算法设计与分析 第四部分

动态规划方法及实例分析

主要内容

- 动态规划的基本概念
- 动态规划的基本步骤
- 动态规划问题求解实例

动态规划的求解对象

- 最优化问题
  - 工程问题中设计参数的选择
  - 有限资源的合理分配
  - 车间作业调度
  - 交通系统的规划
  - 不胜枚举…………..

动态规划（dynamic programming）

- 分治法求解回顾
  - 子问题相互独立，不包含公共子问题
- 动态规划
  - 与分治法类似，也是将问题分解为规模逐渐减小的同类型的子问题
  - 与分治法不同，分解所得的子问题很多都是重复的

动态规划求解实例-矩阵连乘问题

- 矩阵连乘问题（矩阵链乘法）
- 一般描述：
  - 对于给定的n个矩阵，M_1， M_2 ，…， M_n，其中矩阵M_i 和M_j 是可乘的，要求确定计算矩阵连乘积（ $M_1M_2 \ldots M_n$ ）的计算次序，使得按照该次数计算矩阵连乘积时需要的乘法次数最少

矩阵连乘问题

- 例，设有矩阵M_1,M_2,M_3,M_4
- 其维数分别是：10×20, 20×50,50×1 ,1×100
- 现要求出这4个矩阵相乘的结果
- 计算次序可以通过加括号的方式确定
- 当n很大时，n个矩阵连乘的加括号的方法数是指数量级的，逐一检查不现实

矩阵连乘问题

- 目标：求出矩阵连乘M_iM_{i+1}---$M_{j-1}M_j$（i<j）所需的最少乘法次数
- 共有j-i+1个矩阵，可称这个矩阵连乘的规模是j-i+1
- 按照做最后一次乘法的位置进行划分，矩阵连乘一共可分为j-i种情况
- 若已知任一个规模不超过j-i的矩阵连乘所需的最少乘法次数，则可计算出M_iM_{i+1}---$M_{j-1}M_j$所需的最少乘法次数

矩阵连乘问题

- j-i种划分情况表示为通式：
  - ➢$(M_i \text{---} M_k)\ (M_{k+1} \text{---} M_j)$（$i \le k < j$）
- 记第t个矩阵M_t的列数为r_t，并令r_0为矩阵M_1的行数
- $M_i \text{---} M_k$连乘所得是$r_{i-1} \times r_k$维矩阵
- $M_{k+1} \text{---} M_j$连乘所得是$r_k \times r_j$维矩阵
- 这两个矩阵相乘需要做$r_{i-1} \times r_k \times r_j$次乘法

矩阵连乘问题

- 由于已知$(M_i \cdots M_k)$和$(M_{k+1} \cdots M_j)$所需的最少乘法次数，记为m_{ik}和$m_{k+1,j}$
- $(M_i \ldots M_k)(M_{k+1} \ldots M_j)$的矩阵连乘所需的最少乘法次数为：$m_{ik}+m_{k+1,j}+r_{i-1}\times r_k\times r_j$
- 对满足$i\le k<j$ 的共$j-i$种情况逐一进行比较，可得：
- $m_{ij}=\min_{（i\le k<j）}\{m_{ik}+m_{k+1,j}+r_{i-1}\times r_k\times r_j\}$

矩阵连乘问题

- 对于$m_{ij}=\min_{（i\le k<j）}\{m_{ik}+m_{k+1,j}+r_{i-1}\times r_k\times r_j\}$
- $m_{ii}=0$(相当于单个矩阵的情况)
- 首先求出计算M_1M_2，M_2M_3，…，$M_{n-1}M_n$所需的最少乘法次数$m_{i,i+1}(i=1,2,\cdots,n-1)$
- 再基于以上结果，根据m_{ij}的求解公式计算$M_1M_2M_3$，$M_2M_3M_4$, …，$M_{n-2}M_{n-1}M_n$所需的最少乘法次数$m_{i,i+2}(i=1,2,\cdots,n-2)$
- 直至$m_{i,i+3}(i=1,2,\cdots,n-3)$,$m_{i,i+4}(i=1,2,\cdots,n-4)$, $m_{1,n}$

矩阵连乘问题

- 已知M_1,M_2,M_3,M_4
- 其维数分别是：10×20, 20×50,50×1 ,1×100
- 求m_{13}
- 求解$m_{1,n}$的算法

for i=1 to n do $m_{ii} \leftarrow 0$;

for l=1 to n-1 do

for i=1 to n-l do

$\{j \leftarrow i+l; \quad m_{ij} \leftarrow \min_{(i \le k < j)} \{m_{ik}+m_{k+1,j}+r_{i-1} \times r_k \times r_j\}\}$

矩阵连乘问题

- 该方法将时间从brute-force法的指数量级降低到了$\Theta(n^3)$

矩阵连乘问题

- 动态规划相关的重要概念
  - 子问题的高度重复性
  - 最优子结构性质
    - 问题的最优解中包含着其每一个子问题的最优解

矩阵连乘问题

for i=1 to n do $m_{ii} \leftarrow 0$;

for i=1 to n-1 do $d_{i,i+1} \leftarrow i$;

for l=1 to n-1 do

for i=1 to n-l do

$\{j \leftarrow i+l$;

$m_{ij} \leftarrow \min_{（i \le k < j）} \{m_{ik}+m_{k+1,j}+r_{i-1} \times r_k \times r_j\}$

$d_{ij} \leftarrow k'\}$

矩阵连乘问题

- 例$M_1M_2M_3M_4M_5M_6$，其维数分别是30×35，35×15，15×5，5×10，10×20和20×25
- 二维数组(d_{ij})

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

适合用动态规划方法求解的问题

- 若一个问题可以分解为若干个高度重复的子问题，且问题也具有最优子结构性质，就可以用动态规划法求解
- 具体方式：可以递推的方式逐层计算最优值并记录必要的信息，最后根据记录的信息构造最优解

动态规划方法总体思想

- 保存已解决的子问题的答案，在需要时使用，从而避免大量重复计算

Those who cannot remember the past are doomed to repeat it.

-----George Santayana,
The life of Reason,
Book I: Introduction and
Reason in Common
Sense (1905)

动态规划方法解题步骤

- 找出最优解的性质，并刻画其结构特征
- 递归地定义最优值（写出动态规划方程）
- 以自底向上的递推方式计算出最优值
- 根据计算最优值时得到的信息，以递归方法构造一个最优解

最长公共子序列问题（LCS）

- 子序列的概念
- 设$X=< x_1, x_2,\cdots, x_m>$
- 若有$1\le i_1< i_2< \cdots <i_k\le m$
- 使得$Z=< z_1, z_2,\cdots, z_k> = < x_{i1}, x_{i2},\cdots, x_{ik}>$则称Z是X的子序列，记为$Z<X$

最长公共子序列问题（LCS）

- 公共子序列的概念
- 设X，Y是两个序列，且有Z<X和Z<Y
- 则称Z是X和Y 的公共子序列

最长公共子序列问题（LCS）

- 最长公共子序列的概念
- 若Z<X，Z<Y，且不存在比Z更长的X和Y的公共子序列
- 则称Z是X和Y 的最长公共子序列，记为$Z \in LCS(X,Y)$
- 请注意，最长公共子序列往往不止一个

LCS的求解方法

- Brute-force法
- 列出X的所有长度不超过n（即│Y│）的子序列，从长到短逐一进行检查，看其是否为Y的子序列，直到找到第一个最长公共子序列
- 由于X共有2^m个子序列，故此方法对较大m没有实用价值

最长公共子序列问题（LCS）

- 记X_i=﹤x_1，…，x_i﹥即X序列的前i个字符 (1≤i≤m)
- Y_j=﹤y_1，…，y_j﹥即Y序列的前j个字符 (1≤j≤n)
- 假定Z=﹤z_1，…，z_k﹥∈LCS(X,Y)

最长公共子序列问题（LCS）

- 若x_m=y_n (最后一个字符相同)，则有$z_k = x_m = y_n$ 且Z_{k-1}∈LCS(X_{m-1}，Y_{n-1})
- 若x_m≠y_n且z_k≠x_m，则有Z∈LCS(X_{m-1}，Y)
- 若x_m≠y_n且z_k≠y_n则有Z∈LCS(X，Y_{n-1})
- 求LCS(X_{m-1}，Y) 与LCS(X，Y_{n-1})的长度，不是相互独立的，具有重叠性
- 两个序列的LCS中包含了两个序列的前缀的LCS，具有最优子结构性质

最长公共子序列问题（LCS）

- 引入一个二维数组C，用C[i,j]记录X_i与Y_j的LCS的长度

- C[i,j]= $\begin{cases} 0 & \text{若} i=0 \text{或} j=0 \\ C[i-1,j-1]+1 & \text{若} i,j>0 \text{且} x_i = y_j \\ \max\{C[i-1,j],C[i,j-1]\} & \text{若} i,j>0 \text{且} x_i \neq y_j \end{cases}$

最长公共子序列问题（LCS）

- 为了构造出LCS，引入一个m×n的二维数组b，b[i,j]记录C[i,j]是通过哪一个子问题的值求得的，以决定搜索的方向
- 若C[i-1,j]≥C[i,j-1]，则b[i,j]中记入“↑”
- 若C[i-1,j]<C[i,j-1]，则b[i,j]中记入“←”

最长公共子序列问题（LCS）

```
LCS_L(X,Y,m,n,C)
for i=0 to m do C[i,0]←0
for j=1 to n do C[0,j]←0
for i=1 to m do {
  for j=1 to n do{
     if x[i]=y[j] then {C[i,j]←C[i-1,j-1]+1；
                    b[i,j]← “↖” ； }
}else if C[i-1,j]≥C[i,j-1] then {C[i,j]←C[i-1,j]；
                         b[i,j]← “↑” ； }
  }else{C[i,j]←C[i,j-1]；
      b[i,j]← “←” ； }
```

最长公共子序列问题（LCS）

- 输出一个LCS(X,Y)的递归算法

```
LCS_Output(b,X,i,j)
 If i=0 or j=0 then return;
 If b[i,j]= “↖” then      /*X[i]=Y[j]*/
   {LCS_Output(b,X,i-1,j-1);
       输出X[i]; }
 else if b[i,j]= “↑” then  /*C[i-1,j]≥C[i,j-1]*/
    LCS_Output(b,X,i-1,j)
 else if b[i,j]= “←” then  /*C[i-1,j]<C[i,j-1]*/
   LCS_Output(b,X,i,j-1)
```

最长公共子序列问题（LCS）

- 请注意，以上算法不能搜索到所有的LCS
- 没区分C[i-1,j]＞C[i,j-1] 和C[i-1,j]＝C[i,j-1]
- 当满足X[i]与Y[j]不等，且C[i-1,j]＝C[i,j-1]时，要执行b[i,j]←“←↑”，即记录两个搜索方向，才能够据此找出所有的LCS

最优二分搜索树

- 什么是二分搜索树
  - 或者是一棵空树
  - 或者是具有下列性质的二叉树
    - 若左子树不空，则左子树上所有结点的值均小于它的根结点的值
    - 若右子树不空，则右子树上所有结点的值均大于它的根结点的值
    - 左、右子树分别为二叉搜索树

最优二分搜索树

- 常见操作：查询二分搜索树
- 指令MEMBER(x,S)：若x在S中则返回”yes”，否则返回”no”
- 设有n个实数 $a_1<a_2<\ldots<a_n$构成集合S，考察由MEMBER指令构成的序列
- 指令MEMBER(x,S)中的x可能是某个a_i，也可能不在S中，把不在S中的数按区间分为n+1类，以b_0，b_1，…，b_n作为每一类数的代表（虚节点）

最优二分搜索树

- 定义
  - p_i为MEMBER (a_i,S)出现的频率（i=1,2,…,n）
  - q_j为MEMBER (b_j,S)出现的频率（j=0,1,2,…,n）
- 因此有$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$
- 定义一棵二分搜索树的总耗费：

$$\sum_{i=1}^{n} p_i(depth(a_i)+1) + \sum_{j=0}^{n} q_j(depth(b_j))$$

- 最优二分搜索树：耗费最小的二分搜索树

最优二分搜索树

- 分析
  - 假定T_0是最优二分搜索树，它的根是a_k（第k小的数）
  - 则a_k的左子树中必然包含了$\{a_1 \ldots a_{k-1}\}$，
  - a_k的右子树中必然包含了$\{a_{k+1} \ldots a_n\}$。

最优二分搜索树

- 考察一棵树接到另一个结点之下构成一棵新树时耗费的增加
- 设有一棵由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的树
- 按定义该树的耗费为

$$\sum_{l=i+1}^{j} p_l(depth(a_l)+1)+\sum_{l=i}^{j} q_l(depth(b_l))$$

最优二分搜索树

- 当这棵由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的树接到另一个结点之下构成一棵新树时
- 这棵子树中的每个结点的深度在新树中均增加了1
- 该子树在新树中的耗费增加了

$$\sum_{l=i+1}^{j} p_l + \sum_{l=i}^{j} q_l = q_i + (p_{i+1} + q_{i+1}) + \ldots + (p_j + q_j) = W_{ij}$$

最优二分搜索树

- 根据前面的约定以及二分搜索树的性质，任何一颗子树中结点的编号都是连续的
- 而且，最优树中的任何一棵子树，也必然是关于子树中结点的最优树
- 因此最优二分搜索树具有最优子结构性质

最优二分搜索树

- 若规模为$m \le n-1$的最优子树均已知
- 就可以通过逐一计算以a_1，a_2，…，a_n为根的树的耗费来确定（使耗费达到最小的）根a_k并找出最优二分搜索树
- 在上述计算中，规模较小（$m \le n-1$）的最优子树在计算中要多次被用到，因此，该问题具有高度重复性

最优二分搜索树

- 在所有由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的树中，把耗费最小的树记为T_{ij}
- 若树以a_k作为根 ($i+1 \le k \le j$)
  - 则b_i，a_{i+1}，b_{i+1}，…，a_{k-1}，b_{k-1}必然在其左子树中
  - 则b_k，a_{k+1}，b_{k+1}，…，a_j，b_j必然在其右子树中
  - 这样的树中耗费最小的必然是以$T_{i,k-1}$为其左子树，以$T_{k,j}$为其右子树
- 记c_{ij}是最优子树T_{ij}的耗费，则$c_{i,k-1}$是最优子树$T_{i,k-1}$的耗费，$c_{k,j}$是最优子树$T_{k,j}$的耗费

最优二分搜索树

- 考察以a_k $(i+1 \le k \le j)$为根、由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的、耗费最小的树的总耗费
- 由三部分组成
  - 左子树的耗费为：$c_{i,k-1}+ w_{i,k-1}$
  - 右子树的耗费为：$c_{k,j}+w_{k,j}$
  - 根的耗费为：p_k
- 总耗费为：$c_{i,k-1}+ w_{i,k-1}+c_{kj}+w_{k,j}+p_k$
- 总耗费为：$c_{i,k-1}+c_{kj}+w_{i,j}$

最优二分搜索树

- 对于以a_k为根、耗费最小的树的总耗费$c_{i,k-1}+c_{kj}+w_{ij}$
- p_i（i=1,2,…,n），q_j（j=0,1,2,…,n）已知
- 若$w_{i,j-1}$已知，则根据$w_{i,j}= w_{i,j-1}+p_j + q_j$可以计算出$w_{ij}$（由$w_{ij}$的定义）
- 故当$c_{i,k-1}$与c_{kj}已知时，以a_k为根的树的最小总耗费在O(1)时间就可以计算出来

最优二分搜索树

- 根据以上分析
- 分别计算以a_{i+1}，a_{i+2}，…，a_j为根、含有结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j的树的总耗费
- 从中选出耗费最小的树，此即最优子树T_{ij}
- 因此，最优子树T_{ij}的耗费为

$$c_{ij} = \min_{i<k\le j}\{c_{i,k-1} + c_{kj} + w_{ij}\}$$

最优二分搜索树

- 递推求c_{ij}及记录T_{ij}的根的算法

$w_{ii} \leftarrow q_i (i=1,2,\ldots,n)$;　$c_{ii} \leftarrow 0$

for $l \leftarrow 1$ to n do

{ for $i \leftarrow 0$ to $n-l$ do

{$j \leftarrow i+l$;

$w_{i,j} \leftarrow w_{i,j-1} + p_j + q_j$;

$c_{ij} \leftarrow \min_{(i<k<=j)} \{c_{i,k-1} + c_{kj} + w_{ij}\}$;

$r_{ij} \leftarrow k'$;

}

}

最优二分搜索树

$w_{ii} \leftarrow q_i (i=1,2,\ldots,n)$; $c_{ii} \leftarrow 0$

for $l \leftarrow 1$ to n do

{ for $i \leftarrow 0$ to $n-l$ do

{$j \leftarrow i+l$;

$w_{i,j} \leftarrow w_{i,j-1} + p_j + q_j$;

$c_{ij} \leftarrow \min_{(i<k<=j)} \{c_{i,k-1} + c_{kj} + w_{ij}\}$;

$r_{ij} \leftarrow k'$;

}

}

最优二分搜索树

- 动态规划方法：**$\Theta(n^3)$**
  - 三层循环，每个循环至多规模为n
- 穷举法
  - N个结点的二叉树共有$\Omega(4^n/n^{3/2})$个，使用穷举法需要检查指数个数个二分搜索树

最优二分搜索树

- 如何找出最优二分搜索树:根据r_{ij}去找
- 设T_{ij}的根为a_k (r_{ij}记录到的值是k)，则从根开始建结点

```
Build-tree(i,j,r,A)                          /*建立最优子树Tij*/
 {If i≥j return "nill";
  pointer←newnode(nodetype);
  k←rij;                                     /*必有i < k ≤ j*/
  pointer→value←A[k];     /*A[k]即ak*/
  pointer→leftson←Buildtree(i,k-1,r,A);  /*建立最优左子树Ti,k-1*/
  pointer→rightson←Buildertree(k,j,r,A); /*建立最优右子树Tk,j*/
  return pointer;}
```

最优二分搜索树

- 递推求c_{ij}及记录T_{ij}的根的算法可以改进，把算法时间复杂度从$\Theta(n^3)$降到$\Theta(n^2)$
- 可以证明：如果最小耗费树$T_{i,j-1}$和$T_{i+1,j}$的根分别为a_p和a_q，则必有(1)$p\le q$；(2)最小耗费树T_{ij}的根a_k满足$p\le k\le q$
- 因此，求$\min\{c_{i,k-1}+c_{kj}+w_{ij}\}$时，无需在$a_{i+1}\sim a_j$之间去一一尝试，而只要从$a_p\sim a_q$之间去找一个根即可

最优二分搜索树

$w_{ii} \leftarrow q_i (i=1,2,\ldots,n)$; $c_{ii} \leftarrow 0$

for $l \leftarrow 1$ **to** n **do**

{ **for** $i \leftarrow 0$ **to** $n-l$ **do**

{$j \leftarrow i+l$;

$w_{i,j} \leftarrow w_{i,j-1} + p_j + q_j$;

$c_{ij} \leftarrow \min_{(1<k<=j)} \{c_{i,k-1} + c_{kj} + w_{ij}\}$;

$r_{ij} \leftarrow k'$;

}

}

流水作业调度

- 设有n个作业，每一个作业i均被分解为m项任务: T_{i1}, T_{i2}, ┅ , T_{im}(1≤i≤n，故共有n×m个任务)，要把这些任务安排到m台机器上进行加工
- n个作业
- m项任务
- m台机器

流水作业调度

- 如果任务的安排满足下列3个条件，则称该安排为流水作业调度：
  - 1. 每个作业i的第j项任务T_{ij} ($1 \le i \le n$, $1 \le j \le m$) 只能安排在机器P_j上进行加工
  - 2. 作业i的第j项任务T_{ij}($1 \le i \le n$, $2 \le j \le m$)的开始加工时间均安排在第j-1项任务$T_{i,j-1}$加工完毕之后
  - 3. 任何一台机器在任何一个时刻最多只能承担一项任务

流水作业调度

- 最优流水作业调度
- 设任务T_{ij}在机器P_j上进行加工需要的时间为t_{ij}，如果所有的t_{ij} $(1 \le i \le n, 1 \le j \le m)$均已给出，要找出一种安排任务的方法，使得完成这n个作业的加工时间为最少，这个安排称之为最优流水作业调度
- 完成n个作业的加工时间：从安排的第一个任务开始加工, 到最后一个任务加工完毕，其间所需要的时间

流水作业调度

- 注意点
- 优先调度
  - 允许优先级较低的任务在执行过程中被中断，转而去执行优先级较高的任务
- 非优先调度
  - 任何任务一旦开始加工，就不允许被中断，直到该任务被完成
- 流水作业调度一般均指的是非优先调度

流水作业调度

- 当机器数(或称工序数)m≥3时，流水作业调度问题是一个NP-hard问题
- 当m=2时，该问题可有多项式时间的算法
- 为讨论的方便
  - 记t_{i1}为a_i（作业i在P_1上加工所需时间）
  - 记t_{i2}为b_i（作业i在P_2上加工所需时间）

流水作业调度

- 当机器P_1为空闲时，则任何一个作业的第一个任务都可以立即在P_1上执行
- 必有一个最优调度使得在P_1上的加工是无间断的
- 一定有一个最优调度使得在P_2上的加工空闲时间（从0时刻起算）为最小，同时还满足在P_1上的加工是无间断的

流水作业调度

- 如果在P_2上的加工次序与在P_1上的加工次序不同，则只可能增加加工时间（在最好情况下，增加的时间为0）
- 请注意，这里机器数m的值
- 仅需要考虑在P_1和P_2上加工次序完全相同的调度
- 为简化起见，假定所有$a_i \neq 0$

流水作业调度

- 最优调度具有如下性质
  - 在所确定的最优调度的排列中去掉第一个执行作业后，剩下的作业排列仍然还是一个最优调度，即该问题具有最优子结构的性质
  - 在计算规模为n的作业集合的最优调度时，该作业集合的子集合的最优调度会被多次用到，即该问题亦具有高度重复性
- 可以用动态规划方法求解？

流水作业调度

- 设N={1，2，---，n}是全部作业的集合，作业集S是N的子集合即有$S \subseteq N$
- 设对机器P_2需等待t个时间单位以后才可以用于S中的作业加工（t也可以为0即无须等待）
- 记g(S,t)为在此情况下完成S中全部作业的最短时间，则g(S,t)可递归表示为
- $g(S,t)=\min_{i\in S}\{a_i+ g(S-\{i\},b_i+\max\{t-a_i,0\})\}$

流水作业调度

- 当S=N即全部作业开始加工时，t=0。
- $g(N,0)=\min_{1\le i\le n}\{a_i+ g(N-\{i\},b_i)\}$
- 根据上式可以实现计算g(N,0)
- 该算法的时间复杂度为指数量级，因为算法中对N的每一个非空子集都要进行一次计算，而N的非空子集共有2^n-1个
- 因此不能直接使用动态规划方法来求解该问题

流水作业调度

- $\min\{a_j, b_i\} \geq \min\{a_i, b_j\}$ （Johnson不等式）
- 即当$\min\{a_i, a_j, b_i, b_j\}$为$a_i$或者$b_j$时，Johnson不等式成立，此时把i排在前j排在后的调度用时较少
- 反之，若$\min\{a_i, a_j, b_i, b_j\}$为$a_j$或者$b_i$时，则j排在前i排在后的调度用时较少

流水作业调度

- 推广到一般情况
  - 当$\min\{ a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n \}=a_k$时，则对任何$i \neq k$，都有$\min\{a_i, b_k\} \geq \min\{a_k, b_i\}$成立，故此时应将作业k安排在最前面，作为最优调度的第一个执行的作业
  - 当$\min\{ a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n \}= b_k$时，则对任何$i \neq k$，也都有$\min\{a_k, b_i\} \geq \min\{a_i, b_k\}$成立，故此时应将作业k安排在最后面，作为最优调度的最后一个执行的作业

流水作业调度

- n个作业中首先开工（或最后开工）的作业确定之后，对剩下的n-1个作业采用相同方法可再确定其中的一个作业,应作为n-1个作业中最先或最后执行的作业；
- 反复使用这个方法直到最后只剩一个作业为止，即可确定最优调度
- 时间主要耗费在对任务集的排序，因此，其时间复杂度为O(nlgn)

流水作业调度

- 满足1）高度重复性 2）最优子结构性质时，一般采用动态规划法，但偶尔也可能得不到高效的算法
- 若问题本身不是NP-hard问题
  - 进一步分析后就有可能获得效率较高的算法
- 若问题本身就是NP-hard问题
  - 与其它的精确算法相比，动态规划法性能一般不算太坏， 但有时需要对动态规划法作进一步的加工

备忘录方法

- 当某个问题可以用动态规划法求解,但二维数组中有相当一部分元素在整个计算中都不会被用到
- 因此，不需要以递推方式逐个计算二维数组中元素，而采用备忘录方法：数组中的元素只是在需要计算时才去计算，计算采用递归方式，值计算出来之后将其保存起来以备它用

备忘录方法

- 若有大量的子问题无需求解时，用备忘录方法较省时
- 但当无需计算的子问题只有少部分或全部都要计算时，用递推方法比备忘录方法要好（如矩阵连乘，最优二分搜索树）

备忘录方法 LCS

- LCS问题，当$x_i=y_j$时，求C[i,j]只需知道C[i-1,j-1]，而无需用到C[i,0]～C[i,j-1]及C[i-1,j]～C[i-1,n]
- 当只需求出一个LCS时，可能有一些C[p,q]在整个求解过程中都不会用到
- 首先将C[i,0]与C[0,j] 初始化为0
- 其余m×n个C[i,j]全部初始化为-1

备忘录方法 LCS

- 计算C[i,j]的递归算法LCS_L2(X,Y, i,j,C)
- 若x[i]=y[j]，则去检查C[i-1,j-1]
  - 若C[i-1,j-1]> -1（已经计算出来），就直接把C[i-1,j-1]+1赋给C[i,j]，返回
  - 若C[i-1,j-1]=-1（尚未计算出来），就递归调用LCS_L2(X,Y, i-1,j-1,C) 计算出C[i-1,j-1]，然后再把C[i-1,j-1]+1赋给C[i,j]，返回
- 若x[i]不等于y[j]，则检查C[i-1,j]和C[i,j-1]
  - 若两者均 > -1（已经计算出来），则把max{ C[i-1,j], C[i,j-1]}赋给C[i,j]，返回
  - 若C[i-1,j], C[i,j-1] 两者中有一个等于-1（尚未计算出来），或两者均等于-1，就递归调用LCS_L2将其计算出来，然后再把max{ C[i-1,j], C[i,j-1]} 赋给C[i,j]

最长递增子序列问题

- 最长递增子序列问题
  - Longest increasing subsequence, LIS
- 假设$A=<a_1, a_2, \ldots, a_n>$为由n个不同的实数组成的序列
- A的递增子序列L是这样的一个子序列
  - $L=<a_{k_1}, a_{k_2}, \ldots, a_{k_m}>$
  - 其中$k_1 < k_2 < \ldots < k_m$并且$a_{k_1} < a_{k_2} < \cdots < a_{k_m}$
- 最长递增子序列问题就是求A的最长递增子序列，也就是说，需要求最大的m值

最长递增子序列问题

- 求解方法
  - LIS与LCS
  - DP？

History Grading

- 在计算机科学中的许多问题是带约束的最优化问题
- 在一次历史考试中，要求学生按照时间顺序排列若干历史事件
  - 将所有事件按照正确次序排列的学生可得满分
  - 部分正确的学生如何得分？
    - 1 point for each event whose rank matches its correct rank
    - 1 point for each event in the longest (not necessarily contiguous) sequence of events which are in the correct order relative to each other
  - 请按照规则2为这样的问题评分

SKI

- Michael 喜欢滑雪。为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，不得不再次走上坡或者等付升降机来载你。
- Michael 想知道在一个区域中最长的滑坡。

SKI

- 区域由一个二维数组给出。数组的每个数字代表点的高度。
- 当且仅当高度减小，一个人可以从某个点滑向上下左右相邻四个点之一。

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

Wavio Sequence

- Wavio是一个整数序列，具有如下特性
  - Wavio的长度是奇数，即$L = 2 * n + 1$
  - Wavio序列的前$n + 1$个整数是一个严格的递增序列
  - Wavio序列的后$n + 1$个整数是一个严格的递减序列
  - 在Wavio序列中，没有两个相邻的整数是相同的
- 给出一个整数序列，请找出给出序列中的一个子序列，这个子序列是具有最长长度的Wavio序列

最大子段和问题

- n个整数序列$a_1 \ldots a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值
- 当所有整数均为负整数时定义其最大子段和为0
- 根据以上，$\max\left\{0, \max_{1\le i\le j\le n} \sum_{k=i}^{j} a_k\right\}$
  - 例：当$(a_1, a_2, \ldots, a_6)$= (-2,11 -4,13,-5,-2)
  - 此最大子段和为 $\sum_{k=2}^{4} a_k = 20$

最大子段和问题简单算法

- 用数组a[]存储n个整数$a_1 \ldots a_n$

```
void MaxSum(int n， int *a， int& besti， int& bestj)
{       int sum=0;
        for (int i = 1; i <= n; i++)
          for (int j = i; j <= n; j++){
             int thissum=0;
             for (int k = i; k<=j; k++) thissum+=a[k]; //i→j
             if (thissum>sum) {//记录i， j
                sum=thissum;
                besti=i;
                bestj=j;
             }
        }
        return sum;
}
```

可进行改进

显然计算时间是$O(n^3)$

最大子段和算法改进

```
void MaxSum(int n， int *a， int& besti， int& bestj)
{
      int sum=0;
      for (int i = 1; i <= n; i++){
         int thissum=0;
         for (int j = i; j <= n; j++){
            thissum+=a[j];                    //i→n的和
            if (thissum>sum) {
               sum=thissum;
               besti=i;
               bestj=j;
            }
         }
      }
      return sum;
}
```

从算法设计技巧上的改进，
计算时间是$O(n^2)$

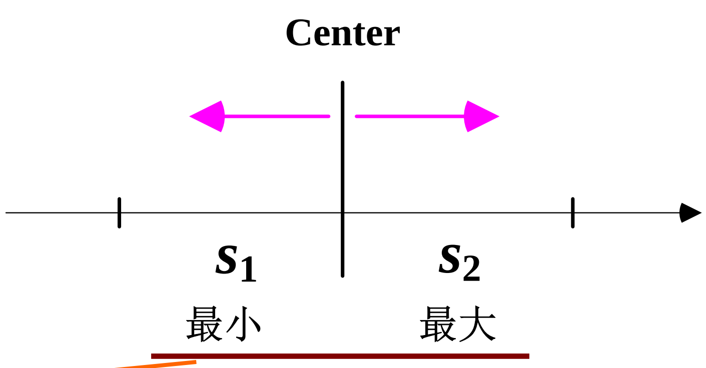
最大子段和算法进一步改进

- 如果将所给的序列$a[1..n]$分为长度相等的两段$a[1:n/2]$和$a[n/2+1:n]$,分别求出这两段最大子段和，则$a[1..n]$的最大子段和有三种情形：
  1. $a[1:n]$的最大子段和与$a[1:n/2]$的最大子段和相同；
  2. $a[1:n]$的最大子段和与$a[n/2+1:n]$的最大子段和相同；
  3. $a[1:n]$的最大子段和为 $\sum_{k=i}^{j} a_k$ ，且 $1 \le i \le n/2$ ，$n/2 + 1 \le j \le n$ 。
- 对于1，2两种情况可以递归求解
- 对于3， $a[n/2]$与$a[n/2+1]$在最优子序列中
  - 可以在a[1:n/2]中计算出 $s_1 = \max\limits_{1 \le i \le n/2} \sum_{k=i}^{n/2} a[k]$
  - 可以在a[n/2+1:n]中计算出 $s_2 = \max\limits_{n/2+1 \le i \le n} \sum_{k=\frac{n}{2}+1}^{i} a[k]$

s_1+s_2即为最优

最大子段和算法进一步改进

```
void MaxSubSum(int *a， int left， int right)
{
   int sum=0;
   if (left==right) sum=a[left]>0?a[left]:0;
   else{ int center=(left+right)/2;
      int leftsum=MaxSubSum(a,left,center);
      int rightsum=MaxSubSum(a,center+1,right);
      int s1=0;      int lefts=0;
      for(int i=center;i>=left;i--){
         lefts+=a[i];   if (lefts>s1) s1=lefts;
      }
      int s2=0;  int rights=0;
      for (int i=center+1; i<=right; i++){
      rights+=a[i];   if (rights>s2) s2=rights;
      }
     sum=s1+s2;
     if (sum<leftsum) sum=leftsum;
     if (sum<rightsum) sum=rightsum;}
return sum;
}
```



最大子段和分治算法分析

- 算法所需的计算时间T(n)满足典型的分治算法递归式：

$$T(n)=\begin{cases} O(1) & n \le c \\ 2T(n/2)+O(n) & n > c \end{cases}$$

- 基于主方法和主定理
  - $T(n)=O(n\log n)$

最大子段和动态规划算法

- 若记 $b[j]=\max\limits_{1\le i\le j}\left\{\sum\limits_{k=i}^{j}a[k]\right\}$，$1\le j\le n$，则所求最大子段和为

$$\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\max_{1\le i\le j}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}b[j]$$

- 由b[j]定义：

  - 当$b[j$-1]>0，$b[j]= b[j$-1]+ $a[j]$
  - 否则，$b[j]= a[j]$

$$\Rightarrow b[j]=\max_{1\le j\le n}\{b[j-1]+a[j],a[j]\}$$

- 据此，可设计出求最大子段和的动态规划算法

最大子段和动态规划算法

```
int MaxSum(int n， int *a)
{
    int sum=0, b=0;//初始化最大子段和为0，b[0]=0
    for (int i = 1; i <= n; i++){
        if (b>0)  b+=a[i];
        else  b=a[i];
        if (b>sum) sum=b;//更新当前找到的最大子段和
    }
    return sum;
}
```

算法时间复杂度$O(n)$

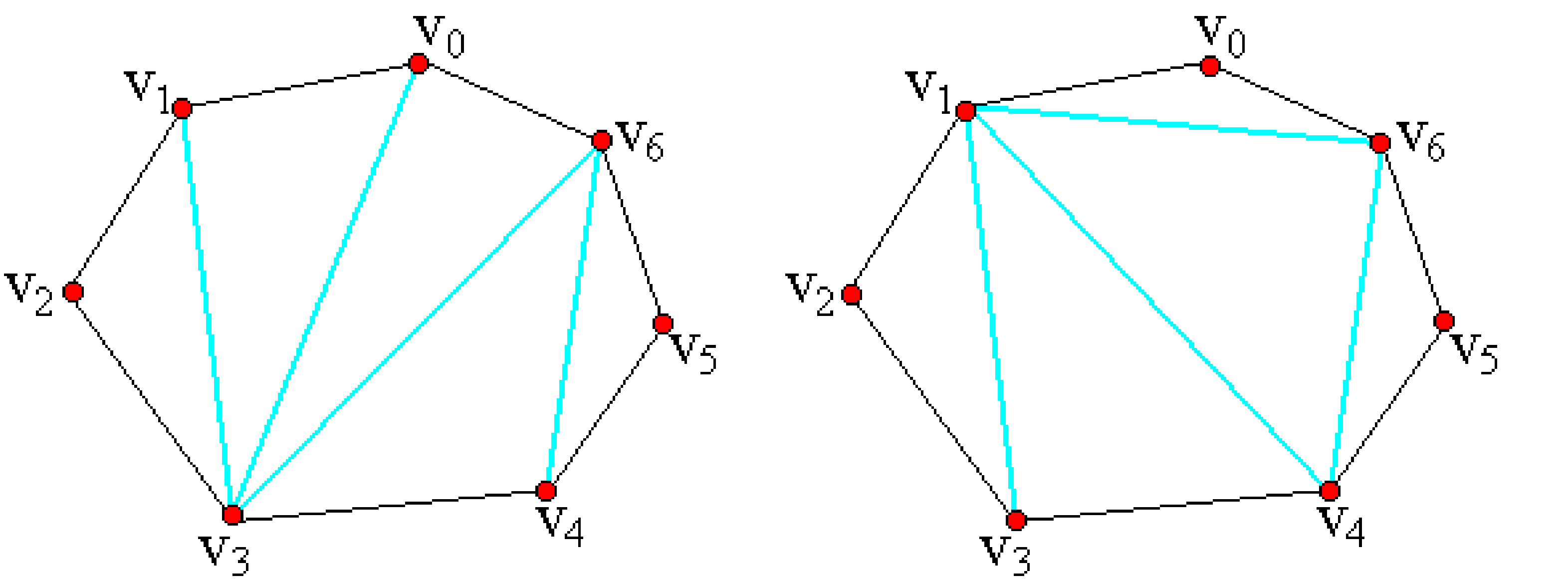
凸多边形最优三角剖分

- 凸多边形的特点：
  - 1围在多边形内的所有点构成多边形内部
  - 2多边形本身构成边界
  - 3其余部分构成多边形外部
- 凸多边形边界上/内部任意两点所连成的直线段上所有点均在凸多边形的内部或边界上
- 用多边形顶点的逆时针序列表示凸多边形，即$P=\{v_0,v_1,\ldots,v_{n-1}\}$表示具有n条边的凸多边形
- 若v_i与v_j是多边形上不相邻的2个顶点，则线段v_iv_j称为多边形的一条弦。弦将多边形分割成2个多边形$\{v_i,v_{i+1},\ldots,v_j\}$和$\{v_j,v_{j+1},\ldots v_i\}$

凸多边形最优三角剖分

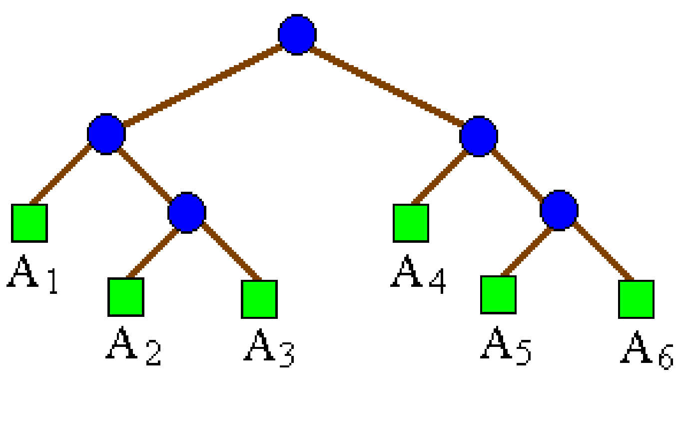
- 多边形的三角剖分是将多边形分割成互不相交的三角形的弦的集合T
- 给定凸多边形P，以及定义在由多边形的边和弦组成的三角形上的权函数w。要求确定该凸多边形的三角剖分，使得即该三角剖分中诸三角形上权之和为最小

凸多边形最优三角剖分

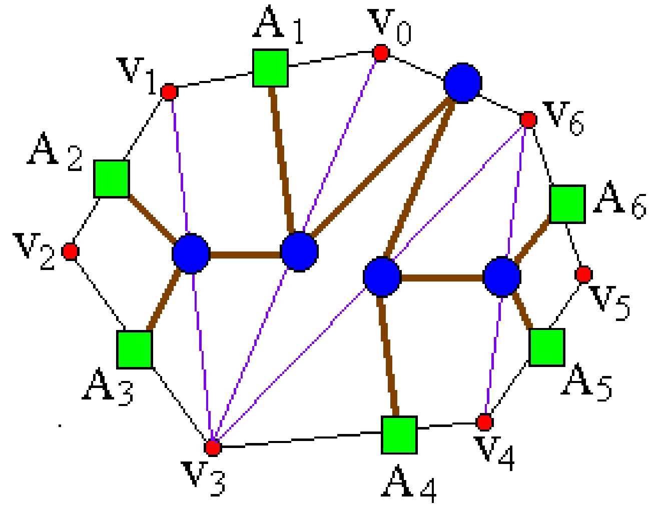


三角剖分的结构及其相关问题

- 一个表达式的完全加括号方式相应于一棵二叉树，称为表达式的语法树
  - 例如，完全加括号的矩阵连乘积 $((A_1(A_2A_3))(A_4(A_5A_6)))$所相应的语法树如图 (a)所示



(a)



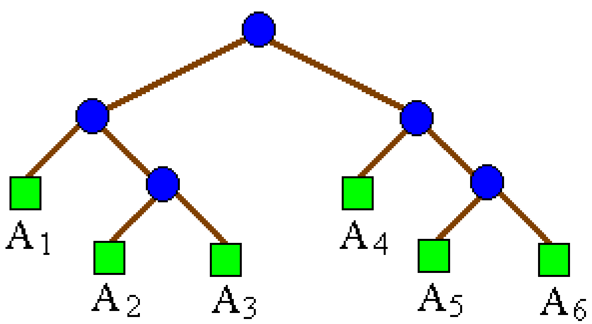
(b)

三角剖分的结构及其相关问题

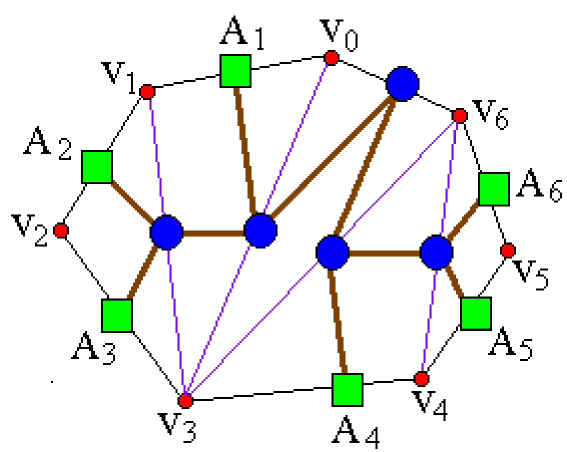
- 凸多边形$\{v_0,v_1,\ldots v_{n-1}\}$的三角剖分也可以用语法树表示。例如，图 (*b*)中凸多边形的三角剖分可用图 (*a*)所示的语法树表示

三角剖分的结构及其相关问题

- 矩阵连乘积中的每个矩阵A_i对应于凸多边形中的一条边$v_{i-1}v_i$。三角剖分中的一条弦v_iv_j，$i<j$，对应于矩阵连乘积$A[i+1:j]$



(a)



(b)

$v_0v_3v_6$将原凸多边形分为多边形$\{v_0 \ldots v_3\} \cup \{v_3 \ldots v_6\}$和$\{v_0v_3v_6\}$三角形。

最优三角剖分最优子结构性质

- 凸多边形的最优三角剖分问题有最优子结构性质
- 事实上，若凸$(n+1)$边形P=$\{v_0,v_1,\ldots,v_n\}$的最优三角剖分T包含三角形$v_0v_kv_n$，$1\le k\le n-1$，则T的权为3个部分权的和：三角形$v_0v_kv_n$的权，子多边形$\{v_0,v_1,\ldots,v_k\}$和$\{v_k,v_{k+1},\ldots,v_n\}$的权之和
- 可以断言，由T所确定的这2个子多边形的三角剖分也是最优的。因为若有$\{v_0,v_1,\ldots,v_k\}$或$\{v_k,v_{k+1},\ldots,v_n\}$的更小权的三角剖分将导致$T$不是最优三角剖分的矛盾

最优三角剖分的递归结构

- 定义$t[i][j]$，$1 \le i < j \le n$为凸子多边形$\{v_{i-1}, v_i, \ldots, v_j\}$的最优三角剖分所对应的权函数值，即其最优值。为方便起见，设退化的多边形$\{v_{i-1}, v_i\}$具有权值0。据此定义，要计算的凸$(n+1)$边形P的最优权值为$t[1][n]$
- $t[i][j]$的值可以利用最优子结构性质递归地计算。当$j-i \ge 1$时，凸子多边形至少有3个顶点。由最优子结构性质，$t[i][j]$的值应为$t[i][k]$的值加上$t[k+1][j]$的值，再加上三角形$v_{i-1}v_kv_j$的权值，其中$i \le k \le j-1$。由于在计算时还不知道k的确切位置，而k的所有可能位置只有$j-i$个，因此可以在这$j-i$个位置中选出使$t[i][j]$值达到最小的位置

最优三角剖分的递归结构

- 由此，t[i][j]可递归地定义为：

$$t[i][j]=\begin{cases}0 & i=j\\ \min\limits_{i\le k<j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

多边形游戏

- 多边形游戏是一个单人玩的游戏，开始时有一个由n个顶点构成的多边形
- 每个顶点被赋予一个整数值
- 每条边被赋予一个运算符“+”或“*”
- 所有边依次用整数从1到n编号

多边形游戏

- 游戏第1步，将一条边删除
- 随后n-1步按以下方式操作：
  - (1)选择一条边E以及由E连接着的2个顶点V1和V2；
  - (2)用一个新的顶点取代边E以及由E连接着的2个顶点V1和V2。将由顶点V1和V2的整数值通过边E上的运算得到的结果赋予新顶点。
- 最后，所有边都被删除，游戏结束。游戏的得分就是所剩顶点上的整数值
- 问题:对于给定的多边形，计算最高得分

多边形游戏

- 设所给的多边形的顶点和边的顺时针序列为op[1],v[1],op[2],v[2],….,op[n],v[n]
- 在所给多边形中，从顶点i(1≤i≤n)开始，长度为j(链中有j个顶点)的顺时针链p(i，j) 可表示为v[i]，op[i+1]，…，v[i+j-1]
- 如果这条链的最后一次合并运算在op[i+s]处发生(1≤s≤j-1)，则可在op[i+s]处将链分割为2个子链p(i，s)和p(i+s，j-s)

多边形游戏

- 对于2个子链p(i，s)和p(i+s，j-s)
- 设m1是对子链p(i，s)的任意一种合并方式得到的值，而a和b分别是在所有可能的合并中得到的最小值和最大值
- 设m2是p(i+s，j-s)的任意一种合并方式得到的值，而c和d分别是在所有可能的合并中得到的最小值和最大值

多边形游戏的最优子结构性质

- 根据上述定义有a≤m1≤b，c≤m2≤d
  - (1)当op[i+s]='+'时，显然有a+c≤m≤b+d
  - (2)当op[i+s]='*'时，有min{ac，ad，bc，bd}≤m≤max{ac，ad，bc，bd}
- 因此，主链的最大值和最小值可由子链的最大值和最小值得到
- 可以根据上述分析递归求解

0-1背包问题

- 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?
- 0-1背包问题是一个特殊的整数规划问题。

$$\max \sum_{i=1}^{n} v_i x_i \qquad \begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

0-1背包问题

设所给0-1背包问题的子问题 $\max \sum_{k=i}^{n} v_k x_k$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_k \in \{0,1\}, i \le k \le n \end{cases}$$

的最优值为$m(i,j)$，即$m(i,j)$是背包容量为j，可选择物品为i, $i+1, \ldots, n$时0-1背包问题的最优值。由0-1背包问题的最优子结构性质，可以建立计算$m(i,j)$的递归式如下。

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

$$m(n, j) = \begin{cases} v_n & j \ge w_n \\ 0 & 0 \le j < w_n \end{cases}$$

0-1背包问题

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。当背包容量c很大时，算法需要的计算时间较多。例如，当$c>2^n$时，算法需要$\Omega(n2^n)$计算时间。

不同类型的背包问题

- 完全背包
- 多重背包
- 混合背包
- 二维背包
- 分组背包
- 有依赖的背包

完全背包问题

- 给出n种物品和一个容量为M的背包，每种物品都有无限件，物品i重量为w_i，价值为p_i，其中$w_i > 0, p_i > 0, 1 <= i <= n$。
- 问将哪些物品装入背包可以使得背包中所放物品总重量不超过M，并且背包中物品的价值总和达到最大。

多重背包问题

- 给出n种物品和一个容量为M的背包，物品i重量为w_i，数量为num_i，价值为p_i，其中$w_i > 0$, $p_i > 0, num_i > 0, 1 <= i <= n$。
- 问将哪些物品装入背包可以使得背包中所放物品总重量不超过M，并且背包中物品的价值总和达到最大。

混合背包问题

- 在基本的0-1背包问题、完全背包和多重背包的基础上，将三者混合起来。
- 也就是说
  - 有的物品只可以取一次或者不取（基本的0-1背包）
  - 有的物品可以取无限次（完全背包）
  - 有的物品可以取得次数有一个上限（多重背包）

二维背包问题

- 二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价。
- 问怎样选择物品可以得到最大的价值。

分组背包问题

- 给出n种物品和一个容量为M的背包，每种物品都有无限件，物品i重量为w_i，价值为p_i，其中$w_i > 0, p_i > 0, 1 <= i <= n$。
- 这n个物品被划分为若干组，每组中的物品相互冲突，最多选一件放入背包。
- 问将哪些物品装入背包可以使得背包中所放物品总重量不超过M，并且背包中物品的价值总和达到最大。

有依赖的背包问题

- 此类问题是基本的0-1背包问题的变形。与基本的0-1背包问题不同的是，物品之间存在某种“依赖”的关系。
- 也就是说，如果物品i依赖于物品j，则表示如果要选物品i，则必须先选物品j。

动态规划方法的解题关键

- 如何递归定义最优值
- 如何构造最优解