 投资额为 $x_n$ , 所获收益为 $G_n(x_n)$
  - 则向剩余n-1项投资: 投资额为 $m - x_n$ , 所获最大收益为 $F_{n-1}(m - x_n)$
- 用递归式描述最优值结构:
  - 最优值  $f[i][j]$ : 向前i项工程投资, 投资额为j时获得的最大收益

$$f[i][j] = \begin{cases} g[1][j] & i=1, 0 \leq j \leq m \\ \max\{f[i-1][j-k] + g[i][k]\} & 0 \leq k \leq j, 1 < i \leq n, \\ & 0 \leq j \leq m \end{cases}$$

- 原问题的最优值在  $f[n][m]$
  - 标记  $d[i][j]$ : 前i项工程投资额为j时, 获得最大收益时, 向当前第i项工程的投资额k
- 构造最优解
  - $d[i][j]$ : 向前i项工程投资为j, 获得最大收益时, 向第i项工程的投资额存放在  $d[i][j]$
  - s为当前剩余投资额, 初始s=m

## 4.7 旅行商问题

### 问题描述:

某售货员要到n个城市去推销商品, 已知各城市之间的旅费, 售货员要选定一条从驻地出发经过每个城市一次, 最后回到驻地的路线, 使总旅费最小

### 图论语言描述:

- 设图  $G = (V, E)$  是一个加权连通图
- 要求:
  - $|V| = n$
  - 边  $(i, j)$  ( $i \neq j$ ) 的费用  $c[i][j]$  为正数
- 图G的一条周游路线是经过V中的每个顶点恰好一次的一条回路
- 周游路线的费用是这条回路上的所有边的费用之和
- TSP问题就是要在图G中找出费用最小的周游路线

### 动态规划求解:

1. 刻画解的结构特征:

设从顶点*i*出发，经过图*G*中各顶点最终返回顶点*i*的回路的最优解结构为

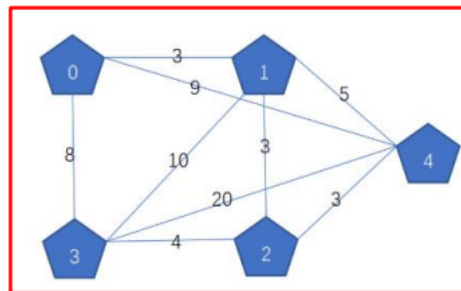
●  $(i, x_2, x_3, \dots, x_n)$ ,  $x_k \in (1, 2, \dots, i-1, i+1, \dots, n)$

● 其中：

➢  $x_i \neq x_k$

➢  $x_2, x_3, \dots, x_n$  是  $n-1$  个顶点的一个排列。

□ 反证法



## 2. 建立递归关系

- 定义函数  $g(k, V_1)$  为从顶点  $k$  出发，经过  $V_1$  中各顶点一次，并最终返回顶点  $i$  的最短路径长度
- 则 TSP 问题的最优值为  $g(i, V - \{i\})$
- 由于 TSP 问题满足最优子结构， $g(i, V - \{i\})$  具有如下递归关系：

•  $g(i, V - \{i\}) = \min \{c[i][k] + g(k, V - \{i, k\})\}, k \in V - \{i\}$

• 边界条件：  $g(k, \Phi)$

•  $= c[k][i]$

- 计算最优值：
  - 输入：  $n$  个城市及费用矩阵  $c$ , 出发城市  $i$
  - 输出：从城市  $i$  出发并返回的最终路径长度
- 构造最优解
  - TSP 问题的最优解是  $path(i, k_1, k_2, \dots, k_{n-1})$
- 算法复杂度：
  - 需要进行计算的次数与  $V$  的大小  $k (k=0, 1, 2, \dots, n-1)$  的子集的个数相关，并且每确定一个  $g(j, A)$ ，需要  $k$  次加法和  $k-1$  比较运算
  - 计算中所需的加法和比较的次数，可以算出时间复杂度为  $O(n^2 \cdot 2^n)$

# 5 回溯法：

## 5.1 回溯法概述

回溯法定义：

回溯法（也称试探法）：将问题候选解按某一顺序逐一枚举和试探的过程

三种可能的情况：

- 回溯：发现当前候选解不可能是可行解或最优解，则直接选下一个候选解
- 试探：当前候选解除了不满足问题规模的要求外，满足所有其他要求，则继续扩大当前候选解规模
- 解：满足包括问题规模在内的所有要求，则该候选解就是问题的一个可行解或最优解

概述：

- 回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树
- 算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解
- 如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索

问题的解：

- 可行解：满足约束条件的解。解空间中的一个子集
- 最优解：使目标函数取极值（极大或极小）的可行解，一个或少数几个

TSP问题:有 $n!$ 种可能解，有些是可行解，只有一个或几个是最优解

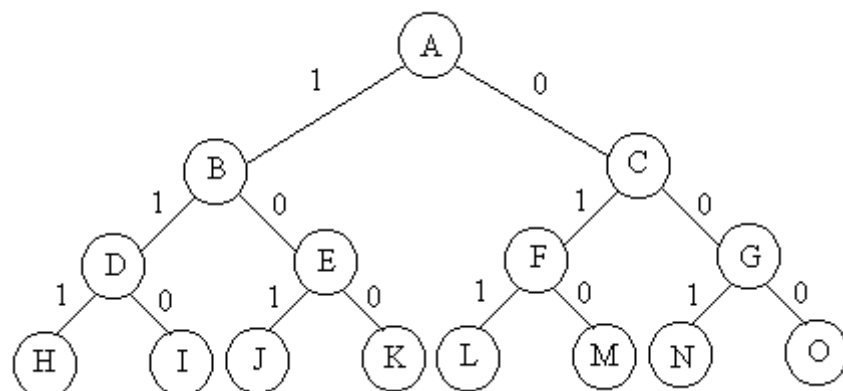
八后问题和图的着色问题:只要可行解，不需要最优解

### 问题的解空间：

定义问题的解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间

例： $n=3$ 的0-1背包问题

用完全二叉树表示的解空间：



### 回溯法基本思想：

- 确定了解空间的组织结构后，回溯法就从开始结点(根结点)出发，以深度优先的方式搜索整个解空间
- 深度优先的问题状态生成法：
  - 对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点
  - 完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子(若有)
  - 在一个扩展结点变成死结点之前，它一直是扩展结点
- 回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已无活结点时为止

### 基本概念：

- 扩展结点：一个正在产生儿子的结点称为扩展结点
- 活结点：一个自身已生成但其儿子还没有全部生成的节点称做活结点（试探）
- 死结点：一个所有儿子已经产生的结点称做死结点（回溯）

### 回溯法的步骤：

- (1) 针对所给问题，定义问题的解空间
- (2) 确定易于搜索的解空间结构
- (3) 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索

### 剪枝函数：

- 为了避免生成那些不可能产生最佳解的问题状态，要不断地利用**剪枝函数**来剪掉那些实际上不可能产生所需解的活结点，以减少问题的计算量
- 具有剪枝函数的深度优先生成法称为回溯法

- 常用剪枝函数：
  - 用约束函数在扩展结点处剪去不满足约束的子树
  - 用限界函数剪去得不到最优解的子树

### 递归回溯：

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法

伪代码：

```
void backtrack (int t) //t为当前扩展结点在解空间树中的深度
{
    if (t>n) output(x);
    else
        for (int i=f(n,t); i<=g(n,t); i++) { //f(n,t)、g(n,t)分别为当前扩展结点处未搜索过的子树的
            起始、终止编号
            x[t]=h(i); //h(i)表示当前扩展结点处的第i个可选值
            if (constraint(t)&&bound(t)) //constraint(t)和bound(t)分别为当前扩展结点处的约束函数和
            限界函数
                backtrack(t+1); //以调用递归函数方式试探
        }
}
```

### 迭代回溯：

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程

伪代码：

```
void iterativeBacktrack () {
    int t=1;
    while (t>0) {
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t); i<=g(n,t); i++) { //f(n,t)、g(n,t)分别为当前扩展结点处未搜索过的子树的
                起始、终止编号
                x[t]=h(i); //当前扩展结点处的第i个可选值
                if (constraint(t)&&bound(t)) { //constraint(t)和bound(t)分别为当前扩展结点处的约束函数
                和限界函数
                    if (solution(t)) output(x);
                    else t++;
                }
            }
            else t--;
    }
}
```

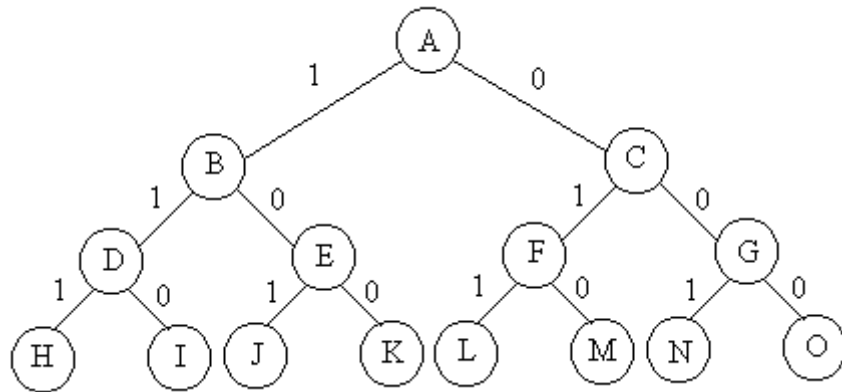
### 子集树与排列树：

- 子集树：
  - 当所给问题是从n个元素的集合S中找出S满足某种性质的子集时，相应的解空间树称为子集树。如n个物品的0-1背包问题所相应的解空间树
  - 遍历子集树需 $O(2^n)$ 计算时间
  - 伪代码：

```

void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=0;i<=1;i++) {
            x[t]=i;
            if (legal(t)) backtrack(t+1);
        }
}

```



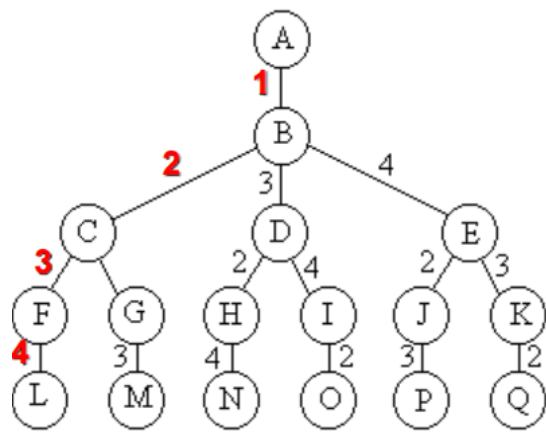
- 排列树:

- 当所给问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。如：旅行售货员问题的解空间树
- 遍历排列树需要 $O(n!)$ 计算时间
- 伪代码:

```

void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=t;i<=n;i++) {
            swap(x[t], x[i]);
            if (legal(t)) backtrack(t+1);
            swap(x[t], x[i]);
        }
}

```



## 5.2 旅行售货员问题 (TSP)

问题描述:

某售货员要到n个城市去推销商品，已知各城市之间的旅费，售货员要选定一条从驻地出发经过每个城市一次，最后回到驻地的路线，使总旅费最小

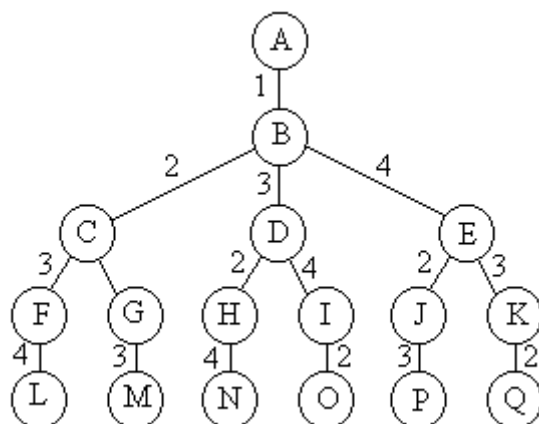
#### 图论语言描述：

- 设图 $G = (V, E)$  是一个加权连通图
- 要求：
  - $|V|=n$
  - 边  $(i,j)$  ( $i$ 不等于 $j$ )的费用 $c[i,j]$ 为正数
- 图G的一条周游路线是经过V中的每个顶点恰好一次的一条回路
- 周游路线的费用是这条回路上的所有边的费用之和
- TSP问题就是要在图G中找出费用最小的周游路线

#### 穷举法

- 由起始点出发的周游路线一共有 $(n-1)!$ 条，即等于除始结点外的 $n-1$ 个结点的排列数
- 旅行售货员问题是一个排列问题
- 穷举法的时间复杂度：  $O(n!)$

$|V|=4$ 旅行售货员问题的解空间：

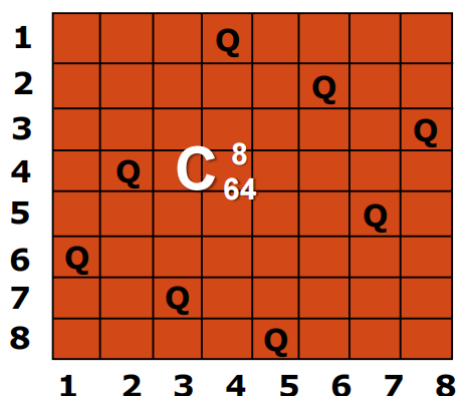


旅行售货员问题的解空间树为排列树，时间复杂度为 $O(n!)$

## 5.3 n皇后问题

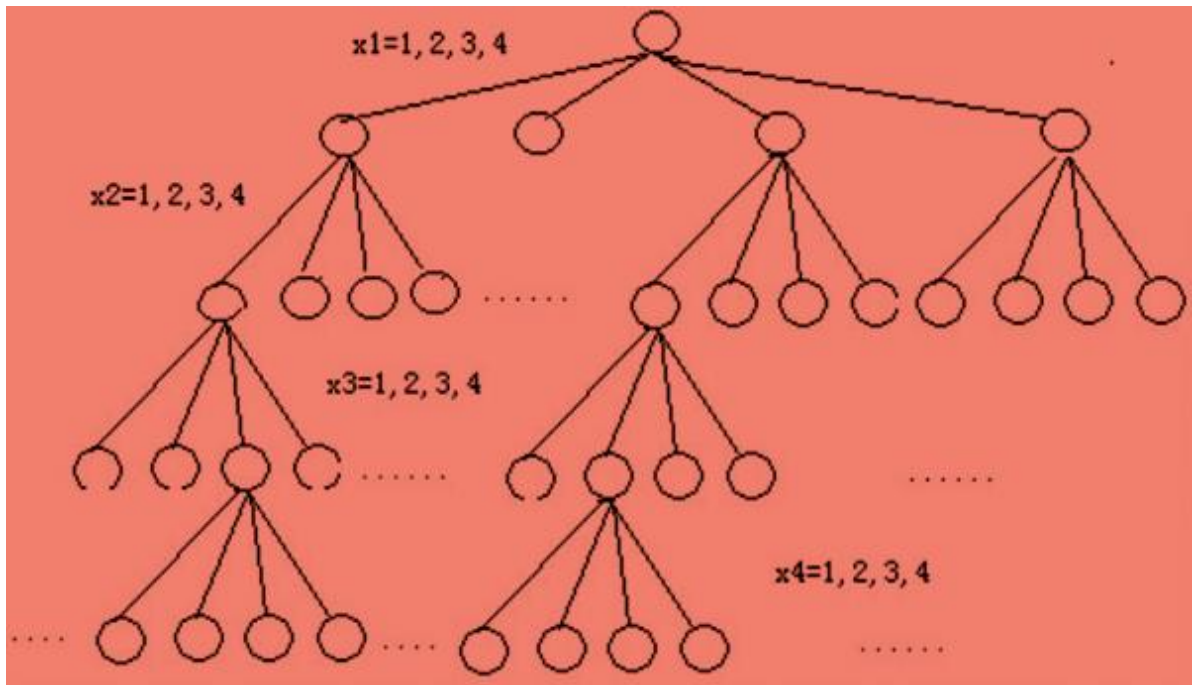
#### 问题描述：

- 在 $n \times n$ 格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子
- n后问题等价于在 $n \times n$ 格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上



### 分析:

- 解向量:  $(x_1, x_2, \dots, x_n)$
- 显约束:  $x_i = 1, 2, \dots, n$  (代表所在列)
- 隐约束:
  - 不同行/列:  $x_i \neq x_j$
  - 不处于同一正、反对角线:  $|i - j| \neq |x_i - x_j|$
- 状态空间树:



### 递归算法伪代码:

```
bool Queen::Place(int k)//检测当前行为k, 列为x[k]的皇后的位置是否合理(与前k-1行比较);当前行为k, 当前列为x[k]
{
    for (int j=1;j<k;j++)
        if ((abs(k-j)==abs(x[j]-x[k])) || (x[j]==x[k])) return false;//是否处于对角线或同一行/列上
    return true;
}
void Queen::Backtrack(int t)//第t行
{
    if (t>n) sum++;输出解;
    else
        for (int i=1;i<=n;i++) {
            x[t]=i;// 设定列值, 选取扩展结点
            if (Place(t)) Backtrack(t+1);//试探第t+1行
        }
}
```

## 5.4 迷宫问题

### 问题描述:

- 迷宫——用A[ ][ ]描述迷宫
- 当A[i,j]=0 表示该房间为空, 当A[i,j]=1 表示该房间封闭

- 解的形式：( $x_1, x_2, \dots, x_n$ ) ( $n$ 的取值不确定)
- 显约束： $X_i = \{1, 2, 3, 4\}$
- 隐约束条件：
  - 当 $A[i,j]=2$  表示该房间已经走过，不能再走
  - $A[i,j]=1$  表示该房间封闭，也不能走
  - 每步走的方向：上下左右

## 5.5 图的m着色问题

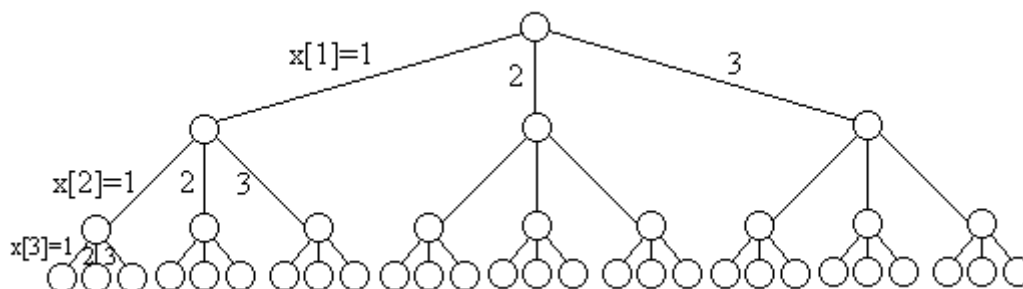
### 问题描述：

- 给定无向连通图 $G$ 和 $m$ 种不同的颜色。用这些颜色为图 $G$ 的各顶点着色，每个顶点着一种颜色。是否有一种着色法使 $G$ 中每条边的2个顶点着不同颜色
- 若一个图最少需要 $m$ 种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数 $m$ 为该图的色数。求一个图的色数 $m$ 的问题称为图的 $m$ 可着色优化问题

### 分析：

- 解向量： $(x_1, x_2, \dots, x_n)$ 表示顶点 $i$ 所着颜色 $x[i]$
- 可行性约束函数：顶点 $i$ 与已着色的相邻顶点颜色不重复

例： $n=3, m=3$ 的解空间树：



## 5.6 0-1背包问题

### 问题描述：

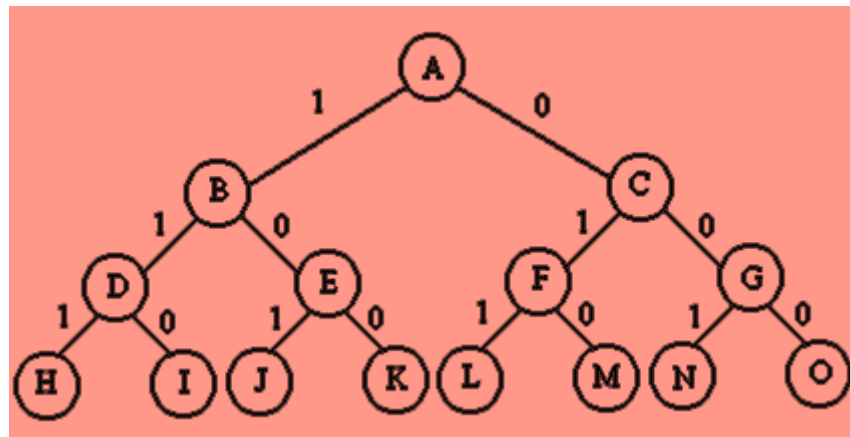
- $n$ 个物体 $v_i$ ，重量 $w_i$ 、价值 $p_i$ ， $0 \leq i \leq n-1$ ，背包的载重量 $C$
- $x_i$ ：物体 $v_i$ 被装入背包的情况， $x_i=0,1$

### 分析：

- 解向量： $(x_0, x_1, \dots, x_{n-1})$
- 状态空间树：有 $2^n$ 个叶子节点，其结点总数有 $2^{n+1}-1$ 个
- 根结点到叶结点的路径，是问题的可能解
- 第 $i$ 层的左子树表示物体 $x_i$ 被装入背包的情况；右子树表示物体 $x_i$ 未被装入背包的情况
- 可行性约束函数： $\sum_{i=0}^n w_i x_i \leq c$

例： $n=4$ 的解空间树：





#### 求解过程：

- 初始化：目标函数上界为0，物体按价值重量比的非增顺序排序
- 搜索过程：尽量沿左儿子结点前进，当不能沿左儿子继续前进时，就得到问题的一个部分解，并把搜索转移到右儿子子树
- 估计由部分解所能得到的最大价值：
  - 估计值高于当前上界：继续由右儿子子树向下搜索，扩大部分解，直到找到可行解；保存可行解，用可行解的值刷新目标函数的上界，向上回溯，寻找其它可行解（右子树中可能有最优解）
  - 若估计值小于当前上界：丢弃当前正在搜索的部分解，向上回溯（右子树中不可能有最优解）

#### 伪代码：

**C**——背包容量

**w[i]**——第i个物品重量

**p[i]**——第i个物品价值

**x[i]**——第i个物品是否装包

**bestp**——当前最优价值

**cp**——当前价值

**cw**——当前重量

```
Backtrack (int i)
{ if(i>n) // 到达叶节点
  { if (bestp<cp) bestp=cp; // 求出最优值
    return; }
  if(cw+w[i]<=c) // 可装入，试探进入左子树
  { x[i]=1;
    cw+=w[i];
    cp+=p[i];
    backtrack(i+1);
    cw-=w[i];
    cp-=p[i]; }
  // 装满背包
  if( bound(i+1)>bestp)
  { x[i]=0;
    backtrack(i+1);
  }
}
```

上界函数：

```

Bound(int i)
{// 计算上界
    cleft = c - cw; // 剩余容量
    b = cp;
    // 以物品单位重量价值递减序装入物品
    while (i <= n && w[i] <= cleft)
    { cleft -= w[i];
      b += p[i];
      i++;
    }

    // 装满背包
    if (i <= n)
        b += p[i]/w[i] * cleft;
    return b;
}

```

## 5.7 子集和问题

### 问题描述:

假设有 $n$ 个不同的正整数，找出这些数中所有使其和为正整数 $m$ 的组合

### 例:

$n=4$ ,  $w=\{11,13,24,7\}$ ,  $m=31$

则相应的子集和数问题的解是 $\{0,0,1,1\}$ ,  $\{1,1,0,1\}$

### 解空间:

- 解的形式:  $(x_1, x_2, \dots, x_n)$
- 显约束条件:  $x_i=0$ 或 $1$
- 隐约束条件: 子集和等于 $m$
- 解空间树

## 5.8 装载问题

### 问题描述:

- 装载问题是最优装载问题的变形
- 有一批共  $n$  个集装箱要装上2艘载重量分别为 $c_1$ 和 $c_2$ 的轮船，其中集装箱  $i$  的重量为 $w_i$ ，且装载问题要求确定是否有一个合理的装载方案可将这  $n$  个集装箱装上这2艘轮船。如果有，找出一种装载方案
- 当 $\sum_{i=1}^n w_i = c_1 + c_2$ 时，装载问题等价于子集和问题
- 当 $c_1 = c_2$ 且 $\sum_{i=1}^n w_i = 2c_1$ ，装载问题等价于划分问题

### 分析:

- 容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案
  - 首先将第一艘轮船尽可能装满
  - 将剩余的集装箱装上第二艘轮船
- 将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近 $c_1$ 。由此可知，装载问题等价于以下特殊的0-1背包问题

$$\begin{aligned} \max \quad & \sum_{i=1}^n w_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n w_i x_i \leq c_1 \\ & x_i \in \{0,1\}, 1 \leq i \leq n \end{aligned}$$

- 解空间：子集树
- 限界函数(不选择当前元素)：当前载重量 $cw$ +剩余集装箱的重量 $r \leq$ 当前最优载重量 $bestw$
- 用回溯法设计解装载问题的 $O(2^n)$ 计算时间算法。在某些情况下(即 $c_1 > 2^n$ )，该算法优于动态规划算法 $O(\min\{c_1, 2^n\})$

## 6 分支限界法

### 6.1 基本思想

概述：

- 一种求解**离散最优化**问题的计算分析方法，又称分支定界法
- 这种方法通常仅需计算和分析**部分允许解**，即可求得最优解
- 同回溯法一样适合解决组合优化问题

分支限界法与回溯法：

- 求解目标不同：回溯法的求解目标是找出解空间树中满足约束条件的**所有解**，而分支限界法的求解目标则是找出满足约束条件的**一个解**，或是在满足约束条件的解中找出在某种意义下的**最优解**
- 搜索方式不同：回溯法以**深度优先**的方式搜索解空间树，而分支限界法则以**广度优先**或以**最小耗费(或最大效益)优先**的方式搜索解空间树

其他：

- 在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点后就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中
- 此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程
- 这个过程一直持续到找到所需的解或活结点表为空时为止

常见的两种分支限界法：

- 队列式分支限界法：按照队列先进先出(FIFO)原则选取下一个节点为扩展节点
- 优先队列式分支限界法：按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点

代价函数（即限界函数）：

- 计算位置：搜索树的结点处
- 值：最大化问题是以该点为根的子树所有可行解的值得上界（最小化问题这是下界）
- 性质：对极大化问题父结点代价不小于子结点的代价（最小化问题相反）
- 界的含义：当前得到的可行解的目标函数的最大值（最小化问题想法）
- 界的初值：最大化问题初值为0（最小化初值为最大值）
- 界的更新：得到更好的可行解时

### 6.2 0-1背包问题

- 首先，要对输入数据进行预处理，将各物品依其单位重量价值从大到小进行排列。

- 在优先队列分支限界法中，结点的优先级由已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和。
- 算法首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点优先队列中。
- 当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。当扩展到叶节点时为问题的最优值。

## 6.3 装载问题

**问题描述：**

- 有一批共n个集装箱要装上2艘载重量分别为C1和C2的轮船，其中集装箱i的重量为wi，且
$$\sum_{i=1}^n w_i \leq c_1 + c_2$$
- 装载问题要求确定是否有一个合理的装载方案可将这 n个集装箱装上这2艘轮船。如果有，找出一种装载方案

**分析：**

容易证明：如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案：

- 首先将第一艘轮船尽可能装满
- 将剩余的集装箱装上第二艘轮船

**队列式分支限界法：**

- 在算法的while循环中，首先检测当前扩展结点的左儿子结点是否为可行结点。如果是则将其加入到活结点队列中。然后将其右儿子结点加入到活结点队列中(右儿子结点一定是可行结点)。2个儿子结点都产生后，当前扩展结点被舍弃
- 技巧：活结点队列中的队首元素被取出作为当前扩展结点，由于队列中每一层结点之后都有一个尾部记-1，故在取队首元素时，活结点队列一定不空。当取出的元素是-1时，再判断当前队列是否为空。如果队列非空，则将尾部标记-1加入活结点队列，算法开始处理下一层的活结点

**算法改进：**

- 节点的左子树表示将此集装箱装上船，右子树表示不将此集装箱装上船。设bestw是当前最优解；Ew是当前扩展结点所相应的重量；r是剩余集装箱的重量。
- 则当Ew+r≤bestw时，可将其右子树剪去。
- 为了确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新bestw的值。

**构造最优解：**

- 为了在算法结束后能方便地构造出与最优值相应的最优解，算法必须存储相应子集树中从活结点到根结点的路径。为此目的，可在每个结点处设置指向其父结点的指针，并设置左、右儿子标志
- 找到最优值后，可以根据parent回溯到根节点，找到最优解。

**优先队列式分支限界法：**

- 装载问题的优先队列式分支限界法用最大优先队列存储活结点表。活结点x在优先队列中的优先级定义为从根结点到结点x的路径所相应的载重量再加上剩余集装箱的重量之和。
- 优先队列中优先级最大的活结点成为下一个扩展结点。以结点x为根的子树中所有结点相应的路径的载重量不超过它的优先级。子集树中叶结点所相应的载重量与其优先级相同。
- 在优先队列式分支限界法中，一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解。此时可终止算法。

## 6.4 旅行售货员问题 (TSP)

**问题描述：**

- 某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小。
- 路线是一个带权图。图中各边的费用(权)为正数。图的一条周游路线是包括V中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。
- 旅行售货员问题的解空间树，从树的根结点到任一叶结点的路径定义了图的一条周游路线。旅行售货员问题要在图G中找出费用最小的周游路线。

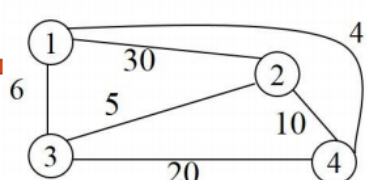
例：

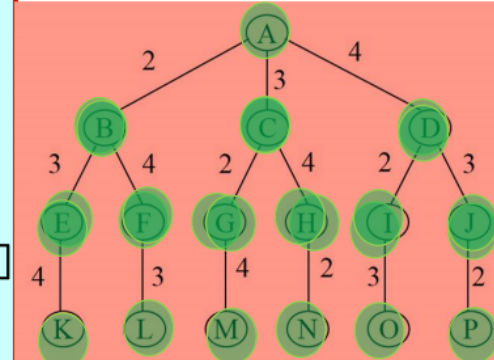
## 实例

### 2 TSP问题的两种分支定界法

**队列式分支限界法：**

- [A] B, C, D => B, C, D
- [B, C, D] E, F => E, F
- [C, D, E, F] G, H => G, H
- [D, E, F, G, H] I, J => I, J
- [E, F, G, H, I, J] K(59)
- [1, 2, 3, 4]
- [F, G, H, I, J] L(66)
- [G, H, I, J] M(25) [1, 3, 2, 4]
- [H, I, J] 1-3-4(26)
- [I, J] 0 => 0(25)
- [J] P => P(59)





分析：

- 算法开始时创建一个最小堆,用于表示活结点优先队列。堆中每个结点的子树费用的下界lcost值是优先队列的优先级。接着算法计算出图中每个顶点的最小费用出边并用minout记录。如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法即告结束。如果每个顶点都有出边，则根据计算出的minout作算法初始化。
- 算法的while循环体完成对排列树内部结点的扩展。对于当前扩展结点，算法分2种情况进行处理：
  1. 首先考虑s=n-2的情形，此时当前扩展结点是排列树中某个叶结点的父结点。如果该叶结点相应一条可行回路且费用小于当前最小费用，则将该叶结点插入到优先队列中，否则舍去该叶结点。
  2. 当s<n-2时，算法依次产生当前扩展结点的所有儿子结点。由于当前扩展结点所相应的路径是x[0:s]，其可行儿子结点是从剩余顶点x[s+1:n-1]中选取的顶点x[i]，且(x[s],x[i])是所给有向图G中的一条边。对于当前扩展结点的每一个可行儿子结点，计算出其前缀(x[0:s], x[i])的费用cc和相应的下界lcost。当lcost<bestc时，将这个可行儿子结点插入到活结点优先队列中。
- 算法中while循环的终止条件是排列树的一个叶结点成为当前扩展结点
  - 当s=n-1时，已找到的回路前缀是x[0:n-1]，它已包含图G的所有n个顶点。
  - 因此，当s=n-1时，相应的扩展结点表示一个叶结点。此时该叶结点所相应的回路的费用等于cc和lcost的值。
  - 剩余的活结点的lcost值不小于已找到的回路费用。它们都不可能导致费用更小的回路。
  - 因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。
- 算法结束时返回找到的最小费用，相应的最优解由数组X给出

