Formulaire Dyn Prog

========== Coin Change ==========

```python
def coinChange_lecture(cs: List[int], S: int) -> list[list[int]]:
    """S: goal,  cs: set of coins"""
    if S == 0 or not cs: return [[]]
    A = [[0] * len(cs) for _ in range(S)]

    def solve_before(s: int):
        for i, c in enumerate(cs):
            if c == s:
                A[s - 1][i] = 1
                return A[s - 1][:]
        tmp: List[List[int]] = []  # we have A[0] -> A[s-1] => find A[s]
        hi = s - 1  # i, hi => 2 pointers on 0 -> s/2 (i), s -> s/2 (hi)
        # e.g. 2: 1 + 1, 3: 1 + 2, 2 + 1
        for i in range(ceil(s / 2)):
            tmp.append([amnt_i + amnt_hi for amnt_i, amnt_hi in zip(A[i], A[hi - 1])])
            hi -= 1
        sol = min(tmp, key=lambda lst: sum(lst))  # min amount of coin used, is min of the sum of each element
        A[s - 1] = sol
        return sol[:]

    for s in range(1, S + 1): solve_before(s)
    return A
```

=== Follow dp template in formulaire ===

```python
def coinChange_classic_dp(coins: list[int], goal: int) -> list[list[int]]:
    if not coins: return [[]]
    K, N = len(coins), goal
    # (N + 1): 0 is necessary for B.S. & I.S. [0..0]
    DP = [[0 for _ in range(K)] for _ in range(N + 1)]

    def incr_ret(lst: list[int], idx: int):
        tmp = lst.copy()
        tmp[idx] += 1
        return tmp

    # DP[k] represents the optimal amount of change that sums up to k
    for k in range(1, N + 1):
        DP[k] = (
            min(
                (incr_ret(DP[k - c_i], i) for i, c_i in enumerate(coins) if k - c_i ≥ 0),
                key=lambda lst: sum(lst),
            )
        )
    return DP
```

other version for `for k loop`

```python
    for k in range(1, N + 1):
        for i, c_i in enumerate(coins):
            if k - c_i ≥ 0:
                DP[k] = min(DP[k], incr_ret(DP[k - c_i], i), key=lambda lst: sum(lst))
            else:
                break
```

========== Min Cost climbing stairs (dp tuto)==========

You are given an integer array `cost` where `cost[i]` is the cost of `i`th step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index `0`, or the step with index `1`.

Return *the minimum cost to reach the top of the floor*.

Brute-force

```python
# HINT Brute Force
def mCCS_bf(self, cost: List[int]) -> int:
    N, j_m = len(cost), 2  # max stair index, max jump
    cost.append(0)
    calls: dict[int, int] = {}

    def F(k: int) -> int:
        if k <= 0: return cost[k]  # B. S.
        if k in calls: calls[k] += 1
        else: calls[k] = 1
        return cost[k] + min((F(k - j) for j in range(1, j_m + 1)))  # I. H.
    # expo. growth rate of calls (fibonacci to be precise, i.e.e 7:3, 6: 5, 4: 13 ... )
    return min(F(N - 1), F(N - 2))  # we can start at 0 or 1
```

Memoization:

```python
# HINT Memoization (cache for recursive calls, Top-Down)
def mCCS_memo(self, cost: List[int]) -> int:
    N, j_m = len(cost), 2  # max stair index, max jump
    cache = [-1] * N
    cost.append(0)

    def F(k: int) -> int:
        if k <= 0: return cost[k]  # B. S.
        if cache[k] != -1: return cache[k] # memoization

        min_cost: int = cost[k] + min((F(k - j) for j in range(1, j_m + 1)))  # I. H.
        cache[k] = min_cost
        return min_cost

    return min(F(N - 1), F(N - 2))  # we can start at 0 or 1

# HINT Dynamic Programming (Bottom-Up)
def minCostClimbingStairs(self, cost: List[int]): # type: ignore
    cost.append(0)
    N = len(cost)
    j_m, dp = 2, [float('inf')] * N
    dp[0], dp[1] = cost[0], cost[1]
    for k in range(2, N):
        # choose the min cost from 0 to stair k
        # knowing the optimal costs from 0 to stairs 1 to k-1
        for j in range(1, j_m + 1):
            dp[k] = min(dp[k], cost[k] + dp[k - j])
        """ NB. for loop is not *absolutely* necessary:
            dp[k] = cost[k] + min([dp[k - j] for j in range(1, j_m + 1)])"""
    return min(dp[-1], dp[-2])
```

========== **Floyd (all pair shortest path : train)** ==========

```python
from copy import deepcopy
from math import inf, isfinite
def dp_all_pair_sp(V: list[int], L: list[list[int | float]]) -> list[list]:
    """Dyn-Prog sol to all pair shortest path problem,
    V: list of vertices, L: adjencency matrix (with weights)
    (non-existant edges must be present with weight inf)"""
    N, D = len(V), deepcopy(L)
    P = [[[i + 1] for _ in range(N)] for i in range(N)]
    # basis step for P
    for i, row in enumerate(D):
        for j, val in enumerate(row):
            if isfinite(val) and j != i:  # edge (i, j) exists
                P[i][j].append(j + 1)
    # I. S.
    for k in range(N):
        for i in range(N):
            for j in range(N):
                crt, candidate = D[i][j], min(D[i][j], D[i][k] + D[k][j])
                if candidate < crt:
                    P[i][j] = P[i][k] + P[k][j][1:] # dont add k twice
                    D[i][j] = candidate
        mprint(P)
    return D
```

============== House Robber ===========

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight* ***without alerting the police***.

```python
def rob2(self, nums: list[int]):
    if not nums: return 0
    if len(nums) == 1: return nums[0]
    N = len(nums)
    dp: list[list[int]] = [[]] * len(nums)

    dp[0], dp[1] = [0], [1] # store index of steps used to avoid reusing them
    # because we cant rob 2 time same house
    print(nums, "\n")
    for k in range(2, N):
        max_mon = max((dp[k - j] for j in range(2, k + 1)), key=lambda lst: sum(nums[i] for i in lst))
        dp[k] = max_mon + [k]

    return (out := max(dp[-1], dp[-2], key=lambda lst: sum(nums[i] for i in lst)), sum(nums[i] for i in out))

def rob(self, nums: list[int]):
    if not nums: return 0
    if len(nums) == 1: return nums[0]

    dp: list[int] = [0] * len(nums)
    N, dp[0], dp[1] = len(nums), nums[0], nums[1]

    for k in range(2, N):
        dp[k] = nums[k] + max(dp[k - j] for j in range(2, k + 1))

    return max(dp[-1], dp[-2])
```

(rob2 is just attempt at not reusing same idx but algo not done)

=== Tribonacci ====

```python
def tribonacci(self, n: int) -> int:
    if n < 2: return n
    if n == 2: return 1

    cache = [0]*(n + 1)
    cache[0], cache[1], cache[2] = 0, 1, 1

    for i in range(3, n + 1):
        # if not computed yet
        if cache[i] == 0:
            cache[i] = cache[i - 1] + cache[i - 2] + cache[i - 3]
        print(cache)

    return cache[n]
```

At least two types of dynamic programming problems can be identified.
Problems such as calculating Fibonacci numbers, or binomial coefficients, are usually specified directly by their recurrence relation: fib n = fib (n-1) + fib (n-2) or binom n k = binom n (k-1) + binom (n-1) (k-1).
Other DP problems, such as the coin change problem, the knapsack problem, or the optimal bracketing problem, are called 'combinatorial optimization problems'. Their specification is often given more broadly by
A) a set of potential candidate solutions,
B) a condition that must be met by a candidate solution, and finally
C) a measure by which the best candidate solution must be chosen (minimizing or maximizing the measure).
In these problems, a recurrence relation tells us how to build partial candidate solutions of some size given we have calculated the partial candidate solutions of all sizes smaller. Your job is then to figure out how to use these (not necessarily optimal) smaller candidate solutions to build a single bigger one (called 'incrementing' a candidate solution). How this is done varies from problem to problem, and, without a more formal treatment of the subject, the best approach is to practice with example problems.

Here are some questions you may want to ask yourself when thinking about such a problem:

- *What does a partial candidate solution look like?*
  In the (0/1) knapsack problem, it is simply a smaller knapsack containing only processed items, which is still a valid solution. But in the coin change problem, a partial candidate solution is a selection of coins which don't make the right change, and so you still have to do more work to get to a valid solution.

- *What options are there for incrementing the partial candidate solutions, and do they satisfy the condition imposed on candidate solutions?*
  In the knapsack problem, for each item processed, there are two options: include it or exclude it from the knapsack in question. But if you include it, you must make sure the added weight does not exceed the capacity. After this combination process, this is where your recurrence relation will, in most cases, have a min or max function applied to the incremented candidate solutions.

- *What parameters can be used to define the state of the algorithm? When do they indicate that it has finished?*
  This is an important one. All tabulation methods (the specific technique here referred to as dynamic programming) produce tables (or compute them). The size of these tables is proportionate to the parameters that define the problem, and so must be used in the recurrence relation. In the 0/1 knapsack problem, the parameters of the algorithm are the number of items that remain to be processed, and the remaining capacity (M(i, w)), with the algorithm being finished when we have calculated M(0, W) (where W is the maximum capacity given as input). This will give you an extra clue of which direction the parameters in the recursive calls should go in.