Problem 1: Holiday Planning

To compute the maximum number of attractions one can visit a dynamic programming approach is used.

The function

```
fn holiday_planning(its: Vec<Vec<usize>>, c: usize, d:usize) -> usize
```

takes as parameters:

- `itineraries:` the `c` itineraries, one for each of the `c` cities that may be visited
- `c` : the number of cities
- `d` : the number of days available

and it iteratively computes the maximum number of attractions.

The function behaves as follows:

- create the dynamic programming array `asf` (acronym for *attractions so far*)
  - `asf[i]` is the maximum number of attractions seen up to the day `i`
- we initialize `asf` as the prefix sum array of the first city
- for each itinerary:
  - create the array `asf_curr_city` , which is the prefix sum of attractions that can be seen in the current city
  - create the array `curr_asf` , which is the dynamic programming array that will be built considering the current city as a possible destination
  - for each day `day`
    - for the number of days (`prev_days`) before the current day (`day`)
      - we compute the maximum number of attractions, considering the current city, setting `curr_asf[day]` as the maximum between
        - `curr_asf[day]` : maximum number of attractions seen if there are no days spent in the current city
        - `asf_curr_city[prev_days] + asf[day - prev_days]`
          - `asf_curr_city[prev_days]` : max number of attractions seen in the current city if `prev_days` are spent there
          - `asf[day - prev_days]` : max number of attractions seen the remaining days up to the day `day` in the previous "optimal" city
    - `asf = curr_asf` : "update" the dynamic programming array with the newly computed one

Time Complexity: $O(\text{cities} \cdot \text{days}^2)$

- three nested loops:
  - the outer loop iterates over each itinerary (one for each city)
  - the middle loop iterates over each day `day` from 0 to the number of available days
  - the inner loop iterates from 0 to `day`

Space Complexity: $O(\text{days})$

- three support arrays (`asf`, `curr_asf`, `asf_curr_city`) of length $\text{days} + 1$ are used

Problem 2: Design Course

To compute the maximum number of topics that can be taught in a course an adaptation to the solution of the "Longest Increasing Subsequence" is employed.

The function:

```
fn design_a_course(mut topics: Vec<(u32, u32)>) -> u32
```

takes as a parameter the array of topics, represented as pairs `(beauty, difficulty)` and returns the solution.

The function behaves as follows:

- sort the topics by their beauty
- computes the longest increasing subsequence by the difficulty, using dynamic programming:
  - define the array `lis`, where `lis[i]` is the *LIS* of the prefix `[0..i-1]` of the array
  - for each current topic i
    - for each topic j that comes before the topic i
      - check if:
        - the current topic i is harder than the topic j (the pedagogical constraint)
        - the current topic i is more interesting than the topic j (the students "pickiness" constraint)
        - the *LIS* of topics that ends with the topic i is smaller that the *LIS* of topics that ends in j plus the topic i (DP recurrence)
      - then `lis[i] = lis[j] + 1`
  - return the maximum value in `lis`

Time Complexity: $O(n^2)$

- Two nested loops: the outer loop runs for each topic, and the inner loop compares the current topic with all previous topics
  - inside the inner loop is performed a constant-time operation

Space Complexity: $O(n)$

- use an additional array of length n, `lis`, to compute the result