Lesson1

> **动态规划大纲**
>
> - 常用模型：背包
> - 不同类型的DP：线性DP，区间DP，计数类DP，数位统计DP，状压DP，树形DP

> **背包问题**
>
> **DP分析方式：从集合角度进行理解**
>
> - **状态表示**：用几维变量表示状态（例如背包问题用两维f(i, j)表示）
>   - **集合**：例如背包问题是所有选法的集合，以背包问题分析，f(i, j)属性对应的集合满足的条件为
>   - 只从前 i 个物品中选
>   - 总体积 ≤ j
>   - **属性**：集合的属性，一般为以下三种
>   - 最大值Max（求最大值时，子集划分不重原则可以进行放松）
>   - 最小值Min
>   - 元素数量
> - **状态计算**：**如何一步步计算出每一个状态。**对于背包问题，如何计算出每个f(i, j)，问题答案为f(N, V)。
>   - **集合划分**： **把当前的集合划分为若干个更小的子集，使得每一个子集都可用前面已经计算出的状态表示。**
>
>   对于背包问题，将f(i, j)分为两大子集，第一类为不含i的选法，第二类为包含i的选法。第一类子集从1~i中选，总体积小于j，且不包含i，可以简化为从1 ~ i-1中选，总体积不超过j，所以该子集的最大价值可用**f(i-1, j)**表示；第二类子集从1~i中选，总体积小于j，且每种选法都包含i，因此我们可以去掉第二类子集所包含的每种选法中的i，且同时去掉i不会导致**原子集**中价值最大的选法发生变化，去掉之后的集合的最大价值可用**f(i-1, j-vi)**表示。且原子集最大价值可用 **f(i-1, j-vi)+wi**表示（曲线救国）。
>
>   因此 **f(i, j) = max(f(i-1, j), f(i-1, j-vi)+wi)**
>
>   **子集划分原则**
>
>   - 不重
>   - **不漏**
>
> DP优化：一般是对DP代码或者DP计算方程做等价变形
>
> **01背包问题**
>
> 每件物品最多只能用一次
>
> 具体实现1：朴素方法
>
> ```
> static int N = 1010;
> ```

```java

static int n, m;
static int[] v  = new int[N], w = new int[N];
static int[][] f = new int[N][N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) {
        ins.nextToken(); v[i] = (int)ins.nval;
        ins.nextToken(); w[i] = (int)ins.nval;
    }

    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++) {
            f[i][j] = f[i-1][j];
            if (j >= v[i]) f[i][j] = Math.max(f[i][j], f[i-1][j-v[i]]+w[i]);
        }

    out.println(f[n][m]);

    out.flush();
}
```

具体实现2：DP优化

能优化的两个条件

- 状态转移方程中，f(i)只用到了f(i-1)，因此可以用滚动数组优化（仍为2维）
- **转移方程右侧的 j 与 j-v[i] 都小于等于 j，因此可以进一步用一维数组进行计算（逆序计算）**

```java
static int N = 1010;

static int n, m;
static int[] v  = new int[N], w = new int[N];
static int[] f = new int[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) {
        ins.nextToken(); v[i] = (int)ins.nval;
        ins.nextToken(); w[i] = (int)ins.nval;
    }

    for (int i=1; i<=n; i++) {
        for (int j=m; j>=0; j--)
            if (j >= v[i]) f[j] = Math.max(f[j], f[j-v[i]]+w[i]);
    }

    out.println(f[m]);
```

```

    out.flush();
}
```

完全背包问题

每件物品有无限个

分析思路

- 状态表示：f[i, j]（f[i, j]存储属性）
  - 集合：只考虑前i个物品，且总体积不大于j的所有选法
  - 属性：最大值
- 状态计算：集合的划分
  - 对于f(i, j)，按第i个物品选多少个将集合划分为若干个子集（0, 1, 2, 3,..., k）

    因此，选k个第i个物品的子集的最大值属性可用**f[i-1, j-k * v[i]]+k * w[i]**表示

    状态转移方程：f[i, j] = Max(f[i, j], f[i-1, j - k * v[i]] + k * w[i])

具体实现：朴素写法，时间复杂度较高（TLE，初级）

```java
static int N = 1010;

static int n, m;
static int[] v = new int[N], w = new int[N];
static int[][] f = new int[N][N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) {
        ins.nextToken(); v[i] = (int)ins.nval;
        ins.nextToken(); w[i] = (int)ins.nval;
    }

    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            for (int k=0; k*v[i]<=j; k++)
                f[i][j] = Math.max(f[i][j], f[i-1][j-k*v[i]]+k*w[i]);

    out.println(f[n][m]);

    out.flush();
}
```

具体实现：优化1（降低时间复杂度，高级）

f[i, j] = Max(f[i-1, j], f[i-1, j-v]+w, f[i-1, j-2v]+2w, f[i-1, j-3v]+3w, ... ）（用v, w分别代替v[i], w[i]) f[i, j-v] = Max(f[i-1, j-v],   f[i-1, j-2v]+w, f[i-1, j-3v]+2w, ...)

所以Max(f[i-1, j-v]+w, f[i-1, j-2v]+2w, f[i-1, j-3v]+3w, ...) = f[i, j-v] + w

所以   **f[i, j] = Max(f[i-1, j], f[i, j-v[i]]+w[i])**

对比01背包 **f[i, j] = Max(f[i-1, j], f[i-1, j-v[i]]+w[i])**

```
static int N = 1010;

static int n, m;
static int[] v = new int[N], w = new int[N];
static int[][] f = new int[N][N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) {
        ins.nextToken(); v[i] = (int)ins.nval;
        ins.nextToken(); w[i] = (int)ins.nval;
    }

    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++) {
            f[i][j] = f[i-1][j];
            if (j>=v[i]) f[i][j] = Math.max(f[i][j], f[i][j-
v[i]]+w[i]);
        }

    out.println(f[n][m]);

    out.flush();
}
```

具体实现：进一步优化（降维，优化时间复杂度，终极）

```
static int N = 1010;

static int n, m;
static int[] v = new int[N], w = new int[N];
static int[] f = new int[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) {
        ins.nextToken(); v[i] = (int)ins.nval;
        ins.nextToken(); w[i] = (int)ins.nval;
    }

```

```
    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            if (j>=v[i]) f[j] = Math.max(f[j], f[j-v[i]]+w[i]);

    out.println(f[m]);

    out.flush();
}
```

多重背包问题

第 i 个物品有 Si 个

分析思路

- 状态表示：f[i, j]（f[i, j]存储属性）
    - 集合：只考虑前i个物品，且总体积不大于j的所有选法
    - 属性：最大值
- 状态计算：集合划分
    - 对于f(i, j)，按第i个物品选多少个将集合划分为若干个子集（0, 1, 2, 3,..., s[i]）

      状态转移方程：f[i, j] =max(f[i, j]，f[i-1, j - k * v[i]] + k * w[i])，且(k<=s[i])

具体实现：朴素版本

```
static int N = 110;

static int n, m;
static int[] s = new int[N], v = new int[N], w = new int[N];
static int[][] f = new int[N][N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=1; i<=n; i++) {
        ins.nextToken(); v[i] = (int)ins.nval;
        ins.nextToken(); w[i] = (int)ins.nval;
        ins.nextToken(); s[i] = (int)ins.nval;
    }

    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            for (int k=0; k<=s[i] && k*v[i]<=j; k++)
                f[i][j] = Math.max(f[i][j], f[i-1][j-k*v[i]]+k*w[i]);

    out.println(f[n][m]);

    out.flush();
}
```

具体实现：优化方案（位运算优化）

优化思路：

- 不能使用完全背包的优化问题

  f[i, j] = Max(f[i-1, j], f[i-1, j-v]+w, f[i-1, j-2v]+2w, ... , f[i-1, j-sv]+sw) （用v, w分别代替v[i], w[i]）

  f[i, j-v] = Max(f[i-1, j-v], f[i-1, j-2v]+w, ..., f[i-1, j-sv]+(s-1)w, f[i-1, j-(s+1)v]+sw)

  且Max(f[i-1, j-v], f[i-1, j-2v]+w, ..., f[i-1, j-sv]+(s-1)w) 在给定f[i, j-v]时并不能计算出来，即Max不能做减法。因此不能采用完全背包的优化方式

- **二进制优化**

  思考：假设s[i]=1023，是否需要枚举0~1023？有无可能用更高效方式进行枚举

  把第i个物品进行打包，每组i的数量分别为：1, 2, 4, 8, ..., 512 （一共10组），且每组最多只能选一次。因此，我们可以利用这10组拼凑出0~1023中的任意一个数，且每组最多只选一次。将每组数量转为二进制表示可以很容易证明这一结论。进一步，将每一组打包后的包裹看成是01背包中的物品，即用10个互不相同的新物品来表示原来的第i个物品。因此时间复杂度降为O(NVlogS)

  例如s=200，则可以将物品打包为：1, 2, 4, 8, 16, 32, 64, 73，即一共7组。

  证明：1, 2, 4, 8, 16, 32, 64可以拼凑出0~127中任意一个数，且0 ~ 127中每个数加上73即可拼凑出73~200中任意一个数，所以从0 ~ 200之内的数都可以被该7组数凑出来，且每一组最多只用一次。

  推广：对于任意一个S，将其分为1, 2, 4, 8, ..., 2^k，k是满足1+2+..+2^k <= S的最大正整数，最后补上

  C = S-(1+2+..+2^k) < 2^(k+1)。这些数可以凑出0~S中任意一个数，且每个数最多只使用一次。

  证明：

  首先1, 2, ..., 2^k可以凑出0~(2^(k+1))-1中任意一个数（二进制性质）

  对于1, 2, ..., 2^k，每个数加上C，则可以凑出C ~ (2^(k+1)) -1+C中任意一个数，且 (2^(k+1)) -1+C = S,

  且C <= 2^(k+1)-1， 所以[0, (2^(k+1))-1] U [C, (2^(k+1)) -1+C] = [0, S]

- **最终优化思路**：对每一个物品i，将其划分为 S[i] - > logS[i] 组物品，拆分之后进行01背包问题求解，因此时间复杂度从O(NVS)降为O(NVlogS)。

```
static int N = 1000*15, M = 2010;

static int n, m;
static int[] v = new int[N], w = new int[N];
static int cnt;
static int[] f = new int[M];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    while (n-- > 0) {
```

```java
            ins.nextToken(); int vv = (int)ins.nval;
            ins.nextToken(); int ww = (int)ins.nval;
            ins.nextToken(); int s = (int)ins.nval;

            int k = 1;
            while (k <= s) {
                cnt++;
                v[cnt] = k*vv;
                w[cnt] = k*ww;
                s -= k; k *= 2; //注意顺序
            }

            if (s > 0) {
                cnt++;
                v[cnt] = s*vv;
                w[cnt] = s*ww;
            }
        }

        for (int i=1; i<=cnt; i++)
            for (int j=m; j>=v[i]; j--)
                f[j] = Math.max(f[j], f[j-v[i]]+w[i]);

        out.println(f[m]);

        out.flush();
    }
```

分组背包问题

物品有n组，每组物品里有若干种物品，每种若干个，且每组里面最多只能选一个物品

分析思路

- 状态表示：f[i, j]（f[i, j]存储属性）
    - 集合：只从**前i组**物品中选，且总体积不大于j的所有选法
    - 属性：最大值（Max）
- 状态计算：集合的划分
    - 对于f(i, j)，**依据第i组物品选哪个或者不选来划分子集。**

      若不选，则为**f(i-1, j)**，若选择第i组里第k个，则为**f(i-1, j-v[i, k])+w[i, k])**

```java
static int N = 110;

static int n, m;
static int[][] v = new int[N][N], w = new int[N][N];
static int[] s = new int[N];
static int[] f = new int[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

```

```java
    for (int i=1; i<=n; i++) {
        ins.nextToken(); s[i] = (int)ins.nval;

        for (int j=1; j<=s[i]; j++) {
            ins.nextToken(); v[i][j] = (int)ins.nval;
            ins.nextToken(); w[i][j] = (int)ins.nval;
        }
    }

    for (int i=1; i<=n; i++)
        for (int j=m; j>=1; j--)
            for (int k=1; k<=s[i]; k++)
                if (j >= v[i][k]) f[j] = Math.max(f[j], f[j-v[i][k]]+w[i][k]);

    out.println(f[m]);

    out.flush();
}
```

背包九讲地址：https://www.bilibili.com/video/BV1qt411Z7nE