动态规划及其优化

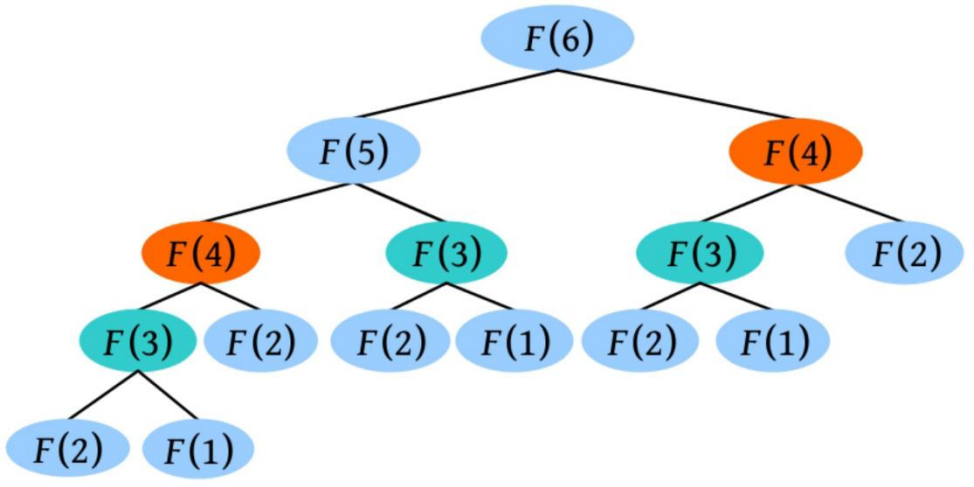
动态规划是理查德·贝尔曼于 1957 年在 **Dynamic Programming** 一书中提出的一种表格处理方法，它把原问题分解为若干子问题，自底向上先求解最小的子问题，把结果存储在表格中，求解大的子问题时直接从表格中查询小的子问题的解，以避免重复计算，从而提高效率。

1　动态规划求解原理

对什么样的问题可以使用动态规划求解呢？首先要分析问题是否具有以下 3 个性质。

（1）最优子结构。最优子结构指问题的最优解包含其子问题的最优解，是使用动态规划的基本条件。

（2）子问题重叠。子问题重叠指求解过程中每次产生的子问题并不总是新问题，有大量子问题是重复的。例如，递归求解斐波那契数列时，有大量子问题被重复求解，如下图所示。动态规划算法利用了子问题重叠的性质，自底向上对每一个子问题都只求解一次，将其结果存储在一个表格中，当再次需要求解该子问题时，直接在表格中查询，无须再次求解，从而提高效率。子问题重叠不是使用动态规划解决问题的必要条件，但更能突出动态规划的优势。



（3）无后效性。在动态规划中会将原问题分解为若干子问题，将每个子问题的求解过程都作为一个阶段，在完成前一阶段后，根据前一阶段的结果求解后一阶段。并且，对当前阶段的求解只与之前阶段有关，与之后阶段无关，这叫作“无后效性”。如果一个问题有后效性，则需要将其转化或逆向求解来消除后效性，然后才可以使用动态规划。

原理 1　动态规划的三个要素

在现实生活中有一类活动，可以将活动过程按顺序分解成若干个相互联系的阶段，在每一阶段都要做出决策，对全部过程的决策是一个决策序列。对每一阶段决策的选取都不是随意确定的，依赖于当前状态，又影响以后的发展。这种把问题看作一个前后关联的具有链状结构的多阶段的过程叫作多阶段决策过程，这种问题就叫作多阶段决策问题。

动态规划把原问题划分为若干子问题，通过求解子问题的解得到原问题的解，每个子问题的求解过程都构成一个阶段，在完成前一阶段的求解后才会进行后一阶段的求解。根据无后效性，**动态规划的求解过程构成一个有向无环图，求解遍历的顺序就是该有向无环图的一个拓扑序。**在有向无环图中，节点对应问题的状态，有向边对应状态之间的转移，对转移的选择对应动态规划中的决策。所以，状态、阶段、决策就是动态规划的三个要素。

例如，使用动态规划求解单源最短路径问题，过程如下。

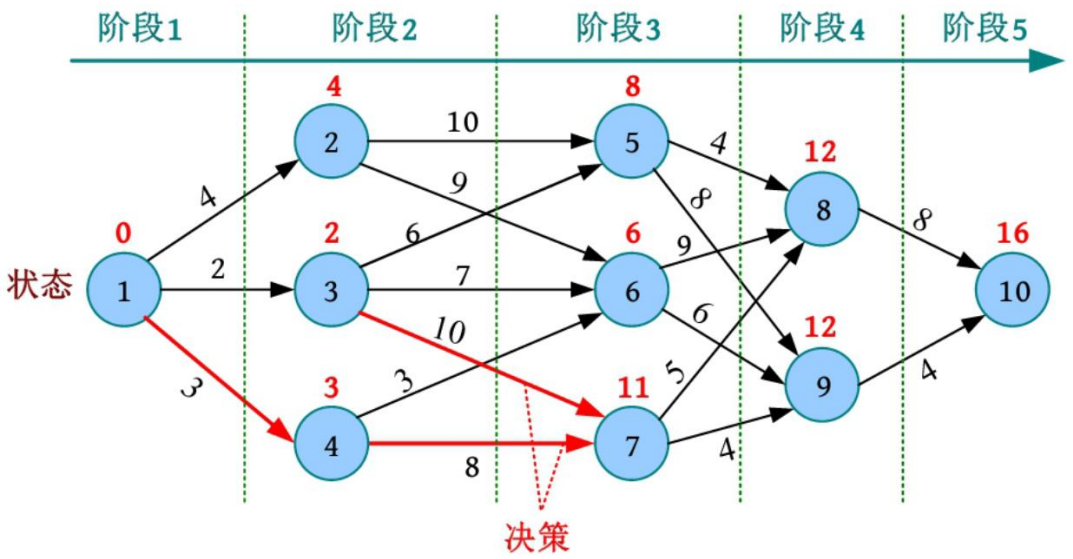
（1）确定状态，dp[i]表示源点到节点 i 的最短距离。

（2）根据拓扑序列划分阶段。

（3）决策选择：考察当前节点的逆邻接点，将所有逆邻接点的最短距离与边权之和取最小值得到 dp[i]，写出状态转移方程，dp[i]=min(dp[j]+w[j][i])，<j, i>∈E。

（4）边界条件：若源点为 1，则令 dp[1]=0。

（5）求解目标：dp[i]，i=2, 3, …, n，如下图所示。



原理 2　动态规划设计方法

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择达到结束状态；或者倒过来，从结束状态开始，通过对中间阶段决策的选择达到初始状态。这些决策形成一个决策序列，同时确定了完成整个过程的一条活动路线，通常是求最优活动路线。动态规划有一定的设计模式，一般分为以下步骤。

（1）状态表示。将问题发展到各个阶段时所处的各种客观情况用不同的状态表示出来，确定状态和状态变量。当然，对状态的选择要满足无后效性。

（2）阶段划分。按照问题的时间特征或空间特征，将问题划分为若干阶段。划分后的阶段一定是有序或可排序的，否则问题无法求解。

（3）状态转移。状态转移指根据上一阶段的状态和决策导出本阶段的状态。根据相邻两阶段各个状态之间的关系确定决策，一旦确定决策，就可以写出状态转移方程。

（4）边界条件。状态转移方程是一个递归式，需要确定初始条件或边界条件。

（5）求解目标。确定问题的求解目标，根据状态转移方程的递推结果得到求解目标。

求解动态规划问题时，如何确定状态和状态转移方程是关键，也是难点。不同的状态和状态转移方程可能产生不同的算法复杂度。动态规划问题灵活多变，在各类算法竞赛中层出不穷，需要多练习、多总结，积累丰富的经验和发挥创造力。

背包问题

背包问题指在一个有容积或重量限制的背包中放入物品，物品有体积、重量、价值等属性，要求在满足背包限制的情况下放置物品，使背包中物品的价值之和最大。根据物品限制条件的不同，背包问题可分为 01 背包问题、完全背包问题、多重背包问题、分组背包问题和混合背包问题等。

原理 1　01 背包

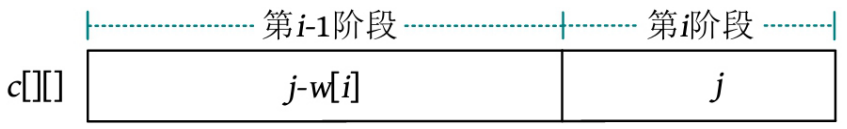
给定 n 种物品，每种物品都有重量 w_i 和价值 v_i，每种物品都只有一个。另外，背包容量为 W。求解在不超过背包容量的情况下将哪些物品放入背包，才可以使背包中的物品价值之和最大。**每种物品只有一个，要么不放入（0），要么放入（1），因此称之为 01 背包。**

假设第 i 阶段表示处理第 i 种物品，第 i-1 阶段表示处理第 i-1 种物品，则当处理第 i 种物品时，前 i-1 种物品已处理完毕，只需考虑第 i-1 阶段向第 i 阶段的转移。

状态表示：c[i][j]表示将前 i 种物品放入容量为 j 的背包中获得的最大价值。

第 i 种物品的处理状态包括以下两种。

- **不放入：**放入背包的价值不增加，问题转化为“将前 i-1 种物品放入容量为 j 的背包中获得的最大价值”，最大价值为 c[i-1][j]。
- **放入：**在第 i 种物品放入之前为第 i-1 阶段，相当于从第 i-1 阶段向第 i 阶段转化。问题转化为“将前 i-1 种物品放入容量为 j-w[i]的背包中获得的最大价值”，此时获得的最大价值就是 c[i-1][j-w[i]]，再加上放入第 i 种物品获得的价值 v[i]，总价值为 c[i-1][j-w[i]]+v[i]。



若背包容量不足，则肯定不可以放入，所以价值仍为前 i-1 种物品处理后的结果；若背包容量充足，则考察在放入、不放入哪种情况下获得的价值更大。状态转移方程：

$$c[i][j]=\begin{cases} c[i-1][j] & ,j<w[i] \\ \max\{c[i-1][j],c[i-1][j-w[i]]+v[i]\} & ,j\geqslant w[i] \end{cases}$$

1. 算法步骤

1）初始化

初始化 c[][]数组 0 行 0 列为 0：c[0][j]=0，c[i][0]=0，其中 i=0, 1, 2, …, n，j=0, 1, 2, …, W，表示第 0 种物品或背包容量为 0 时获得的价值均为 0。

2）循环阶段

（1）按照状态转移方程处理第 1 种物品，得到 c[1][j]，j=1, 2, …, W。

（2）按照状态转移方程处理第 2 种物品，得到 c[2][j]，j=1, 2, …, W。

（3）以此类推，得到 c[n][j]，j=1, 2, …, W。

3）构造最优解

c[n][W]就是不超过背包容量时可以放入物品的最大价值（最优值）。若还想知道具体放入了哪些物品，则需要根据 c[][]数组逆向构造最优解。对此可以用一维数组 x[]来存储解向量，x[i]=1 表示第 i 种物品被放入背包，x[i]=0 表示第 i 种物品未被放入背包。

（1）初始时 i=n，j=W。

（2）若 c[i][j]>c[i−1][j]，则说明第 i 种物品被放入背包，令 x[i]=1，j−=w[i]；若 c[i][j]≤c[i−1][j]，则说明第 i 种物品没被放入背包，令 x[i]=0。

（3）i--，转向第 2 步，直到 i=1 时处理完毕。

此时已经得到解向量（x[1], x[2], …, x[n]），直接输出该解向量，也可以仅把 x[i]=1 的物品序号 i 输出。

2. 图解

有 5 个物品，重量分别为 2、5、4、2、3，价值分别为 6、3、5、4、6。背包的容量为 10。求解在不超过背包容量的前提下将哪些物品放入背包，才可以使背包中的物品价值之和最大。

	1	2	3	4	5
w[]	2	5	4	2	3

	1	2	3	4	5
v[]	6	3	5	4	6

（1）初始化。c[i][j]表示将前 i 种物品放入容量为 j 的背包中可以获得的最大价值。初始化 c[][]数组第 0 行第 0 列为 0。

c[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										
5	0										

（2）按照状态转移方程处理第 1 种物品（i=1），w[1]=2，v[1]=6，如下图所示。

$$c[i][j]=\begin{cases} c[i-1][j] & , j < w[i] \\ \max\{c[i-1][j], c[i-1][j-w[i]]+v[i]\} & , j \geqslant w[i] \end{cases}$$

c[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0										
3	0										
4	0										
5	0										

其中：

- j=1 时，c[1][1]=c[0][1]=0；
- j=2 时，c[1][2]=max{c[0][2], c[0][0]+6}=6；
- j=3 时，c[1][3]=max{c[0][3], c[0][1]+6}=6；
- j=4 时，c[1][4]=max{c[0][4], c[0][2]+6}=6；
- j=5 时，c[1][5]=max{c[0][5], c[0][3]+6}=6；
- j=6 时，c[1][6]=max{c[0][6], c[0][4]+6}=6；
- j=7 时，c[1][7]=max{c[0][7], c[0][5]+6}=6；
- j=8 时，c[1][8]=max{c[0][8], c[0][6]+6}=6；
- j=9 时，c[1][9]=max{c[0][9], c[0][7]+6}=6；
- j=10 时，c[1][10]=max{c[0][10], c[0][8]+6}=6。

（3）按照状态转移方程处理第 2 种物品（i=2），w[2]=5，v[2]=3，如下图所示。

c[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0										
4	0										
5	0										

其中：

- j=1 时，c[2][1]=c[1][1]=0；
- j=2 时，c[2][2]=c[1][2]=6；
- j=3 时，c[2][3]=c[1][3]=6；
- j=4 时，c[2][4]=c[1][4]=6；
- j=5 时，c[2][5]=max{c[1][5], c[1][0]+3}=6；
- j=6 时，c[2][6]=max{c[1][6], c[1][1]+3}=6；
- j=7 时，c[2][7]=max{c[1][7], c[1][2]+3}=9；
- j=8 时，c[2][8]=max{c[1][8], c[1][3]+3}=9；
- j=9 时，c[2][9]=max{c[1][9], c[1][4]+3}=9；
- j=10 时，c[1][10]=max{c[1][10], c[1][5]+3}=9。

（4）按照状态转移方程处理第 3 种物品（i=3），w[3]=4，v[3]=5，如下图所示。

c[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0										
5	0										

其中：

- j=1 时，c[3][1]=c[2][1]=0；
- j=2 时，c[3][2]=c[2][2]=6；
- j=3 时，c[3][3]=c[2][3]=6；
- j=4 时，c[3][4]=max{c[2][4], c[2][0]+5}=6；
- j=5 时，c[3][5]=max{c[2][5], c[2][1]+5}=6；
- j=6 时，c[3][6]=max{c[2][6], c[2][2]+5}=11；
- j=7 时，c[3][7]=max{c[2][7], c[2][3]+5}=11；
- j=8 时，c[3][8]=max{c[2][8], c[2][4]+5}=11；
- j=9 时，c[3][9]=max{c[2][9], c[2][5]+5}=11；
- j=10 时，c[3][10]=max{c[2][10], c[2][6]+5}=11。

（5）按照状态转移方程处理第 4 种物品（i=4），w[4]=2，v[4]=4，如下图所示。

c[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0	0	6	6	10	10	11	11	15	15	15
5	0										

其中：

- j=1 时，c[4][1]=c[3][1]=0；
- j=2 时，c[4][2]=max{c[3][2], c[3][0]+4}=6；
- j=3 时，c[4][3]=max{c[3][3],+4}=c[3][1]6；
- j=4 时，c[4][4]=max{c[3][4], c[3][2]+4}=10；
- j=5 时，c[4][5]=max{c[3][5], c[3][3]+4}=10；
- j=6 时，c[4][6]=max{c[3][6], c[3][4]+4}=11；
- j=7 时，c[4][7]=max{c[3][7], c[3][5]+4}=11；
- j=8 时，c[4][8]=max{c[3][8], c[3][6]+4}=15；
- j=9 时，c[4][9]=max{c[3][9], c[3][7]+4}=15；
- j=10 时，c[4][10]=max{c[3][10], c[3][8]+4}=15。

（6）按照状态转移方程处理第 5 种物品（i=5），w[5]=3，v[5]=6，如下图所示。

c[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0	0	6	6	10	10	11	11	15	15	15
5	0	0	6	6	10	12	12	16	16	17	17

其中：

- j=1 时，c[5][1]=c[4][1]=0；
- j=2 时，c[5][2]=c[4][2]=6；
- j=3 时，c[5][3]=max{c[4][3], c[4][0]+6}=6；
- j=4 时，c[5][4]=max{c[4][4], c[4][1]+6}=10；
- j=5 时，c[5][5]=max{c[4][5], c[4][2]+6}=12；
- j=6 时，c[5][6]=max{c[4][6], c[4][3]+6}=12；
- j=7 时，c[5][7]=max{c[4][7], [=c[4]4]+6}16；
- j=8 时，c[5][8]=max{c[4][8], =c[4][5]+6}16；

- j=9 时，c[5][9]=max{c[4][9], c[4][6]+6}=17；
- j=10 时，c[5][10]=max{c[4][10], c[4][7]+6}=17。

（7）构造最优解：

①读取 c[5][10]>c[4][10]，说明第 5 种物品被放入背包，即 x[5]=1，j=10−w[5]=7；

②发现 c[4][7]=c[3][7]，说明第 4 种物品没被放入背包，即 x[4]=0；

③发现 c[3][7]>c[2][7]，说明第 3 种物品被放入背包，即 x[3]=1，j=j−w[3]=3；

④发现 c[2][3]=c[1][3]，说明第 2 种物品没被放入背包，即 x[2]=0；

⑤发现 c[1][3]>c[0][3]，说明第 1 种物品被放入背包，即 x[1]=1，j=j−w[1]=1，如下图所示。

c[][]	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	6	6	6	9	9	9	9
3	0	0	6	6	6	6	11	11	11	11	11
4	0	0	6	6	10	10	11	11	15	15	15
5	0	0	6	6	10	12	12	16	16	17	17

3. 算法实现

1）求解放入背包的物品最大价值 c[i][j]表示将前 i 种物品放入容量为 j 的背包中可以获得的最大价值。对每种物品都进行计算，背包容量 j 为 1～W，若物品重量大于背包容量，则不放此物品，c[i][j]=c[i−1][j]；否则比较放与不放此物品哪种使背包内的物品价值最大，即 c[i][j]=max(c[i−1][j], c[i−1][j−w[i]]+v[i])。

算法代码：

```
for(i=1;i<=n;i++)//计算 c[i][j]
     for(j=1;j<=W;j++)
          if(j<w[i])  //若物品重量大于背包容量，则不放此物品
               c[i][j]=c[i-1][j];
          else    //否则比较放与不放此物品哪种使背包内的物品价值最大
               c[i][j]=max(c[i-1][j],c[i-1][j-w[i]]+v[i]);
cout<<"放入背包的最大价值为:"<<c[n][W]<<endl;
```

2）最优解构造

根据 c[][]数组的计算结果逆向递推最优解，若 c[i][j]>c[i−1][j]，则说明第 i 种物品被放入背包，令 x[i]=1，j−=w[i]；若 c[i][j]≤c[i−1][j]，则说明第 i 种物品没被放入背包，令 x[i]=0。

算法代码：

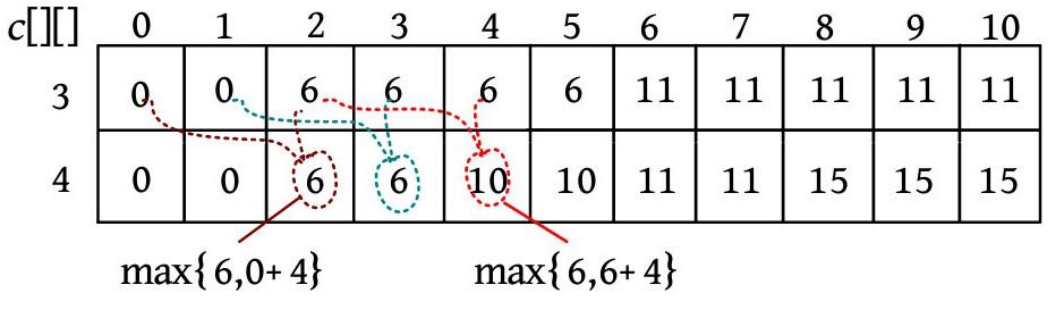
```
//逆向构造最优解
j=W;
for(i=n;i>0;i--)
    if(c[i][j]>c[i-1][j]){
          x[i]=1;
          j-=w[i];
    }
    else
        x[i]=0;
cout<<"放入背包的物品为:";
for(i=1;i<=n;i++)
    if(x[i]==1)
          cout<<i<<"  ";
```

算法分析：本算法使用了两层 for 循环，**时间复杂度为 O(nW)**；使用了二维数组 c[n][W]，**空间复杂度为 O(nW)**。

4. 算法优化

根据求解过程可以看出，依次处理 1..n 的物品，当处理第 i 种物品时，只需第 i-1 种物品的处理结果，若不需要构造最优解，则放入第 i-1 种物品之前的处理结果已经没用了。

例如，处理到第 4 种物品（w[4]=2，v[4]=4）时，只需第 3 种物品的处理结果（上一行）。求第 j 列时，若 j<w[4]，则照抄上一行；若 j≥w[4]，则需要将上一行第 j 列的值与上一行第 j-w[4]列的值+v[4]的值进行比较，取最大值，如下图所示。

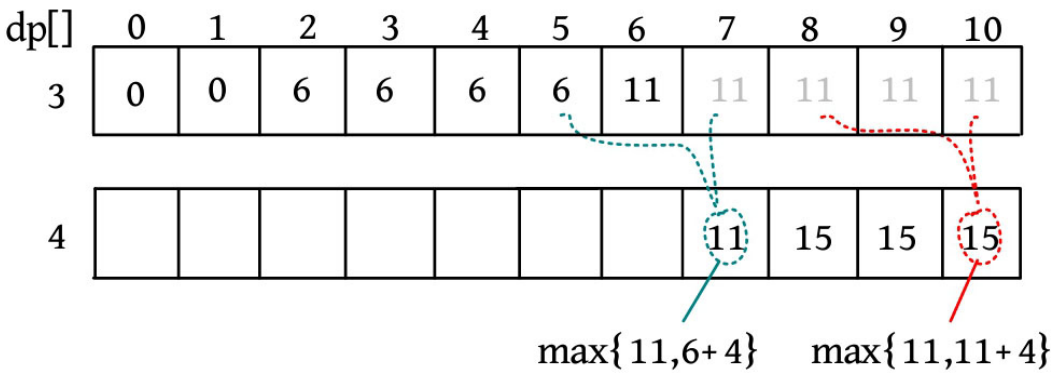


既然只需上一行当前列和前面列的值，那么只用一个一维数组倒推就可以了。

状态表示：dp[j]表示将物品放入容量为 j 的背包中可以获得的最大价值。

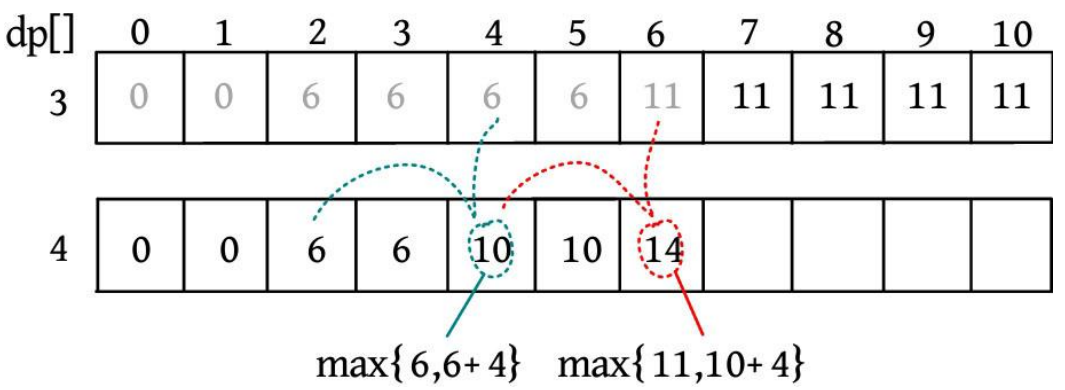
状态转移方程：dp[j]=max{dp[j], dp[j-w[i]]+v[i]}。

倒推的计算过程如下图所示。



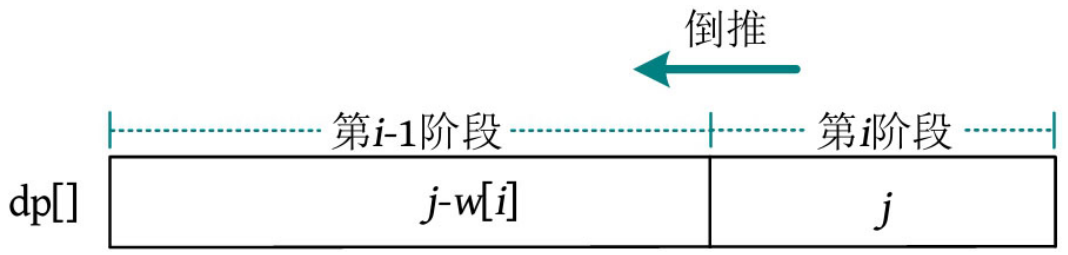
为什么不正推呢？下面进行推理。

正推的情况：求解 dp[4]时，将当前值与 dp[2]+4 进行比较，取最大值，发现将第 4 种物品放入的价值最大，结果为 10；求解 dp[6]时，将当前值与 dp[4]+4 进行比较，求最大值，发现将第 4 种物品放入的价值最大，结果为 14；此时第 4 种物品被放入两次，因为计算 dp[6]时，dp[4]不是第 3 种物品处理完毕的结果，而是放入第 4 种物品更新后的结果，如下图所示。

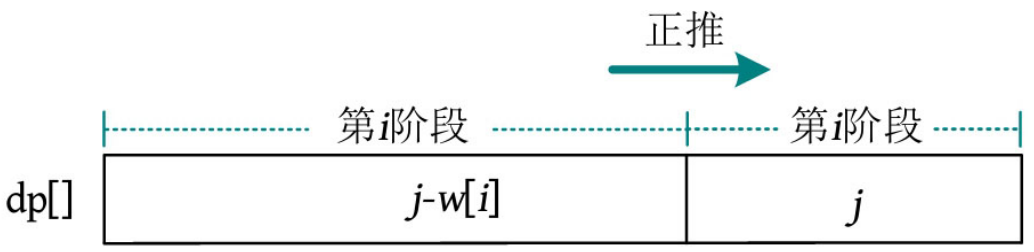


第 i 阶段表示处理第 i 种物品时，前 i-1 种物品已被处理完毕。倒推时，**从后往前推**，前面的值还未更新，仍为第 i-1 阶段的结果，这意味着总是用第 i-1 阶段的结果更新第 i 阶段，即从第 i-1 阶段向第 i 阶段进行状态转移。

第 i-1 阶段的结果不包括第 i 种物品，**保证第 i 种物品最多只被放入背包 1 次**，如下图所示。



正推时，从前往后推，前面的值已被更新为第 i 阶段，这意味着总是用第 i 阶段的结果更新第 i 阶段，即从第 i 阶段向第 i 阶段进行状态转移。**这样第 i 种物品可能被放入背包多次**，如下图所示。



在 01 背包问题中，每种物品只有一个，最多被放入 1 次，所以必须采用倒推形式求解。若每种物品有多个且可被放入多次（完全背包），则采用正推求解，见下一小节的内容。

算法代码：

```
void opt2(int n,int W){//采用 01 背包优化一维数组
     for(i=1;i<=n;i++)
          for(j=W;j>=w[i];j--)//逆向循环（倒推）
               dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
}
```

算法分析：本算法包含两层 for 循环，**时间复杂度为 O(nW)**；使用了一维数组 dp[W]，**空间复杂度为 O(W)。**

原理 2　完全背包

给定 n 种物品，每种物品都有重量 w_i 和价值 v_i，其数量没有限制。背包容量为 W，求解在不超过背包容量的情况下如何放置物品，使背包中物品的价值之和最大。

假设第 i 阶段表示处理第 i 种物品，因为第 i 种物品可以被多次放入，所以相当于从第 i 阶段向第 i 阶段转移。

根据对 01 背包算法优化的分析，可以采用一维数组正推，这样每种物品都可被多次放入。

状态表示：dp[j]表示将物品放入容量为 j 的背包中可以获得的最大价值。

状态转移方程：dp[j]=max{dp[j], dp[j-w[i]]+v[i]}。

算法代码：

```
void comp_knapsack(int n,int W){//完全背包问题
     for(i=1;i<=n;i++)
          for(j=w[i];j<=W;j++)//正序循环（正推）
               dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
}
```

算法分析：本算法**时间复杂度为 O(nW)**，**空间复杂度为 O(W)。**

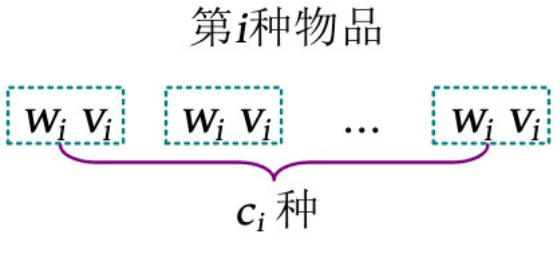
原理 3　多重背包

给定 n 种物品，每种物品都有重量 w_i 和价值 v_i，每种物品的数量都可以大于 1 但是有限制。第 i 种物品有 c_i 个。背包容量为 W，求解在不超过背包容量的情况下如何放置物品，可以使背包中物品的价值之和最大。

我们可以将多重背包问题**通过暴力拆分或二进制拆分转化为 01 背包问题，也可以通过数组优化对物品数量进行限制。**

1. 暴力拆分

暴力拆分指将第 i 种物品看作 ci 种独立的物品，每种物品只有一个，转化为 01 背包问题。状态表示和状态转移方程与 01 背包问题中的相同，如下图所示。



算法代码：

```
void multi_knapsack1(int n,int W){//暴力拆分
     for(i=1;i<=n;i++)
          for(k=1;k<=c[i];k++)//多一层循环
               for(j=W;j>=w[i];j--)
                    dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
}
```

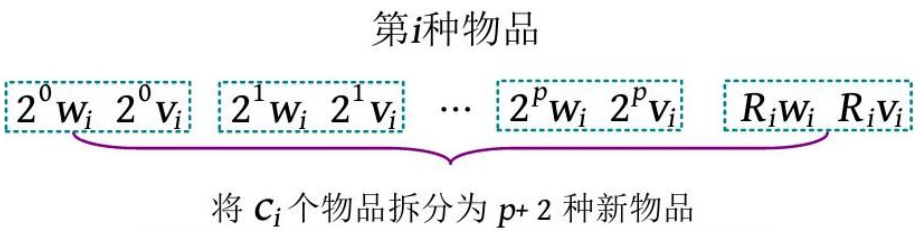
算法分析：本算法包含 3 层 for 循环，时间复杂度为 $O(W\sum c_i)$，空间复杂度为 O(W)。

2. 二进制拆分

当物品满足 c[i]×w[i]≥W 时，可以认为这种物品是不限数量的，按照完全背包的方法求解即可；否则可以采用二进制拆分，将 c[i]个物品拆分成若干种新物品。

一定存在一个最大的整数 p，使得 $2^0+2^1+2^2+…+2^p \le c[i]$，将剩余部分用 R_i 表示，$R_i=c[i]-(2^0+2^1+2^2+…+2^p)$。可以将 c[i]拆分为 p+2 个数：$2^0$, 2^1, 2^2, …, 2^p, R_i，例如，若 c[i]=9，则可以将 9 拆分为 2^0, 2^1, 2^2, $9-(2^0+2^1+2^2)$，即 1, 2, 4, 2，相当于将 9 个物品分成 4 堆，第 1 堆有 1 个物品，第 2 堆有 2 个物品，第 3 堆有 4 个物品，第 4 堆有 2 个物品。可以将每堆物品都看作一种新物品。

将 c[i]个物品拆分为 p+2 种新物品，每种新物品对应的重量和价值都如下图所示。



进行二进制拆分后，c[i]个物品被拆分为 p+2 种新物品，每种新物品只有一个，转化为 01 背包问题。

算法代码：

```
void multi_knapsack2(int n,int W){//二进制拆分
     for(i=1;i<=n;i++){
          if(c[i]*w[i]>=W){//转化为完全背包问题
               for(j=w[i];j<=maxn;j++)
                    dp[j]=min(dp[j],dp[j-w[i]]+v[i]);
          }
          else{
               for(int k=1;c[i]>0;k<<=1){//二进制拆分
                    int x=min(k,c[i]);
                    for(int j=W;j>=w[i]*x;j--)//转化为 01 背包问题
                         dp[j]=min(dp[j],dp[j-w[i]*x]+k*v[i]);
                    c[i]-=x;
               }
          }
     }
}
```

算法分析：本算法包含 3 层 for 循环，将 c[i]个物品拆分为 p+2 种新物品需要 O(logc_i)时间，**时间复杂度为 O(W∑logc_i)**，**空间复杂度为 O(W)**。

3. 数组优化

用 num[j]数组记录容量为 j 时放入了多少个第 i 种物品，以满足物品数量限制。

算法代码：

```
void multi_knapsack3(int n,int W){//数组优化
    for(int i=1;i<=n;i++){
       memset(num,0,sizeof(num));//统计数量
       for(int j=w[i];j<=W;j++){
          if(dp[j]<dp[j-w[i]]+v[i]&&num[j-w[i]]<c[i]){
             dp[j]=dp[j-w[i]]+v[i];
             num[j]=num[j-w[i]]+1;
          }
       }
    }
}
```

算法分析：本算法包含两层 for 循环，**时间复杂度为 O(nW)**，**空间复杂度为 O(W)。**

原理 4　分组背包

给定 n 组物品，第 i 组有 c_i 个物品，第 i 组的第 j 个物品有重量 w_{ij} 和价值 v_{ij}，背包容量为 W，在不超过背包容量的情况下每组最多选择一个物品，求解如何放置物品可使背包中物品的价值之和最大。

因为**每组最多选择一个物品，所以可以将每组都看作一个整体，这就类似于 01 背包问题。**

处理第 i 组物品时，前 i-1 组物品已处理完毕，只需考虑从第 i-1 阶段向第 i 阶段转移。

状态表示：c[i][j]表示将前 i 组物品放入容量为 j 的背包中可以获得的最大价值。

对第 i 组物品的处理状态如下。

• 若不放入第 i 组物品，则放入背包的价值不增加，问题转化为“将前 i−1 组物品放入容量为 j 的背包中可以获得的最大价值”，最大价值为 c[i-1][j]。

• 若放入第 i 组的第 k 个物品，则相当于从第 i-1 阶段向第 i 阶段转移，问题转化为“将前 i−1 组物品放入容量为 j-w[i][k]的背包中可以获得的最大价值”,此时获得的最大价值是 c[i-1][j-w[i][k]] ,再加上放入第 i 组的第 k 个物品获得的价值 v[i][k] ,总价值为 c[i-1][j-w[i][k]]+v[i][k]。

如果背包容量不足，不可以放入，则价值仍为前 i−1 组物品处理后的结果；如果背包容量允许，则考察放入或不放入哪种获得的价值更大。

状态转移方程：

$$c[i][j]=\begin{cases} c[i-1][j] & ,j<w_{ik} \\ \max\limits_{1\leqslant k\leqslant c_i}\{c[i-1][j],c[i-1][j-w[i][k]+v[i][k]\} & ,j\geqslant w_{ik} \end{cases}$$

和 01 背包一样，我们可以将**分组背包优化为一维数组，然后倒推，从而实现从第 i-1 阶段向第 i 阶段转移时每组最多选择一个物品。**

状态表示：dp[j]表示放入容量为 j 的背包时可以获得的最大价值。

状态转移方程：dp[j]=max(dp[j], dp[j-w[i][k]]+v[i][k])。

算法代码：

```
void group_knapsack1(int n,int W){//分组背包
    for(int i=1;i<=n;i++)
        for(int j=W;j>=0;j--)
            for(int k=1;k<=c[i];k++)//枚举组内的各个物品
                if(j>=w[i][k])
                    dp[j]=max(dp[j],dp[j-w[i][k]]+v[i][k]);
}
```

算法分析：本算法包含 3 层 for 循环，**时间复杂度为 O(W∑c_i)**，**空间复杂度为 O(W)。**

注意：枚举组内各个物品的个数 k 一定在最内层循环中，若将其放在 j 的外层，则变为多重背包的暴力拆分算法，因为这会出现组内物品被多次放入的情况，就变成了多重背包问题。

原理 5　混合背包

如果在一个问题中有些物品只可以取一次（01 背包），有些物品可以取无限次（完全背包），有些物品可以取的次数有一个上限（多重背包），则该种问题属于混合背包问题。

1. 01 背包+完全背包

01 背包问题和完全背包问题混合时，根据物品的类别选择倒推或正推求解即可，伪代码如下：

```
for i=1..N
    if 第 i 种物品属于 01 背包问题
        for v=V..0
            f[v]=max{f[v],f[v-c[i]]+w[i]};
    else if 第 i 种物品属于完全背包问题
        for v=0..V
            f[v]=max{f[v],f[v-c[i]]+w[i]};
```

2. 01 背包+完全背包+多重背包

若三种背包问题混合，则分别判断物品所属类别进行处理即可，伪代码如下：

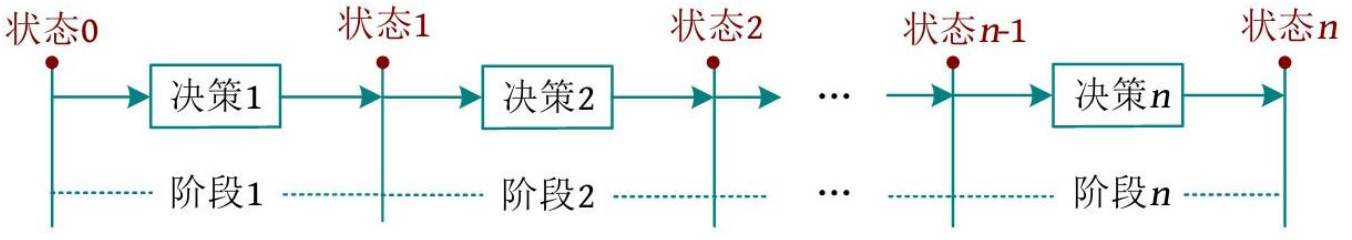
```
for i=1..N
    if 第 i 种物品属于 01 背包问题
        ZeroOnePack(c[i],w[i])
    else if 第 i 种物品属于完全背包问题
        CompletePack(c[i],w[i])
    else if 第 i 种物品属于多重背包问题
        MultiplePack(c[i],w[i],n[i])
```

混合背包问题并不是什么难题，但将它们组合起来可能会难倒不少人。只要基础扎实，领会基本背包问题的思想，就可以把困难的问题拆分成简单的问题来解决。

线性 DP

具有线性阶段划分的动态规划算法叫作线性动态规划（简称线性 DP）。

若状态包含多个维度，则每个维度都是线性划分的阶段，也属于线性 DP，如下图所示。

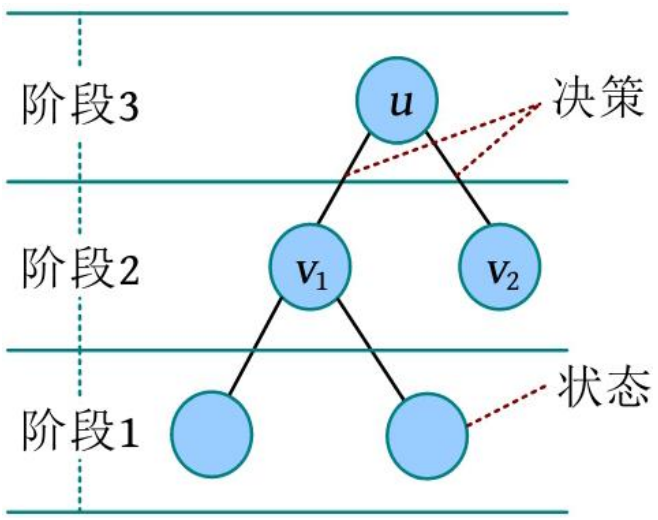


区间 DP

区间 DP 属于线性 DP 的一种，以区间长度作为 DP 的阶段，以区间的左右端点作为状态的维度。一个状态通常由被它包含且比它更小的区间状态转移而来。阶段（长度）、状态（左右端点）、决策三者按照由外到内的顺序构成三层循环。

树形 DP

在树形结构上实现的动态规划叫作树形 DP。动态规划自身是多阶段决策问题，而树形结构有明显的层次性，正好对应动态规划的多个阶段。树形 DP 的求解过程一般为自底向上，将子树从小到大作为 DP 的“阶段”，将节点编号作为 DP 状态的第 1 维，代表以该节点为根的子树。树形 DP 一般采用深度优先遍历，递归求解每棵子树，回溯时从子节点向上进行状态转移。在当前节点的所有子树都求解完毕后，才可以求解当前节点。



数位 DP

数位 DP 是与数位相关的一类计数类 DP，一般用于统计[l, r]区间满足特定条件的元素个数。数位指个位、十位、百位、千位等，数位 DP 就是在数位上进行动态规划。数位 DP 在实质上是一种有策略的穷举方式，在子问题求解完毕后将其结果记忆化就可以了。

状态压缩 DP

在动态规划状态设计中，若状态是一个集合，例如 S={1, 0, 1, 1, 0}，则表示第 1、2、4 个节点被选中（从右向左对应 0～4 号节点）。若集合的大小不超过 N，则集合中的每个元素都是小于 K 的正整数，可以把这个集合看作一个 N 位 K 进制数，以一个[0, K^N-1]的十进制整数作为 DP 状态。可以将 S={1, 0, 1, 1, 0}看作一个 5 位二进制数 10110，其对应的十进制数为 21。

这种将集合作为整数记录状态的一类算法叫作状态压缩 DP。在状态压缩 DP 中，状态的设计直接决定了程序的效率或者代码长短。我们需要积累经验，根据具体问题分析本质，才能更好地找出恰当的状态表示、状态转移方程和边界条件。

尽管用了一个十进制数据存储二进制状态，但因为操作系统是二进制的，所以在编译器中也可以采用位运算解决这个问题。在状态压缩 DP 中广泛应用位运算操作，常见的位运算如下。

（1）与：&，表示按位与运算，两个都是 1 才是 1。x&y 表示将两个十进制数 x、y 在二进制下按位与运算，然后返回其十进制下的值。例如，3(11)&2(10)=2(10)。

（2）或：|，表示按位或运算，有一个是 1 就是 1。x|y 表示将两个十进制数 x、y 在二进制下按位或运算，然后返回其十进制下的值。例如 3(11)|2(10)=3(11)。

（3）异或：^，表示按位异或运算，两个不相同时才是 1。x^y 表示将两个十进制数 x、y 在二进制下按位异或运算，然后返回其十进制下的值。例如 3(11)^2(10)=1(01)。

（4）左移：<<，表示左移操作。x<<2 表示将 x 在二进制下的每一位都向左移动两位，最右边用 0 填充，相当于让 x 乘以 4。每向左移动一位，都相当于乘以 2。

（5）右移：>>，表示右移操作。x>>1 表示将 x 在二进制下的每一位都向右移动一位，最右边用符号位填充，低位舍弃，相当于对 x/2 向 0 取整，3/2=1，(-3)/2=-1。

1. 旅行商问题

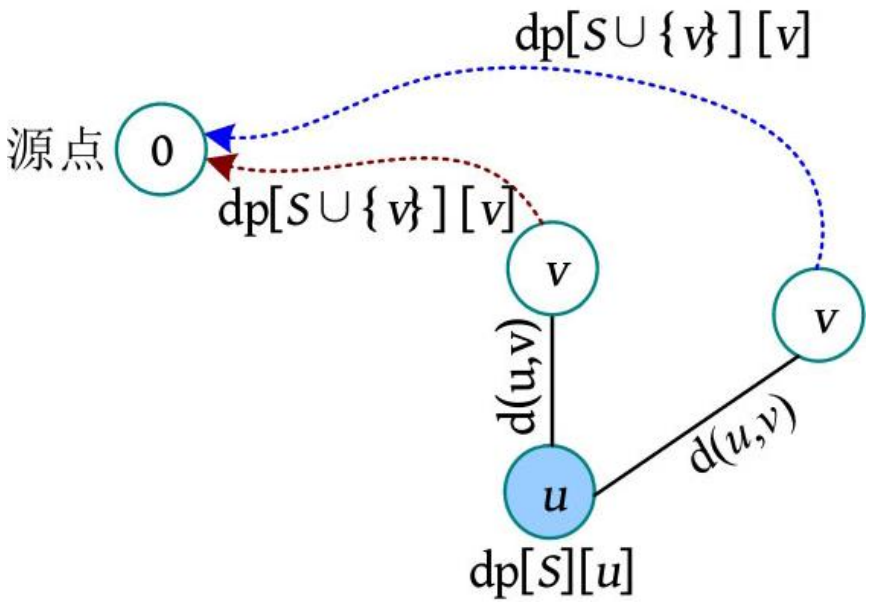
著名的旅行商问题（Traveling Salesman Problem，TSP）指一个旅行商从一个城市出发，经过每个城市一次且只有一次回到原来的地方，要求经过的距离最短。TSP 问题是一个 NP 难题，目前没有多项式时间的高效算法。若采用搜索+剪枝，则该算法的时间复杂度为 O(n!)，数据量大时，这种方法无法解决，可以尝试采用动态规划解决。

假设已访问的节点集合为 S（将源点 0 当作未被访问的节点，因为从 0 出发，所以要回到 0），当前所在的节点为 u。

状态表示：dp[S][u]表示已经访问的节点集合为 S，从 u 出发走完所有剩余节点回到源点的最短距离。

状态转移方程：若当前 u 的邻接点 v 未访问，则 dp[S][u]由两部分组成，第 1 部分是 u 到 v 的边值，第 2 部分是从 v 出发走完所有剩余节点再回到源点的最短距离。访问完 v 之后，已访问的节点集合变为 S∪{v}，若 u 有多个未被访问的邻接点 v，则取最小值。

$dp[S][u]=\min\{dp[S\cup\{v\}][v]+d[u][v]|v\notin S\}$，如下图所示：



临界条件：dp[(1<<n)-1][0]=0，表示若所有节点都已被访问，则此时已经没有剩余节点，从 0 节点出发走完所有剩余节点回到源点的最短距离为 0。

旅行商问题可以采用递推和记忆化递归两种方法求解。

1. 递推

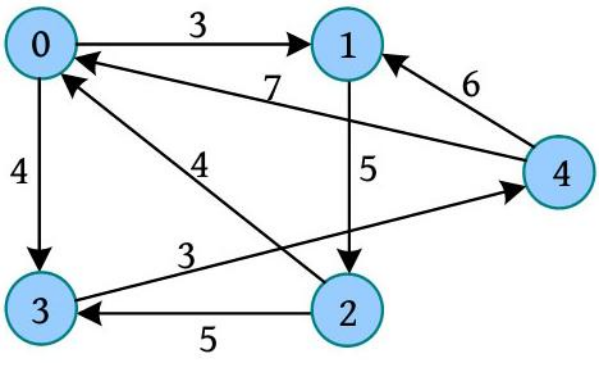
以递推方法求解旅行商问题指从临界条件开始枚举每一种状态 S，若 u=0 或者 u 已被访问，而 v 未被访问，则判断是否可以借助 v 更新 dp[S][u]，直到求解出答案 dp[0][0]。时间复杂度为 $O(n^2 2^n)$。

算法代码：

```
void Traveling(){//计算 dp[S][u]
     dp[(1<<n)-1][0]=0;//注意：1<<n 一定要加括号
     for(int S=(1<<n)-2;S>=0;S--)
       for(int u=0;u<n;u++)
          for(int v=0;v<n;v++){
             if((u!=0&&!(S>>u&1))||g[u][v]==INF) continue;//可以加约束条件，不加太多状态
             if(!(S>>v&1)&&dp[S][u]>dp[S|1<<v][v]+g[u][v]){
               dp[S][u]=dp[S|1<<v][v]+g[u][v];
               path[S][u]=v;//记录后继节点
             }
          }
}
```

图解：

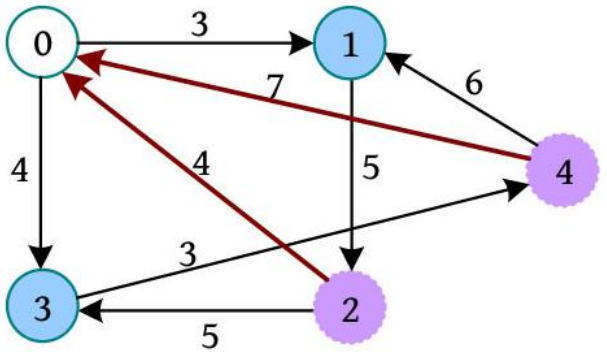
一个有向图如下图所示，求从 0 节点出发经过每个节点一次且只有一次回到 0 节点的最短路径。



（1）初始条件，dp[(1<<n)-1][0]=0，即 dp[31][0]=0，S 对应的二进制为 11111，该二进制从低位到高位分别表示 0～4 号节点是否已被访问，0 表示未被访问，1 表示已被访问。初始时所有节点都已被访问，当前位置在 0 节点。

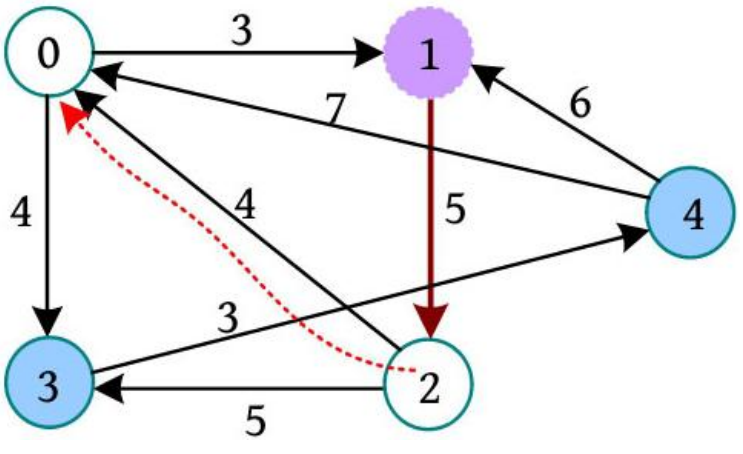
（2）枚举每一种状态，已被访问的节点集合 S 从 11110 枚举到 00000。当 S=11110 时，0 节点未被访问，更新以下结果。

- 从 2 节点出发到达 0 节点，最短距离为 4，dp[11110][2]=dp[11111][0]+4=4。
- 从 4 节点出发到达 0 节点，最短距离为 7，dp[11110][4]=dp[11111][0]+7=7。



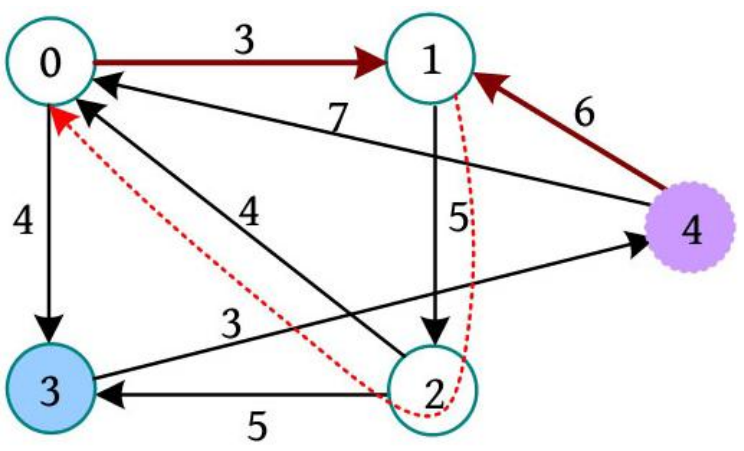
（3）当 S=11010 时，0、2 节点未被访问，更新以下结果。

- 从 1 节点出发经过 2 节点到达 0 节点，最短距离为 9，dp[11010][1]=dp[11110][2]+5=9。



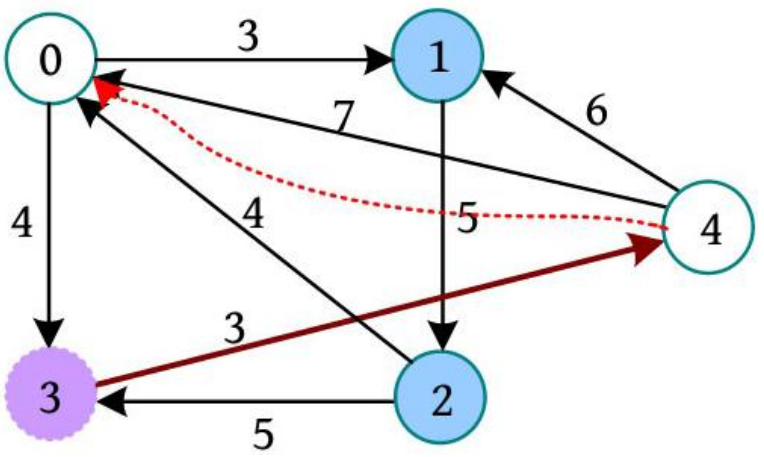
（4）当 S=11000 时，0、1、2 节点未被访问，更新以下结果。

- 从 0 节点出发经过 1、2 节点到达 0 节点，最短距离为 12，dp[11000][0]=dp[11010][1]+3=12。
- 从 4 节点出发经过 1、2 节点到达 0 节点，最短距离为 15，dp[11000][4]=dp[11010][1]+6=15。



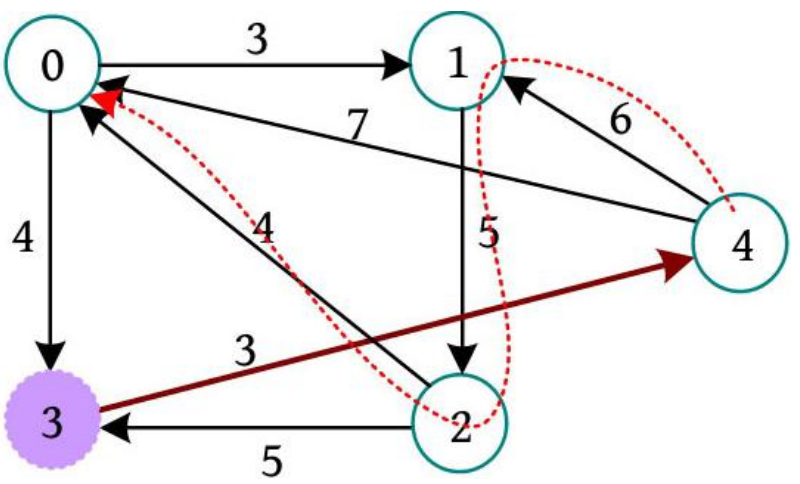
（5）当 S=01110 时，0、4 节点未被访问，更新以下结果。

- 从 3 节点出发经过 4 节点到达 0 节点，最短距离为 10，dp[01110][3]=dp[11110][4]+3=10。



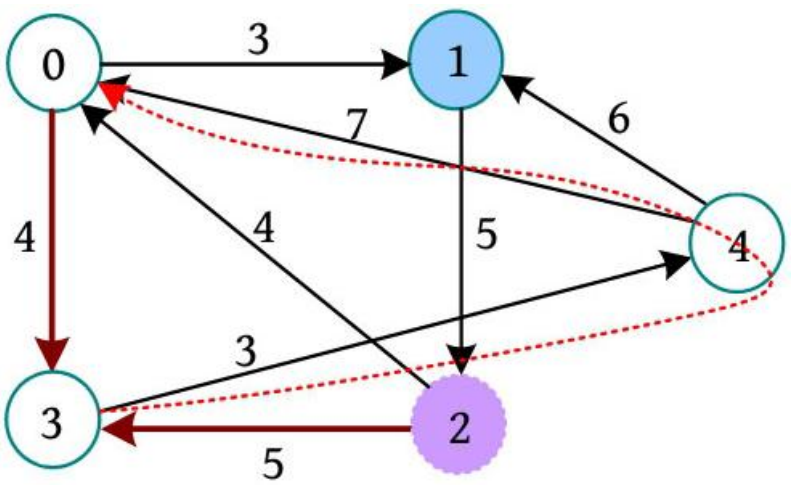
（6）当 S=01000 时，0、1、2、4 节点未被访问，更新以下结果。

- 从 3 节点出发经过 4、1、2 节点到达 0 节点，最短距离为 18，dp[01000][3]=dp[11000][4]+3=18。



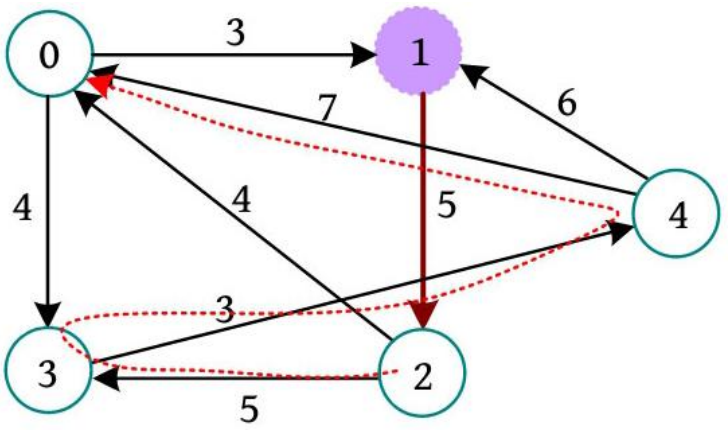
（7）当 S=00110 时，0、3、4 节点未被访问，更新以下结果。

- 从 0 节点出发经过 3、4 节点到达 0 节点，最短距离为 14，dp[00110][0]=dp[01110][3]+4=14。
- 从 2 节点出发经过 3、4 节点到达 0 节点，最短距离为 15，dp[00110][2]=dp[01110][3]+5=15。



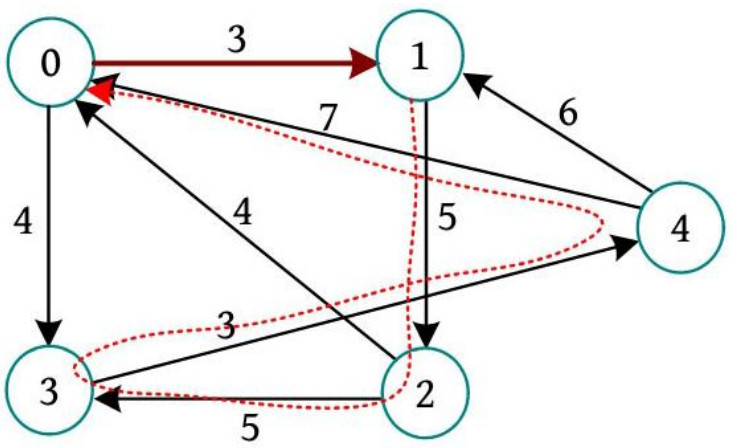
（8）当 S=00010 时，0、2、3、4 节点未被访问，所以更新。

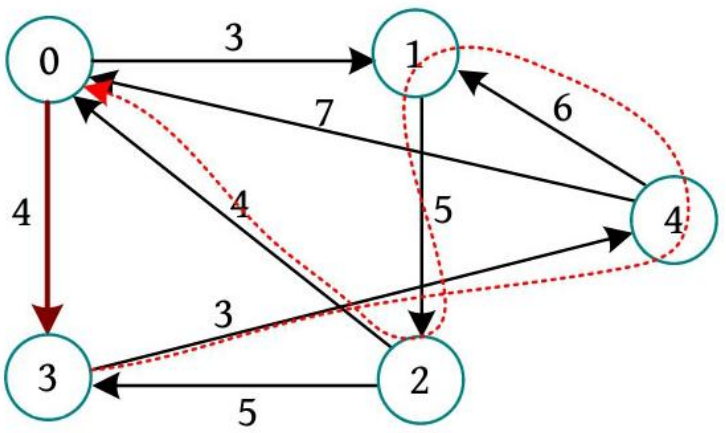
- 从 1 节点出发经过 2、3、4 节点到达 0 节点，最短距离为 20，dp[00010][1]=dp[00110][2]+5=20。



（9）当 S=00000 时，0、1、2、3、4 节点未被访问，更新以下结果。

- 从 0 节点出发经过 1、2、3、4 节点到达 0 节点，最短距离为 23，dp[00000][0]=dp[00010][1]+3=23；
- 从 0 节点出发经过 3、4、1、2 节点到达 0 节点，最短距离为 22，dp[00000][0]=dp[01000][3]+4=22。





（10）输出最短路径：0→3→4→1→2→0；最短路径长度为 22。

2. 记忆化递归

采用记忆化递归的方法求解旅行商问题时，首先确定递归结束的条件和递归式，然后记忆化递归即可。

（1）递归结束的条件为上面递推方法的初始条件：当 S=(1<<n)-1 且 u=0 时，返回 dp[S][0]=0。

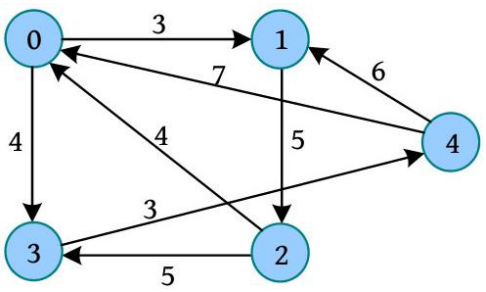
（2）递归式：Traveling(S, u)=min(ans, Traveling(S|1<<v, v)+g[u][v])。

（3）记忆化递归：将 dp[][]数组初始化为-1，若已赋值，则直接返回。

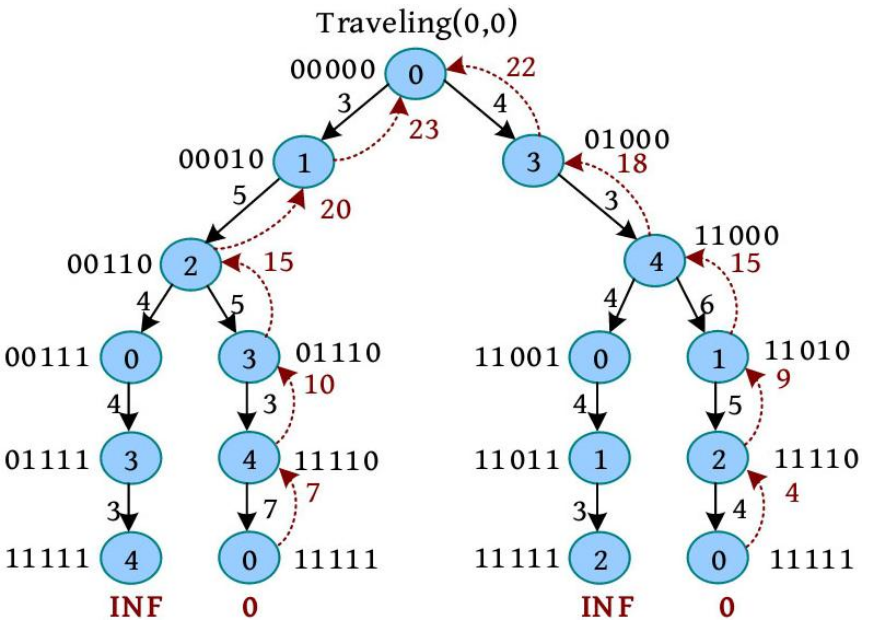
算法代码：

```
int Traveling(int S,int u) {//计算dp[S][u]，记忆化递归
    if(dp[S][u]>=0)//记忆化递归
        return dp[S][u];
    if(S==(1<<n)-1&&u==0) //递归结束条件
        return dp[S][u]=0;
    int ans=INF;
    for(int v=0;v<n;v++)
        if(!(S>>v&1)&&g[u][v]!=INF)
            ans=min(ans,Traveling(S|1<<v,v)+g[u][v]);
    return dp[S][u]=ans;
}
```

示例：一个有向图如下图所示，求从 0 节点出发经过每个节点一次且只有一次回到 0 节点的最短路径。



（1）从 Traveling(0, 0)开始递归，达到结束条件时返回。递归树如下图所示：



（2）状态转移情况：

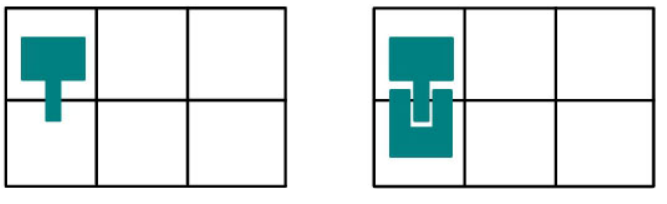
- 从 4 节点出发到达 0 节点，最短距离为 7，dp[11110][4]=dp[11111][0]+7=7；
- 从 3 节点出发经过 4 节点到达 0 节点，最短距离为 10，dp[01110][3]=dp[11110][4]+3=10；
- 从 2 节点出发经过 3、4 节点到达 0 节点，最短距离为 15，dp[00110][2]=dp[01110][3]+5=15；
- 从 1 节点出发经过 2、3、4 节点到达 0 节点，最短距离为 20，dp[00010][1]=dp[00110][2]+5=20；
- 从 0 节点出发经过 1、2、3、4 节点到达 0 节点，最短距离为 23，dp[00000][0]=dp[00010][1]+3=23；
- 从 2 节点出发到达 0 节点，最短距离为 4，dp[11110][2]=dp[11111][0]+4=4；
- 从 1 节点出发经过 2 节点到达 0 节点，最短距离为 9，dp[11010][1]=dp[11110][2]+5=9；
- 从 4 节点出发经过 1、2 节点到达 0 节点，最短距离为 15，dp[11000][4]=dp[11010][1]+6=15；
- 从 3 节点出发经过 4、1、2 节点到达 0 节点，最短距离为 18，dp[01000][3]=dp[11000][4]+3=18；
- 从 0 节点出发经过 3、4、1、2 节点到达 0 节点，最短距离为 22，dp[00000][0]=dp[01000][3]+4=22。

最短路径：0→3→4→1→2→0；最短路径长度为 22。

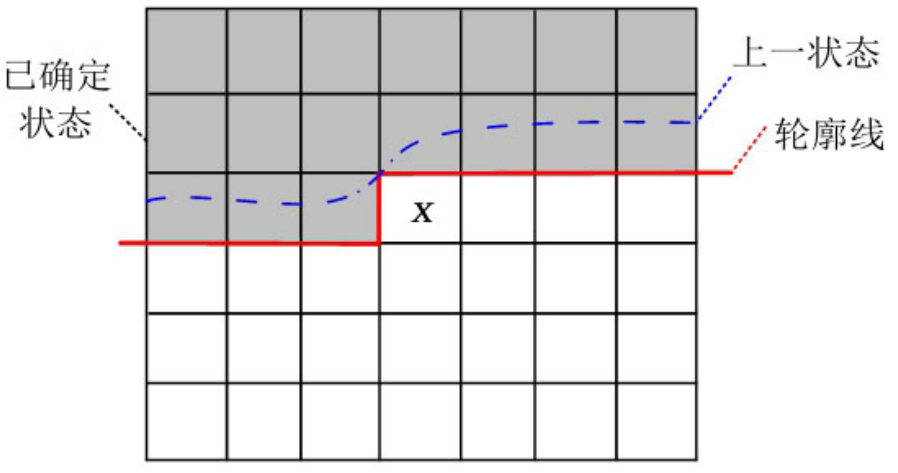
插头 DP

插头 DP 是一类特殊的状态压缩 DP，又叫作**轮廓线 DP**，通常**用于解决二维空间的状态压缩问题**，且每个位置的取值都只与临近的几个位置有关，**适用于超小数据范围、网格图、连通性等问题。**

插头：一个格子通过某些方向与另一个格子相连，这些连接的位置叫作“插头”。可以这样理解，网格图上的每一个格子都是一块拼图，两块拼图的接口就是插头。



轮廓线：若从左上角开始处理，则灰色表示已确定状态，白色表示未确定状态，**已确定状态和未确定状态之间的分界线叫作“轮廓线”。**若按行从左向右逐格求解，则 x 位置是当前待确定状态的格子。x 的处理方案只与上一状态有关。



动态规划优化

动态规划是解决多阶段决策优化问题的一种方法。动态规划高效的**关键在于减少了“冗余”**，即减少了不必要或重复计算的部分。动态规划在自底向上的求解过程中，记录了子问题的求解结果，**避免了重复求解子问题**，提高了效率。

动态规划的时间复杂度计算抽象公式如下：

时间复杂度=状态总数×每个状态的决策数×每次状态转移所需的时间

可以从三个方面进行动态规划**优化：状态总数、每个状态的决策数和每次状态转移所需的时间。**

（1）减少状态总数的基本策略包括修改状态表示（状态压缩、倍增优化）、选择适当的 DP 方向（双向 DP）等。

（2）减少状态的决策数，最常见的是利用最优决策单调性进行四边不等式优化、剪枝优化。

（3）减少状态转移所需的时间，可采用预处理、合适的计算方法和数据结构优化。

原理 1　倍增优化

倍增，顾名思义，指成倍增加。若问题的状态空间特别大，则一步一步递推的算法复杂度太高，可以利用倍增思想，只考察 2 的整数次幂位置，快速缩小求解范围，直到找到所需的解。

原理 2　数据结构优化

可以利用数据结构（二分、哈希表、线段树、状态数组）优化，解决查找、区间最值、前缀和等问题。

原理 3　单调队列优化

单调队列是一种特殊的队列，可以在队列两端进行删除操作，并始终维护队列的单调性。

单调队列有两种单调性：元素的值严格单调（递减或递增），**元素的下标严格单调**（递减或递增）。单调队列只可以从队尾入队，但可以从队尾或队首出队。当状态转移为以下两种情况时，考虑优化。

状态转移方程形如 dp[i]=min{dp[j]+f[j]}，0≤j<i。在这种情况下，下界不变，i 增加 1 时，j 的上界也增加 1，决策的候选集合只扩大、不缩小，仅用一个变量维护最值。用一个变量 val 维护(0, i)区间中 dp[j]+f[j]的最小值即可。

状态转移方程形如 dp[i]=min{dp[j]+f[j]}，i-a≤j≤i-b。在这种情况下，i 增加 1 时，j 的上界、下届同时增加 1，在一个新的决策加入候选集时，需要把过时（前面一个超出区间的）的决策从候选集合中剔除。例如，当前 j 的范围为[2, 4]，当 i 增加 1 时，j 的范围变为[3, 5]，此时 2 已过时（不属于[3, 5]区间）。

当决策的取值范围的上、下界均单调变化时，每个决策都在候选集合中插入或删除最多一次，可以用一个单调队列维护[i-a, i-b]区间 dp[j]+f[j]的最小值。

原理 4　斜率优化

动态规划算法的状态转移方程为 dp[i]=min(dp[j]+f(i, j)), L(i)≤j≤R(i)。

若 f(i, j)仅与 i、j 中的一个有关，则可以采用单调队列优化；若 f(i, j)与 i、j 均有关，则可以采用斜率优化。

原理 5　四边不等式优化

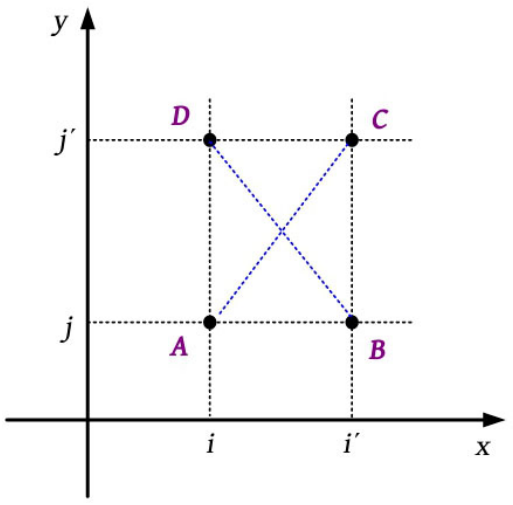
这里以猴子派对问题（HDU3506）为例，讲解四边不等式优化。

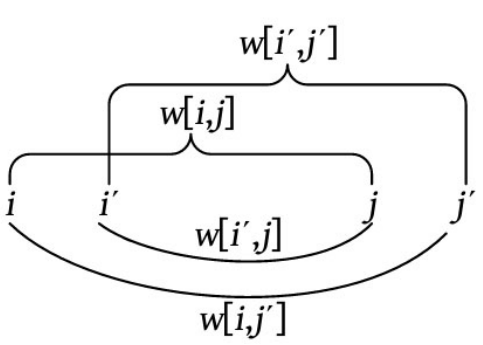
$$m[i][j]=\begin{cases}0, & i=j\\ \min\limits_{i\le k\le j}(m[i][k]+m[k+1][j]+w(i,j)), & i<j\end{cases}$$

s[i][j]表示取得最优解 Min[i][j]的最优策略位置。

四边不等式：当函数 w[i，j]满足 w[i, j]+w[i', j']≤w[i', j]+w[i, j'], i≤i'≤j≤j'时，称 w 满足四边形不等式。

在四边不等式的坐标表示中，A+C≤B+D。在四边不等式的区间表示中，w[i, j]+w[i', j']≤w[i', j]+w[i, j']。





区间包含关系单调：当函数 w[i, j]满足 w[i', j]≤w[i, j'], i≤i'≤j≤j'时，称 w 关于区间包含关系单调。

定理 1：若函数 w[i, j]满足四边不等式和区间包含关系单调，则 m[i, j]也满足四边不等式。

定理 2：若函数 m[i, j]满足四边不等式，则最优决策 s[i, j]有单调性。m[i, j]满足四边不等式是使用四边不等式优化的必要条件。

下面证明 3 个问题：

（1）w[i, j]满足四边不等式和区间单调性；

（2）m[i, j]也满足四边不等式；

（3）s[i, j]有单调性。

证明 1：w[i, j]满足四边不等式。

在石子归并问题中，因为 $w[i,j]=\sum_{l=i}^{j}a[l]$ ，所以 w[i, j]+w[i', j']=w[i', j]+w[i, j']，则 w[i, j]满足四边形不等式，同时由于 a[i]≥0，可知 w[i, j]满足区间包含关系单调性。

证明 2：m[i, j]满足四边不等式。

对满足四边形不等式的单调函数 w[i, j]，可推知由递归式定义的函数 m[i, j]也满足四边形不等式，即 m[i, j]+m[i', j']≤m[i', j]+m[i, j']，i≤i'≤j≤j'。

对四边形不等式中“长度”l=j'−i 进行归纳：当 i=i'或 j=j'时，不等式显然成立。由此可知，当 l≤1 时，函数 m[i, j]满足四边不等式。

下面分两种情形进行讲解。

情形 1：i<i'=j<j'

在这种情形下，四边形不等式可简化为**反三角不等式：m[i, j]+m[j, j']≤m[i, j']。**

设 k=min{p|m[i, j']=m[i, p]+m[p+1, j']+w[i, j']}，再分两种情形 k≤j 或 k>j。下面只讨论 k≤j 的情况，k>j 同理。

k≤j：

$$\begin{aligned} m[i,j]+m[j,j'] &\leqslant w[i,j]+m[i,k]+m[k+1,j]+m[j,j'] \\ &\leqslant w[i,j']+m[i,k]+m[k+1,j]+m[j,j'] \\ &\leqslant w[i,j']+m[i,k]+m[k+1,j'] \\ &= m[i,j'] \end{aligned}$$

情形 2：i<i'<j<j'

仍需再分两种情形讨论，即 z≤y 或 z>y。下面只讨论 z≤y 的情况，z>y 同理。

由 i<z≤y≤j，有：

$$\begin{aligned} m[i,j]+m[i',j'] &\leqslant w[i,j]+m[i,z]+m[z+1,j]+w[i',j']+m[i',y]+m[y+1,j'] \\ &\leqslant w[i,j']+w[i',j]+m[i',y]+m[i,z]+m[z+1,j]+m[y+1,j'] \\ &\leqslant w[i,j']+w[i',j]+m[i',y]+m[i,z]+m[z+1,j']+m[y+1,j] \\ &= m[i,j']+m[i',j] \end{aligned}$$

综上所述，m[i, j]满足四边形不等式。

证明 3：s[i, j]有单调性。

令 s[i, j]= min{k | m[i, j]=m[i, k]+m[k+1, j]+w[i, j]}，由函数 m[i, j]满足四边形不等式，可以推出函数 s[i, j]的单调性，即

s[i, j-1]≤s[i, j]≤s[i+1, j]，i≤j

当 i=j 时，单调性显然成立。所以下面只讨论 i<j 的情形。由于对称，只要证明 s[i, j]≤s[i+1, j]，便可证明 s[i, j-1]≤s[i, j]。

令 mk[i, j]=m[i, k]+m[k+1, j]+w[i, j]。要证明 s[i, j]≤s[i+1, j]，只要证明对所有 i<k'≤k≤j 且 mk[i, j]≤mk'[i, j]，mk [i+1, j]≤mk'[i+1, j] 成立即可。

因为 m[i, j]满足四边形不等式，所以 m[i, k']+m[i+1, k]≤m[i, k]+m[i+1, k']

移项可得：m[i+1, k']-m[i+1, k]≥m[i, k']-m[i, k]

根据 k 的最优性：m[i, k']+m[k'+1, j]≥m[i, k]+m[k+1, j]

所以

$$\begin{aligned} & m_{k'}[i+1,j]-m_k[i+1,j] \\ &= m[i+1,k']+m[k'+1,j]+w[i+1,j]-(m[i+1,k]+m[k+1,j]+w[i+1,j]) \\ &= m[i+1,k']-m[i+1,k]+m[k'+1,j]-m[k+1,j] \\ &\geqslant m[i,k']-m[i,k]+m[k'+1,j]-m[k+1,j] \\ &= m[i,k']+m[k'+1,j]-(m[i,k]+m[k+1,j]) \\ &\geqslant 0 \end{aligned}$$

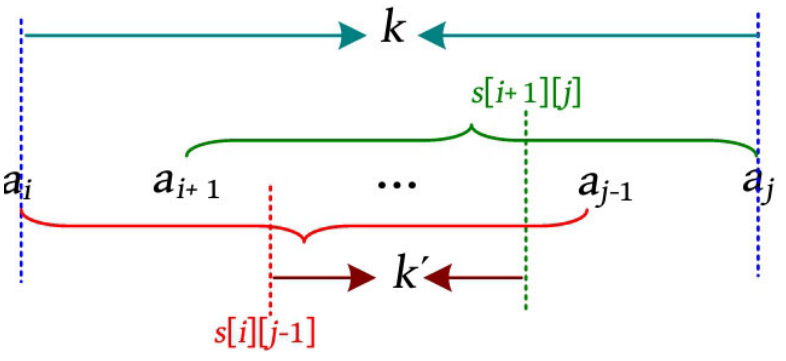
说明对于 m[i+1, j]，k 比 k'更优。

综上所述，当 m 满足四边形不等式且函数 s[i, j]有单调性时，可以采用四边不等式优化。

于是，利用 s[i, j]的单调性，得到优化的状态转移方程如下：

$$m[i][j]=\begin{cases} 0, & i=j \\ \min\limits_{s[i][j-1]\leqslant k\leqslant s[i+1][j]}(m[i][k]+m[k+1][j]+w(i,j)), & i<j \end{cases}$$

普通枚举算法的时间复杂度为 O(n^3)，利用四边形不等式优化，限制 k=i, …, j-1 的取值范围为 s(i, j-1) ~ s(i+1, j)，s[i][j]表示取得最优解 dp[i][j]的位置，达到优化效果 O(n^2)。



经过优化，算法时间复杂度可以减少至 O(n^2)。 $\sum_{d=2}^{n}\sum_{i=1}^{n-d+1}(s[i+1][j]-s[i][j-1]+1)$ ，因为公式中 j=i+d-1，所以：

$$\sum_{d=2}^{n}\sum_{i=1}^{n-d+1}(s[i+1][i+d-1]-s[i][i+d-2]+1)$$

$$=\sum_{v=2}^{n}\left\{\begin{array}{l}(s[2][d]-s[1][d-1]+1\\+s[3][d+1]-s[2][d]+1\\+s[4][d+2]-s[3][d+1]+1\\+\cdots\\+s[n-d+2][n]-s[n-d+1][n-1]+1)\end{array}\right\}$$

$$=\sum_{d=2}^{n}(s[n-d+2][n]-s[1][d-1]+n-d+1)$$

$$\leqslant\sum_{d=2}^{n}(n-1+n-d+1)$$

$$=\sum_{d=2}^{n}(2n-d)$$

$$\approx O(n^2)$$

上述方法利用了四边形不等式推出最优决策的单调性，减少了每个状态转移的状态数，降低了算法的时间复杂度。

上述方法是有普遍性的。状态转移方程与上述递归公式类似，且 w[i, j]满足四边形不等式的动态规划问题都可以采用相同的优化方法解决，例如最优二叉排序树等。