

软件工程专业导论

问题求解与程序设计

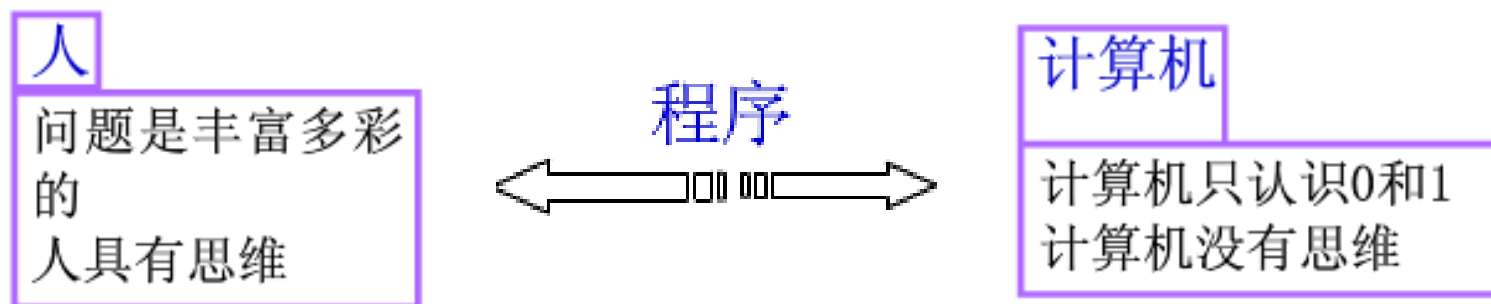
学习目标：

- 求解过程
- 算法



1. 问题求解

需要解决问题的人 \longleftrightarrow 没有思维的计算机



人和计算机通过程序进行沟通

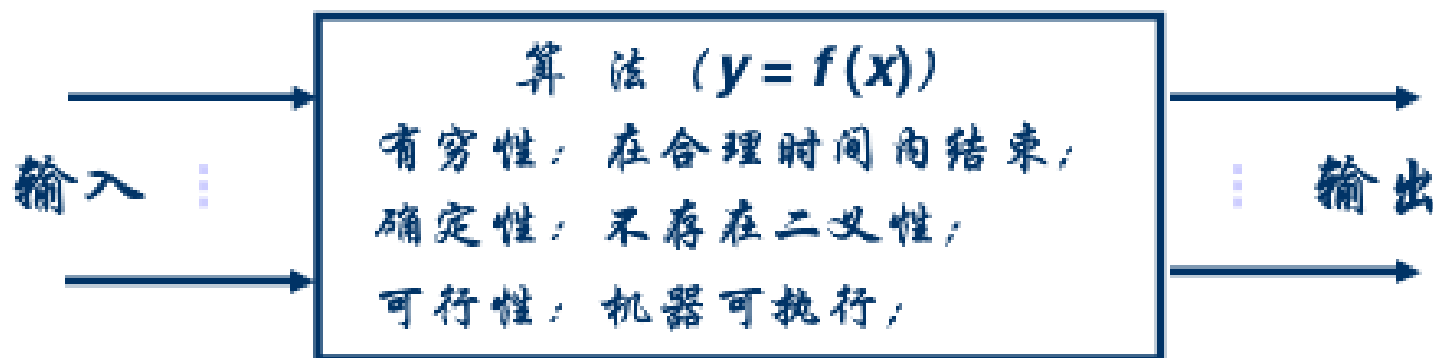


1. 问题求解



2. 算法

- 算法是一个有穷规则的集合，它用规则规定了解决某一特定类型问题的运算序列，或者规定了任务执行或问题求解的一系列步骤
- 基本特征：
 - 输入、输出、有穷性、确定性、可行性



2. 算法

寻找两个正整数的最大
公约数的欧几里德算法

输入：正整数 M 和正整数 N

输出： M 和 N 的最大公约数(设 $M>N$)

算法步骤：

Step 1. M 除以 N ,记余数为 R

Step 2. 如果 R 不是 0 ，将 N 的值赋给 M ,
 R 的值赋给 N , 返回Step 1; 否则, 最大
公约数是 N , 输出 N , 算法结束。

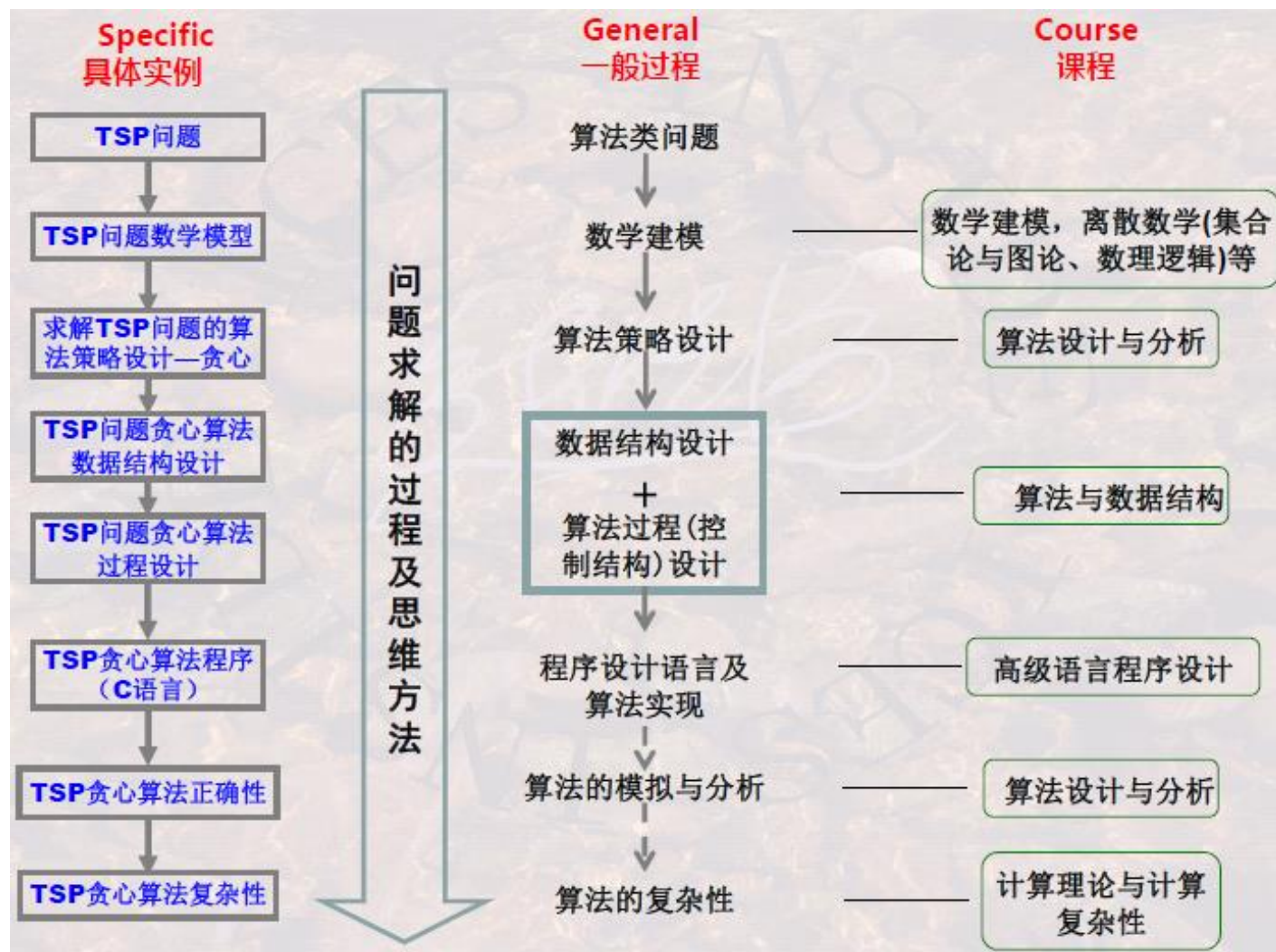
基本运算：除法、赋值、逻辑判断

典型的“重复/循环”与“迭代”



	M	N	R	最大公约数
具体问题	32	24		?
算法计算过程				
1	32	24	8	
2	24	8	0	8
具体问题	31	11		?
算法计算过程				
1	31	11	9	
2	11	9	2	
3	9	2	1	
4	2	1	0	1

2. 算法



2. 算法

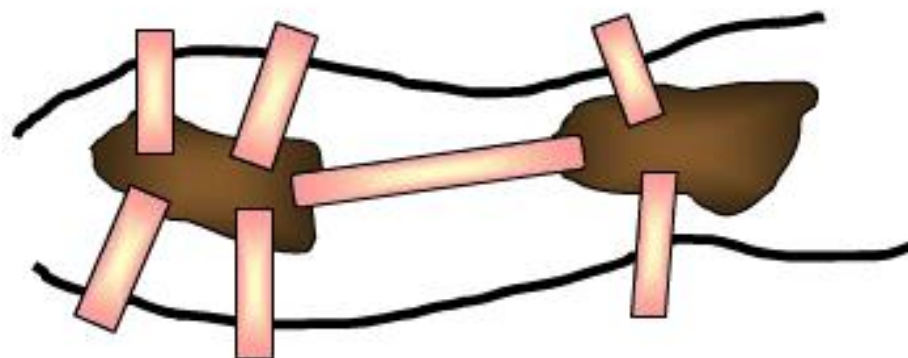


数学建模是用数学语言描述实际现象的过程，即建立数学模型的过程。
数学模型是对实际问题的一种数学表述，是关于部分现实世界为某种目的的一个抽象的简化的数学结构。

2. 算法

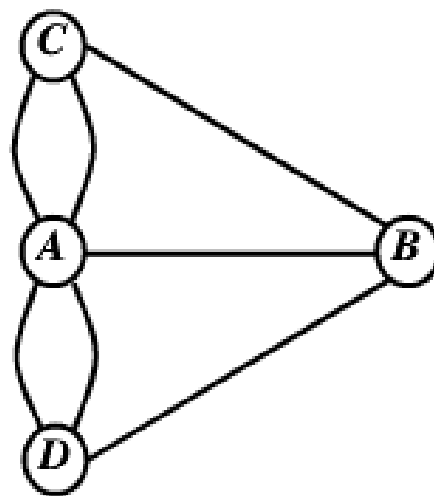
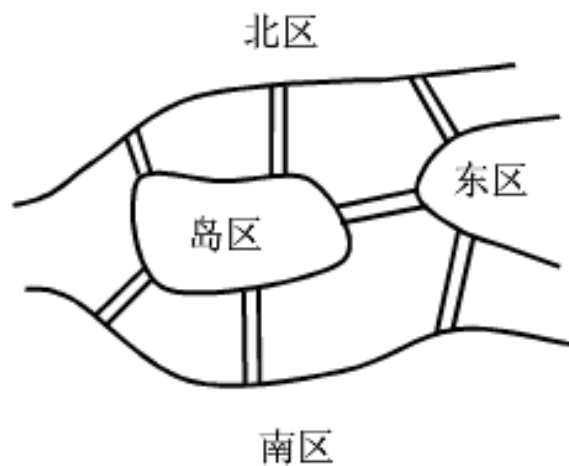
● 哥尼斯堡七桥问题

【问题】 17世纪的东普鲁士有一座哥尼斯堡城（现在叫加里宁格勒，在波罗的海南岸），普雷格尔河流过城中，在河中有两座小岛，全城共有七座桥将4个城区连接起来，于是，产生了一个有趣的问题：一个人是否能在一次步行中穿越全部的七座桥后回到起点，且每座桥只经过一次。



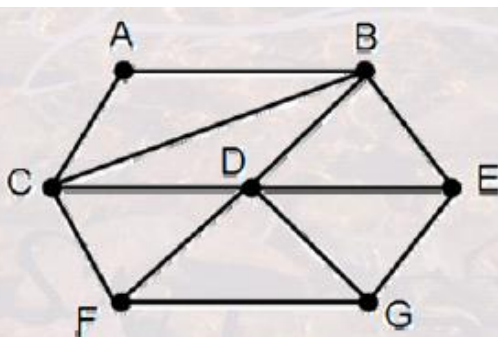
2. 算法

- 哥尼斯堡七桥问题 → 图 (Graph)
 - 陆地 --> 节点
 - 连接陆地的桥梁 --> 连接节点的边



2. 算法

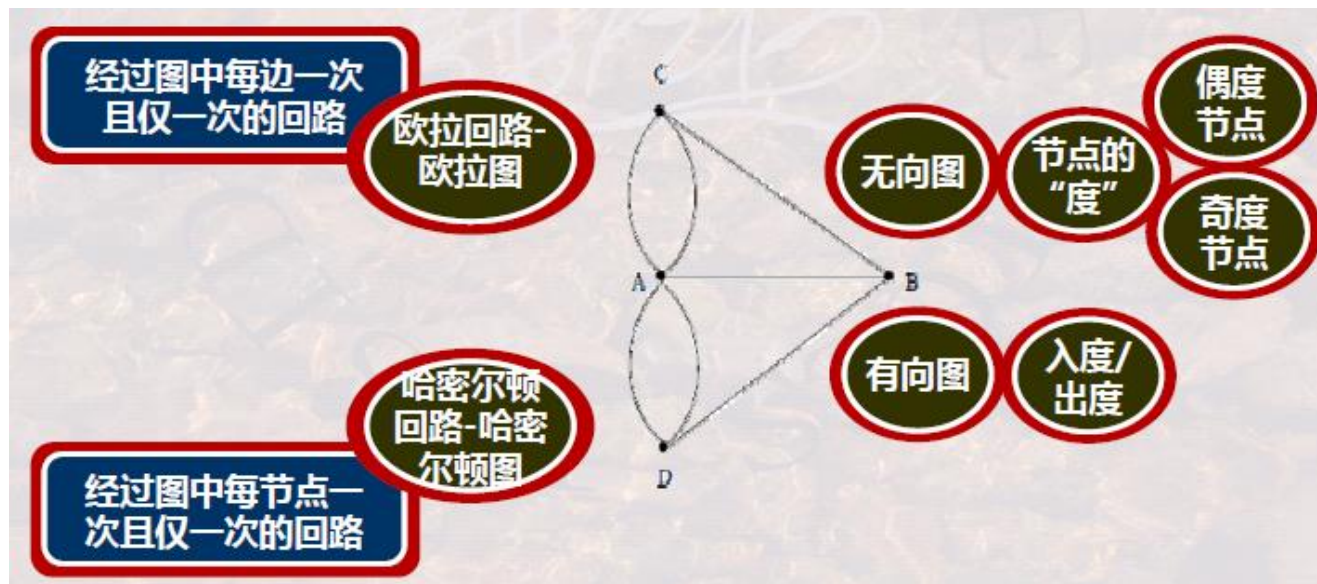
- 哥尼斯堡七桥问题 → 图（Graph）
 - 寻找走遍这7座桥且只许走过每座桥一次最后又回到原出发点的路径
- 一般性问题
 - 对给定的任意一个河道图与任意多座桥判定可能不可能每座桥恰好走过一次



如能抽象成数学模型，则可将一个具体问题的求解，推广为一类问题的求解！

2. 算法

- 哥尼斯堡七桥问题 → 图 (Graph)
 - 连通：两个节点间有路径相连接
 - 可达：从一个节点出发能够到达另一个节点
 - 回路：从一个节点出发最后又回到该节点的一条路径



2. 算法

- 哥尼斯堡七桥问题 → 图 (Graph)
 - 凡是由偶度点组成的连通图，一定可以一笔画成。画时可以把任一偶度点为起点，最后一定能以这个点为终点画完此图；
 - 凡是只有两个奇度点的连通图，其余都为偶度点，一定可以一笔画成。画时必须以一个奇度点为起点，另一个奇度点为终点；
 - 其余情况都不能一笔画出

2. 算法

- 哥尼斯堡七桥问题 → 图 (Graph)

【**算法**】 设函数 **EulerCircuit** 求解七桥问题，算法描述如下：
输入：二维数组 **mat[4][4]**
功能：计算七桥问题中奇数桥的结点个数
输出：通奇数桥的结点个数 **count**

Step1: **count** 初始化为0

Step2: 下标 **i** 从0到 **n-1** 重复执行下列操作：

 step2.1: 计算第 **i** 行元素之和 **degree**;

 step2.2: 如果 **degree** 为奇数，则 **count+=1**;

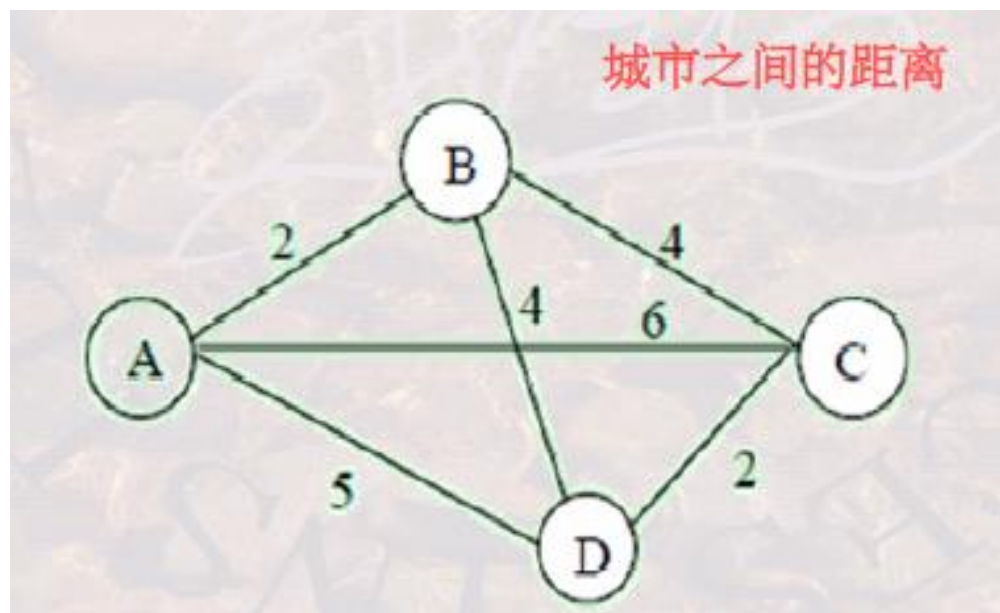
Step3: 返回 **count**

2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题), 威廉哈密尔顿爵士和英国数学家克克曼T.P.Kirkman于19世纪初提出TSP问题
- TSP问题: 有若干城市, 任何两个城市之间的距离都是确定的, 现要求一旅行商从某城市出发必须经过每一城市且只能在每个城市逗留一次, 最后回到原出发城市, 问如何事先确定好一条最短的路线使其旅行的费用最少

2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)



经过图中每节点一次且仅一次的回路

哈密尔顿回路-哈密尔顿图

赋权-哈密尔顿图

经过图中每节点一次且仅一次的回路；连接节点的边有“权值”

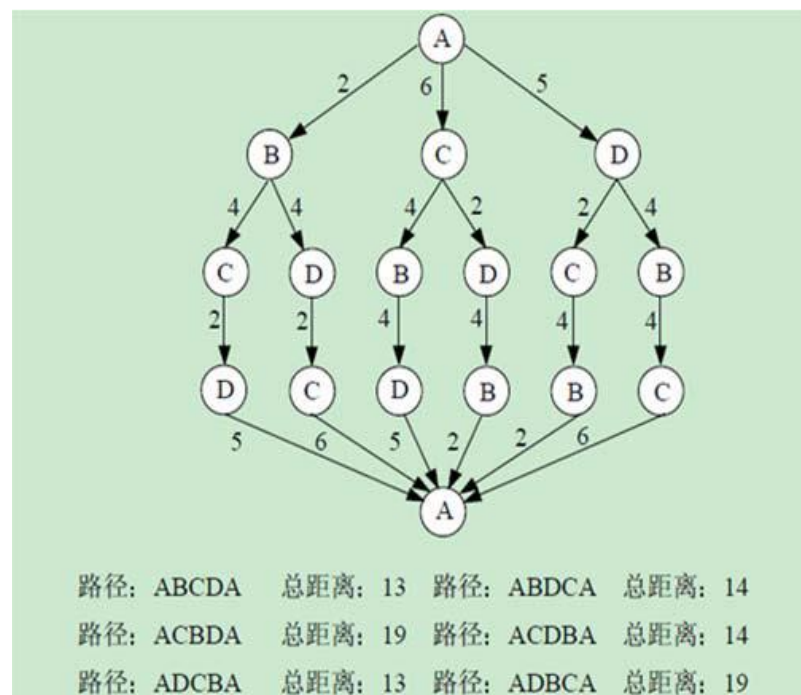
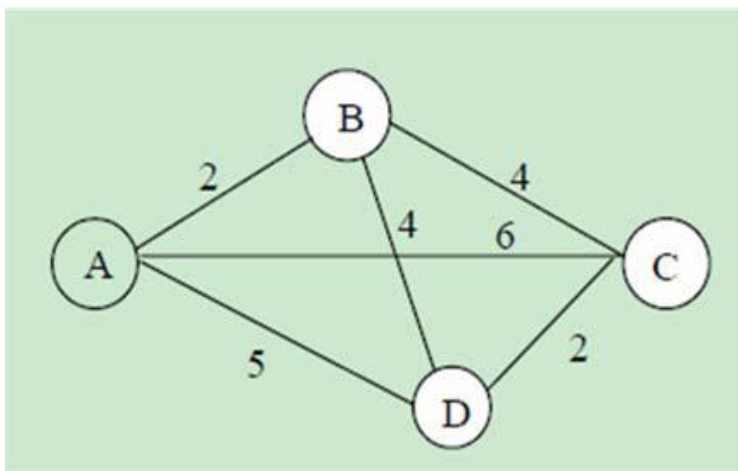
2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)



2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)
 - 遍历算法：列出每一条可供选择的路线，计算出每条路线的总里程，最后从中选出一条最短的路线。



2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)
 - 遍历算法：列出每一条可供选择的路线，计算出每条路线的总里程，最后从中选出一条最短的路线。
 - 存在的问题：组合爆炸
 - 路径组合数目： $(n-1)!$
 - 20个城市，遍历总数 1.216×10^{17}
 - 计算机以每秒检索1000万条路线的计算速度，需386年

2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)
 - 其他算法?
 - 遍历、搜索算法
 - 分治算法
 - 贪心算法
 - 动态规划算法
 - 遗传算法
 -

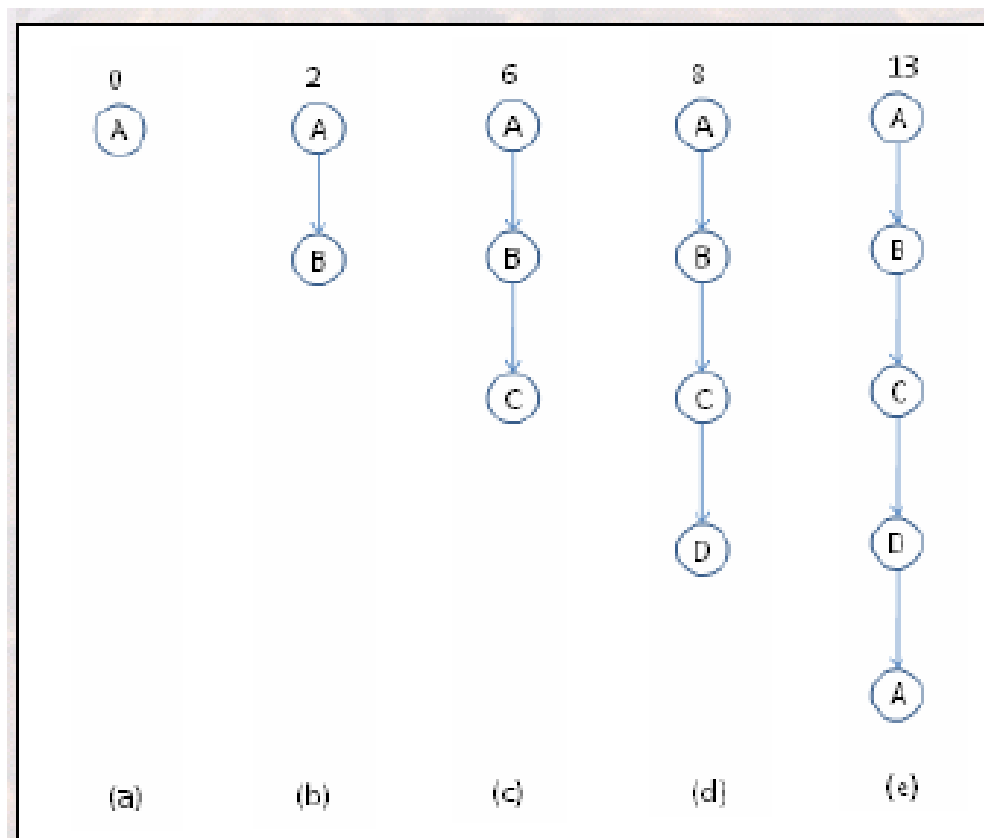
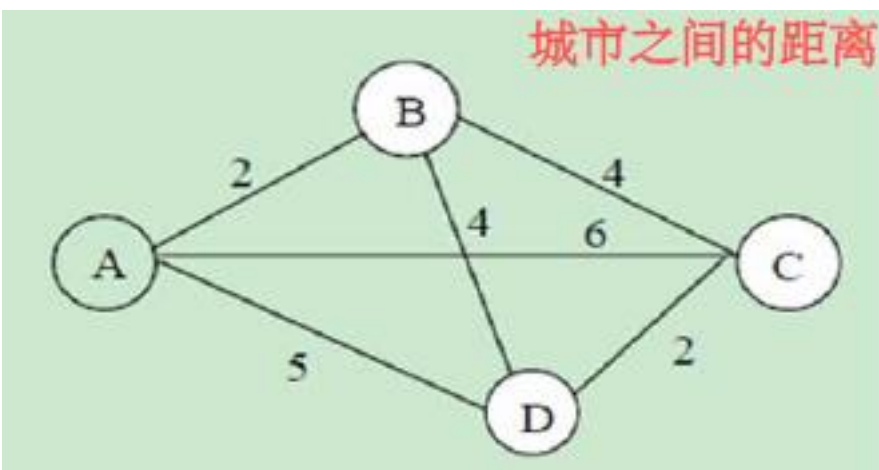


2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)
 - 贪心算法
 - 从某一个城市开始，每次选择一个城市，直到所有城市都被走完
 - 每次在选择下一个城市的时候，只考虑当前情况，保证迄今为止经过的路径总距离最短

2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)



2. 算法



- 算法的数据结构设计：问题或算法相关的数据之间的逻辑关系及存储关系的设计
 - 如何将数学模型中的数据转为计算机可以存储和处理的数据？
- 算法的控制结构设计：算法的计算规则或计算步骤设计
 - 如何构造和表达处理的规则，以便能够按规则逐步计算出结果？

2. 算法

- TSP问题(Traveling Salesman Problem, 旅行商问题)

城市映射为编号: A---1, B---2, C---3, D---4

城市间距离关系: 表或二维数组D, 用 $D[i][j]$ 或 $D[i,j]$ 来确定欲处理的每一个元素

城市编号	1	2	3	4
1		2	6	5
2	2		4	4
3	6	4		2
4	5	4	2	

D

$D[2][3]$

访问路径/解: 一维数组S, 用 $S[j]$ 来确定每一个元素

S	1	4	3	2
	S[1]	S[2]	S[3]	S[4]

{A->D->C->B->A}

2. 算法

- TSP问题的贪心算法
 - 依次访问过的城市编号被记录在 $S[1], S[2], \dots, S[N]$ 中, 即第 l 次访问的城市记录在 $S[l]$ 中
 - 从第1个城市开始访问起, 将城市编号1赋值给 $S[1]$
 - 第 l 次访问的城市为城市 j , 其距第 $l-1$ 次访问城市的距离最短

$S[1] \ S[2] \ S[3] \ S[4]$
S

1			
---	--	--	--

$l = 2, 3, 4$ ---当前要找第几个
 $j = 1, 2, 3, 4$ ---城市号

D

城市编号	1	2	3	4
1		2	6	5
2	2		4	4
3	6	4		2
4	5	4	2	

2. 算法

- TSP问题的贪心算法
 - 依次访问过的城市编号被记录在 $S[1], S[2], \dots, S[N]$ 中, 即第 l 次访问的城市记录在 $S[l]$ 中
 - 从第1个城市开始访问起, 将城市编号1赋值给 $S[1]$
 - 第 l 次访问的城市为城市 j , 其距第 $l-1$ 次访问城市的距离最短

Start of the Algorithm

- (1) $S[1]=1$;
- (2) $Sum=0$;
- (3) 初始化距离数组 $D[N, N]$;
- (4) $l=2$;
- (5) 从所有未访问过的城市中查找距离 $S[l-1]$ 最近的城市 j ;
- (6) $S[l]=j$;
- (7) $l=l+1$;
- (8) $Sum=Sum+Dtemp$;
- (9) 如果 $l \leq N$, 转步骤(5), 否则, 转步骤(10);
- (10) $Sum=Sum+D[1, j]$;
- (11) 逐个输出 $S[N]$ 中的全部元素;
- (12) 输出 Sum 。

End of the Algorithm

2. 算法

● TSP问题的贪心算法

(5.1) $K=2$;
 (5.2) 将 $Dtemp$ 设为一个大数(比所有两个城市之间的距离都大)
 (5.3) $L=1$;
 (5.4) 如果 $S[L]==K$, 转步骤5.8; //该城市已出现过, 跳过
 (5.5) $L=L+1$;
 (5.6) 如果 $L < I$, 转5.4;
 (5.7) 如果 $D[K, S[I-1]] < Dtemp$, 则 $j=K$; $Dtemp = D[K, S[I-1]]$;
 (5.8) $K=K+1$;
 (5.9) 如果 $K \leq N$, 转步骤5.3。

S S[1] S[2] S[3] S[4]
 1

$I = 2, 3, 4$ ---当前要找第几个

$L = 1, 2, \dots, I-1$ ---从第1个访问过的, 到第 $I-1$ 个访问过的

$K = 2, 3, 4$ ---城市号

2. 算法

- TSP问题的贪心算法

S

S[1]

S[2]

S[3]

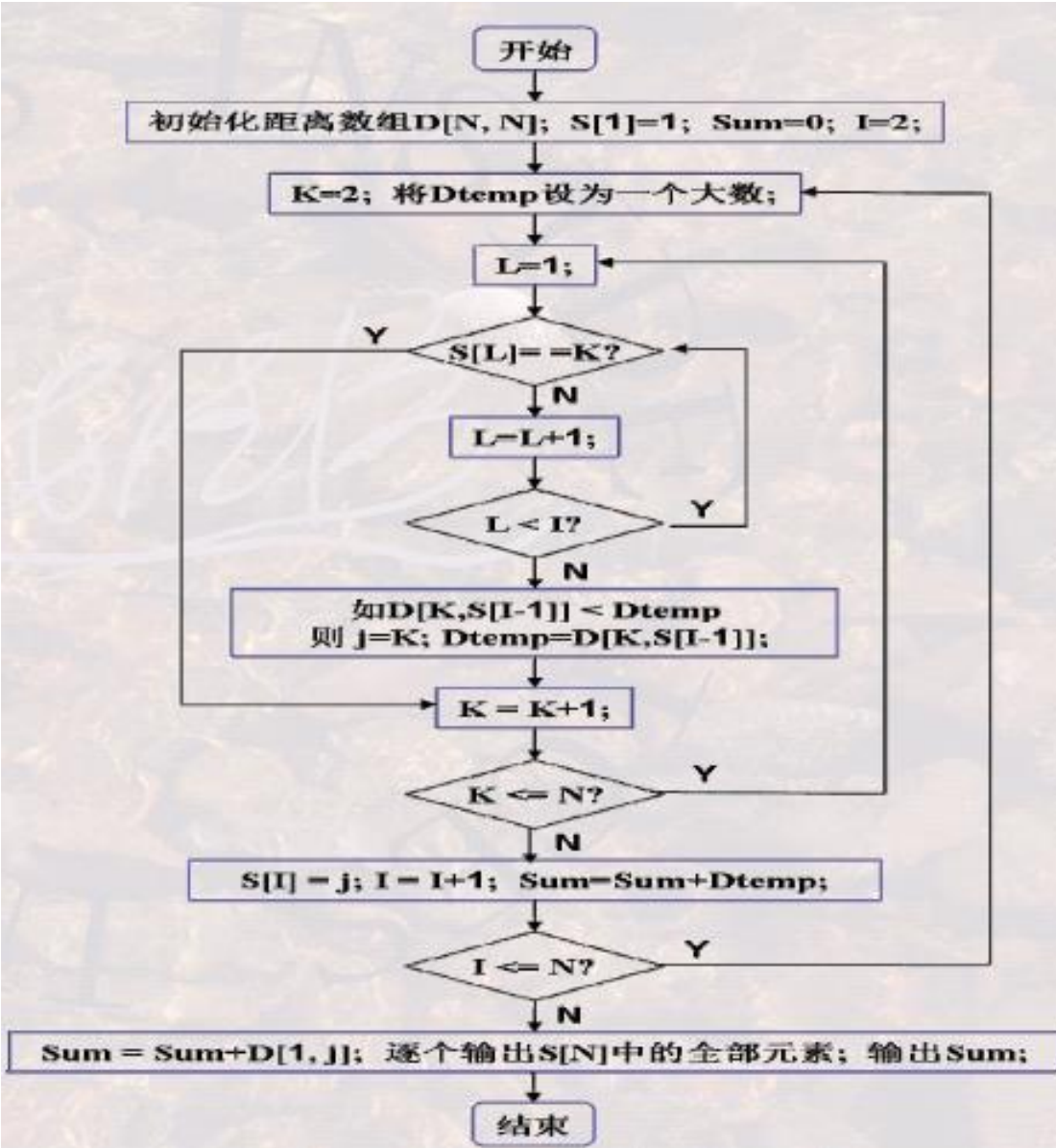
S[4]

1

$I = 2, 3, 4$ ---当前要找第几个

$K = 2, 3, 4$ ---城市号

$L = 1, 2, \dots, I-1$ ---从第1个访问过的, 到第I-1个访问过的



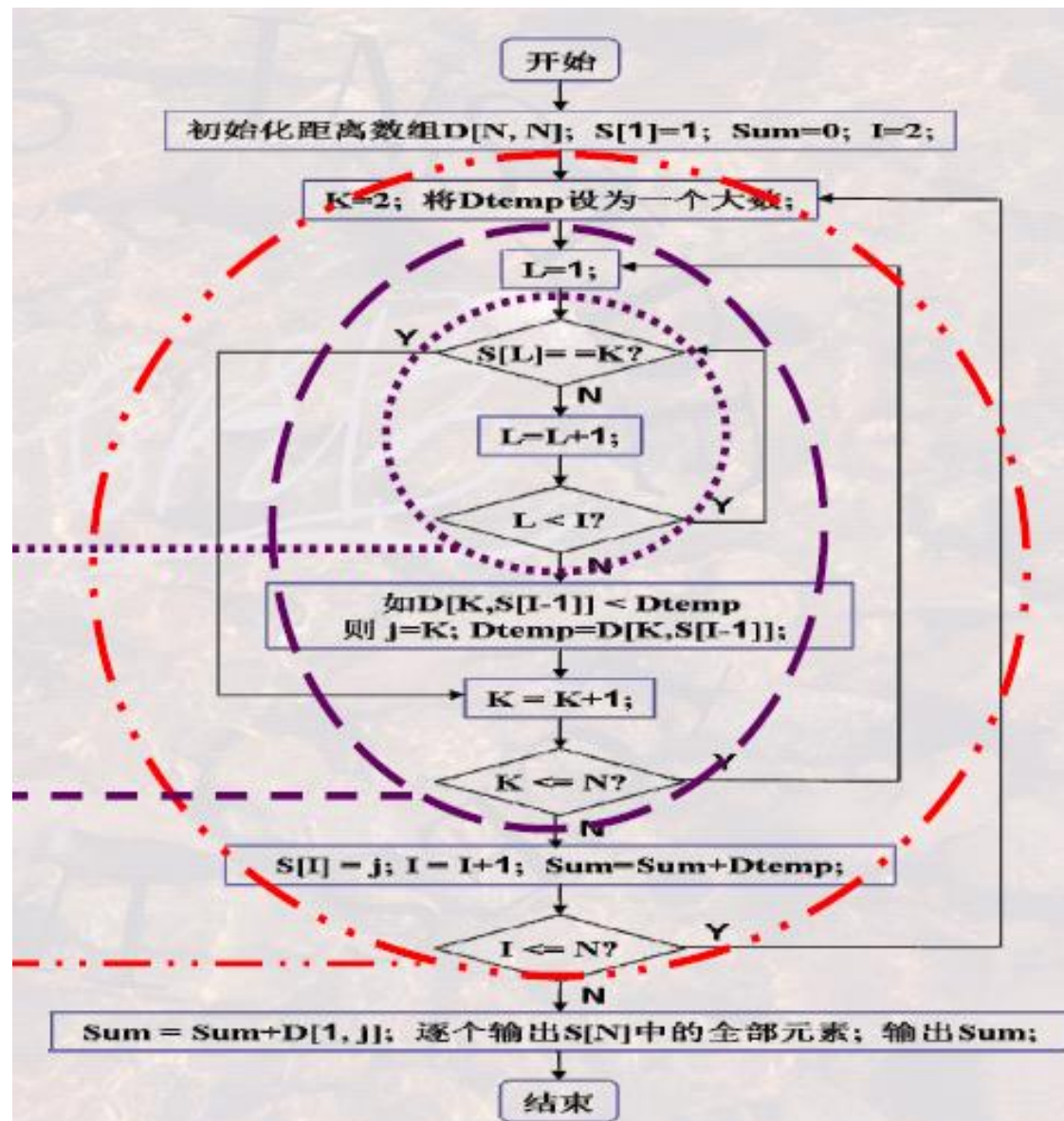
2. 算法

● TSP问题的贪心算法

内层循环, L 从 1 至 $I-1$, 循环判断第 K 个城市是否是已访问过的城市, 如是则不参加最小距离的比较;

中层循环, K 从第 2 个城市至第 N 个城市循环, 判断 $D[K, S[I-1]]$ 是否是最小值, j 记录了最小距离的城市号 K 。

外层循环, I 从 2 至 N 循环; $I-1$ 个城市已访问过, 正在找与第 $I-1$ 个城市最近距离的城市; 已访问过的城市号存储在 $S[]$ 中。



2. 算法



2. 算法



2. 算法

- 算法是正确的吗?
- 算法获得的解是最优的吗?
- 两种评价方法:
 - 证明方法：利用数学方法证明；
 - 仿真分析方法：产生或选取大量的、具有代表性的问题实例，利用该算法对这些问题实例进行求解，并对算法产生的结果进行统计分析

2. 算法

- 算法的复杂性分析
 - 时间复杂性：如果一个问题的规模是 n ，解这一问题的某一算法所需要的时间为 $T(n)$ ，它是 n 的某一函数， $T(n)$ 称为这一算法的“时间复杂性”
 - 大 O 记法
 - 空间复杂性：算法在执行过程中所占存储空间的大小

2. 算法

- 算法的复杂性分析

```
sum=0;                                (1次)  
for( i=1; i<=n; i++)                  (n次)  
{ for( j=1; j<=n; j++)                (n²次)  
    { sum++; }                        (n²次)  
}
```

解: $T(n) = 2n^2 + n + 1 = O(n^2)$

2. 算法

$O(n^3)$ 与 $O(3^n)$ 的差别, $O(n!)$ 与 $O(n^3)$ 的差别

问题规模n	计算量
10	10!
20	20!
100	100!
1000	1000!
10000	10000!

$$20! = 1.216 \times 10^{17}$$

$$20^3 = 8000$$

$O(n^3)$	$O(3^n)$
0.2秒	4×10^{28} 秒 =1015年
注：每秒百万次，n=60，1015年相当于10亿台计算机计算一百万年	

$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^b)$

$O(b^n)$, $O(n!)$

2. 算法

- 递归算法与实现
- 分治策略的递归
- 回溯策略的递归

2. 算法

- 递归算法与实现

- 一个使用函数自身给出定义的函数称为递归函数
- 一个直接或间接地调用自身的算法称为递归算法

2. 算法

● 递归算法与实现

阶乘的递归定义：

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

写成函数形式则为：

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

2. 算法

- 递归算法与实现

```
int fact( int n ){  
    if( n==1 ) return 1;  
    else return ( n*fact( n-1 ) );  
}
```

递归调用自身

2. 算法

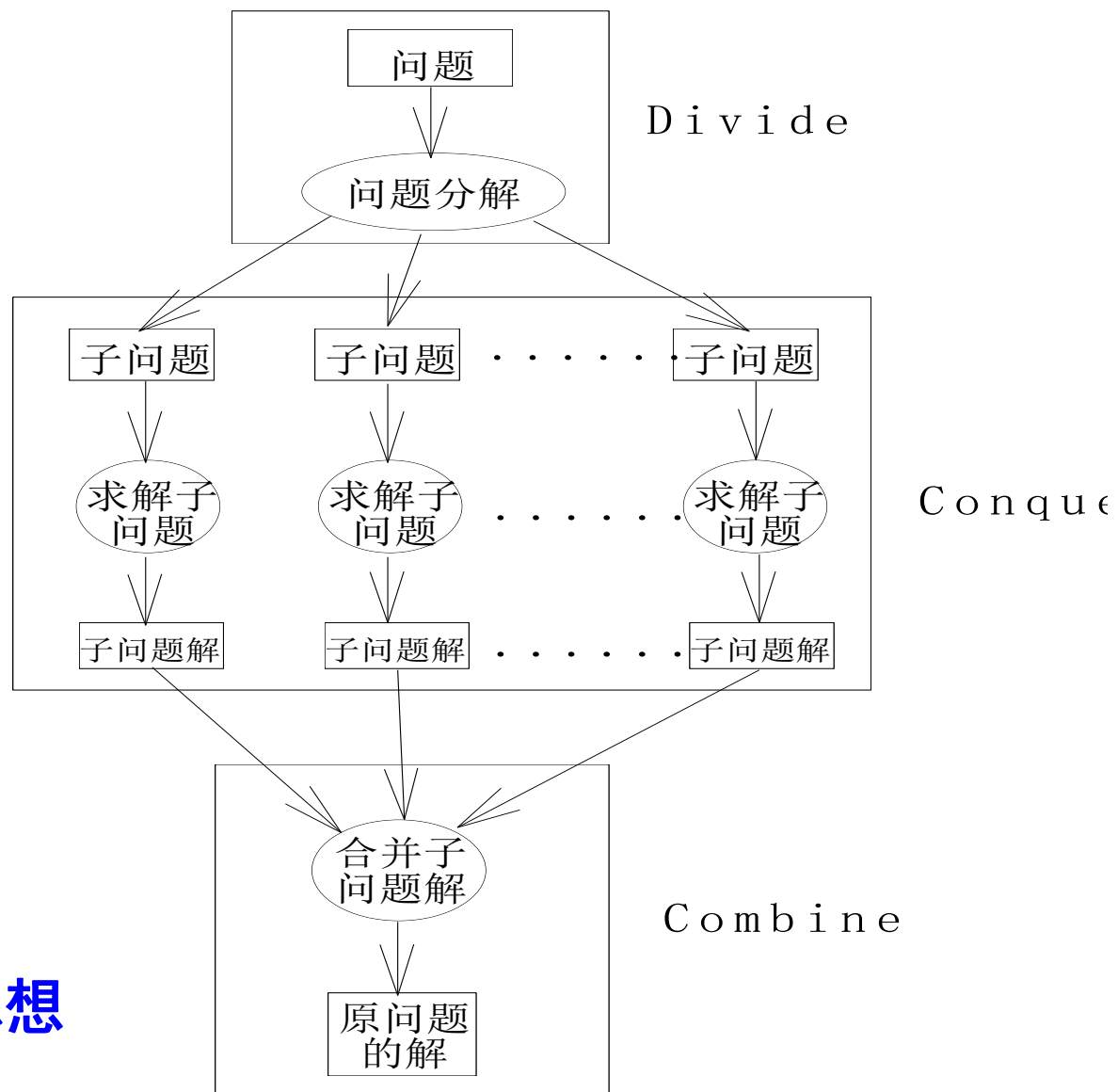
- 求解同一个问题可以有多种算法

```
int fact( int n ){
    if( n==1 ) return 1;
    else return ( n*fact( n-1 ) );
}
```

```
int f( int n ) {
    int i, product;
    product=1
    if( n==0 ) return 1;
    for( i=1; i<=n; i++ ) {
        product = product * i ;
    }
    return product;
}
```

2. 算法

- 分治策略的递归



分治策略的基本思想

2. 算法

- 分治策略的递归

1. 将一个规模为 n 的问题分解为 k 个规模较小的子问题
2. 这些子问题相互独立且与原问题相同（同一个算法求解）
3. 递归地解这些子问题，然后将各子问题的解合并得到原问题的解

2. 算法

- 分治策略的递归

算法的设计模式：

```
Divide_and_Conquer(P) {  
    If ( |P| <=n0)   Adhoc(P);
```

不可分解子问题的基本
子算法，如赋初始值。

```
    Divide P into small subinstances ;
```

分解成子问题

```
    P1, P2, ... , Pk;
```

```
    for(i=1;i<=k; i++) yi = Divide_and_Conquer(Pi);
```

递归调用

```
    Return Merge(y1,y2, ... , yk);
```

合并

```
}
```

2. 算法

- 归并排序 (Merge Sort)

排序问题： 给定的 n 个元素集合 $a[0: n-1]$ 进行排序。

归并排序算法基本思想：

1. 当 $n = 1$ 时终止排序。
2. 否则，将待排序元素分成大致相同的两个子集合，分别对两个子集合进行排序。
3. 将排好序的两个子集合合并排序。

2. 算法

归并排序算法基本思想：

1. 当 $n=1$ 时终止排序。
2. 否则，将待排序元素分成大致相同的两个子集合，分别对两个子集合进行排序。
3. 将排好序的两个子集合合并排序。

```
void MergeSort( int a[ ], int left, int right)
```

```
{
```

```
    if (left < right) { // 至少有两个元素
```

```
        int mid = (left + right)/2; //取中点
```

```
        MergeSort( a, left, mid);
```

左半部分递归调用

```
        MergeSort( a, mid+1, right);
```

右半部分递归调用

```
        Merge( a, left, mid, mid+1, right);
```

合并

```
    }
```

```
}
```

2. 算法

- 归并排序 (Merge Sort)

6 5 3 1 8 7 2 4

2. 算法

- 回溯策略的递归

0-1背包问题:

$$Q = \text{Max} \{ p_1 x_1 + p_2 x_2 + \dots + p_n x_n \}$$

$$(x_1, \dots, x_n) \in X$$

$$\text{其中 } X = \{ (x_1, x_2, \dots, x_n) \mid w_1 x_1 + w_2 x_2 + \dots + w_n x_n \leq C \}$$

例. $N = 3$ 时的0-1背包问题，考虑下面的具体例子。

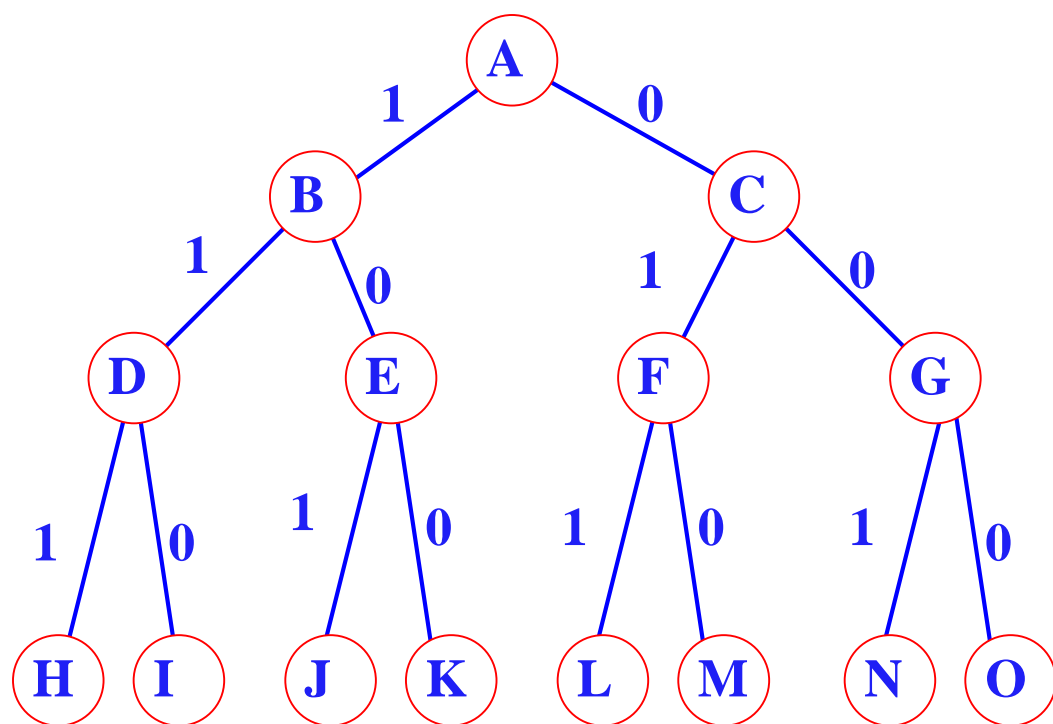
$$W = [16, 15, 15], \quad P = [45, 25, 25], \quad C = 30。$$

分析：问题的解空间为 $\{ (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) \}$



2. 算法

- 回溯策略的递归



0-1背包问题:

问题参数:

$W = [16 , 15 , 15] ,$

$P = [45 , 25 , 25] ,$

$C = 30 .$

问题的解空间:

$\{ (0,0,0), (0,1,0),$
 $(0,0,1), (1,0,0),$
 $(0,1,1), (1,0,1),$
 $(1,1,0), (1,1,1) \}$

2. 算法

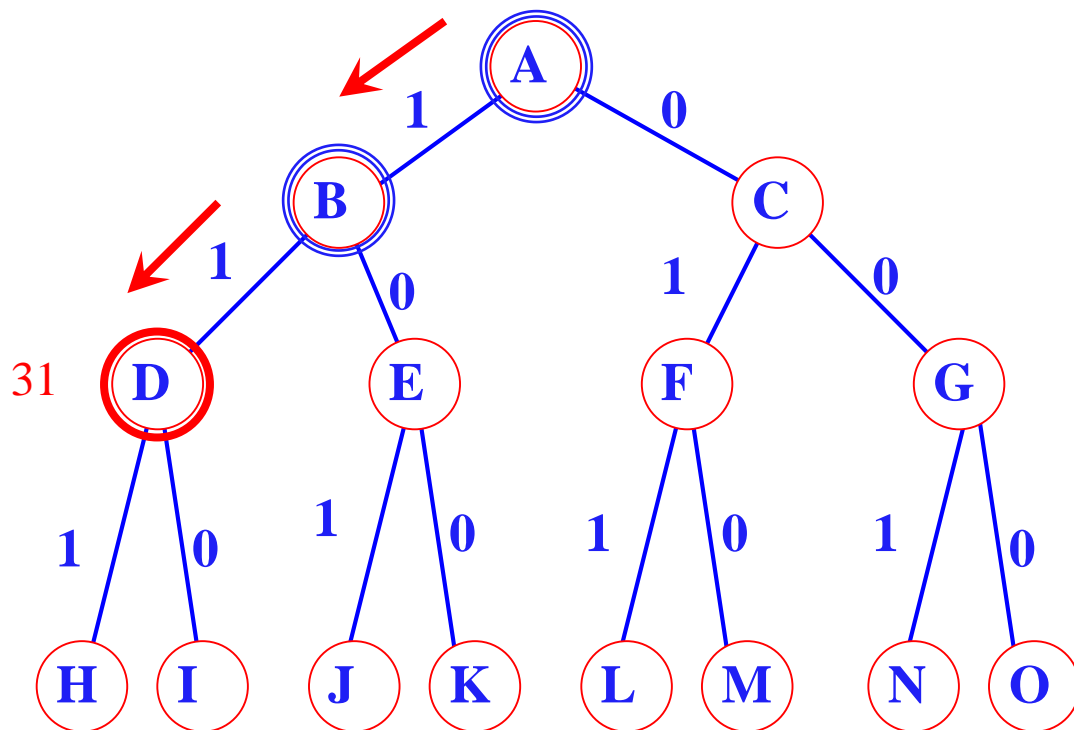
当前装载

70

节点堆栈

A

B



0-1背包问题:

问题参数:

$W = [16, 15, 15],$

$P = [45, 25, 25],$

$C = 30.$

问题的解空间:

$\{ (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) \}$

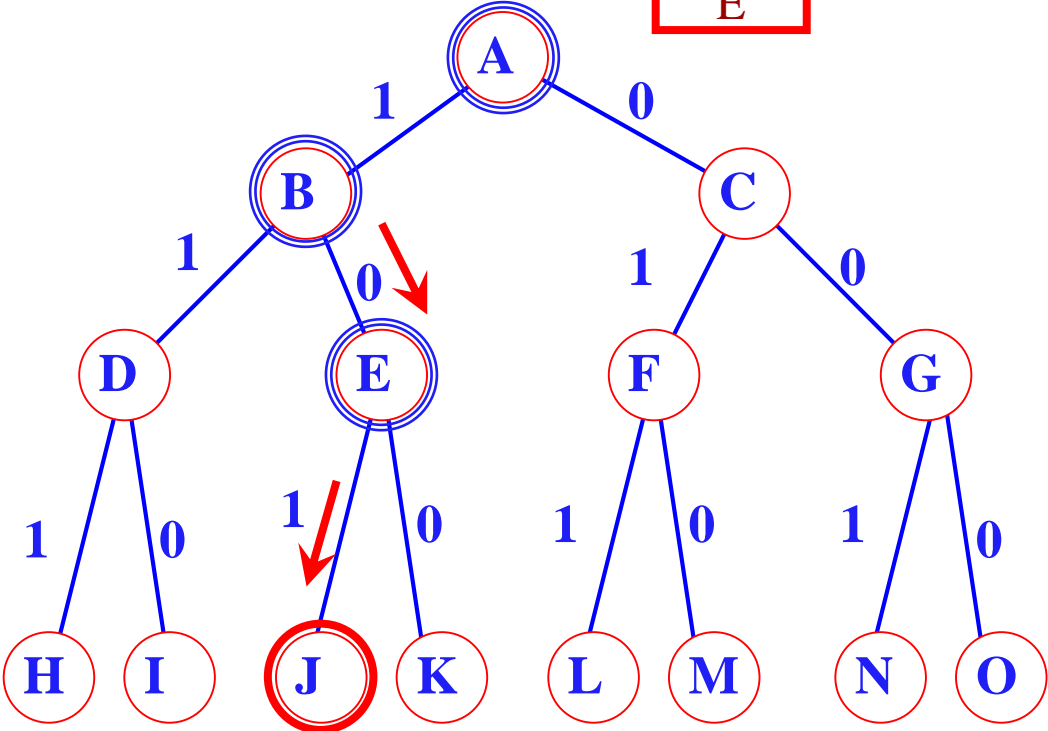
2. 算法

当前装载

70

节点堆栈

A
B
E



0-1背包问题:

问题参数:

$W = [16 , 15 , 15] ,$
 $P = [45 , 25 , 25] ,$
 $C = 30 .$

问题的解空间:

$\{ (0,0,0), (0,1,0),$
 $(0,0,1), (1,0,0),$
 $(0,1,1), (1,0,1),$
 $(1,1,0), (1,1,1) \}$

2. 算法

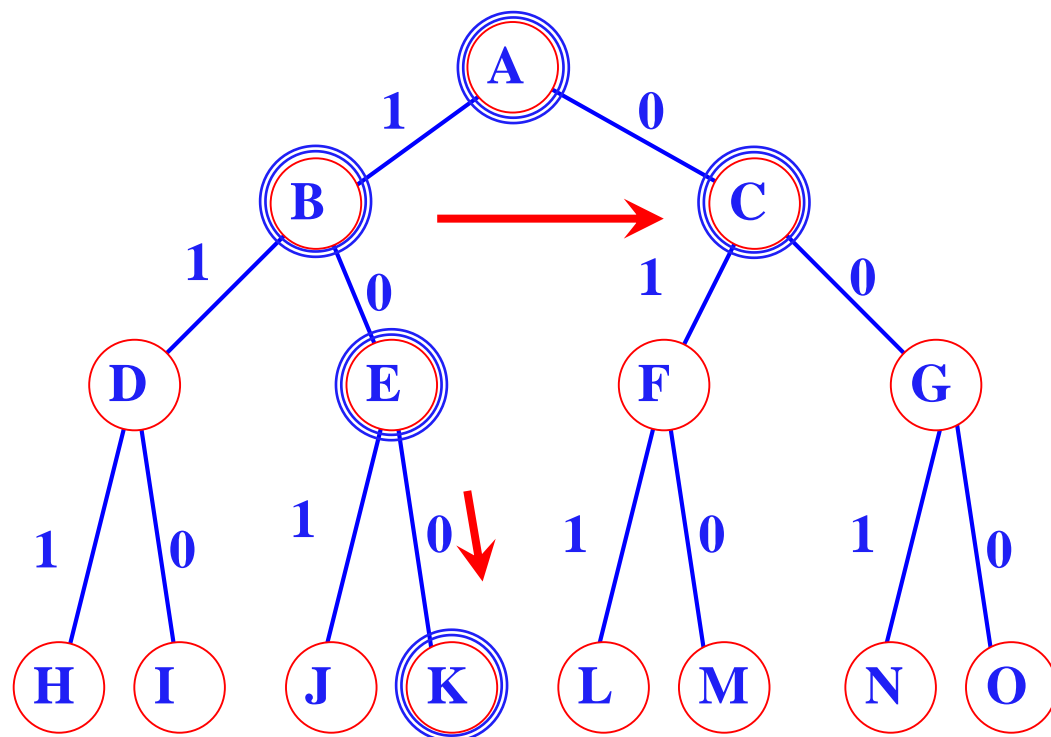
当前装载

45

节点堆栈

A

B



45

0-1背包问题:

问题参数:

$W = [16, 15, 15]$,

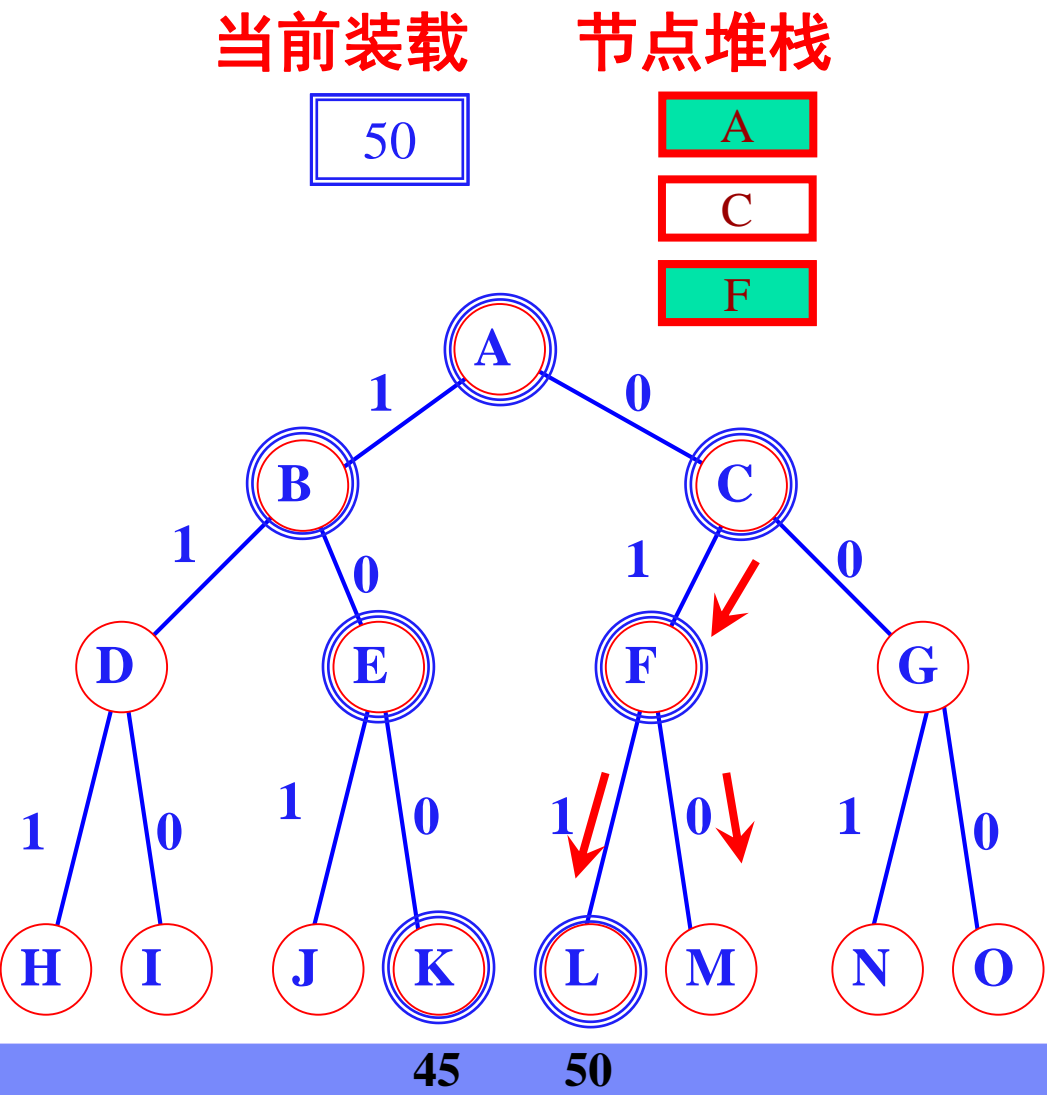
$P = [45, 25, 25]$,

$C = 30$ 。

问题的解空间:

$\{ (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) \}$

2. 算法



0-1背包问题:

问题参数:

$W = [16 , 15 , 15] ,$
 $P = [45 , 25 , 25] ,$
 $C = 30 .$

问题的解空间:

$\{ (0,0,0), (0,1,0),$
 $(0,0,1), (1,0,0),$
 $(0,1,1), (1,0,1),$
 $(1,1,0), (1,1,1) \}$

2. 算法

当前装载

25

节点堆栈

C

0-1背包问题:

问题参数:

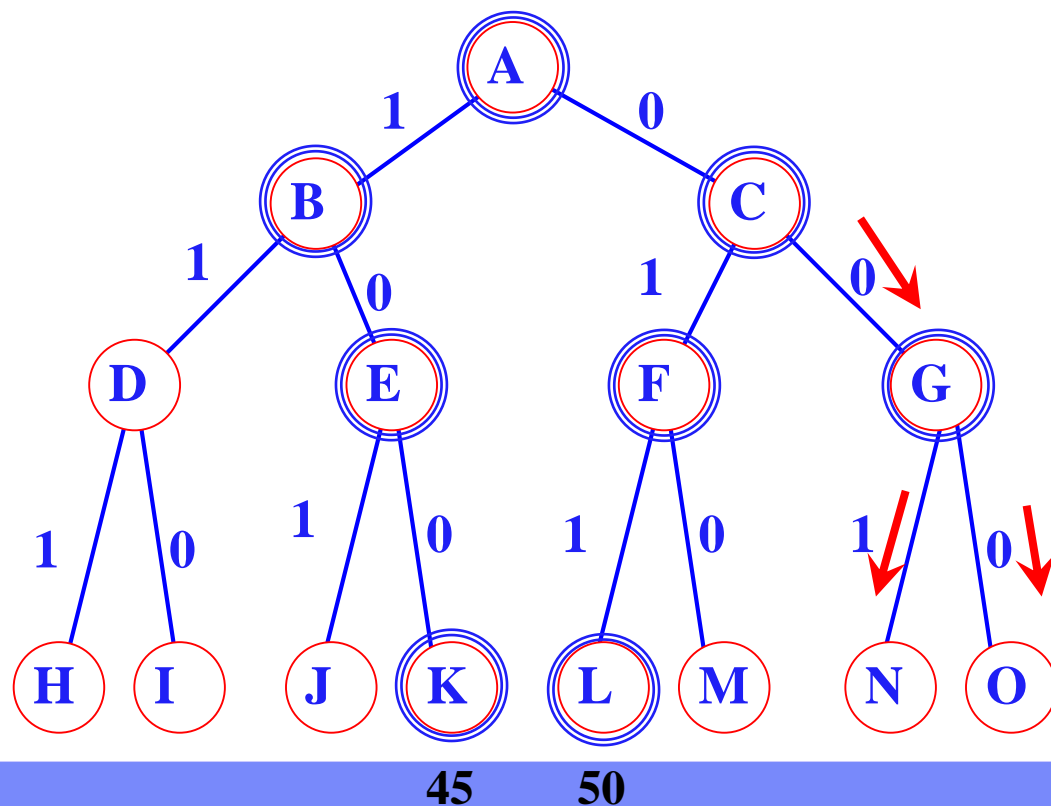
$W = [16, 15, 15]$,

$P = [45, 25, 25]$,

$C = 30$ 。

问题的解空间:

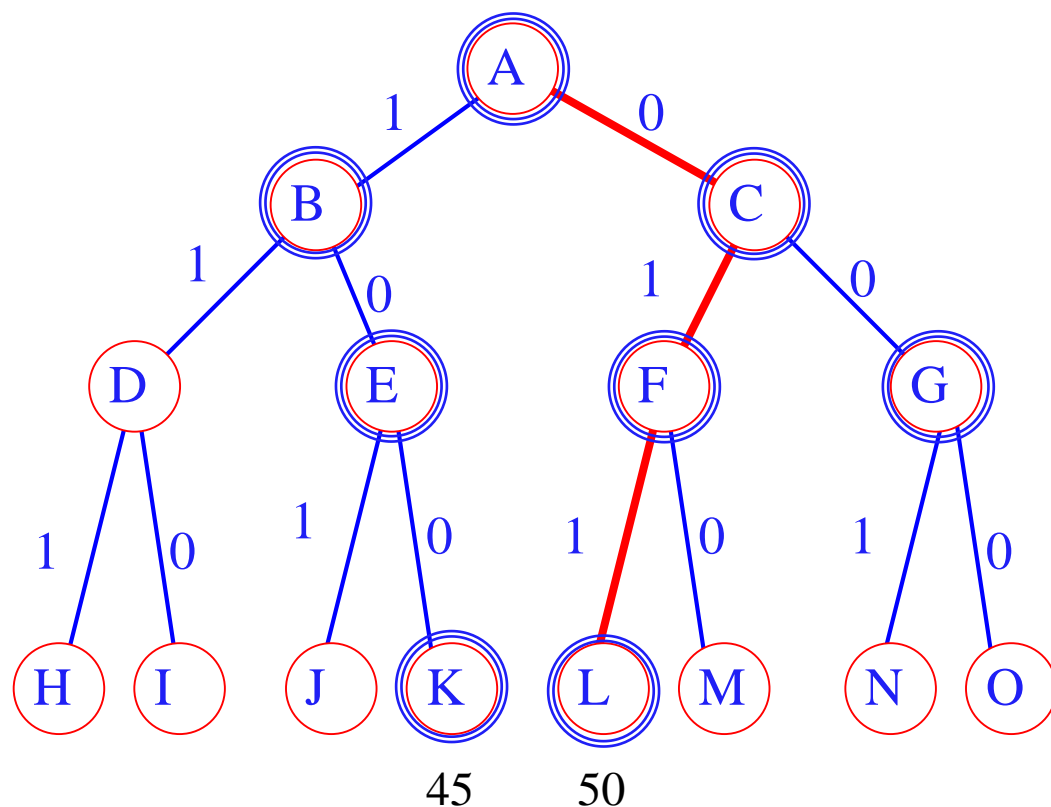
$\{ (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) \}$



2. 算法

最优解为 $(0, 1, 1)$

最大值为 50



0-1背包问题:

问题参数:

$W = [16, 15, 15]$,

$P = [45, 25, 25]$,








$C = 30$ 。

问题的解空间:

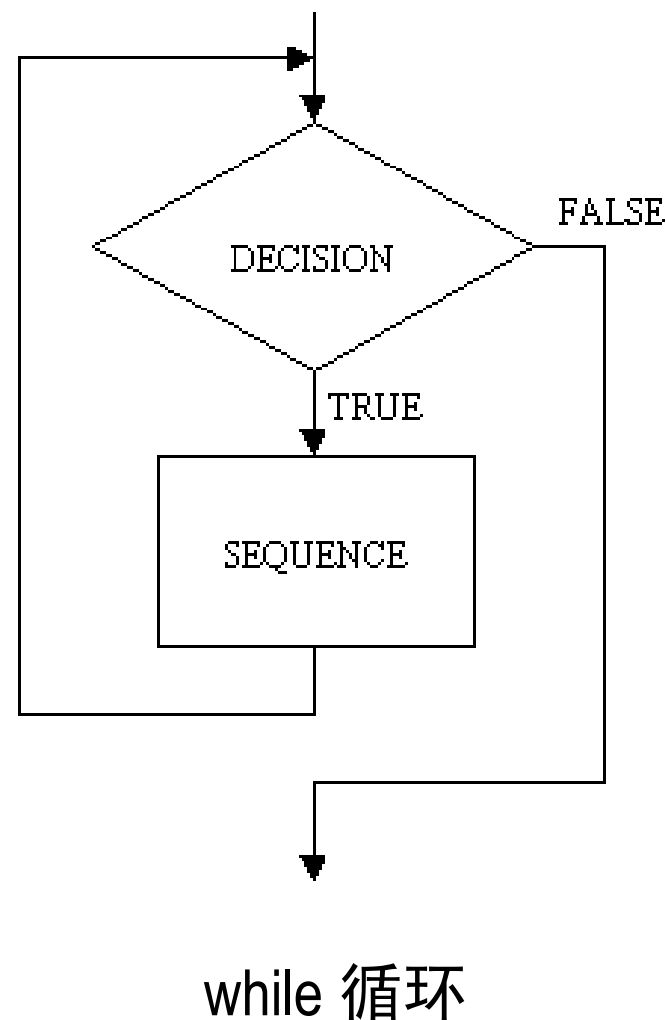
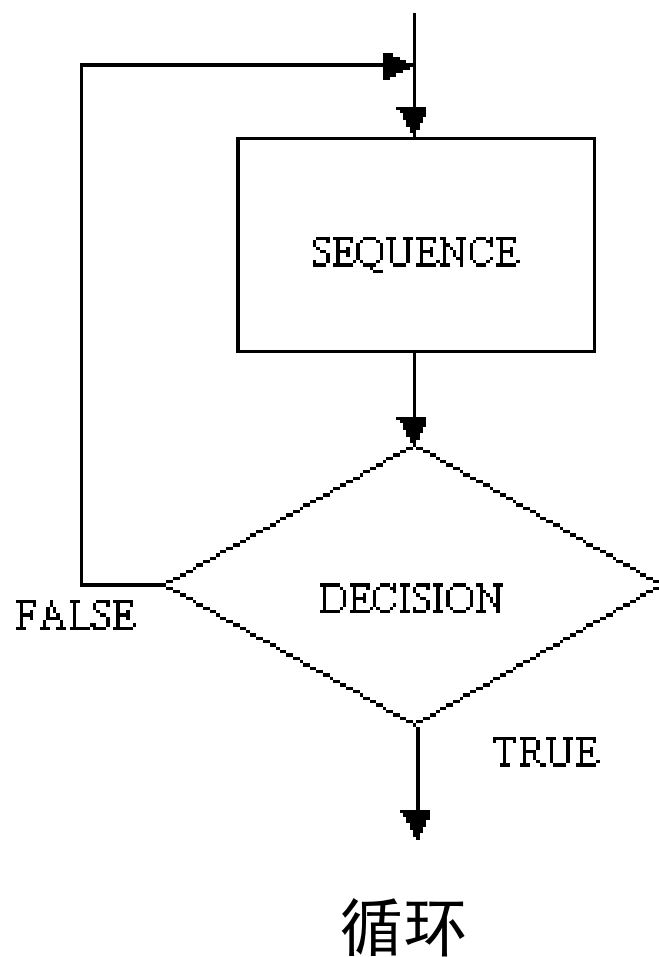
$\{ (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) \}$

2. 算法

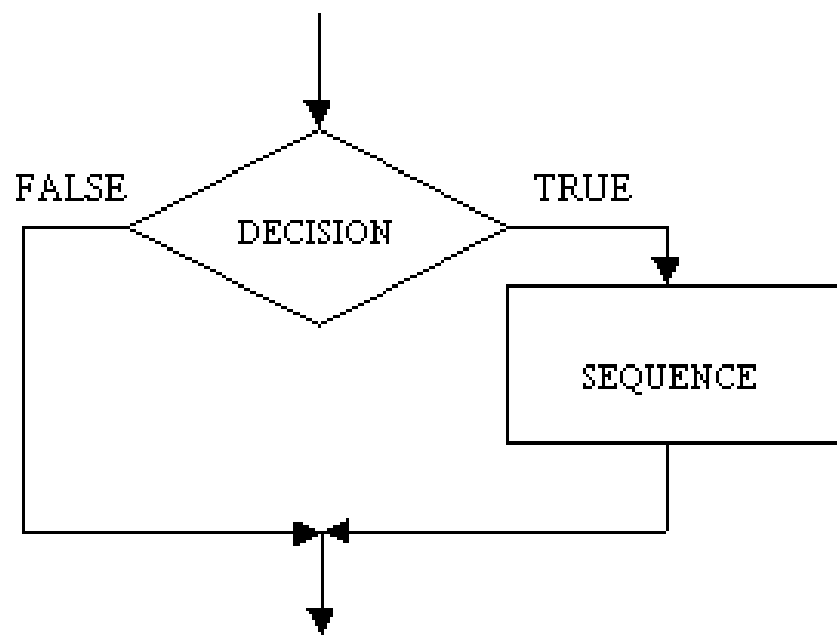
算法描述：
程序流程图

图形符号	名称	含义
	起止框	表示算法的开始或结束
	处理框	表示处理或运算等功能
	输入/输出框	表示进行输入/输出操作
	判断框	判读是否满足给定条件， 决定执行两条路径中的某一条路径
	控制流	程序执行路径，箭头代表方向
	页内连接框	连接同一页内流程图的不同部分
	页间连接框	连接不同页流程图的不同部分

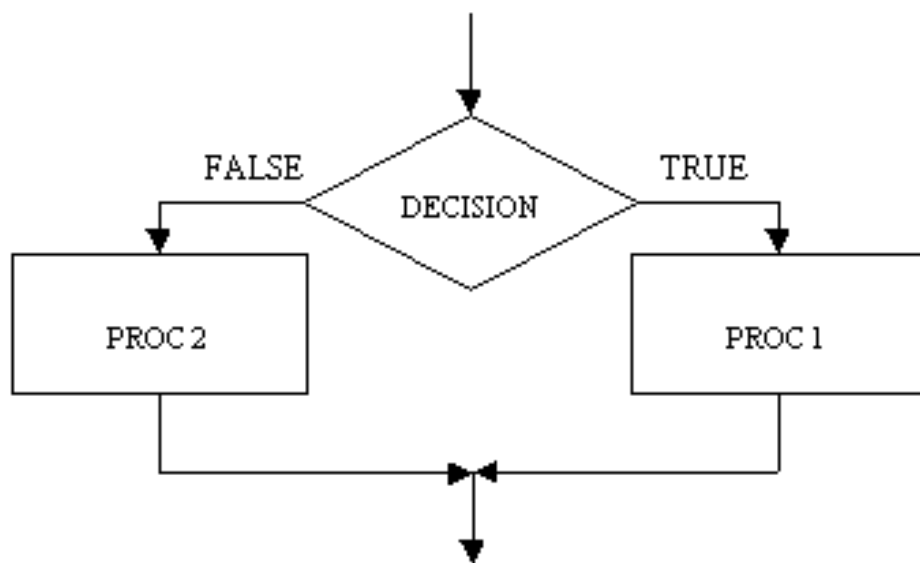
2. 算法



2. 算法

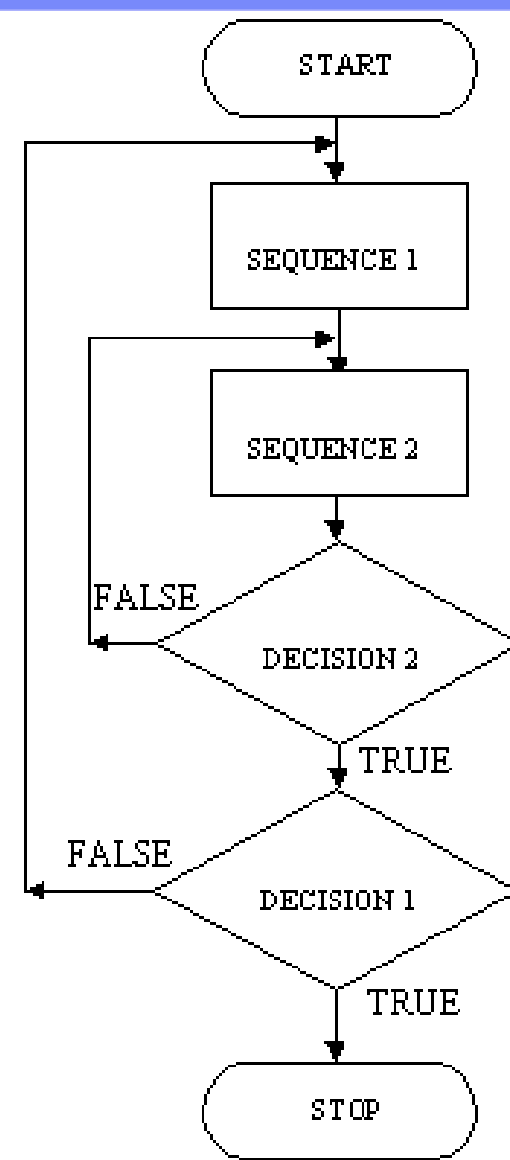
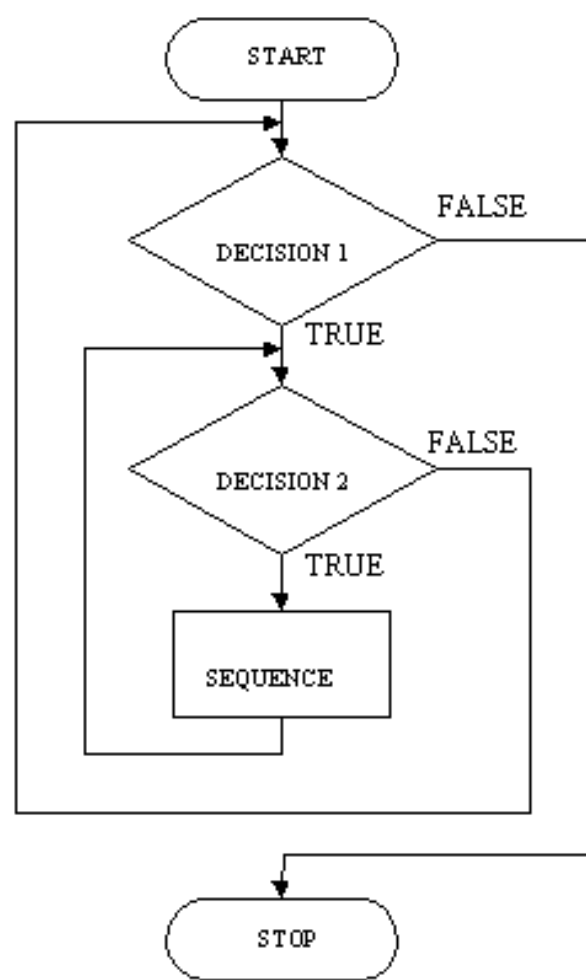


if ... then



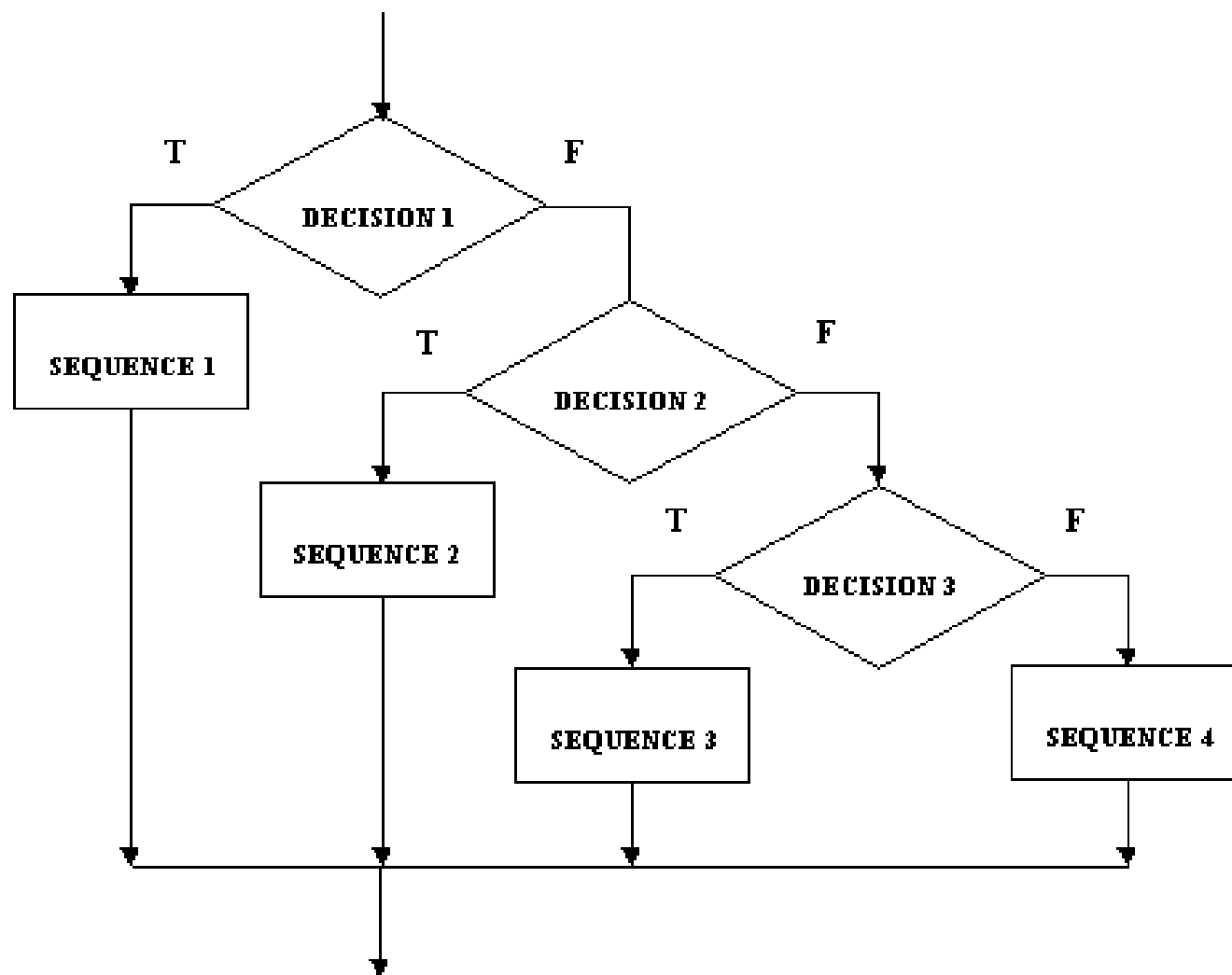
if ... then ... else

2. 算法



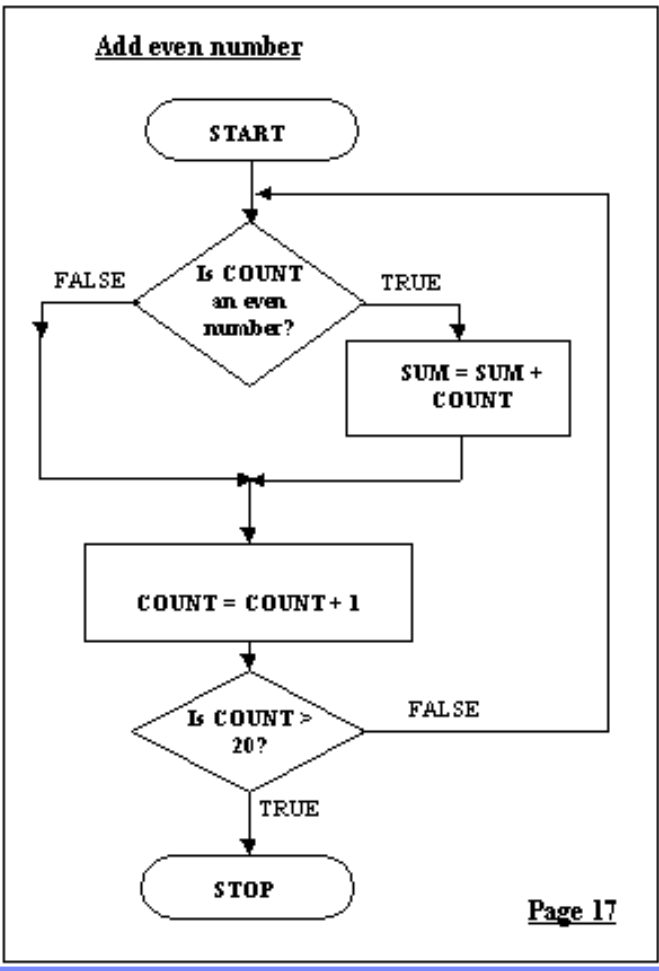
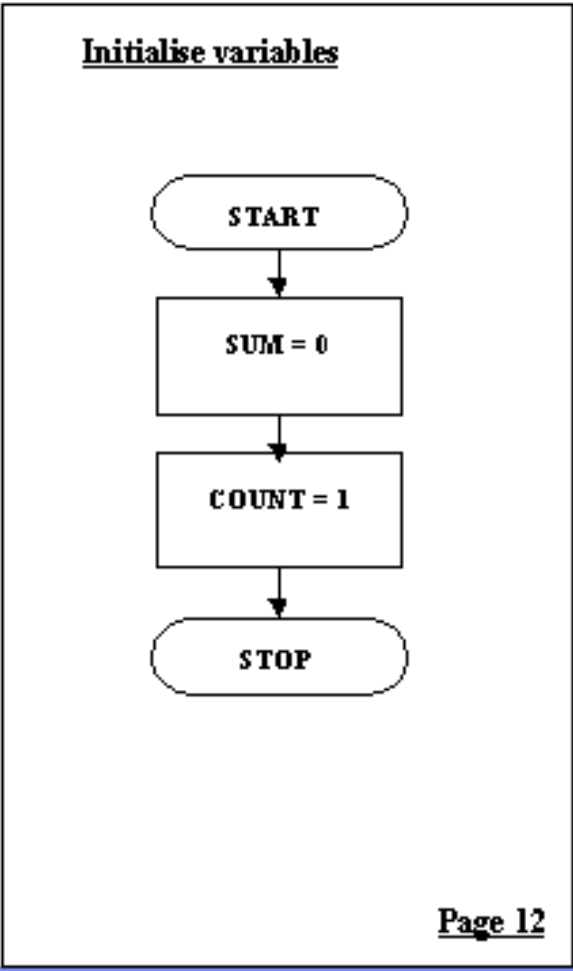
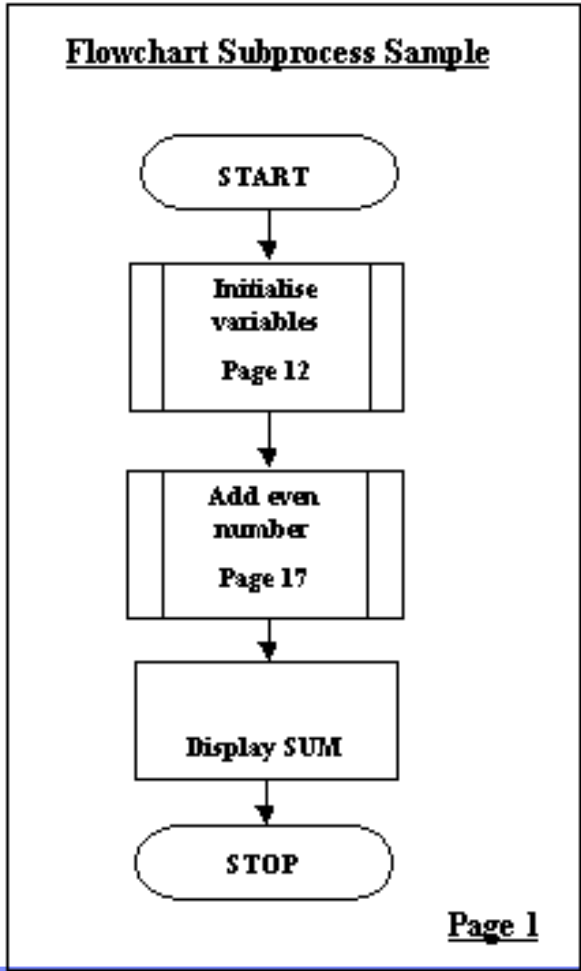
循环嵌套

2. 算法



多路选择

2. 算法



小结

