动态规划

动态规划思路

- 刻画一个最优解的结构特征
- 递归定义最优解的值
- 计算最优解的值，通常采取自低向上的方法
- 利用计算出的信息构造一个最优解

钢条切割问题

> 给定一段长度为n英寸的钢条和一个价格表$p_i(i = 1, 2, \cdots, n)$, 求切割钢条方案，使得销售收益r_n 最大。注意，如果长度为n英寸的钢条的价格p_i足够大，最优解可能就是完全不需要切割

- 刻画一个最优解的结构特征

  将钢条从左边切割长度为i的一段，只对右边剩下长度为$n-i$的一段继续切割，对左边的一段不再切割。

- 递归定义最优解的值

  $$r_n = \max_{1 \le i \le n}(p_i + r_{n-1})$$

- 计算最优解的值

```
def cut_rot(p: list) -> int:
    n = len(p)
    r = [0] * (n+1)
    for j in range(1, n+1):
        q = -1
        for i in range(j + 1):
            q = max(q, p[i] + r[j - i])
        r[j] = q
    return r[n]
```

- 利用计算出的信息构造一个最优解

```
def cut_rot(p: list):
    n = len(p)
    r = [0] * (n+1)
    s = [0] * (n+1)
    for j in range(1, n+1):
        q = -1
        for i in range(j+1):
            if q < p[i] + r[j - i]:
                q = p[i] + r[j - i]
                s[j] = i
            r[j] = q
    return (r[n], s[n])
```

活动选择问题

> 假定有n个活动的集合$S = \{a_1, a_2, \cdots, a_n\}$,这些活动使用同一个资源，而这个活动在某个时刻只能供一个活动使用。每个活动a_i都有一个开始时间s_i和一个结束时间f_i,其中$0 \le s_i < f_i < \infty$。如果被选中，任务a_i 发生在半开区间$[s_i, f_i)$期间。如果两个活动a_i和a_j 满足$[s_i, f_i)$和$[s_j, f_i)$不重叠，则称它们是兼容的。在活动选择问题中，我们希望选出一个最大兼容集。假定活动已按结束时间的单调递增顺序排序:
>
> $f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n$

解法一

- 刻画一个最优解的结构特征

  假设S_{ij}表示在活动a_i结束之后，a_j开始之前的所有活动的集和，记A_{ij}为S_{ij} 中活动的最大兼容子集且$a_k \in A_{ij}$,于是可以得到两个子问题: 寻找S_{ik}中的兼容子集，寻找S_{kj}中的兼容子集。令$A_{ik} = A_{ij} \bigcap S_{ik}$,$A_{kj} = A_{ij} \cup a_k \cup A_{kj}$, $A_{ij} = A_{ik} \cup a_k \cup A_{kj}$

  于是得到$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

- 递归定义最优解的值

$$c_{[i,j]} = \begin{cases} 0, S_{ij} = \emptyset \\ c_{[i,k]} + c_{[k,j]} + 1, S_{ij} \neq \emptyset \end{cases}$$

- 计算最优解的值

```
def act_sel(s: list[int], f: list[int]):
    n = len(s)
    c = [[0 for j in range(n+2)] for i in range(n+1)]
    for j in range(1, n+2):
        for i in range(j):
            for k in range(i+1, j):
                if s[k] > s[i] and f[k] < s[j]:
                    c[i][j] = c[i][k] + c[k][j] + 1
    return c[0][n+1]
```

- 利用计算出的信息构造一个最优解

```
def act_sel(s: list[int], f: list[int]):
    n = len(s)
    r = []
    c = [[0 for j in range(n+2)] for i in range(n+1)]
    for j in range(1, n+2):
        for i in range(j):
            for k in range(i+1, j):
                if s[k] > s[i] and f[k] < s[j]:
                    c[i][j] = c[i][k] + c[k][j] + 1
                    r.append(k)
    return (c[0][n+1], r)
```

解法二

- 刻画一个最优解的结构特征

  假设S为所有活动的集和，若$a_K \in S$,那么可以对a_K执行两种操作:选和不选，定义A_{k-1}为a_k之前在此模式下执行的最优解，当不选择a_k时，显然$|A_k| = |A_{k-1}|$,当选择a_k时，记a_m为结束时间最接近但小于a_k开始时间的活动，由此可以推出$|A_k| = |A_m| + 1$

- 递归定义最优解的值

  $$c_i = \max_{0 \le m < i}(c_{i-1}, c_m + 1)$$

- 计算最优解的值

```
def act_sel(s: list[int], f: list[int]):
    n = len(s)
    c = [0] * (n+1)
    for i in range(1, n+1):
        j = i - 1
        pos = s[j]
        while f[j - 1] > pos:
            j -= 1
        c[i] = max(c[i - 1], c[j-1] + 1)
    return c[n]
```

- 利用计算出的信息构造一个最优解

```
def act_sel(s: list[int], f: list[int]):
    n = len(s)
    c = [0] * (n+1)
    s = []
    for i in range(1, n+1):
        j = i - 1
        pos = s[j]
        while f[j - 1] > pos:
            j -= 1
        c[i] = max(c[i - 1], c[j-1] + 1)
        if c[i-1] > c[j - 1] + 1:
            c[i] = c[i-1]
        else:
            c[i] = c[j-1] + 1
            s.append(i)
    return (c[n], s)
```

0-1背包问题

> 一个正在抢劫商店的小偷发现了n 件商品，第i件商品价值v_i美元,重w_i磅，v_i和w_i都是整数。这个小偷希望拿走价值尽量高的商品，但他的背包最多容纳w磅重的商品，w是一个整数。他应该拿哪些商品？

- 刻画一个最优解的结构特征

  假设dp_{ij}表示j磅第i件商品的最优解，第i件有两种决策：选和不选。

  如果从第i件商品考虑，不选第i件商品后，包内最大容纳量为j。在此容纳量下，我们完成了$i-1$次决策，最优解可表示为$dp_{(i-1)j}$，同理选择第i件商品后，包内最大容纳量为$j-weight_i$在此容纳量下，我们完成了$i-1$次决策，最优解可表示为

  $dp_{(i-1)(j-weight_i)}$

- 递归定义最优解的值

$$dp_{ij} = \max(dp_{(i-1)j}, dp_{(i-1)(j-weight_i)} + value_i)$$

- 计算最优解的值

```
def com_sel(value:list[int], weight:list[int], W:int):
    n = len(value)
    dp = [[0 for j in range(W+1)] for i in range(n)]
    #初始化
    for j in range(weight[0], W+1):
        dp[0][j] = value[0]
    for j in range(1, W+1):
        for i in range(1, n):
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]] + value[i])
    return dp[n-1][W]
```

- 利用计算出的信息构造一个最优解

```
def com_sel(value: list[int], weight: list[int], W: int):
    n = len(value)
    dp = [[0 for j in range(W+1)] for i in range(n)]
    s = []
    # 初始化
    for j in range(weight[0], W+1):
        dp[0][j] = value[0]
    for j in range(1, W):
        for i in range(1, n):
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]] + value[i])
    for i in range(1, n):
        if dp[i-1][W-weight[i]] + value[i] < dp[i-1][W]:
            dp[i][W] = dp[i-1][W]
        else:
            dp[i][W] = dp[i-1][W-weight[i]] + value[i]
            s.append(i)
    return dp[n-1][W], s
```

在本章中，每个步骤都要进行一次选择，但选择通常依赖子问题的解。因此，我们通常以一种自底向上的方式求解，先求子问题，然后是较大的问题。然而，是否可以做出局部最优选择，并将该选择加入最优解中，这引出了我们的下一章—**贪心选择**