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)
```