Formulaire Greedy + Autres

========== Kruskal: ==========

```python

def in_tree(v: int, tree: set[tuple[int, int]]) -> bool:
    for edge in tree:
        if v in edge: return True
    return False

def find_tree(v: int, trees: list[set[tuple[int, int]]]) -> int: #set[tuple[int, int]]:
    for i, t in enumerate(trees):
        if in_tree(v, t):
            return i
    s = f"{v} should be in trees since it contains all edges"
    raise ValueError(s)

def merge_tree(t1_idx: int, t2_idx: int, trees: list[set[tuple[int, int]]]):
        t1, t2 = trees[t1_idx], trees[t2_idx]
        trees.remove(t1)
        trees[t2_idx] = t1.union(t2)


def kruskal(A: list[list[int]]) -> set[tuple[int, int]]:  # sourcery skip: identity-comprehension
    """A: adjacency matrix, Return: MST"""
    N = len(A)
    if not A or not A[0]: return set()
    if N == 1: return {(0, 0)}

    trees, S = [], []  # S: sorted list of edges (w.r.t. weight)
    trees: list[set[tuple[int, int]]] = []
    for i in range(N):
        # list of trees at start : list of trees of length 1 i.e. each edge in a singleton
        for j in range(N):
            if A[i][j] == 0: continue  # (no edge)
            trees.append({(i, j)})
            S.append([(i, j), A[i][j]])

    S.sort(key=lambda kv: kv[1], reverse=True)  # min should be last
    F: set[tuple[int, int]] = set()  # forest which will hold/be the MST
    while S and len(F) < N:
        e, cost = S.pop()
        v1, v2 = e
        t1_idx, t2_idx = find_tree(v1, trees), find_tree(v2, trees)
        if trees[t1_idx] != trees[t2_idx]:
            F.add(e)
            merge_tree(t1_idx, t2_idx, trees)
    return F
```

2sum: "2 pointers"

```python
# find indices i,j such that nums[i] + nums[j] = target
def twoSum(self, nums: List[int], target: int) -> List[int]:
    left, right = 0, len(nums) - 1
    while nums[left] + nums[right] != target:
        if nums[left] + nums[right] < target:
            left += 1
        else:
            right -= 1
    return [left + 1, right + 1]
```

========== DFS & BFS: ==========

```python
class Node:
    """Node of binary tree. i.e. has 2 children until max depth is atteined"""
    MAX_DEPTH = 4  # hard-coded for test purposes
    # not max_depth -1 bcs values starts at 2**0 (1 | 2, 3 | 4, 5,6,7) so we added +1 as well
    MAX_VAL = 2 ** (MAX_DEPTH) - 1
    def __init__(self, val: int) -> None: self.val = val
    def get_adjacents(self) -> List[Self]:
        """Return: nodes adjacent to `self`. (i.e. children for a tree)"""
        val_left, val_right, val_top = self.val * 2, self.val * 2 + 1, max(self.val // 2, 1)  # last make graph undirecte
        return [] if val_right > Node.MAX_VAL else [Node(val_top), Node(val_left), Node(val_right)]


def BFS(root: Node, goal: int) -> Optional[Node]:
    visited: Set[int] = set()
    queue: List[Node] = [root]  # emulate queue behavior with list
    while len(queue) > 0:
        crt = queue.pop(0)  # queue => First in, First Out
        visited.add(crt.val)
        if crt.val == goal: return crt
        for node in crt.get_adjacents():
            # avoid cycle
            if node.val not in visited: queue.append(node)
            # adds to end => prioritize node on same depth
    return None

def DFS2(root: Node, goal: int) -> Optional[Node]:
    visited: Set[int] = set()
    stack: List[Node] = [root]
    while len(stack) > 0:
        crt = stack.pop(-1)
        visited.add(crt.val)
        if crt.val == goal: return crt
        for node in crt.get_adjacents():
            if node.val not in visited: stack.append(node)
    return None

def graph_traversal(root: Node, goal: int, is_bfs: bool):
    """if `is_bfs` search for `goal` with BFS else do it with DFS """
    visited: Set[int] = set()
    live_nodes: List[Node] = [root]
    def queue(): return live_nodes.pop(0)  # First In First Out (BFS)
    def stack(): return live_nodes.pop(-1) # First In Last Out (DFS)
    next_node = queue if is_bfs else stack
    while len(live_nodes) > 0:
        crt = next_node()
        visited.add(crt.val)
        if crt.val == goal: return crt
        for node in crt.get_adjacents():
            if node.val not in visited: live_nodes.append(node)
    return None
def DFS(root: Node, goal: int) -> Optional[Node]:
    out: Optional[Node] = None
    visited: Set[int] = set()
    def rec(crt: Node):
        nonlocal out
        visited.add(crt.val)
        for node in crt.get_adjacents():
            if node.val == goal:
                out = node
                return
            if node.val not in visited: rec(node)
            if out is not None: return
    rec(root)
    return out
if __name__ == "__main__":
    root = Node(1)
    found = graph_traversal(root=root, goal=16, is_bfs=True)
    print("Not Found" if found is None else f"Found: {found.val}")
```

```python
def dijkstra(V: list[int], M: list[list[int | float]], start: int, goal: int):
    """start, goal: indices of start and goal vertex in V"""
    N = len(V)
    C, visited = [inf] * N, set()
    C[start] = 0
    path: list[int] = []
    pq: list[tuple[int, int]] = [(start, 0)] # node, cost

    def add_priority(node: int, cost: int):
        pq.append((node, cost))
        for i in range(len(pq) - 1, 0, -1):
            crt, next = pq[i], pq[i - 1]
            if crt[1] > next[1]: pq[i], pq[i - 1] = next, crt
            else: break

    def get_adjacent(node: int) -> list[int]:
        return [i for i in range(N) if i != node and isfinite(M[node][i])]

    enode = None
    while len(pq) > 0:
        enode = pq.pop()[0]
        if enode in visited: continue
        visited.add(enode)
        if enode == goal: return C, path
        for node in get_adjacent(enode):
            if node in visited: continue
            cost = M[enode][node] + C[enode]
            if cost < C[node]:
                C[node] = cost
                add_priority(node, cost) # type: ignore
    return C, path
```

================ Dijkstra ================

```python
def dijkstra(V: list[int], M: list[list[int | float]], start: int, goal: int):
    """start, goal: indices of start and goal vertex in V"""
    N = len(V)
    C, visited = [inf] * N, set()
    C[start] = 0
    path: list[int] = []
    pq: list[tuple[int, int]] = [(start, 0)] # node, cost

    def add_priority(node: int, cost: int):
        pq.append((node, cost))
        for i in range(len(pq) - 1, 0, -1):
            crt, next = pq[i], pq[i - 1]
            if crt[1] > next[1]: pq[i], pq[i - 1] = next, crt
            else: break

    def get_adjacent(node: int) -> list[int]:
        return [i for i in range(N) if i != node and isfinite(M[node][i])]

    enode = None
    while len(pq) > 0:
        enode = pq.pop()[0]
        if enode in visited: continue
        visited.add(enode)
        if enode == goal: return C, path
        for node in get_adjacent(enode):
            if node in visited: continue
            cost = M[enode][node] + C[enode]
            if cost < C[node]:
                C[node] = cost
                add_priority(node, cost) # type: ignore
    return C, path
```

Formulaire D&C

========== Majority Element ==========

Given an array `nums` of size `n`, return *the majority element*.

The majority element is the element that appears more than `⌊n / 2⌋` times. You may assume that the majority element always exists in the array.

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        def G(nums): return nums[0]
        if len(nums) == 1: return G(nums)
        N = len(nums)
        def is_majority(m):
            nonlocal N
            count = 0
            for k in nums:
                if  k == m:
                    count += 1
                    if count > N / 2: return True
            return False

        def REDUCE(crt_list: List[int]):
            out = []
            for i in range(0, len(crt_list) - 1, 2):
                crt = crt_list[i]
                if crt == crt_list[i+1]: out.append(crt)
            return out


        def me_rec(crt: List[int]):
            n = len(crt)
            if n == 1: return G(nums)
            # if length of crt is odd ⇒ check if crt[-1] is majority element
            # if its not ⇒ rec call for reduce(crt)
            if n % 2 == 1:
                last = crt[-1]
                if is_majority(last): return last
                else: crt.pop()
            return me_rec(REDUCE(crt))
        # REDUCE can create a majority element but can never remove one.
        # hence ⇒ check if one returned by me_rec was created or is genuine
        candidate = me_rec(nums)
        return candidate if is_majority(candidate) else None
```

(leetcode version):
(ça marche vraiment)

```python
from math import ceil
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        if len(nums) == 1: return nums[0]
        nums.sort()
        ok = len(nums)
        return nums[ok//2]
```

Bin search on non sorted list: (i.e. keep idx then sort)

```python
# find indices i,j such that nums[i] + nums[j] = target
def twoSum(self, nums: List[int], target: int) -> List[int]:  # noqa: E999
    numss = sorted([(idx, x) for idx, x in enumerate(nums)], key=lambda x: x[1])
    pairs = set()
    # iterate on each element i and binary search on target - nums[i]
    for idx, xi in numss:
        to_search = target - xi
        crt_pair = (xi, to_search)
        if crt_pair not in pairs: found_idx = self.bin_search(numss, to_search)
        if found_idx is not None and numss[found_idx][0] ≠ idx:
            return [idx, numss[found_idx][0]]
        else:
            pairs.add(crt_pair)
            pairs.add((to_search, xi))
    return [None, None]
```

========== Max Bin Tree ==========

You are given an integer array `nums` with no duplicates. A **maximum binary tree** can be built recursively from `nums` using the following algorithm:

1. Create a root node whose value is the maximum value in `nums`.
2. Recursively build the left subtree on the **subarray prefix** to the **left** of the maximum value.
3. Recursively build the right subtree on the **subarray suffix** to the **right** of the maximum value.

Return *the **maximum binary tree** built from* `nums`.

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val; self.left = left; self.right = right
class Solution:
    def _max(self, nums: List[int]) -> Tuple[int, int]:
        crt_idx, crt = 0, nums[0]
        for i, x in enumerate(nums):
            if x > crt: crt_idx, crt = i, x
        return crt_idx, crt

    def constructMaximumBinaryTree(self, nums: List[int]) -> Optional[TreeNode]:
        def rec(acc: List[int]):
            if acc is None or len(acc) == 0: return None
            idx_max, vmax = self._max(acc)
            return TreeNode(vmax, rec(acc[:idx_max]), rec(acc[idx_max + 1 :]))
        return rec(nums)

    # IN PLACE:
    def constructMaximumBinaryTree2(self, nums: List[int]) -> Optional[TreeNode]:
        def _max(p: int, q: int):
            crt_idx, crt = p, nums[p]
            for i in range(p + 1, q + 1):
                x = nums[i]
                if x > crt: crt_idx, crt = i, x
            return crt_idx, crt

        def rec(p: int, q: int):
            if p ≥ q:
                if p == q: return TreeNode(nums[p])
                return None
            idx_max, vmax = _max(p, q)
            return TreeNode(vmax, rec(p, idx_max - 1), rec(idx_max + 1, q))

        return rec(0, len(nums) - 1)
```

========== Binary Search ==========

```python
def binary_search(self, nums: List[int], goal: int) -> Optional[int]:
    if not nums: return None

    def bs_rec(p: int, q: int):
        if p ≥ q: return p if nums[p] == goal else None
        m = (p + q) // 2
        val = nums[m]
        if val < goal: return bs_rec(m + 1, q)
        elif val > goal: return bs_rec(p, m)
        else: return m
    return bs_rec(0, len(nums) - 1)
```

Formulaire BT

========== PowerSet BT ==========

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        out = [[]]  # noqa: F841
        def max_k(k: Tuple[int, int]): return max(k[0], k[1])

        def T(x: List[List[int]], k:Tuple[int, int], N: int):
            return list(range(max_k(k), N))

        def recBT(x: List[List[int]], k:Tuple[int, int], path:List[int], N: int):
            # we go through nums in increasing order of indices
            if max_k(k) ≥ N: return # if we arrived at the end of said indices ⇒ stop
            path.append(None) # ⇒ else store (or not) elements starting from max(k, old_i) (old_i is the iteration index with which recBT was called)
            _T = T(x, k, N)
            for i in _T:
                path[-1] = nums[i]
                if nums[i] not in path[:-1] and path not in out: # doesnt cover permutation i.e. [1,2,3] and [1, 3, 2]
                    out.append(path.copy())
                    print(path.copy())
                    recBT(x, (k[0] + 1, i), path.copy(), N)

        recBT([[]], (0, 0), [], len(nums))
        return out
```

========== N-Queens ==========

```python
def solveNQueens(self, n: int) -> List[List[str]]:
    if n < 3: return [[]]
    out = []
    def is_same_diag(coord1: Tuple[int, int], coord2: Tuple[int, int]) -> bool:
        # c1 same diag c2 ⟺ (z:= c1-c2 ⇒ z_1 = z_2)
        z = (coord1[0] - coord2[0], coord1[1] - coord2[1])
        return z[0] == z[1]

    def is_same_anti_diag(coord1: Tuple[int, int], coord2: Tuple[int, int]) -> bool:
        # c1 same anti_diag c2 ⟺ (z:= c1-c2 ⇒ z_1 = -z_2)
        z = (coord1[0] - coord2[0], coord1[1] - coord2[1])
        return z[0] == -z[1]

    # position at x[i] indicate column of the queen on the i-th row
    # indicates available columns-indices at step k given moves at steps 0..k-1
    def T(x: List[int], k, N):
        base = set(range(N))
        # remove those that would induce colum collision
        for i in range(k): base.discard(x[i])  # if not present do nothing
        return base

    def B(x: List[int], k: int, N: int):
        tocheck = x[k]
        coord_1 = k, tocheck
        for i in range(k):
            crt = x[i]
            # coord_1, coord_2 = (k, tocheck), (i, crt)
            coord_2 = i, crt
            if is_same_diag(coord_1, coord_2) or is_same_anti_diag(coord_1, coord_2):
                return False
        return True

    def P(x: List[int], k, N):
        if x[0] is None: return False
        # At each step check for diag collision between x[i] and each other x[i-j] for all j < i
        queen = 0
        for i, xi in enumerate(x):
            if i > k: break  # iterate through x[0] → x[k] (included)
            if xi is None: return False
            if not B(x, i, N): return False
            queen += 1
        return queen ≥ N # return if enough queen were in x

    def bt_rec(x: List[List[int]], k: int, N: int):
        nonlocal out
        if k ≥ N: return
        for col in T(x, k, N):
            x[k] = col
            if B(x, k, N):
                if P(x, k, N): out.append(x[: k + 1].copy())
                bt_rec(x, k + 1, N)

    bt_rec([None]*n, 0, n)
    return self.to_dum_formatting(out, n)
```

======== 2 sum BT ================

```python
class Solution:
    def twoSum(self, nums: list[int], target: int) -> list[int]:
        S_i = set(range(len(nums)))
        pairs: dict[int, set[int]] = dict()
        for i in S_i:
            pairs[nums[i]] = set()

        def T(x, k, N):
            used = [] if x[0] is None else pairs[nums[x[0]]]
            out = S_i.difference(set(x[:k]).union(used))
            if x[0] is not None and x[k] is not None:
                pairs[nums[x[0]]].add(nums[x[k]])
            return out

        def sumk(x, k):
            summ, i = 0, 0
            for xi in x:
                summ += nums[xi]
                i += 1
                if i > k: return summ
            return 0

        def B(x, k, N):
            summ = sumk(x, k)
            return summ ≤ target if target ≥ 0 else summ ≥ target

        def P(x, k, N):
            if k ≠ 1: return False
            summ = sumk(x, k)
            return summ == target
        out = None

        def rBT(x: list[int], k: int, N: int) -> list[int] | None:
            nonlocal out
            for y in T(x, k, N):
                # print(y)
                x[k] = y
                if B(x, k, N) and nums[x[k]] not in pairs[nums[x[0]]]:
                    # print("")
                    if P(x, k, N):
                        out = x[: k + 1]
                        return out
                    if k < N - 1:
                        rBT(x, k + 1, N)
                if out:
                    return out

        N = len(nums)
        return rBT([None] * N, 0, N)
```

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.

Formulaire B&B

========== **Shortest Path** ==========

```python
class Cell:
    def __init__(self, coords: Tuple[int, int], board: List[List[int]]) -> None:
        row, col = coords
        self.row = row
        self.col = col
        self.free = board[row][col] == 0

    def __eq__(self, __value: object) -> bool:
        if not isinstance(__value, Cell): return False
        return self.row == __value.row and self.col == __value.col

class Node:
    def __init__(self, pos: Cell, cost: int, state: List[Cell], last_move: Optional[int]) -> None:
        self.pos = pos
        self.cost = cost
        self.state = state.copy()
        self.last_move = last_move

    # Return: Depth i.e. nb of parent from root up to `self`
    def depth(self) -> int: return len(self.state)

def up(row: int, col: int): return row - 1, col
def right(row: int, col: int): return row, col + 1
def down(row: int, col: int): return row + 1, col
def left(row: int, col: int): return row, col - 1

UP, RIGHT, DOWN, LEFT = 0, 1, 2, 3
MOVES: Dict = {UP: up, RIGHT: right, DOWN: down, LEFT: left}


# Return: opposite of `move` i.e. LEFT ⇒ RIGHT...
def opposite(move: int): return move + 2 % len(MOVES)

def valid_moves(node: Node, N: int, M: int) -> Set[int]:
    """NxM: dimension of the board. i.e.  N:nb of rows, M: nb of columns.
    Return moves that would not cause an index out of bounds exception.
    (not guaranteed to be actually usable ⇒ doesnt check if next cell is free or not)"""
    out: Set[int] = set(range(UP, LEFT + 1))
    i, j = node.pos.row, node.pos.col
    if i ≥ N-1: out.discard(DOWN)
    if j ≥ M-1: out.discard(RIGHT)
    if i ≤ 0: out.discard(UP)
    if j ≤ 0: out.discard(LEFT)
    return out
```

```python
def listOfChildren(node: Node, board: List[List[int]]) -> Set[int]:
    not_out_of_bounds: Set[int] = valid_moves(node, len(board), len(board[0]))
    # remove last move to avoid having 2 consecutive moves cancelling each other
    if node.last_move is not None: not_out_of_bounds.discard(opposite(node.last_move))
    to_remove: Set[int] = set()
    cell = node.pos
    for move in not_out_of_bounds:
        new_row, new_col = MOVES[move](cell.row, cell.col)
        if board[new_row][new_col] == 1: to_remove.add(move) # if cell is blocked remove it

    return not_out_of_bounds.difference(to_remove)

def cost(cell: Cell, depth: int, goal: Cell) -> int:
    crt_row, crt_col = cell.row, cell.col
    goal_row, goal_col = goal.row, goal.col
    l1_norm = abs(crt_row - goal_row) + abs(crt_col - goal_col) # g(x) i.e. distance
    return depth + l1_norm

def addToLiveNodes(pq: List[Node], node: Node):
    """Emulate behavior of add PriorityQueue """
    pq.append(node)
    for i in range(len(pq) - 1, 0, -1): # iterate from the end
        crt, next = pq[i], pq[i - 1]
        if next.cost < crt.cost :
            pq[i], pq[i - 1] = next, crt # keep sorted decreasing order
        else: break # rest is sorted

def nextENode(pq: List[Node]) -> Node: return pq.pop()

def P(node: Node, goal: Cell): return node.pos == goal


def shortest_path(board: List[List[int]], _start: Tuple[int, int], _goal: Tuple[int, int]) -> Node:
    start, goal = Cell(_start, board), Cell(_goal, board)
    live_nodes = []
    root = Node(start, 0, [], None)
    enode = root
    while not P(enode, goal):
        for move in listOfChildren(enode, board):
            new_cell: Cell = Cell(MOVES[move](enode.pos.row, enode.pos.col), board)
            new_node = Node(new_cell, cost(new_cell, enode.depth() + 1, goal), [*enode.state, enode.pos], move)
            addToLiveNodes(live_nodes, new_node)

        enode = nextENode(live_nodes)
    return enode
```

========== Task scheduling ==========

Dans ce problème, on veut assigner n tâches à n agents, sachant que chaque agent est rémunéré et chaque tâche est assignée à un agent différent.

C'est un problème que l'on peut couramment rencontrer: assigner des emplacements à des bâtiments devant être construits, assigner des évènements à des organisteurs candidats, etc... La rémunération des agents peut être représentée par une matrice des coûts, où l'élément (i, j) représente le coût de l'exécution de la tâche i par l'agent j.

Le but de l'exercice est d'implémenter une stratégie Branch-and-Bound pour résoudre ce problème, avec la matrice des coûts comme paramètre. Pour ce faire, nous proposons de décrire un état, ou solution partielle, par la liste des tâches déjà assignées et des agents déjà assignés (on commence donc avec des listes vides). De cette manière, on peut définir le coût d'une solution partielle par la somme des coûts minimaux pour chaque tâche restante, augmentée du coût effectif des assignations effectuées jusque là. Prenons par exemple la matrice des coûts ci-dessous:

```python
from math import inf

COST_MATRIX = []
class Node:
    def __init__(self, cost: int, state: List[Tuple[int, int]]):
        """affectation: task_index, agent_index , state: list of tuple[task_index, agent_index]"""
        self.cost = cost
        self.state = state.copy()
    def affectation(self) -> Tuple[int, int]: return self.state[-1]

def cost(affectations: List[Tuple[int, int]], cost_mat: List[List[int]]) -> int:
    h = sum(cost_mat[task_idx][cost_idx] for (task_idx, cost_idx) in affectations)  # cost accumulated up to here
    used_idx = {affect[0] for affect in affectations}
    g, k = 0, len(affectations)
    for row in cost_mat[k:]:
        min_idx, min_cost = None, inf  # no minimum yet
        for a_idx, a_cost in enumerate(row):
            if a_cost < min_cost and a_idx not in used_idx:
                used_idx.discard(min_idx)  # agent `min_idx` isn't affected anymore
                min_idx, min_cost = a_idx, a_cost
                used_idx.add(a_idx)
        g += min_cost
    return g + h


def listOfChildren(parent: Node, cost_mat: List[List[int]]):
    old_affectations = parent.state
    used_agents = {affectation[1] for affectation in old_affectations}
    # agent already used in previous tasks
    N, task_index = len(cost_mat), parent.affectation()[0] + 1  # nb of agents & tasks
    free_agents = [idx for idx in range(N) if idx not in used_agents]
    return [
        Node(
            cost=cost(old_affectations + [(task_index, agent_idx)], cost_mat),
            state=old_affectations + [(task_index, agent_idx)],
        )
        for agent_idx in free_agents
    ]

def addToLiveNodes(node: Node, pq: List[Node]):
    """Add while maintining priority queue order"""
    pq.append(node)
    for i in range(len(pq) - 1, 0, -1):
        crt, next = pq[i], pq[i - 1]
        # smallest at the end
        if crt.cost > next.cost: pq[i], pq[i - 1] = next, crt
        else: break  # already sorted

def P(node: Node, cost_mat: List[List[int]]) -> bool:
    return len(node.state) == len(cost_mat)  # affectations completed?


def branch_bound(cost_mat: List[List[int]]) -> Tuple[List[Tuple[int, int]], int]:
    """Return: affectations, i.e. list of coordinates"""
    root_affect = 0, cost_mat[0].index(min(cost_mat[0]))
    root = Node(cost([root_affect], cost_mat), [root_affect])
    live_nodes = []
    enode = root
    while not P(enode, cost_mat):
        for node in listOfChildren(enode, cost_mat):
            addToLiveNodes(node, live_nodes)
        enode = live_nodes.pop()
        print(enode)
    return enode.state, enode.cost
```

=== Other interesting cost function ===

```python
def W(node_val: int):
    # w(i) => weight of node i. indexing starts at 2 (because node values starts at 1 and node 1 has no cost)
    if node_val - 2 ≥ len(WEIGHTS): print(node_val)
    return WEIGHTS[node_val - 2]

def cost(child_val: int, parent: Node):
    # computed as cost up to there (parent) + distance (child's weight) from Minimum weight
    return parent.cost + (W(child_val) - MIN_W)
```

========== 15 Puzzle (taquin) ==========

```python
class M:
    UP, RIGHT, DOWN, LEFT = 0, 1, 2, 3
    ALL = [UP, RIGHT, DOWN, LEFT]

def swap(board: List[List[int | None]], pos1: Tuple[int, int], pos2: Tuple[int, int]):
    r1, c1 = pos1
    r2, c2 = pos2
    tmp = board[r1][c1]
    board[r1][c1] = board[r2][c2]
    board[r2][c2] = tmp

def apply_move(board: List[List[int | None]], move: int, blank_pos: Tuple[int, int]):
    """Apply given move on board (switch relevant cells)
    and returns the new position of the whitespace"""
    mr, mc = 0, 0  # modifiers: row, column
    match move:
        case M.UP: mr -= 1
        case M.RIGHT: mc += 1
        case M.DOWN: mr += 1
        case M.LEFT: mc -= 1
    orow, ocol = blank_pos # old row old col
    nr, nc = orow + mr, ocol + mc
    swap(board, blank_pos, (nr, nc))
    return nr, nc

def count_misplaced(board: List[List[int | None]]) -> int:
    N, misp = len(board), 0
    for i, row in enumerate(board):
        for j, val in enumerate(row):
            if val is not None and i * N + j + 1 ≠ val: misp += 1
    return misp

def cost(board: List[List[int | None]], state: List[int]) -> int:
    """state: List of moves to apply to get the board from which to compute the cost
    start_pos: position of blank space for root node."""
    h = len(state)
    return h + count_misplaced(board)
```

```python
class Node:
    def __init__(self, state: List[int], board: List[List[int | None]], old_bpos: Tuple[int, int]):
        """
        state: list of moves (a move is an int in [0, 3]) to apply to attain
        this node in the game tree.
        NB: state[-1] is the last move made that "connects" to this node
        board: state of the board for parent/adjacent node
        old_bpos: position of the blank space in the parent config
        bpos: position of blank space in this config,
        bd: resulting board after having applied all moves in state"""
        bd = deepcopy(board)
        self.state = state[:]  # copy
        self.bpos = apply_move(bd, state[-1], old_bpos) if state else old_bpos
        self.bd = bd  # storing them all is at worst O(3^(n^2)) space
        self.cost = cost(self.bd, self.state)

    def last_move(self): return self.state[-1] if self.state else None
    def __repr__(self): return f"({moves_to_str(self.state)}, bpos={self.bpos}, cost={self.cost})"

def opposite(move: int) -> int:
    """Return: opposite of `move` i.e. the one that reverse it (UP ⟹ DOWN ...)"""
    return (move + 2) % len(M.ALL)


def next_move_not_out(node: Node) -> Set[int]:
    """Return: sets of next moves that won't cause index out of bounds"""
    N = len(node.bd)
    moves = set(M.ALL)
    row, col = node.bpos
    if row ≤ 0: moves.discard(M.UP)
    if col ≤ 0: moves.discard(M.LEFT)
    if row ≥ N-1: moves.discard(M.DOWN)
    if col ≥ N-1: moves.discard(M.RIGHT)
    return moves

def listOfChildren(node: Node) -> List[Node]:
    next_moves: Set[int] = next_move_not_out(node)
    last_move: Optional[int] = node.last_move()
    if last_move: next_moves.discard(opposite(last_move))
    return [Node(node.state + [move], node.bd, node.bpos) for move in next_moves]

def addToLiveNodes(pq: List[Node], node: Node):
    pq.append(node)
    for i in range(len(pq) - 1, 0, -1):
        crt, next = pq[i], pq[i-1]
        if crt.cost > next.cost:
            pq[i], pq[i-1] = next, crt
        else: break

def P(node: Node) -> bool: return count_misplaced(node.bd) ≤ 0

def fifteen_puzzle(board: List[List[int | None]]) -> Node:
    blank_pos = 0, 0
    for i, row in enumerate(board):
        for j, val in enumerate(row):
            if val is None : blank_pos = i, j

    live_nodes: List[Node] = []
    root = Node([], board, blank_pos)
    enode: Node = root
    while not P(enode):
        for node in listOfChildren(enode): addToLiveNodes(live_nodes, node)
        print([node.cost for node in live_nodes])
        enode = live_nodes.pop()
        print(enode); mprint(enode.bd); print(" ")  # noqa: E702
    return enode
```