问题1 — 课程分配 / 活动选择

我们有以下活动：[S={a_1，a_2，\dots，a_n}] 每个活动（a_i）有开始时间（s_i）和结束时间（f_i）。

假设活动按结束时间排序：[f_1 \le f_2 \le \点 f_n\ 点 \ ]

定义边界活动（a_a）和（a_p）之间的兼容集合（\hat S_{a，p}）：活动（a_k\in \hat S_{a，p}）必须满足：[f_a \le s_k\quad\text{and}\quad f_k \le s_p]

鉴于DP复发情况：[c[a，p]= \开始{案件} 0， & \hat S_{a，p}=\emptyset \\ \Max\{c[a，k]+c[k，p]+1 \mid a_k\in \hat S_{a，p}\}， & \hat S_{a，p}\ne\emptyset \结束{案件}]

（1） 最优子结构→动态规划适用

如果区间内的最优解（（a_a， a_p））选择某活动（a_k\in\hat S_{a，p}），那么剩余的选择活动必须完全处于两个独立的子区间中：

- 左子问题：（（a_a， a_k）之间的活动
- 右子问题：（（a_k， a_p）之间的活动

因为任何与（a_k）兼容的活动要么完全在（a_k）之前，要么完全在（a_k）之后，左右选择之间没有冲突。

因此，最优解满足：[c[a，p] = c[a，k] + 1 + c[k，p]] 对于最佳分裂点（k）。
这正是**最优子结构**性质，因此适用动态规划。

在[1]中：

```
# Example activities: (name, start, finish)
activities = [
    ("a1", 1, 4),
    ("a2", 3, 5),
    ("a3", 0, 6),
    ("a4", 5, 7),
    ("a5", 3, 9),
    ("a6", 5, 9),
    ("a7", 6, 10),
    ("a8", 8, 11),
```

```python
    ("a9", 8, 12),
    ("a10", 2, 14),
    ("a11", 12, 16),
]

# Sort by finish time (as required)
activities = sorted(activities, key=lambda x: x[2])

activities
```

出局[1]:

```
[('a1', 1, 4),
 ('a2', 3, 5),
 ('a3', 0, 6),
 ('a4', 5, 7),
 ('a5', 3, 9),
 ('a6', 5, 9),
 ('a7', 6, 10),
 ('a8', 8, 11),
 ('a9', 8, 12),
 ('a10', 2, 14),
 ('a11', 12, 16)]
```

（2）自上而下DP（重复）——`RecuSelect(s, f, a, p)`

我们增加了两个哨兵（边界）活动：

- （a_0）：结束于（-\infty）
- （a_{n+1}）：从（+\infty）开始

那么完整的问题是。 `RecuSelect(0, n+1)`

想法：对于每个区间 （（a，p）），尝试每个兼容的 （k\in\hat S_{a，p}）， 选择最大化（c[a，k]+1+c[k，p]），并在两侧递归。

伪代码：

```
RecuSelect(a, p):
    if Ŝ_{a,p} is empty: return []
```

```
pick k in Ŝ_{a,p} that maximizes ( 1 + c[a,k] + c[k,p] )

return RecuSelect(a,k) + [k] + RecuSelect(k,p)
```

在[6]中：

```python
def add_sentinels(acts):
    # acts: list of (name, s, f) sorted by finish time
    sentinel_start = ("a0", float("-inf"), float("-inf"))
    sentinel_end   = ("a_end", float("inf"), float("inf"))
    return [sentinel_start] + acts + [sentinel_end]

acts = add_sentinels(activities)

names = [x[0] for x in acts]
s = [x[1] for x in acts]
f = [x[2] for x in acts]

n = len(acts) - 2  # original n
(n, names[:3], names[-3:])
```

出局[6]：

```
(11, ['a0', 'a1', 'a2'], ['a10', 'a11', 'a_end'])
```

（3） 自下而上的DP（计票）

设为区间内兼容活动的最大数量（（a，p））。`dp[a][p] = c[a,p]`

我们通过增加间隔长度来填充。
我们还存储 = argmax （k） 以重建列表。`dp choice[a][p]`

伪代码：

```
for len = 2 .. N:
  for a:
    p = a + len
    dp[a][p] = 0
    choice[a][p] = None
    for k in candidates(a,p):
        cand = dp[a][k] + 1 + dp[k][p]
        if cand > dp[a][p]:
```

```
            dp[a][p] = cand
            choice[a][p] = k
```

In [8]:
```
def candidates_between(a, p, s, f):
    # return indices k where activity k can be scheduled between boundary a and p
    cand = []
    for k in range(a+1, p):
        if f[a] <= s[k] and f[k] <= s[p]:
            cand.append(k)
    return cand
```

In [11]:
```
total_bu, sel_bu, dp_table = bottomup_activity_select(s, f)
total_bu, [names[i] for i in sel_bu]
```

Out[11]:
```
(4, ['a1', 'a4', 'a8', 'a11'])
```

In [12]:
```
def greedy_schedule(activities):
    # activities: list of (name, start, finish)
    acts = sorted(activities, key=lambda x: x[2])  # sort by finish time
    selected = []
    last_finish = float("-inf")

    for name, start, finish in acts:
        if start >= last_finish:
            selected.append((name, start, finish))
            last_finish = finish

    return selected
```

In [13]:
```
greedy_sel = greedy_schedule(activities)

print("DP optimum count:", total_bu)
print("Greedy count    :", len(greedy_sel))
print("Greedy selected :", greedy_sel)

assert total_bu == len(greedy_sel)
print("DP and Greedy match ✅")
```

```
DP optimum count: 4
Greedy count    : 4
Greedy selected : [('a1', 1, 4), ('a4', 5, 7), ('a8', 8, 11), ('a11', 12, 16)]
DP and Greedy match ✅
```

Greedy Optimality Proof (Earliest Finish Time)

Claim. The algorithm that repeatedly selects the compatible activity with the earliest finish time returns a maximum-size set of mutually compatible activities.

Key Lemma (Greedy Choice Property / Exchange Argument)

Let be the first activity chosen by Greedy, i.e., the one with the **earliest finish time** among all activities. Let be any optimal solution, and let be the first activity in (the one that finishes first inside). `g1 O o1 O O`

Because has the earliest finish time overall, we have: [f(g_1) \le f(o_1).] `g1`

Now construct a new schedule: [O' = (O \setminus {o_1}) \cup {g_1}.] is still feasible: since finishes no later than , every activity in that starts after also starts after . So has the **same number of activities** as , hence is also optimal and **starts with `g1`**. `O' g1 o1 O o1 g1 O' O O'`

Optimality by Induction

After choosing , the remaining problem is to select the maximum number of activities that start after . By the lemma, there exists an optimal solution whose first choice is , and the subproblem has the same form as the original one. Applying the same argument recursively, Greedy produces an optimal solution for every subproblem. `g1 f(g1) g1`

Therefore, GreedySchedule is optimal for the unweighted activity-selection problem. ✅

Conclusion

- DP (top-down / bottom-up) computes the optimal value using optimal substructure and overlapping subproblems.
- Greedy (earliest finish time) achieves the same optimum for the unweighted case, and is provably optimal.

```
In [ ]:
```