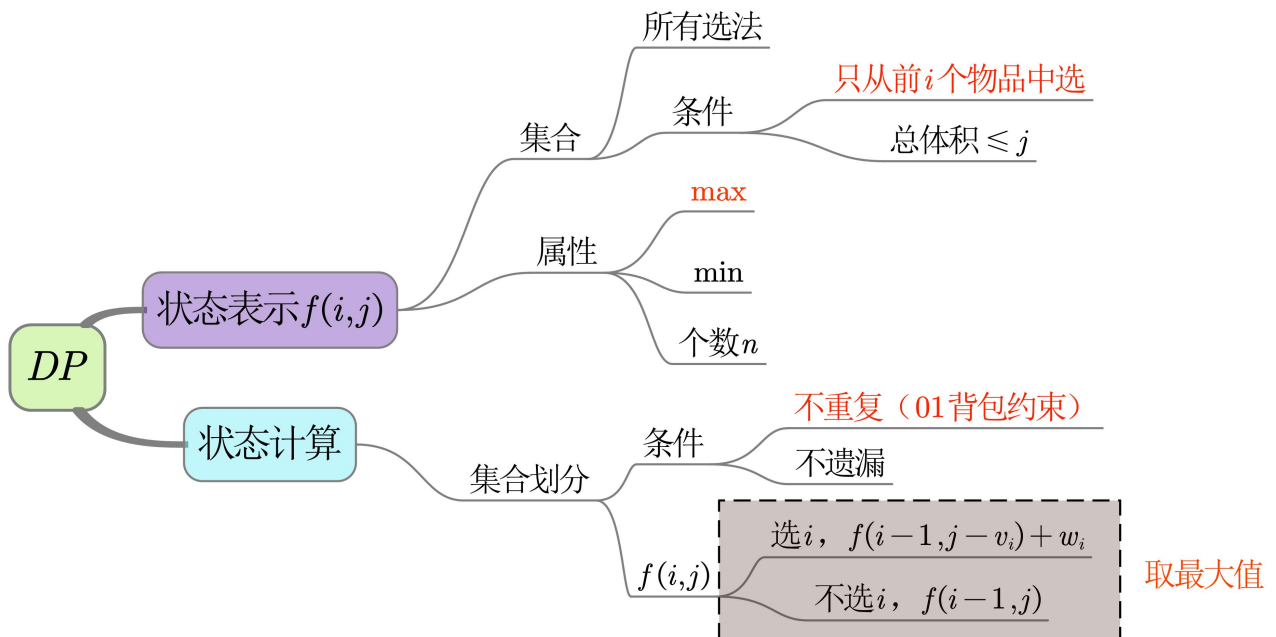


01背包问题



n 个物品，每个物品的体积为 v_i ，价值是 w_i ，背包的容量是 m

若每个物品最多只能装一个，且不能超过背包容量，则背包的最大价值是多少？

```
int n; // 物品总数
int m; // 背包容量
int[] w; // 重量
int[] v; // 价值
```

// $dp[i][j]$ 的含义：在考虑前 i 个物品后，背包容量为 j 条件下的最大价值

```
/* 二维形式 */
```

```
int[][] dp = new int[n][m]; // 表示考虑前  $i$  个物品后，背包容量为  $j$  时的最大价值
```

```
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){
        if(w[i] > j){ // 当前重量装不进，价值等于前 i-1 个物品
            dp[i][j] = dp[i-1][j];
        } else { // 能装，需判断
```

```

        dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-w[i]]
+ v[i]);
    }
}
}

return dp[n][m];

```

/* 因为dp[i][j] 始终用的是 dp[i-1][..] 这一行的数据进行更新，可以将其压缩成一维数组 */

// 使dp[i][j] 始终使用的是 左边的数据进行更新 dp[i-..][j]，为了避免写后读，所以应该倒序遍历容量。

/* 一维形式 */

```

int[] dp = new int[m]; // dp[j] 表示背包容量为j的情况下的最大价值

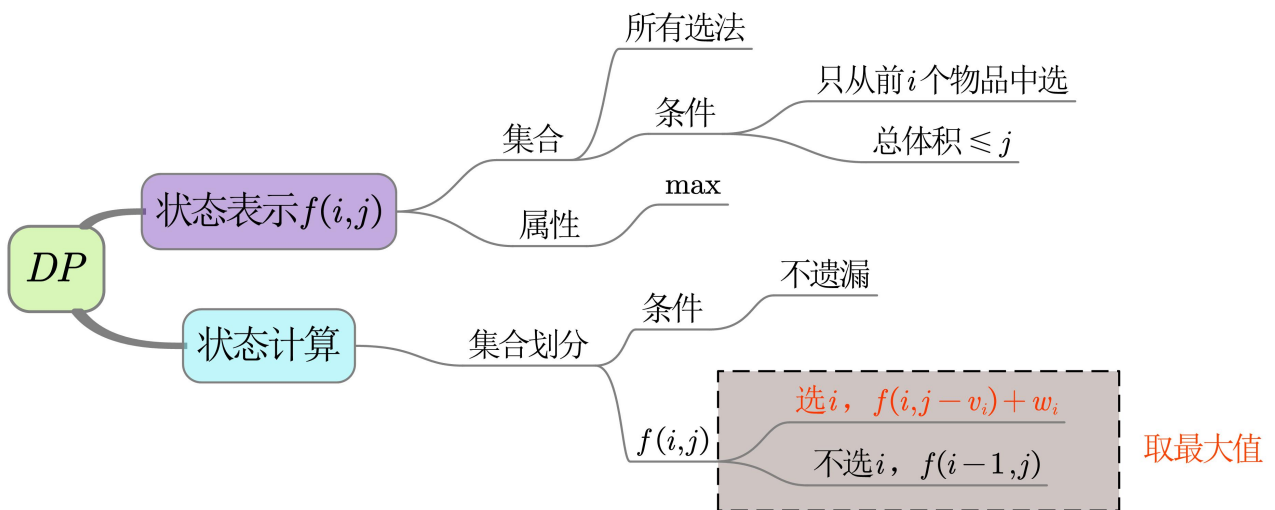
for(int i = 1; i <= n; i++){
    for(int j = m; j >= w[i]; j--){ // 注意是倒序，否则出现写后读错误
        dp[j] = Math.max(dp[j], dp[j-w[i]] + v[i]);
    }
}

return dp[m];

```

完全背包问题

在01背包的基础上，条件变为每个物品可以取任意个



状态 $dp[i,j]$ 表示：所有只考虑前 i 个物品，且总体积不大于 j 的所有选法

属性 max:

采用这样的思维：

1. 先去掉 k 个物品 i
2. 求出此时的 $\max(dp[i-1, j - k * w[i]])$
3. 再把 k 个物品加回来

得到 $dp[i-1][j - k * w[i]] + k * v[i]$

状态的划分就是对第 i 个物品取 $0, 1, 2, 3, \dots, k-1, k$ 个

```
int n; // 物品总数
int m; // 背包容量
int[] w; // 重量
int[] v; // 价值
```

// $dp[i][j]$ 的含义：在考虑前 i 个物品后，背包容量为 j 条件下的最大价值

```

/*      二维形式      */
int[][] dp = new int[n][m]; // 在考虑前i个物品后，背包容量为j条件下的最大价值

for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){
        dp[i][j] = dp[i-1][j]; // 对于不选第 i 个的情况；
        for(int k = 0; k * w[i] <= j; k++){
            dp[i][j] = Math.max(dp[i][j], dp[i-1][j - k * w[i]] +
v[i] * k);
        }
    }
}
return dp[n][m];

```

优化思路

$dp[i,j] = \max(dp[i-1,j], dp[i-1,j-w_i] + v_i, dp[i-1,j-2w_i] + 2v_i, dp[i-1,j-3w_i] + 3v_i, \dots, dp[i-1,j-kw_i] + k*v_i)$ ①

变形一下：

$dp[i,j-w_i] = \max(dp[i-1,j-w_i], dp[i-1,j-2w_i] + v_i, dp[i-1,j-3w_i] + 2v_i, dp[i-1,j-4w_i] + 3v_i, \dots, dp[i-1,j-kw_i] + (k-1)*v_i)$

再变形得：

$dp[i,j-w_i] + v_i = \max(dp[i-1,j-w_i] + v_i, dp[i-1,j-2w_i] + 2v_i, dp[i-1,j-3w_i] + 3v_i, dp[i-1,j-4w_i] + 4v_i, \dots, dp[i-1,j-kw_i] + k*v_i)$

此等式右边与等式①中深色部分相同：

所以可以得到

```

dp[i,j] = max(dp[i-1,j], dp[i,j-w_i] + v_i)

```

```

/* 优化后 */
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){
        if(j < w[i]){
            dp[i][j] = dp[i-1][j];
        }else{
            dp[i][j] = Math.max(dp[i-1][j], dp[i][j- w[i]] + v[i]);
        }
    }
}

return dp[n][m];

```

```

// 压缩成一维

int[] dp = new int[m]; //f[j]表示背包容量为j条件下的最大价值

for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){ // 因为变为一维，此时 j初始值应该为 w[i]
        if(j < w[i]){
            dp[i][j] = dp[i-1][j]; // 压缩成一维后，变为 dp[j]=dp[j]恒
            成立，可以抹去
        }else{
            dp[i][j] = Math.max(dp[i-1][j], dp[i][j- w[i]] + v[i]);
            // dp[j] = Math.max(dp[j], dp[j-w[i]] + v[i]);
        }
    }
}

/*优化后*/
for(int i = 1; i <= n; i++){
    for(int j = w[i]; j <= m; j++){ // 注意这里是顺序
        dp[j] = Math.max(dp[j], dp[j-w[i]] + v[i]);
    }
}

return dp[m];

```

多重背包问题

限定：第*i*件物品最多拿 s_i 件

$$dp[i,j] = \max(dp[i-1,j], dp[i-1,j-w_i] + v_i, dp[i-1,j-2w_i] + 2v_i, \dots, dp[i-1,j-s_iw_i] + s_i*v_i)$$

①

变形一下：

$$dp[i,j-w_i] = \max(dp[i-1,j-w_i], dp[i-1,j-2w_i] + v_i, dp[i-1,j-3w_i] + 2v_i, \dots, dp[i-1,j-s_iw_i] + (s_i-1)v_i, dp[i-1,j-(s_i+1)w_i] + s_i*v_i)$$

再变形得：

$$dp[i,j-w_i] + v_i = \max(dp[i-1,j-w_i] + v_i, dp[i-1,j-2w_i] + 2v_i, dp[i-1,j-3w_i] + 3v_i, \dots, dp[i-1,j-s_iw_i] + s_i v_i, dp[i-1,j-(s_i+1)w_i] + (s_i+1)*v_i)$$

可以看出多了最后一项，无法按照完全背包的方式优化

```
int n;           // 物品总数
int m;           // 背包容量
int w[n];        // 重量
int v[n];        // 价值
int s[n];        // 物品数量
int[][] dp = new int[n][m]; // dp[i][j] 表示在考虑前i个物品后，背包容量为j条件下的最大价值

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

return dp[n][m];
```

二进制优化

已知 $1, 2, 4, \dots, 2^k$ 可以由系数0和1线性组合出 $0-2^{k+1}-1$ 。

考虑更一般的情况，若想线性组合出 $0-S$ ，且 $S < 2^{k+2}$ ，则猜测可由 $1, 2, 4, \dots, 2^k, C$ 组合出，其中 $C < 2^{k+1}$ 。显然，在 C 一定存在的情况下，可得到的数的范围为 $C-S$ 。

由于 $C < 2^{k+1}$ ，则 $C \leq 2^{k+1}-1$ ，故可用 $1, 2, 4, \dots, 2^k, C$ 和一定的系数组成表示任何 $< 2^{k+2}$ 的数。

因此对于有 $s[i]$ 件的某个物品 i ，可以打包成 $\log(s[i])$ 包物品，每包有 $1, 2, 4, \dots, 2^k, C$ 件物品，其中 $k = \log(s[i]) - 1$ ，通过这种情况可以将 $s[i]$ 件物品从 0 到 $s[i]$ 分别进行枚举，从而转化为01背包。

```
/*      二进制优化      */
int n;           // 物品总数
int m;           // 背包容量
int w[N];        // 重量
int v[N];        // 价值
int s[N];

int cnt = 0;
for(int i = 1; i <= n; i++){
    int a = w[i];
    int b = v[i];
    int s = s[i];

    // 读入物品个数时顺便打包
    int k = 1;    // 当前包裹的大小
    while(k <= s){
        cnt++;    // 实际物品种数（即 打包后的包裹总数）
        w[cnt] = a*k;
        v[cnt] = b*k;
        k *= 2;   // 包裹容量倍增
    }

    if(s > 0){
        // 不足的单独放一个包裹，即C
        cnt++;
        w[cnt] = a*s;
        v[cnt] = b*s;
    }
}
```

```

    }
}
// 更新物品种数
n = cnt;

int[][] dp = new int[n][m];

// 转化为 0 1 背包问题
for(int i = 1; i <= n; i++){
    for(int j = m; j >= w[i]; j--){
        dp[j] = Math.max(dp[j], dp[j - w[i]]+v[i]);
    }
}

return dp[m];

```

分组背包问题

每组物品最多拿一个

枚举第 i 组物品选哪个 或者 都不选

```
dp[i-1, j] // 一个都不选
```

```
dp[i-1, j-w[i,k]] + v[i,k]; // 从前 i 件物品中选了 第 k 件
```

```

int n;           // 物品总数
int m;           // 背包容量
int v[N][S];     // 重量
int w[N][S];     // 价值
int s[N];        // 各组物品种数

// int[][] dp = new int[n][m];

for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){
        for(int k = 0; k < s[i]; k++){
            //if(j >= w[i][k]){

```



```

        //dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-w[i]
[k]] + v[i][k]);
    //}else{
        dp[i][j] = dp[i-1][j];
    //}

    /* 优化成一维 */
    if(w[i][k] <= j){
        dp[j] = Math.max(dp[j], dp[j-w[i][k]] + v[i][k]);
    }
}
}
}

return dp[n][m];

```