背包问题

- 01背包问题
- 完全背包问题
- 多重背包问题
- 混合三种背包问题
- 转化为01背包问题
  - 完全背包 -> 01背包问题
  - 多重背包问题 -> 01背包问题
  - 混合背包问题 -> 01背包问题
- 分组背包问题
- 恰好装满
- 求方案总数
- 二维背包问题
- 求最优方案

DP数组优化

- 滚动数组
- 状态压缩

子串问题

- 最长上升子串
- 最长公共子串
- 最大子串和
- 最长回文子串
  - 解法一：翻转字符串
  - 解法二：直接DP

子序列问题

- 最长上升子序列
- 最长公共子序列
- 最长回文子序列

背包问题

01背包问题

> 给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。我们每种物品能拿的数量是0个或者1个。

01背包问题指的就是要么拿要么不拿

最小规模的问题：当背包能装0千克，或者没有物品的时候，总价值是0元（初始化）

往上一点看：如果能装x千克，有y件物品，此时的总价值。

状态转移方程：

- dp[i, j] 表示在背包容量为j的情况下，考虑前i种物品，能装的最大价值
- dp[i, j] = max{dp[i-1, j - w[i]] + v[i], dp[i-1, j]}

滚动数组优化：

- 可以用两个数组，也可以用一个数组

初始化：dp[0~n, 0] = dp[0, 0~n] = 0;

完全背包问题

> 给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。我们每种物品能拿的数量是无限个。

状态转移方程：

- `dp[i, j]` 表示在背包容量为j的情况下，考虑前i种物品，能装的最大价值
- `dp[i, j] = max{dp[i-1, j - k * w[i]] + k * v[i]}` 枚举每一个可能的k(0<=k<=j/w[i])

滚动数组优化：

- 可以用两个数组，也可以用一个数组

多重背包问题

> 给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。我们每种物品能拿的数量是`n[i]`个。

状态转移方程：

- `dp[i, j]` 表示在背包容量为j的情况下，考虑前i种物品，能装的最大价值
- `dp[i, j] = max{dp[i-1, j - k * w[i]] + k * v[i]}` 枚举每一个可能的k (0<=k<=min{j/w[i],n[i]})

滚动数组优化：

- 可以用两个数组，也可以用一个数组

混合三种背包问题

> 给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。我们每种物品能拿的数量是可能是1个，可能是有限个，也有可能是无限个。

状态转移方程：

- dp[i, j] 表示在背包容量为j的情况下，考虑前i种物品，能装的最大价值
- dp[i, j] = max{dp[i-1, j - k * w[i]] + k * v[i]} 枚举每一个可能的k (0<=k<=min{j/w[i],n[i]})

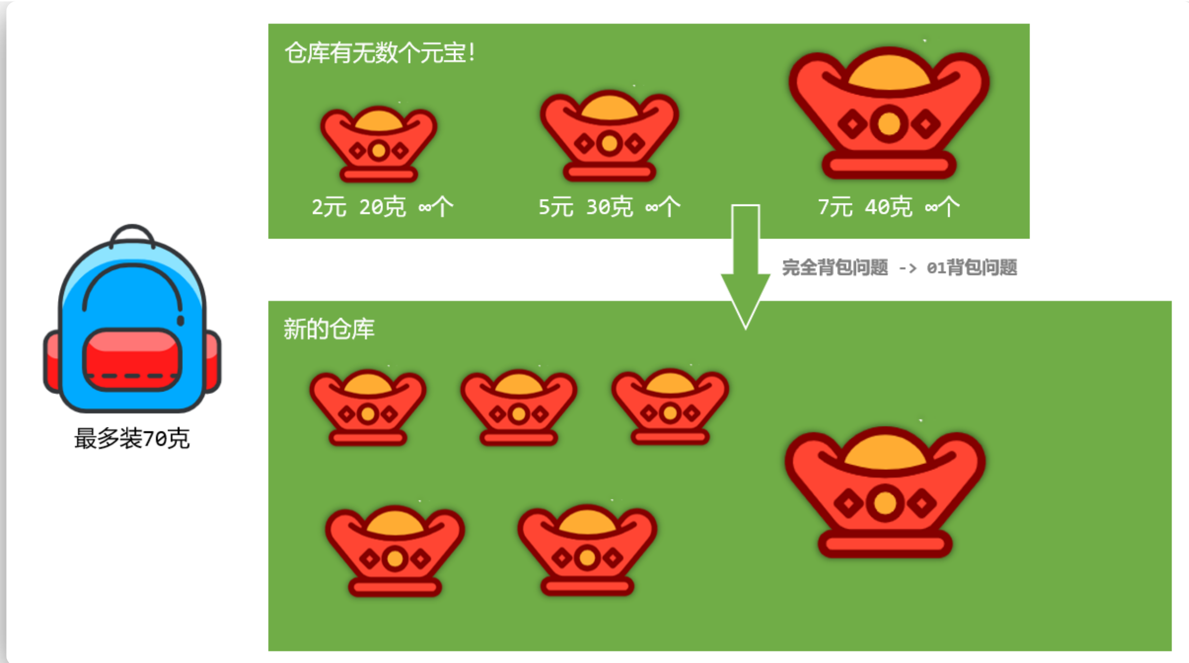
滚动数组优化：

- 可以用两个数组，也可以用一个数组

转化为01背包问题

将每种物品复制多（在合理范围内min{j/w[i],n[i]}）个，再把拆分多个的物品看作是不一样的物品。

完全背包 -> 01背包问题



多重背包问题 -> 01背包问题

混合背包问题 -> 01背包问题

分组背包问题

> 有n个物品，被分为k组，每一组只能拿一种，否则就会冲突，问此时最优解?

```
for k ← 1 to K
    for v ← V to 0
        for all item i in group k
            F[v] ← max{F[v], F[v − C_i] + W_i}
```

以每组为循环单位，再循环每一个容量，最后遍历本组每一个物品取最大值即可。

dp[i, j]表示前i组，在背包容量是j的情况下的最优解

dp[i, j] = max{dp[i-1, j], max{dp[i-1, j-w[k]] + v[k]}}

恰好装满

> 添加一个限制就是必须恰好装满背包。

- 初始化：dp[0~n, 0] = dp[0, 0~n] = 0;
- dp其他位置值都为 -inf

求方案总数

> 求装满背包或将背包装至某一指定容量的方案总数。

初始化：dp[0~n, 0] = dp[0, 0~n] = 1;

- dp[i, j] = sum(dp[i-1, j], dp[i, j-w[i]]) // j >= w[i]

二维背包问题

> 二维背包问题是指每个背包有两个限制条件（比如重量和体积限制），选择物品必须要满足这两个条件。

- dp[i, j, k] = max{dp[i-1, j - w[i], k - 体积[i]] + v[i], dp[i-1, j, k]}

求最优方案

对于01背包问题：dp[i, j] = max{dp[i-1, j - w[i]] + v[i], dp[i-1, j]}

`cot<<dp[n, W]<<endl;`

如果当前状态等于拿了当前物品，且

```
if(dp[n, W] == dp[n-1, W-w[i]] + v[i]) {
    // 所以就拿了第i个物品
} else {
    // 没有拿第i个物品
}
```

对于多重背包问题：dp[i, j] = max{dp[i-1, j - k * w[i]] + k * v[i]} 枚举每一个可能的k (0<=k<=min{j/w[i],n[i]})

```
for k in 可能的范围 {
    if(dp[n, W] == dp[n-1, W-k * w[i]] + k * v[i]) {
        // 所以就拿了 k个 第i个物品
    } else {
        // 没有拿第i个物品
    }
}
```

DP数组优化

这也是为什么动态规划比记忆化搜索好。

滚动数组

例如01背包问题：dp[i, j] = max{dp[i-1, j - w[i]] + v[i], dp[i-1, j]} O(nm)

定义为dp0, dp1 O(m)

优化的思路可以看状态转移方程的依赖关系。

状态压缩

面试题 01.01. 判定字符是否唯一

难度 简单 135

实现一个算法，确定一个字符串 `s` 的所有字符是否全都不同。

示例 1：

```
输入：s = "leetcode"
输出：false
```

示例 2：

```
输入：s = "abc"
输出：true
```

```
// 定义一个数组，如果出现了，就标记为true
bool vis[26] = {false}
// 定义一个数字，2^5,字符串是abcb
0 0 0 0 0 = 0    // 初始情况
0 0 0 0 1 = 1    // 遍历到a
0 0 0 1 1 = 3    // 遍历到b
0 0 1 1 1 = 7    // 遍历到c
// 最后遍历到b，发现b已经访问过了，所以返回false
```

状态压缩就是集合到数字的转变。

dfs(vector& vec) {} ==> dfs(int num) {}

子串问题

> abcd => ab, bc, abc, 不可以"ac"
>
> 子串一定是连续的子序列

最长上升子串

> nums = [1, 2, 5, 6, 3, 2, 7]

把当前问题规模最小化：只有两个元素：[2, 5], [5, 6]

dp[i]表示以**第i个元素结尾**的最长上升子串的最优解

```
if (当前元素比上一个大或等于)
    dp[i] = dp[i-1] + 1
if (当前元素比上一个小)
    dp[i] = 1
```

初始化：dp[0] = 0

最长公共子串

> s1 = "abbabb" i=4"abbb" -> max{"abb"和"aab"或 "abba"和"aa"}
>
> s2 = "aab" j=3"aba"

把当前问题规模最小化：s1或s2为空字符串，那么答案就等于0

如果s2长度为1，那么最长公共子串是多长

dp[i, j]表示s1的前i个字母和s2的前j个字母所构成的最长公共子串

```
if(s1[i]==s2[j]) {
    dp[i, j] = dp[i-1, j-1] + 1
} else {
    dp[i, j] = 0
}
```

最大子串和

> nums = [11, -10, 20, 11, -3]

把当前问题规模最小化：只有两个元素 [10, 20], [1, 20]

[0, 20] 和 [11, -10, 20]

定义dp：dp[i]表示以第i个元素结尾的子串最大和

```
dp[i] = max{dp[i-1] + nums[i], nums[i]}
```

dp[0] = 0;

最长回文子串

> s = "abccbc"

解法一：翻转字符串

将s翻转得到re_s，再找最长公共子串即可。

解法二：直接DP

把当前问题规模最小化：

- 字符串为空：0
- 字符串长度为1：1

"c" <- "bcb"

"ac" <- "bacb"

初始化的时候dp[i, j] = -1，除了dp[i, i] = 1, dp[0, 0] = 0

dp[i, j]表示以i开始，j结尾的字符串，所构成的最长回文子串长度

如果s[i]==s[j]并且s[i+1, j-1]可以构成回文子串

- dp[i, j] = dp[i+1, j-1] + 2

如果s[i]!=s[j]

- dp[i, j] = max{dp[i+1, j], dp[i, j-1]}

遍历顺序：

i -> 0~n

j -> 0~i

子序列问题

> abcd => ab, bc, abc，也可以"ac"
>
> 子序列不一定要连续

最长上升子序列

> nums = [2, 2, 5, 6, 3, 1, 7]

把当前问题规模最小化：只有两个元素：[2, 7]，[6, 7]

dp[i]表示以**第i个元素结尾**的最长上升子序列的最优解

```
for(k -> 看1~i-1个元素){
    if (当前元素比第k个元素大或等于)
        dp[i] = dp[k] + 1
}
当且仅当前面都没有比第i个小，就取1
```

初始化：dp[0] = 0

最长公共子序列

> s1 = "abbabb" i=4"abba" -> max{"abb"和"aab" 或 "abba"和"aa"}
>
> s2 = "aab" j=3"aab"

把当前问题规模最小化：s1或s2为空字符串，那么答案就等于0

如果s2长度为1，那么最长公共子序列是多长

dp[i, j]表示s1的前i个字母和s2的前j个字母所构成的最长公共子序列

```
if(s1[i]==s2[j]) {
    dp[i, j] = dp[i-1, j-1] + 1
} else {
    dp[i, j] = max{dp[i-1, j], dp[i, j-1]}
}
```

最长回文子序列

> s = "abccbc"

把当前问题规模最小化：

- 字符串为空：0
- 字符串长度为1：1

"c" <- "acb"

"ac" <- "bacb"

dp[i, j]表示以i开始，j结尾的字符串，所构成的最长回文子串长度

如果s[i]==s[j] dp[i, j] = dp[i+1, j-1] + 2

如果s[i]!=s[j] dp[i, j] = max{dp[i+1, j], dp[i, j-1]}