1 DP 基本

1.1 DP 的两种编程方法

以斐波那契数列计算为例: $F_1 = F_2 = 1, F_i = F_{i-1} + F_{i-1}, i >= 3$

简单递归

```
def classic_fib(n):
  if n == 1 or n == 2:
    return 1
  return classic_fib(n-1) + classic_fib(n-2)
```

1.1.1 自顶向下结合记忆化

依然采取递归的程序结构, 但是计算返回值前先检查待求结果是否已经被计算出. 解决每一个子问题之后存储结果, 需要时直接返回已经缓存的结果.

```
memo = dict()
  def memo_fib(n):
    global memo
    if n == 1 or n == 2:
      return 1
    elif n in memo.keys():
      return memo[n]
    memo[n] = memo_fib(n-1) + memo_fib(n-2)
    return memo[n]
```

1.1.2 自底向上结合制表

规避了递归编程. 在解决大问题时先解决小问题, 逐步递推到大问题. 递推过程一般需要填写多维表格 dp, 编码时会使用若干 for 循环体填表.

```
def dp_fib(n):
  dp = list()
  dp.append(1)
  dp.append(1)
  for i in range(2, n):
    dp.append(dp[i-2] + dp[i-1])
  return dp[-1]
```

```
n = 10
print(classic_fib(n))
print(memo_fib(n))
print(dp_fib(n))
>>
55
55
55
```

1.2 DP 的设计与实现

以 0/1 背包问题为例 (**hdu 2602**), 背包容量为 C, 骨头数量为 N, 已知每块骨头的价值 v_i 和体积 c_i, 求背包内骨头的最大可能价值

1.2.1 dp 状态设计

总共有 N 块骨头, 背包总体积为 C. 令二维数组 $dp[i][j]$ 为第 i 块骨头放进容量为 j 的背包后, 背包内的最大价值. 考虑到自底向上递推需要的初始化, $i \in [0, N]; j \in [0, C]$, 且 $dp[0][x] = 0, dp[x][n] = 0$ (放入 0 块物品的总价值为 0, 空间为 0 的总价值为 0)

1.2.2 状态转移方程

对于容量为 c 的物块, 当前背包的容量为 j, 放不放入背包有两种情况:

1. $c > j$, 无法放入, $dp[i][j] = dp[i-1][j]$. 直接继承第 i-1 个物品放入空间为 j 的背包后, 背包的最大价值.

2. $c <= j$, 可以放入, 此时分为两种情况

放入: 容量为 j 的背包需要腾出 c 来存放物品, 因此总价值为 $dp[i][j-c]$ (1)

不放入: 同情况 1, 总价值仍为 $dp[i-1][j]$ (2)

此情况下的最终容量应为 $max\{(1), (2)\}$

1.2.3 递推实现

```python
N, C = map(int, input().split())
vs = list(map(int, input().split()))
cs = list(map(int, input().split()))
dp = [[0 for _ in range(C+1)]for _ in range(N+1)]
# fill dp table, skip the first line
for i in range(1, N+1):
    v = vs[i-1]
    c = cs[i-1]
    for j in range(C+1):
        # judge
        if c > j:
            dp[i][j] = dp[i-1][j]
        else:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-c] + v)

for line in range(N+1):
        print(dp[line])
    print(f'\nfinal result: {dp[N][C]}')
```

1.2.4 记忆化实现

```python
def cal_dp(i, j, c=0, v=0):
    global dp
    if i == 0 or j == 0:
        return 0
    if dp[i][j] != 0:
        return dp[i][j]
    if c > j:
        return cal_dp(i-1, j)
    else:
        ans = max(cal_dp(i-1, j-c) + v, cal_dp(i-1, j))
        dp[i][j] = ans
        return ans

N, C = map(int, input().split())
vs = list(map(int, input().split()))
```

```
cs = list(map(int, input().split()))
dp = [[0 for _ in range(C+1)]for _ in range(N+1)]
for i in range(1, N+1):
  c = cs[i-1]
  v = vs[i-1]
  for j in range(C+1):
    dp[i][j] = cal_dp(i, j, c, v)


for line in range(N+1):
  print(dp[line])
print(f'\nfinal result: {dp[N][C]}')
```

1.3 滚动数组简化优化空间占用

多维数组在空间占用上较大, 而在状态转移方程中, 新一行的计算只依赖于上一行, 不再需要访问更早之前计算出的结果. 因此可以复用之前的空间.

但是滚动数组损失了中间信息, 最终将无法输出具体的方案.

1.3.1 交替滚动

dp 定义为 $dp[2][j]$, 行号使用 new 和 old 分别指向当前计算的行和上一行.

hdu 2602 递推的交替滚动实现.

```
def swap(a, b):
  temp = b
  b = a
  a = temp
  return a, b

N, C = map(int, input().split())
vs = list(map(int, input().split()))
cs = list(map(int, input().split()))
dp = [[0 for _ in range(C+1)]for _ in range(2)]
# 此处的01赋值是为了在接下来的循环中, new和old的取值更加符合直觉
new = 0
old = 1
# fill dp table, skip the first line
for i in range(1, N+1):
  v = vs[i-1]
  c = cs[i-1]
  new, old = swap(new, old)
  for j in range(C+1):
    # judge
    if c > j:
      dp[new][j] = dp[old][j]
    else:
      dp[new][j] = max(dp[old][j], dp[old][j-c] + v)

print(dp[new][C])
```

1.3.2 自我滚动

缺, 待补

1.4 经典线性 DP 与优化策略

1.4.1 0/1 背包问题

同前述, 略

1.4.2 多重背包问题 二进制拆分优化

在 0/1 背包问题的基础上, 每种物体除了重量 w, 价值 v 外, 还有一个属性数量 m. 即同一种物体可以在背包中放置多个.

解法 1 转化为简单 0/1 背包: 取消数量属性, 同一种的多个物体直接视为多个物体, 只是重量和价值相等. 编码时, 这种做法会导致超时.

二进制拆分优化: 将每种物体的数量从小到大拆分为 $2^0 + 2^1 + ...$ 的组合, 若不能恰好被 2 的指数拆分, 最后一个为余数. 假设一种物体有 15 个, 将 15 按照二进制分解为 $15 = 1 + 2 + 4 + 8$; 假如有 16 个, 拆分为 $1 + 2 + 4 + 8 + 1$, 拆分到 8 时, 当前组合的数量为 2^3, 剩余 $1 < 2 \cdot 2^3$, 直接作为最后一个组合的数量.

将每一种物体都进行上述拆分, 得到的多组物体视作新的需要放入背包的物体, 进行简单的 0/1 背包问题求解.

原理: 任何数都可以分解为若干 2 的指数与余数的和, 即 $x = (\Sigma 2^i) + r$, 因此, 对于同一种物体, 在背包中放置 x 个, 等价于从经过二进制分解的多个物体组合中放置数量恰好的几个组合.

问题转化思路为: 一种物体可以放置多个 $\Longrightarrow$ 将十进制的组合问题转换为 2 进制数的组合问题以减少组合数量, 减少运行时间 $\Longrightarrow$ **二进制分解优化** $\Longrightarrow$ 新的物品 (重量, 价值) 组合 $\Longrightarrow$ 0/1 背包问题

洛谷 P1776

输入: 第一行为两个整数 n 和 W, 分别表示宝物种数和采集车的最大载重。

接下来 n 行每行三个整数: w_i, c_i, m_i, 表示第 i 个物体的价值, 体积, 数量.

输出: 输出仅一个整数，表示在采集车不超载的情况下收集的宝物的最大价值。

```python
def swap(a, b):
  return b, a

n, W = map(int, input().split())
a = [None for _ in range(n)]
for i in range(n):
  obj = list(map(int, input().split()))
  a[i] = obj
new_w = []
new_v = []

for i in range(n):
  v, w, m = a[i]
  j = 1
  while j > 0:
    # add in new packaged objects
    new_w.append(j*w)
    new_v.append(j*v)
    # update remained m
    a[i][2] -= j
    # update j
    if 2*j < a[i][2]:
      j *= 2
    else:
      j = a[i][2]

# 0/1 package prob. of new_w, new_v
n = len(new_w)
dp = [[0 for _ in range(W+1)] for _ in range(2)]
new = 0
old = 1
for i in range(n):
  w = new_w[i]
  v = new_v[i]
  new, old = swap(new, old)
  for j in range(W+1):
    if w > j:
      dp[new][j] = dp[old][j]
    else:
```

```
40         dp[new][j] = max(dp[old][j], dp[old][j-w] + v)
41
42 print(dp[new][W])
```