2.3 记录结果再利用的“动态规划”

➡动态规划（DP: Dynamic Programming）是算法的设计方法之一，在程序设计竞赛中经常被选作题材。在此，我们考察一些经典的DP问题，来看看DP究竟是何种类型的算法。

2.3.1 记忆化搜索与动态规划

01背包问题

有 n 个重量和价值分别为 w_i, v_i 的物品。从这些物品中挑选出总重量不超过 W 的物品，求所有挑选方案中价值总和的最大值。

⚠限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant w_i, v_i \leqslant 100$
- $1 \leqslant W \leqslant 10000$

样例

输入

```
n = 4
(w, v) = {(2, 3), (1, 2), (3, 4), (2, 2)}
W = 5
```

输出

```
7（选择第0、1、3号物品）
```

这是被称为背包问题的一个著名问题。这个问题要如何求解比较好呢？不妨先用最朴素的方法，针对每个物品是否放入背包进行搜索试试看。这个想法实现后的结果请参见如下代码：

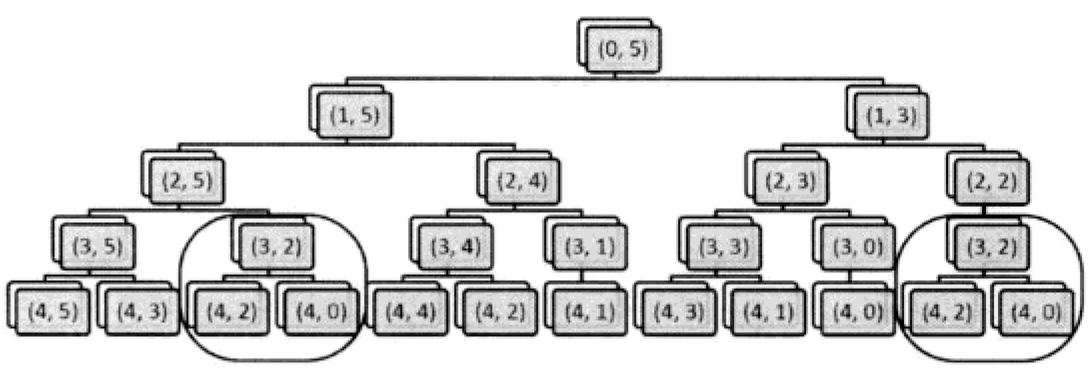
```
// 输入
int n, W;
int w[MAX_N], v[MAX_N];

// 从第i个物品开始挑选总重小于j的部分
```

```
int rec(int i, int j) {
  int res;
  if (i == n) {
    // 已经没有剩余物品了
    res = 0;
  } else if (j < w[i]) {
    // 无法挑选这个物品
    res = rec(i + 1, j);
  } else {
    // 挑选和不挑选的两种情况都尝试一下
    res = max(rec(i + 1, j), rec(i + 1, j - w[i]) + v[i]);
  }
  return res;
}

void solve() {
  printf("%d\n", rec(0, W));
}
```

只不过，这种方法的搜索深度是n，而且每一层的搜索都需要两次分支，最坏就需要$O(2^n)$的时间，当n比较大时就没办法解了。所以要怎么办才好呢？为了优化之前的算法，我们看一下针对样例输入的情形下rec递归调用的情况。



递归地调用

如图所示，rec以(3,2)为参数调用了两次。如果参数相同，返回的结果也应该相同，于是第二次调用时已经知道了结果却白白浪费了计算时间。让我们在这里把第一次计算时的结果记录下来，省略掉第二次以后的重复计算试试看。

```
int dp[MAX_N + 1][MAX_W + 1]; // 记忆化数组

int rec(int i, int j) {
  if (dp[i][j] >= 0) {
    // 已经计算过的话直接使用之前的结果
    return dp[i][j];
  }
  int res;
  if (i == n) {
    res = 0;
  } else if (j < w[i]) {
```

```
    res = rec(i + 1, j);
  } else {
    res = max(rec(i + 1, j), rec(i + 1, j - w[i]) + v[i]);
  }
  // 将结果记录在数组中
  return dp[i][j] = res;
}

void solve() {
  // 用-1表示尚未计算过，初始化整个数组
  memset(dp, -1, sizeof(dp));
  printf("%d\n", rec(0, W));
}
```

这微小的改进能降低多少复杂度呢？对于同样的参数，只会在第一次被调用到时执行递归部分，第二次之后都会直接返回。参数的组合不过nW种，而函数内只调用2次递归，所以只需要$O(nW)$的复杂度就能解决这个问题。只需略微改良，可解的问题的规模就可以大幅提高。这种方法一般被称为记忆化搜索。

专栏　使用memset进行初始化

虽然 memset 按照 1 字节为单位对内存进行填充，−1 的每一位二进制位都是 1，所以可以像 0 一样用 memset 进行初始化。通过使用 memset 可以快速地对高维数组等进行初始化，但是需要注意无法初始化成 1 之类的数值。

专栏　穷竭搜索的写法

如果对记忆化搜索还不是很熟练的话，可能会把前面的搜索写成下面这样

```
// 目前选择的物品价值总和是sum，从第i个物品之后的物品中挑选重量总和小于j的物品
int rec(int i, int j, int sum) {
  int res;
  if (i == n) {
    // 已经没有剩余物品了
    res = sum;
  } else if (j < w[i]) {
    // 无法挑选这个物品
    res = rec(i + 1, j, sum);
  } else {
    // 挑选和不挑选的两种情况都尝试一下
    res = max(rec(i + 1, j, sum), rec(i + 1, j - w[i], sum + v[i]));
  }
  return res;
}
```

在需要剪枝的情况下，可能会像这样把各种参数都写在函数上，但是在这种情况下会让记忆化搜索难以实现，需要注意。

接下来，我们来仔细研究一下前面的算法利用到的这个记忆化数组。记$dp[i][j]$为根据rec的定义，从第i个物品开始挑选总重小于j时，总价值的最大值。于是我们有如下递推式

$$dp[n][j] = 0$$

$$dp[i][j] = \begin{cases} dp[i+1][j] & （j < w[i]） \\ \max(dp[i+1][j], dp[i+1][j-w[i]] + v[i]) & （其他） \end{cases}$$

如上所示，不用写递归函数，直接利用递推式将各项的值计算出来，简单地用二重循环也可以解决这一问题。

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

dp[3][3]=max(dp[4][3], dp[4][1]+2)

i\j	0	1	2	3	4	5
0	0	2	3	5	6	7
1	0	2	2	4	6	6
2	0	0	2	4	4	6
3	0	0	2	2	2	2
4	0	0	0	0	0	0

dp[0][5]=max(dp[1][5], dp[1][3]+3)

```
int dp[MAX_N + 1][MAX_W + 1]; / DP数组

void solve() {
  for (int i = n - 1; i >= 0; i--) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[i][j] = dp[i + 1][j];
      } else {
        dp[i][j] = max(dp[i + 1][j], dp[i + 1][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[0][W]);
}
```

这个算法的复杂度与前面相同，也是$O(nW)$，但是简洁了很多。以这种方式一步步按顺序求出问题的解的方法被称作动态规划法，也就是常说的DP。解决问题时既可以按照如上方法从记忆化搜索出发推导出递推式，熟练后也可以直接得出递推式。

专栏　注意不要忘记初始化

因为全局数组的内容会被初始化为0，所以前面的源代码中并没有显式地将初项=0进行赋值，不过当一次运行要处理多组输入数据时，必须要进行初始化，这点一定要注意。

专栏 各种各样的DP

刚刚讲到 DP 中关于 i 的循环是逆向进行的。反之，如果按照如下的方式定义递推关系的话，关于 i 的循环就能正向进行。

$$dp[i+1][j] := \text{从前 } i \text{ 个物品中选出总重量不超过 } j \text{ 的物品时总价值的最大值}$$

$$dp[0][j] = 0$$

$$dp[i+1][j] = \begin{cases} dp[i][j] & (j < w[i]) \\ \max(dp[i][j], dp[i][j-w[i]]+v[i]) & (\text{其他}) \end{cases}$$

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

dp[3][4]=max(dp[2][4], dp[2][1]+4)

```
void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[i + 1][j] = dp[i][j];
      } else {
        dp[i + 1][j] = max(dp[i][j], dp[i][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[n][W]);
}
```

此外，除了运用递推方式逐项求解之外，还可以把状态转移想象成从“前i个物品中选取总重不超过j时的状态”向“前i+1 个物品中选取总重不超过j”和“前i+1 个物品中选取总重不超过j+$w[i]$时的状态”的转移，于是可以实现成如下形式：

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

dp[3][1]=max(dp[3][1],dp[2][1]), dp[3][4]=max(dp[3][4],dp[2][1]+4)

```
void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      dp[i + 1][j] = max(dp[i + 1][j], dp[i][j]);
```

```
        if (j + w[i] <= W) {
          dp[i + 1][j + w[i]] = max(dp[i + 1][j + w[i]], dp[i][j] + v[i]);
        }
      }
    }
    printf("%d\n", dp[n][W]);
  }
```

如果像上述所示，把问题写成从当前状态迁移成下一状态的形式的话，需要注意初项之外也需要初始化（这个问题中，因为价值的总和至少是0，所以初值设为0就可以了，不过根据问题也有可能需要初始化成无穷大）。同一个问题可能会有各种各样的解决方法，诸如搜索的记忆化或者利用递推关系的DP，再或者从状态转移考虑的DP等，不妨先把自己最喜欢的形式掌握熟练。但是，有些问题不用记忆化搜索也许很难求解，反之，不用DP复杂度就会变大的情况也会有，所以最好要掌握各种形式的DP。

最长公共子序列问题

给定两个字符串 $s_1s_2\cdots s_n$ 和 $t_1t_2\cdots t_n$。求出这两个字符串最长的公共子序列的长度。字符串 $s_1s_2\cdots s_n$ 的子序列指可以表示为 $s_{i_1}s_{i_2}\cdots s_{i_m}$ $(i_1<i_2<\cdots<i_m)$的序列。

限制条件

- $1 \leqslant n, m \leqslant 1000$

样例

输入

```
n = 4
m = 4
s = "abcd"
t = "becd"
```

输出

```
3 ("bcd")
```

这个问题是被称为最长公共子序列问题（LCS，Longest Common Subsequence）的著名问题。不妨用如下方式定义试试看：

$dp[i][j]$:=$s_1\cdots s_i$和$t_1\cdots t_j$对应的LCS的长度

由此，$s_1\cdots s_{i+1}$和$t_1\cdots t_{j+1}$对应的公共子列可能是

当$s_{i+1}=t_{j+1}$时，在$s_1\cdots s_i$和$t_1\cdots t_j$的公共子列末尾追加上s_{i+1}

$s_1\cdots s_i$和$t_1\cdots t_{j+1}$的公共子列

$s_1\cdots s_{i+1}$和$t_1\cdots t_j$的公共子列

三者中的某一个，所以就有如下的递推关系成立①

$$dp[i+1][j+1]=\begin{cases}\max(dp[i][j]+1,dp[i][j+1],dp[i+1][j]) & (s_{i+1}=t_{j+1})\\ \max(dp[i][j+1],dp[i+1][j]) & (\text{其他})\end{cases}$$

这个递推式可用$O(nm)$计算出来，$dp[n][m]$就是LCS的长度。

j\i	0	1(b)	2(e)	3(c)	4(d)
0	0	0	0	0	0
1(a)	0	0	0	0	0
2(b)	0	1	1	1	1
3(c)	0	1	1	2	2
4(d)	0	1	1	2	3

DP数组

```
// 输入
int n, m;
char s[MAX_N], t[MAX_M];

int dp[MAX_N + 1][MAX_M + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (s[i] == t[j]) {
        dp[i + 1][j + 1] = dp[i][j] + 1;
      } else {
        dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j]);
      }
    }
  }
  printf("%d\n", dp[n][m]);
}
```

2.3.2 进一步探讨递推关系

完全背包问题

有 n 种重量和价值分别为 w_i, v_i 的物品。从这些物品中挑选总重量不超过 W 的物品，求出挑选物品价值总和的最大值。在这里，每种物品可以挑选任意多件。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant w_i, v_i \leqslant 100$
- $1 \leqslant W \leqslant 10000$

① 如果稍微思考一下就能发现$s_{i+1}=t_{j+1}$时，只需令$dp[i+1][j+1]=dp[i][j]+1$就可以了。

样例

输入

```
n = 3
(w, v) = {(3, 4), (4, 5), (2, 3)}
W = 7
```

输出

```
10（0号物品选1个，2号物品选2个）
```

这次同一种类的物品可以选择任意多件了。我们再试着写出递推关系。

令$dp[i+1][j]$:=从前i种物品中挑选总重量不超过j时总价值的最大值。那么递推关系为：

$$dp[0][j] = 0$$
$$dp[i+1][j] = \max\{dp[i-k\times w[i]] + k\times v[i] \mid 0 \leqslant k\}$$

让我们试着编写一下按照这个递推关系求解的程序：

```
int dp[MAX_N + 1][MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      for (int k = 0; k * w[i] <= j; k++) {
        dp[i + 1][j] = max(dp[i + 1][j], dp[i][j - k * w[i]] + k * v[i]);
      }
    }
  }
  printf("%d\n", dp[n][W]);
}
```

这次的程序成了三重循环。关于k的循环最坏可能从0循环到W，所以这个算法的复杂度为$O(nW^2)$，这样并不够好。

我们来找一下这个算法中多余的计算（即已经知道结果的计算）。

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	8	8
2	0	0	0	4	5	5	8	9
3	0	0	3	4	6	7	9	10

DP数组

在$dp[i+1][j]$的计算中选择$k(k \geqslant 1)$个的情况，与在$dp[i+1][j-w[i]]$的计算中选择$k-1$个的情况是相同

的，所以$dp[i+1][j]$的递推中$k \geqslant 1$部分的计算已经在$dp[i+1][j-w[i]]$的计算中完成了。那么可以按照如下方式进行变形：

$$
\begin{aligned}
&\max\{dp[i][j-k\times w[i]]+k\times v[i] \mid 0\leqslant k\}\\
&=\max(dp[i][j],\max\{dp[i][j-k\times w[i]]+k\times v[i] \mid 1\leqslant k\})\\
&=\max(dp[i][j],\max\{dp[i][(j-w[i])-k\times w[i]+k\times v[i] \mid 0\leqslant k\}+v[i])\\
&=\max(dp[i][j],dp[i+1][j-w[i]]+v[i])
\end{aligned}
$$

这样一来就不需要关于k的循环了，便可以用$O(nW)$时间解决问题。

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	8	8
2	0	0	0	4	5	5	8	9
3	0	0	3	4	6	7	9	10

DP数组

```
void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[i + 1][j] = dp[i][j];
      } else {
        dp[i + 1][j] = max(dp[i][j], dp[i + 1][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[n][W]);
}
```

此外，此前提到的01背包问题和这里的完全背包问题，可以通过不断重复利用一个数组来实现。

01背包问题的情况

```
int dp[MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = W; j >= w[i]; j--) {
      dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
  }
  printf("%d\n", dp[W]);
}
```

完全背包问题的情况

```
int dp[MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = w[i]; j <= W; j++) {
      dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
  }
  printf("%d\n", dp[W]);
}
```

像这样书写的话，两者的差异就变成只有循环的方向了。重复利用数组虽然可以节省内存空间，但使用得不好将有可能留下bug，所以要格外小心。不过出于节约内存的考虑，有时候必须要重复利用数组。也存在通过重复利用能够进一步降低复杂度的问题。这些我们会在后面介绍。

专栏 DP数组的再利用

除上面的情况之外，还有可能通过将两个数组滚动使用来实现重复利用。例如此前的

```
dp[i+1][j]=max(dp[i][j], dp[i+1][j-w[i]]+v[i])
```

这一递推式中，$dp[i+1]$计算时只需要$dp[i]$和$dp[i+1]$，所以可以结合奇偶性写成如下形式：

```
int dp[2][MAX_W + 1]; // DP数组

void solve() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= W; j++) {
      if (j < w[i]) {
        dp[(i + 1) & 1][j] = dp[i & 1][j];
      } else {
        dp[(i + 1) & 1][j] = max(dp[i & 1][j], dp[(i + 1) & 1][j - w[i]] + v[i]);
      }
    }
  }
  printf("%d\n", dp[n & 1][W]);
}
```

01背包问题之2

有n个重量和价值分别为w_i, v_i的物品。从这些物品中挑选总重量不超过W的物品，求所有挑选方案中价值总和的最大值。

限制条件

- $1 \leqslant n \leqslant 100$
- $1 \leqslant w_i \leqslant 10^7$
- $1 \leqslant v_i \leqslant 100$
- $1 \leqslant W \leqslant 10^9$