程序设计艺术与方法学

第四讲　动态规划

合肥工业大学计算机与信息学院　2015-09

爬楼梯

- 一个楼梯有20级，每次走1级或2级，从底走到顶一共有多少种走法？

分析（例1）

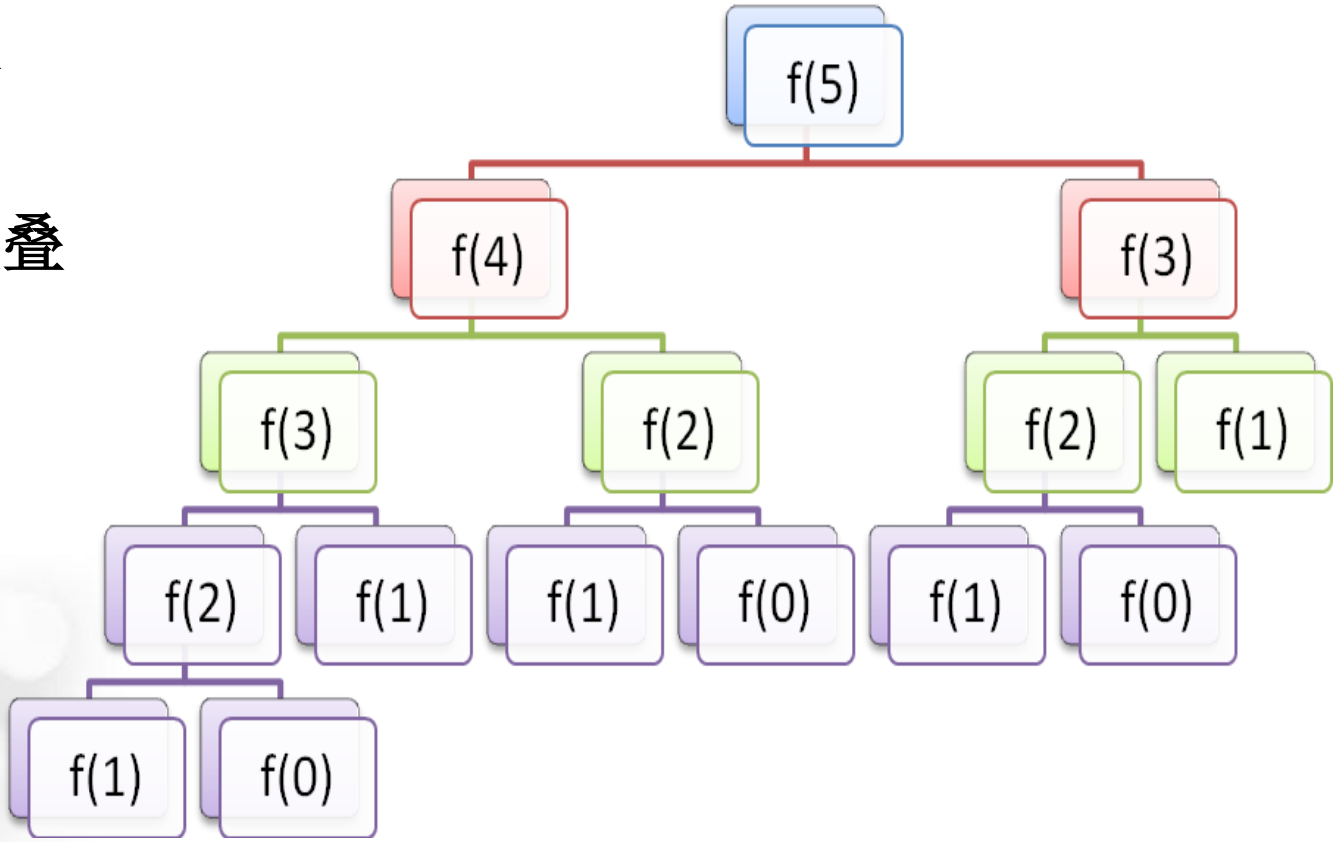
- 假设从底走到第n级的走法有f(n)种，走到第n级有两个方法
- 一个是从第n-1级走1步，另一个是从第n-2级走2步，前者有f(n—1)种方法，后者有f(n-2)种方法
- 所以f(n)=f(n-1)+f(n-2)
- 易知f(0)=1，f(1)=1

分析（例1）

```
int f(int n) {
        if(n==0 || n==1)
return 1;
        return f(n-1) + f(n-2);
}
void main() {
        cout<<f(20)<<endl;
}
```

分析

- 可递归求解
- 子问题有重叠



解决办法（例2）

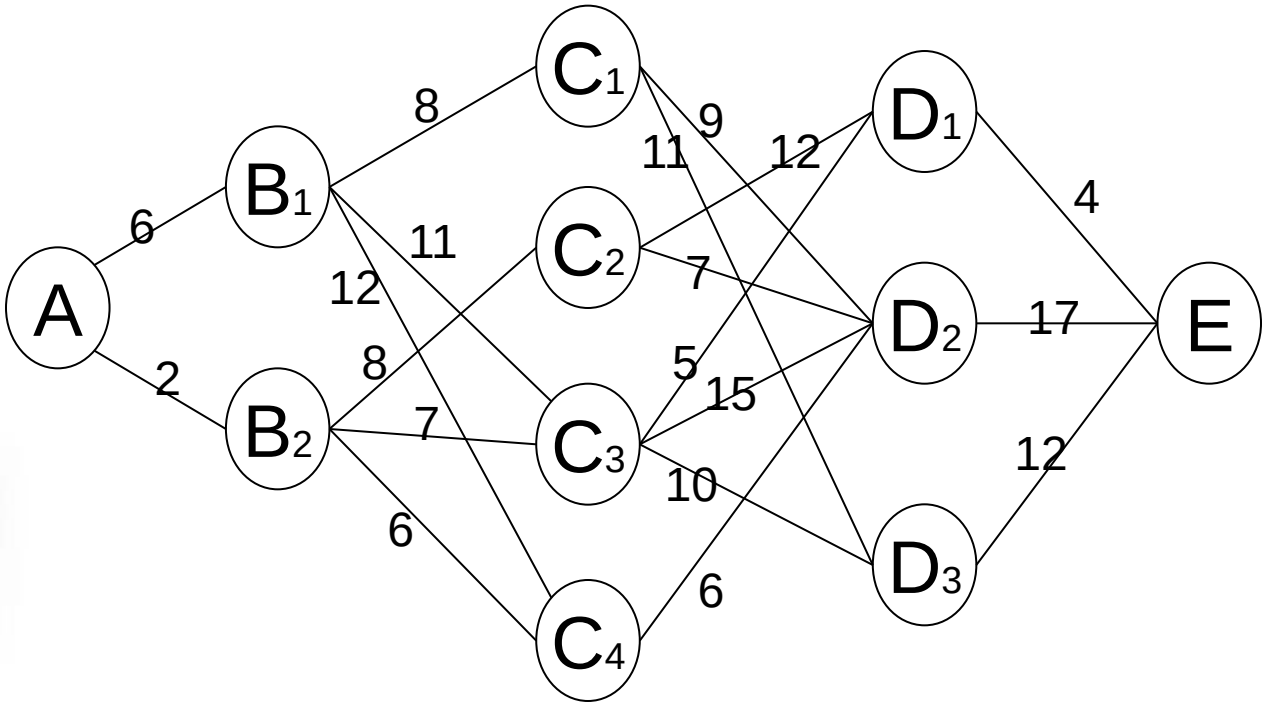
每个子问题及其解记录下来，只求解一次。

```
int dp[100];        //用于保存f(n)的结果
int f(int n) {
        if(dp[n]) return dp[n];//如果问题已经被求解，则直接返回结果
        int result;
        if(n==0 || n==1) result=1;
        else result = f(n-1) + f(n-2);
        return dp[n] = result;      //记录问题的解，并返回
}

void main() {
        memset(dp, 0, sizeof(dp));          //初始化结果集
        cout<<dp(20)<<endl;
}
```

再来一题

如图，各结点代表城市，两结点间连线上数字表示城市间的距离。试找出从结点A到结点E的最短距离



分析

深度优先搜索法来解决此问题，该问题的递归式为

$$MinDist(v) = \min_{u \in \delta(v)} \{w(v,u) + MinDist(u)\}$$

其中δ(v)是与v相邻的节点的集合，w(v,u)表示从v到u的边的长度

算 法

```
int MinDistance(v)
{
        if ( v == E )
            return 0;
        else {
                min = maxint;
                for ( 所有没有访问过的节点i ) {
                    if ( v和i相邻 ) {
                                标记i访问过了;
                                t = v到i的距离+MinDistance(i);
                                标记i未访问过;
                                if ( t<min )  min=t;
                }
                        return min;
        }
}
```

MinDistance(A)就是从A到E的最短距离

算法分析

- 除了已经访问过的城市外，其他城市都要访问，所以时间复杂度为O(n!)
- 观察一下这个算法:
  - 求B1到E的最短距离时，先求C3到E的最短距离
  - 求B2到E的最短距离时，C3到E又求了一次
  - 同样可知，D1到E的最短距离被求了四遍
- 如果求解过程中，将求得的最短距离“记录在案”，随时调用，就可以避免这种情况。

算法改进

- 进一步，改递归为递推，减少递归调用开销
- 如图可知，A只和Bi相邻，Bi只和Ci相邻,...，原问题的解决过程划分为4个阶段，设S1={A}, S2={B1,B2},S3={C1,C2,C3,C4},S4={D1,D2,D3}，Fk(u)表示从Sk中的点u到E的最短距离
- F1(A)就是从A到E的最短距离，T(n)=O(n)

$$F_k(u) = \min_{u \in S_k, v \in S_{k+1} \cap \delta(u)} \{w(u,v) + F_{k+1}(v)\}$$

$$F_5(E) = 0$$

算法改进

```
void DynamicProgramming()
{
    F5[E]=0;
    for ( i=4; i>=1; --I )
            for ( each u ∈Sk ) {
            Fk[u]:=无穷大;
            for ( each v∈Sk+1∩δ(u) )
                  if ( Fk[u]>w(u,v)+Fk+1[v] )
                        Fk[u]:=w(u,v)+Fk+1[v];
        }
    输出F1[A];
}
```

4.1 动态规划的概念和基本要素

4.2 动态规划的基本思想和适用条件

4.3 动态规划实例分析

4.1 动态规划的概念和基本要素

- 动态规划(dynamic programming)是运筹学的一个分支，是求解决策过程(decision process)最优化的数学方法
- 20世纪50年代初美国数学家R.E.Bellman等人在研究多阶段决策过程(multistep decision process)的优化问题时，提出了著名的最优化原理(principle of optimality)，把多阶段过程转化为一系列单阶段问题，逐个求解，创立了解决这类过程优化问题的新方法——动态规划
- 1957年出版了他的名著*Dynamic Programming*，这是该领域的第一本著作

多阶段决策过程

- **多阶段决策过程:** 是指这样的一类特殊的活动过程，问题可以按时间顺序分解成若干相互联系的阶段，在每一个阶段都要做出决策，全部过程的决策是一个决策序列。

- **多阶段决策问题:**使整个活动的总体效果达到最优的问题。
  - [例] 工厂生产某种产品，每单位(千件)的成本为1(千元)，每次开工的固定成本为3(千元)，工厂每季度的最大生产能力为6(千件)。经调查，市场对该产品的需求量第一、二、三、四季度分别为 2，3，2，4(千件)。如果工厂在第一、二季度将全年的需求都生产出来，自然可以降低成本(少付固定成本费)，但是对于第三、四季度才能上市的产品需付存储费，每季每千件的存储费为0.5(千元)。还规定年初和年末这种产品均无库存。试制订一个生产计划，即安排每个季度的产量，使一年的总费用(生产成本和存储费)最少

决策过程的分类

- 根据过程的时间变量是离散的还是连续的，分为
  - 离散时间决策过程(discrete-time decision process)，即多阶段决策过程
  - 连续时间决策过程(continuous-time decision process)
- 根据过程的演变是确定的还是随机的，分为
  - 确定性决策过程(deterministic decision process)
  - 随机性决策过程(stochastic decision process)
- 其中应用最广的是确定性多阶段决策过程

动态规划模型的基本要素

1.阶段： 对整个过程的自然划分

2.状态： 每个阶段开始时过程所处的自然状况

3.决策： 当一个阶段的状态确定后，可以作出各种选择从而演变到下一阶段的某个状态，这种选择手段称为决策(decision)

4.策略： 决策组成的序列称为策略(policy)

5.状态转移方程

6.指标函数和最优值函数

7.最优策略和最优轨线

动态规划模型的基本要素

- **1.阶段：对整个过程的自然划分**
  - 通常根据时间顺序或空间特征来划分阶段
  - 阶段变量一般用k=1,2,..,n表示
- **2.状态：每个阶段开始时过程所处的自然状况**
  - 能够描述过程的特征并且具有无后向性
  - 描述状态的变量称状态变量(state variable)
  - 变量允许取值的范围称允许状态集合(set of admissible states)
  - 用x_k表示第k阶段的状态变量，它可以是一个数或一个向量。用X_k表示第k阶段的允许状态集合

动态规划模型的基本要素

- **3.决策：当一个阶段的状态确定后，可以作出各种选择从而演变到下一阶段的某个状态，这种选择手段称为决策(decision)**
  - 描述决策的变量称决策变量(decision variable)
  - 变量允许取值的范围称允许决策集合(set of admissible decisions)
  - 用$u_k(x_k)$表示第k阶段处于状态x_k时的决策变量，它是x_k的函数，用$U_k(x_k)$表示了x_k的允许决策集合
- **4.策略：决策组成的序列称为策略(policy)**
  - 初始状态x_1开始的全过程的策略记作$p_{1n}(x_1)$，即$p_{1n}(x_1)=\{u_1(x_1), u_2(x_2),...,u_n(x_n)\}$，类似
  - $p_{kn}(x_k)=\{u_k(x_k),...,u_n(x_n)\}$，$p_{kj}(x_k)=\{u_k(x_k),...,u_j(x_j)\}$
  - 对于每一个阶段k的某一给定的状态x_k，可供选择的策略$p_{kj}(x_k)$有一定的范围，称为允许策略集合(set of admissible policies)，用$P_{1n}(x_1),P_{kn}(x_k),P_{kj}(x_k)$表示

动态规划模型的基本要素

- **5.状态转移方程:**
  - 在确定性过程中，一旦某阶段的状态和决策为已知，下阶段的状态便完全确定。用状态转移方程(equation of state)表示这种演变规律。

$$x_{k+1} = T_k(x_k, u_k(x_k)) \quad , k = 1, 2, \cdots, n \qquad (1)$$

动态规划模型的基本要素

6.指标函数和最优值函数

- 指标函数(objective function)是衡量过程优劣的数量指标，它是关于策略的数量函数
- 从阶段k到阶段n的指标函数用$V_{kn}(x_k, p_{kn}(x_k))$表示，$k=1, 2, ..., n$
- 能够用动态规划解决的问题的指标函数应具有可分离性，即V_{kn}可表为$x_k, u_k, V_{k+1\,n}$的函数，记为：

$$V_{kn}(x_k, u_k, x_{k+1}, \cdots, x_{n+1}) = \varphi_k(x_k, u_k, V_{k+1\,n}(x_{k+1}, \cdots, x_{n+1})) \qquad (2)$$

- 其中函数φ_k是一个关于变量$V_{k+1\,n}$单调递增的函数。这一性质是动态规划的适用前提

动态规划模型的基本要素

- 过程在第j阶段的**阶段指标**取决于状态x_j和决策u_j，用$v_j(x_j, u_j)$表示
- 阶段k到阶段n的指标由$v_j(j = k, k+1, .. n)$组成，常见的形式有：
  - 阶段指标之和，即

$$V_{kn}(x_k, u_k, \cdots, x_{n+1}) = \sum_{j=k}^{n} v_j(x_j, u_j) \qquad (3)$$

  - 阶段指标之积，即

$$V_{kn}(x_k, u_k, \cdots, x_{n+1}) = \prod_{j=k}^{n} v_j(x_j, u_j) \qquad (4)$$

  - 阶段指标之极大(或极小)，即

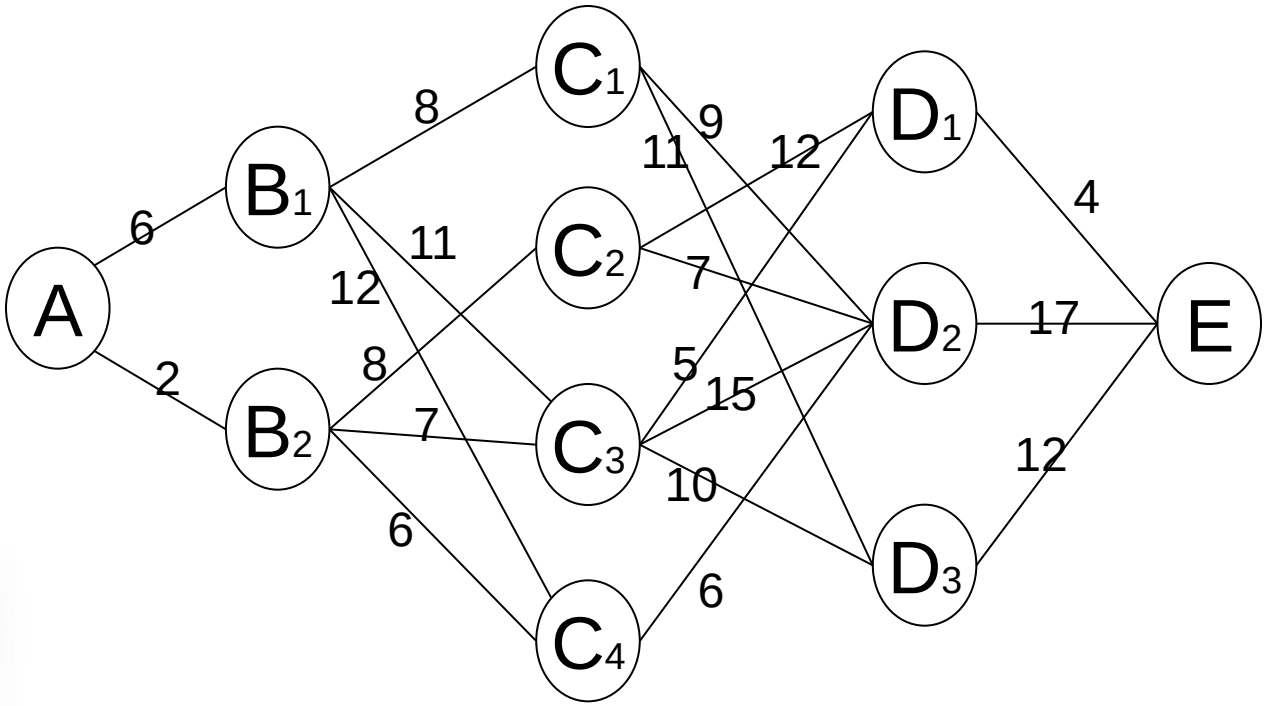
$$V_{kn}(x_k, u_k, \cdots, x_{n+1}) = \max_{k \le j \le n} v_j(x_j, u_j) \quad or \quad \min_{k \le j \le n} v_j(x_j, u_j) \qquad (5)$$

- (3)-(5)三个指标函数的形式都满足最优性原理

动态规划模型的基本要素

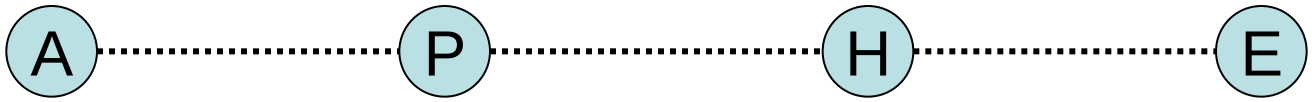
- **7.最优策略和最优轨线**
  - 使指标函数V_{kn}达到最优值的策略是从k开始的后部子过程的最优策略，记作$p_{kn}^*=\{u_k^*, \ldots, u_n^*\}$
  - p_{1n}^*又是全过程的最优策略，简称最优策略(optimal policy)
  - 从初始状态$x_1(=x_1^*)$出发，过程按照p_{1n}^*和状态转移方程演变所经历的状态序列$\{x_1^*, x_2^*, .., x_{n+1}^*\}$称最优轨线(optimal trajectory)

如图，各结点代表城市，两结点间连线上数字表示城市间的距离。试找出从结点A到结点E的最短距离



动态规划的基本方程

生活常识告诉我们，最短路径有一个重要性质：如果有起点A经过P点和H点而到达终点E是一条最短路线，则由点P经过H到达终点E的这条子路线也为最优



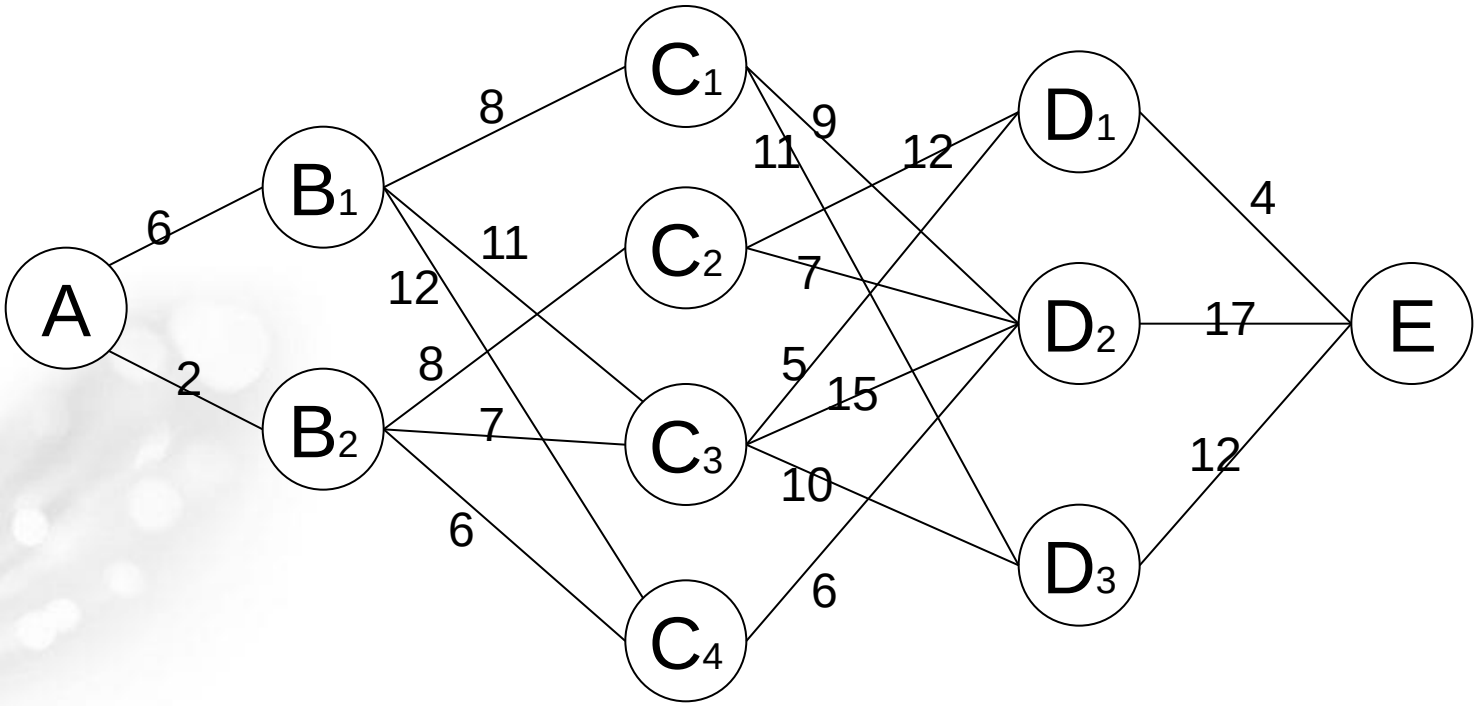
根据这一特性，寻找最短路径的方法，就是从最后一段开始，用由后向前逐渐递推的方法求出个点到E点的最短路线：**动态规划的方法就是从终点逐渐向始点方向寻找最短路线的一种方法**

动态规划的基本方程

k=4时，有$f_4(D_1)=4$, $f_4(D_2)=17$, $f_4(D_3)=12$

k=3时，$f_3(C_1) = \min\{d_3(C_1, D_2)+ f_4(D_2), d_3(C_1, D_3)+ f_4(D_3)\} = 23$

同理可得其它个点，如下图



动态规划的基本方程

- 根据上面的例子可以得到动态规划的基本方程：

$$\begin{cases} f_k(x_k) = \underset{u_k \in U_k(x_k)}{opt} \{\varphi(v_k(x_k,u_k), f_{k+1}(x_{k+1}))\}, \ x_{k+1} = T_k(x_k,u_k), \ k = 1,2,\cdots,n \\ f_{n+1}(x_{n+1}) = \delta(x_{n+1}) \end{cases} \tag{9}$$

  - 其中 $f_{n+1}(x_{n+1}) = \delta(x_{n+1})$ 是决策过程的终端条件，δ为一个已知函数，当x_{n+1}只取固定的状态时称固定终端当x_{n+1}可在终端集合X_{n+1}中变动时称自由终端

- 最优指标函数满足

$$opt\{V_{1n}\} = \underset{x_1 \in X_1}{opt}\{f(x_1)\} \tag{10}$$

4.1 动态规划的概念和基本要素

4.2 动态规划的基本思想和适用条件

4.3 动态规划实例分析

4.2 动态规划的基本思想和适用条件

- **基本定理**
  - 对于初始状态$x_1 \in X_1$，策略p_{1n}*={u_1*,..u_n*}是最优策略的充要条件是对于任意的k,1<k<=n,有

$$V_{1n}(x_1, p_{1n}^*) = \varphi\left(\underset{p_{1k-1} \in P_{1k-1}(x_1)}{opt} [V_{1k-1}(x_1, p_{1k-1})], \underset{p_{kn} \in P_{kn}(x_k)}{opt} [V_{kn}(x_k, p_{kn})] \right) \quad (8)$$

- **推论 (最优化原理)**
  - 若p_{1n}* $\in P_{1n}(x_1)$是最优策略，则对于任意的k, 1<k<n,它的子策略p_{kn}*对于由x_1和$p_{1,k-1}$*确定的以x_k*为起点的第k到n后部子过程而言，也是最优策略

4.2 动态规划的基本思想和适用条件

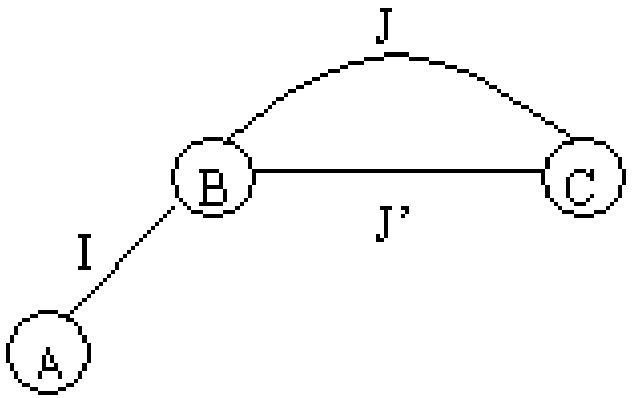
1. 动态规划的关键在于正确写出基本的递推关系式和恰当的边界条件（即基本方程）。
2. 多阶段决策过程中，动态规划方法是既把当前一段和未来各段分开，又把当前效益和未来效益结合考虑的一种最优化方法。故每段决策的选取是从全局来考虑的，与该段的最优答案一般是不同的。
3. 求整个问题的最优决策时，由于初始状态已知，每段的决策都是该段状态的函数，故最优策略所经过的各段状态便可逐次变换得到，从而确定了最优路线。

4.2 动态规划的基本思想和适用条件

- **1最优化原理（最优子结构性质）**
  - 一个最优化策略的子策略总是最优的
  - 一个问题满足最优化原理称其有最优子结构性质

例：如图若路线I和J是A到C的最优路径，则根据最优化原理，路线J必是从B到C的最优路线。

-这可用反证法证明：假设有另一路径J'是B到C的最优路径，则A到C的路线取I和J'比I和J更优，矛盾



2 无后向性

- 将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性

- 如果用前面的记号来描述无后向性，就是：对于确定的x_k，无论$p_{1,k-1}$如何，最优子策略p_{kn}*是唯一确定的，这种性质称为无后向性

3子问题的重叠性

- 动态规划将原来具有指数级复杂度的搜索算法改进成了具有多项式时间的算法。其中的关键在于**解决冗余**，这是动态规划算法的根本目的
- 动态规划实质上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其它的算法
- 设原问题的规模为n，当**子问题树**中的子问题总数是n的超多项式函数，而**不同**的子问题数只是n的多项式函数时，动态规划法显得特别有意义，此时动态规划法具有线性时间复杂性
- 能够用动态规划解决的问题还有一个显著特征：**子问题的重叠性**。这个性质并不是动态规划适用的必要条件，但是如果该性质无法满足，动态规划算法同其他算法相比就不具备优势。

4.2 动态规划的基本思想和适用条件

- 一般来说，只要一个问题可以划分成规模更小的子问题，并且原问题的最优解中包含了子问题的最优解（即满足最优子化原理），则可以考虑用动态规划解决

- **动态规划**的实质是**分治思想**和**解决冗余**，是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略

动态规划与分治法和贪心法的关系

- 都是将问题实例归纳为更小的、相似的子问题，并通过求解子问题产生一个全局最优解。
- 贪心法的当前选择可能要依赖已经作出的所有选择，但不依赖于有待于做出的选择和子问题。因此贪心算法是自顶向下，一步一步作出选择。
- 分治法中的各个子问题是独立的 (即不包含公共的子问题)，因此一旦递归地求出各子问题的解后，便可自下而上地将子问题的解合并成问题的解。
- 动态规划允许子问题不独立（即可包含公共子子问题），也允许通过自身子问题的解作出选择。

4.2 动态规划的基本思想和适用条件

- 贪心算法的例子——最小生成树
- 分治算法的例子——巨人和魔鬼

4.1 动态规划的概念和基本要素

4.2 动态规划的基本思想和适用条件

4.3 动态规划实例分析

动态规划算法的基本步骤

1. 划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段
2. 选择状态：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来
3. 确定决策并写出状态转移方程：决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。如果确定了决策，状态转移方程也就写出来了。但事实上，常常是反过来根据相邻两段的各状态之间的关系来确定决策
4. 写出规划方程（包括边界条件）：动态规划的基本方程是规划方程的通用形式化表达式

标准动态规划的基本框架

```
1.  对fn+1(xn+1)初始化;   {边界条件}
2.  for k:=n downto 1 do
3.      for 每一个xk∈Xk do
4.        for 每一个uk∈Uk(xk) do
           begin
5.           fk(xk):=一个极值;                {∞或－∞}
6.           xk+1:=Tk(xk,uk);                 {状态转移方程}
7.           t:=φ(fk+1(xk+1),vk(xk,uk));      {基本方程(9)式}
8.           if  t比fk(xk)更优 then fk(xk):=t; {计算fk(xk)的最优值}
          end;
9.  t:=一个极值;                              {∞或－∞}
10. for 每一个x1∈X1 do
11.     if f1(x1)比t更优 then t:=f1(x1);      {按照10式求出最优指标}
12. 输出t;
```

标准动态规划的基本框架

- 实际上按以下几个步骤进行：
  1. 分析最优解的性质，并刻划其结构特征；
  2. 递归地定义最优值；
  3. 以自底向上的方式或自顶向下的记忆化方法（备忘录法）计算出最优值；
  4. 据计算最优值时得到的信息，构造一个最优解；
- 步骤(1)--(3)是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤(4)可以省略，若需要求出问题的一个最优解，则必须执行步骤(4)

4.3 动态规划的实例分析

生产计划问题

- 工厂生产某种产品，每单位(千件)的成本为1(千元)，每次开工的**固定成本**为3(千元)，工厂每季度的**最大生产能力**为6(千件)。
- 市场对该产品的**需求量**第一、二、三、四季度分别为 2，3，2，4(千件)。如果工厂在第一、二季度将全年的需求都生产出来，自然可以降低成本(少付固定成本费)，但是对于第三、四季度才能上市的产品需付存储费，每季每千件的**存储费**为0.5(千元)。
- 还规定年初和年末这种产品均无库存。试制订一个生产计划，即安排每个季度的产量，使一年的总费用(生产成本和存储费)最少。

4.3 动态规划的实例分析

- 按照计划时间自然划分**阶段**
- **状态**定义为每阶段开始时的存储量x_k
- **决策**为每个阶段的产量u_k，记每个阶段的需求量（已知）为d_k(2, 3, 2, 4)
- **状态转移方程**为：

$$x_{k+1} = x_k + u_k - d_k, x_k \geq 0, k = 1,2,3,4$$

- **阶段指标**:生产成本费用和存储费用之和

$$v_k(x_k, u_k) = 0.5x_k + \begin{cases} 3+u_k, u_k > 0 \\ 0 \quad\quad , u_k = 0 \end{cases}$$

4.3 动态规划的实例分析

- **指标函数**V_{kn}为v_k之和
- **最优值函数**$f_k(x_k)$为从第k阶段的状态x_k出发到过程终结的最小费用
- 动态规划的基本方程：

$$f_k(x_k)=\min_{u_k\in U_k}\{v_k(x_k,u_k)+f_{k+1}(x_{k+1})\},\quad \text{k=4, 3, 2, 1}$$

- 过程终结时允许存储量为0,终端条件为：

$$f_{4+1}(x_{4+1})=0$$

- 对应的f_4如表所示

$d_4 = 4$

x_4	0	1	2	3	4
u_4	4	3	2	1	0
f_4	7	6.5	6	5.5	2

- 对应的f_3如表所示

$d_3 = 2$

x_3	0	1	2	3	4	5	6
u_3	6	5	4	3	2	1	0
f_3	11	10.5	10	9.5	9	8.5	5

- 对应的f_2如表所示

x_2	0	1	2	3	4	5	6
u_2	3	2	1	0	0	0	0
f_2	17	16.5	16	12.5	12.5	12.5	12.5

4.3 动态规划的实例分析

- 对于f_1，由于x_1=0，所以u_1的取值范围是2~6，由公式

$$f_k(x_k)=\min_{u_k\in U_k}\{v_k(x_k,u_k)+f_{k+1}(x_{k+1})\},\quad k=4,3,2,1$$

- 可得f_1=20.5，对应的决策为5，0，6，0。

4.3 动态规划的实例分析

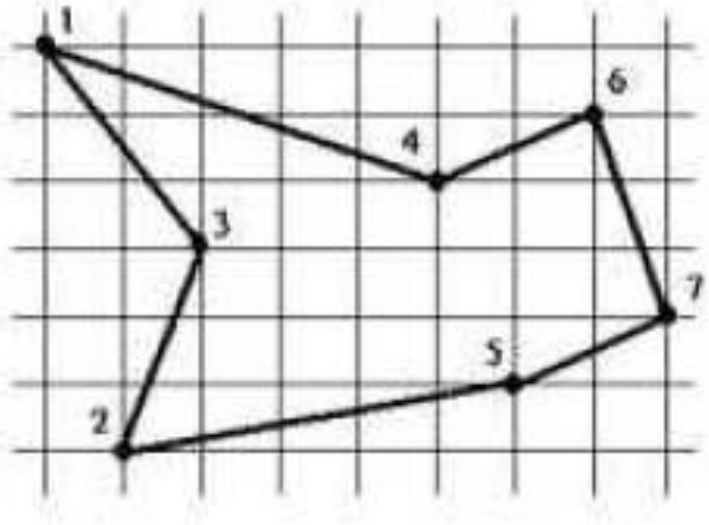
采草药（例3）

- 辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。
- 医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。
- 如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。” 如果你是辰辰，你能完成这个任务吗？

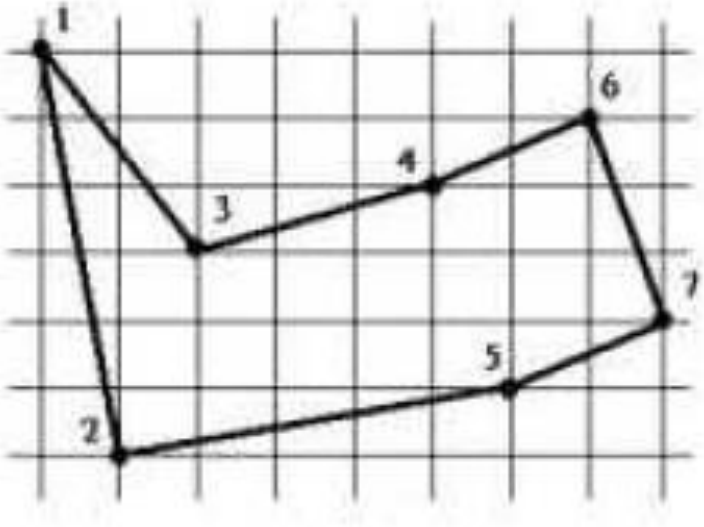
Bitonic旅行路线

- 欧几里德货郎担问题是对平面给定的n个点确定一条连结各点的、闭合的游历路线问题。
- 图(a)给出了七个点问题的解。Bitonic旅行路线问题是欧几里德货郎担问题的简化，这种旅行路线先从最左边开始，严格地由左至右到最右边的点，然后再严格地由右至左到出发点，求路程最短的路径长度。图（b）给出了七个点问题的解。
- 请设计一种多项式时间的算法，解决Bitonic旅行路线问题。

- Bitonic旅行路线



(a)



(b)

4.3 动态规划的实例分析

Bitonic旅行路线（例4）

- 第一步：将n个点按X坐标递增的顺序排列成一个序列L=<点1，点2，…，点n>。显然L[n]为最右点，L[1]为最左点。由于点1往返经过二次(出发一次，返回一次)，因此点1拆成两个点：L[0]=L[1]=点1。
- 设：
  1. $W_{i,j}$——边<i,j>的距离；
  2. $W_{i..j}$——j-i条连续边<i,i+l>,<i+l,i+2>，…,<j-2,j-1>,<j-1,j>的距离和。由点i至点j的连续边组成的路径称为连续路径；
  3. d[i]——从点i出发由左至右直到n点后再由右至左到达点i-1的最短距离。

$$d[i] = \min_{i \le j \le N-1} \{W_{i\ldots j} + d[j+1] + W_{j+1,i-1}\} \qquad (1 \le i \le N)$$

计算矩阵连乘积

- 在科学计算中经常要计算矩阵的乘积。若A是一个p×q的矩阵，B是一个q×r的矩阵，则其乘积C=AB是一个p×r的矩阵。计算C=AB总共需要pqr次的乘法。现在的问题是，给定n个矩阵$\{A_1,A_2,\cdots,A_n\}$。其中A_i与A_{i+1}是可乘的，i=1,2,…,n-1。要求计算出这n个矩阵的连乘积$A_1A_2\cdots A_n$
- 看一个计算3个矩阵$\{A_1,A_2,A_3\}$的连乘积的例子。设3个矩阵的维数:10×100，100×5和5×50。按$((A_1A_2)A_3)$算，需10×100×5+10×5×50=7500次乘法。若按$(A_1(A_2A_3))$算，则需要的乘法次数为100×5×50+10×100×50=75000

计算矩阵连乘积

- **1.分析最优解的结构**
  - 记矩阵连乘积$A_iA_{i+1}\cdots A_j$为$A_{i\cdots j}$
  - 设计算次序在矩阵A_k和A_{k+1}之间将矩阵链断开，$1<=k<n$，则完全加括号方式为$((A_1\cdots A_k)(A_{k+1}\cdots A_n))$
  - 先计算$A_{1\cdots k}$和$A_{k+1\cdots n}$，然后将所得的结果相乘才得到$A_{1\cdots n}$。显然其总计算量为计算$A_{1\cdots k}$的计算量加上计算$A_{k+1\cdots n}$的计算量，再加上$A_{1\cdots k}$与$A_{k+1\cdots n}$相乘的计算量
  - 关键特征是：计算$A_{1\cdots n}$的一个最优次序所包含的计算$A_{1\cdots k}$和$A_{k+1\cdots n}$的次序也是最优的

计算矩阵连乘积

- **2.建立递归关系**
  - 设计算$A_{i\cdots j}$，$1\le i\le j\le n$，所需的最少数乘次数为m[i,j]，原问题的最优值为m[1,n]
- 当i=j时，$A_{i\cdots j}=A_i$为单一矩阵，无需计算，因此m[i,i]=0，i=1,2,…,n
  - m[i,j]可以递归地定义为：

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

  - 将对应于m[i,j]的断开位置k记录在s[i,j]

计算矩阵连乘积

3.计算最优值

- 注意递归计算过程中不同的子问题个数为n平方
- 用动态规划算法解此问题，可依据递归式以自底向上的方式进行计算，并保存已解决的子问题答案，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
- 下面给出的计算m[i,j]动态规划算法中，输入是序列$P=\{p_0,p_1,\cdots,p_n\}$，输出最优值m[i,j]外，还有使$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$达到最优的断开位置k=s[i,j]，$1\le i\le j\le n$

计算矩阵链连乘的最优断开位置

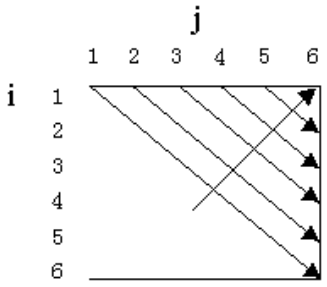
```
public static void matrixChain(int [] p, int [][] m, int [][] s)
{
    int n=p.length-1;
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int r = 2; r <= n; r++)
      for (int i = 1; i <= n - r+1; i++) {
        int j=i+r-1;
        m[i][j] = m[i+1][j]+ p[i-1]*p[i]*p[j];
        s[i][j] = i;
        for (int k = i+1; k < j; k++) {
          int t = m[i][k] + m[k+1][j] + p[i-1] p[k] p[j];
          if (t < m[i][j]) {
            m[i][j] = t;
            s[i][j] = k;}
          }
        }
```

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5 = 0+2500+35\times15\times20 = 13000 \\ m[2][3]+m[4][5]+p_1p_3p_5 = 2625+1000+35\times5\times20 = 7125 \\ m[2][4]+m[5][5]+p_1p_4p_5 = 4375+0+35\times10\times20 = 11375 \end{cases}$$

算法复杂度分析：

算法**matrixChain**的主要计算量取决于算法中对r，i和k的3重循环。循环体内的计算量为O(1)，而3重循环的总次数为$O(n^3)$。因此算法的计算时间上界为$O(n^3)$。算法所占用的空间显然为$O(n^2)$。



(a) 计算次序

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

计算矩阵连乘积

4.构造最优解

- s[i,j]中的数k告诉我们计算矩阵链$A_{i...j}$的最佳方式应在矩阵A_k和A_{k+1}之间断开，即最优的加括号方式应为$(A_{1...k})(A_{k+1\cdots n})$
- 因此，从s[i,j]记录的信息可知计算$A_{1...n}$的最优加括号方式为 $(A_{1\cdots s[1,n]})(A_{s[1,n]+1\cdots n})$
- $A_{1\cdots s[1,n]}$的最优加括号方式为$(A_{1\cdots s[1,s[1,n]]})(A_{s[1,s[1,n]]+1\cdots s[1,n]})$。同理$A_{s[1,n]+1\cdots n}$的最优加括号方式在s[s[1,n]+1,n]处断开
- 照此递推下去，最终可以确定$A_{s[1,n]+1\cdots n}$的最优完全加括号方式

计算矩阵连乘积

- 下面算法是按s指示$A=\{A_1,A_2,\ldots,A_n\}$子链$A_{i\ldots j}$连乘积的算法

```
Matrix MATRIX_CHAIN_MULTIPLY(A,s,i,j);
{
    if ( j>i ) {
        X←MATRIX_CHAIN_MULTIPLY(A,s,i,s[i,j]);
        Y←MATRIX_CHAIN_MULTIPLY(A,s,s[i,j]+1,j);
        return MATRIX_MULTIPLY(X,Y);
            //计算并返回矩阵X*Y的值
    }
    else return (Ai);
}
```

- 计算A1…n调用MATRIX_CHAIN_MULTIPLY(A,s,1,n)即可