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)
```