exercise4.

Problem 1. Courses Allocation: Dynamic Programming VS Greedy Computing.

1. Yes, the problem exhibits optimal substructure, so DP can be used. If the optimal solution A_{ap} (between activities a_a and a_p) includes some activity a_k, then a_k divides the problem into two independent subproblems:

   Left: activities between a_a and a_k → solved by $c[a,k]$
   Right: activities between a_k and a_p → solved by $c[k,p]$

   The total is: $|A_{ap}| = |A_{ak}| + |A_{kp}| + 1$ (the "+1" counts a_k itself)
   If A_{ak} were not optimal for $\hat{S}_{ak}$, we could swap in a better solution A'_{ak}, getting $|A'_{ak}| + |A_{kp}| + 1 > |A_{ap}|$
   this contradicts the assumed optimality of A_{ap}. The same argument applies to A_{kp}.
   (DP works need two conditions:)
   ① Optimal substructure → optimal solutions to subproblems combine into an optimal overall solution
   ② Overlapping subproblems → when trying different choice of a_k, the subproblems $c[a,k]$ and $c[k,p]$ appear repeatedly, so memoization avoids redundant computation.

2. use memoization (top-down). Add two sentinel activities: a_0 with $f[0]=0$ (finishes before everything) and a_{n+1} with $s[n+1]=\infty$ (starts after everything). For each subproblem (a,p) we try every compatible activity a_k in between, recursively solve both sides, and pick the best.

   Re
   Visualizing the Subproblem Splitting: Original Problem: Find compatible activities in range (a, P)

```
                        (a, P)
                       /  |   \
choose a_k :      (a,k)  a_k  (k,P)
                  [left]  1   [right]
```

```
RecuSelect (s, f, a, p):
    if (a, p) in memo: return memo[(a,p)] //already computed. reuse.
    // find compatible activities between a_a and a_p.
    Ŝ_ap = {k : a<k<p, f[a] ≤ s[k] and f[k] ≤ s[p] }

    if Ŝ_ap is empty:        // no compatible activity
        memo[(a,p)] = []
        return []

    best = []
    for each k in Ŝ_ap :    // try each compatible a_k.
        left = RecuSelect (s, f, a, k)  // solve left subproblem
        right = RecuSelect (s, f, k, p) // solve right subproblem
        candidate = left + [a_k] + right  // combine
        if |candidate| > |best| :
            best = candidate

    memo[(a, p)] = best
    return best
```

Initial call: RecuSelect $(s, f, 0, n+1)$

Complexity: Time $O(n^3)$ — $O(n^2)$ subproblems,

each tries $O(n)$ choices. Space $O(n^2)$ for memo table.

3. fill a 2D table c[a][p] from small subproblems to large ones. We also. keep an act [a][p] table to record which a_k was chosen, so we can reconstruct the actual solution afterwards.

```
TabulationSelect (s, f, n):
    c[0..n+1][0..n+1] = 0 //c[a][p] = max number of compatible activities
    act [0..n+1][0..n+1] = 0 // act[a][p] = chosen activity index
    for length = 2 to n+1 :  // increasing subproblem size
        for a=0 to n+1 - length : //start index
            p = a + length   //end index
            for k = a+1 to p-1 : //try each activity in between
                if f[a] ≤ s[k] and f[k] ≤ s[p] : // a_k is compatible
                    if c[a][k] + c[k][p] +1 > c[a][p] : //better solution found
                        c[a][p] = c[a][k] + c[k][p] +1
                        act[a][p] = k
    return Reconstruct (act, 0, n+1)
Reconstruct (act, a, p) :  // trace back to get the activity list
    k = act[a][p]
    if k == 0 : return []
    return Reconstruct (act, a, k) + [a_k] + Reconstruct (act, k, p)
```

Complexity : Time $O(n^3)$ — three nested loops. ~~Space O(n^2)~~

Space $O(n^2)$ — for c and act tables.

Greedy Approach:

I. Greedy Choice

The Greedy choice is: always select the activity with the earliest finish time that is compatible with the last selected activity.

Choosing the earliest-finishing activity leaves the most remaining time for future activities, maximizing the total number we can fit.

II. First sort by finish time (the problem states this is already done). The scan through activities, greedily picking each one that starts after the last selected activity finishes.

```
GreedySchedule(s, f, n):
    Sort activities by finish time: f[1] ≤ f[2] ≤ ... ≤ f[n]

    A = {a₁}    // always pick the first activity (earliest finish)
    k = 1       // index of last selected activity

    for i = 2 to n:
        if s[i] ≥ f[k]: //activity i starts after last selected finishes
            A = A ∪ {aᵢ} // select it
            k = i  // update last selected
    return A
```

Complexity: $O(n\log n)$ for sorting $+ O(n)$ for selection $= O(n\log n)$

much faster than DP's $O(n^3)$

III: Proof

Method 1: Induction.

Let $G = \{g_1, \dots, g_m\}$ be the greedy solution, $O = \{o_1, \dots, o_{m^*}\}$ any optimal solution.

g_1 has the earliest finish time among all activities, so $f(g_1) \le f(o_1)$
we can replace o_1 with g_1 in O — the solution remains valid and optimal.

Inductive step:

Assume $\{g_1, \dots g_k\}$ can be extended to an optimal solution. The greedy algorithm picks g_{k+1} with the earliest finish time after g_k, so $f(g_{k+1}) \le f(O_{k+1})$. Replacing O_{k+1} with g_{k+1} in O preserves validity (all subsequent activities remain compatible since g_{k+1} finishes no later). Thus $\{g_1, \dots, g_{k+1}\}$ can also be extended to an optimal solution.

2. Stay-Ahead Argument.

$f(g_i) \le f(O_i)$ for all i — greedy never finishes later than optimal at each step.

$f(g_1) \le f(O_1)$ since g_1 has the globally earliest finish time.

Inductive step:

Assume $f(g_i) \le f(O_i)$. Then $s(O_{i+1}) \ge f(O_i) \ge f(g_i)$
so O_{i+1} is also compatible with g_i. Since greedy picks the earliest compatible finish time, $f(g_{i+1}) \le f(O_{i+1})$

So since greedy stays ahead at every step, $|G| \ge |O^*|$.
If $|G| < |O^*|$, then after g_m there would still be a compatible activity — but greedy would have selected it.
Therefore $|G| = |O^*|$.

3. Contradiction.

Assume greedy is not optimal: $|G| = m < m^* = |O^*|$. Let j be the first index where $g_j \ne O_j$. Since greedy picks the earliest finish, $f(g_j) \le f(O_j)$.
Replace O_j with g_j in O — still valid, same size. Repeat this exchange until O agrees with G on all m positions. Since $m < m^*$, O has extra activities after g_m that are compatible. But greedy would have selected them (it stops only when nothing is compatible) — contradiction. Therefore $m = m^*$. #