Problem 1

Return a non-consecutive subsequence from an array composing the maximum sum.
If all negative values or null, return an empty array.

a) Solution

See external file MaxSet.py.

b) Time Complexity: $O(n)$

At its upper bound, function max_independent_set() never exceeds $O(n)$.

It creates memo array **cache** of input size n, the size of **nums**.
The procedure loops $n - 2$ times, calling the recurrence relation at $O(1)$ complexity:

```
for indice in range(2, len(nums)):
cache[indice] = max(cache[indice - 1], nums[indice] + cache[indice - 2])
```

After solving its subproblems, the code backtracks an entire n length.
At its worst, **indice** never decrements by 2, iterating down to indice 0.

```
indice = len(cache) - 1
while indice > -1:
if nums[indice] > 0 and cache[indice] != cache[indice - 1]:
subseq.append(nums[indice])
indice -= 2
else:
indice -= 1
```

Problem 2

Return the power set of set n of distinct numbers.

a) Solution

See external file PowerSet.py.

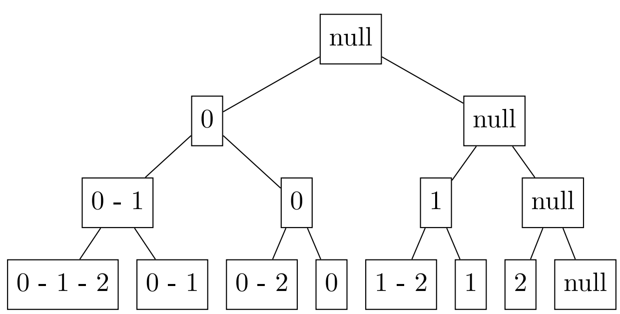
b) Time Complexity: $O(2^n)$

Function powerset_helper() is $O(2^n)$. Upper-bound, it generates 2^n subsets at each subproblem.

```
if pointer > len(input) - 1:
    result.append(choices\_made[:])
    return
choices\_made.append(input[pointer])
powerset\_helper(pointer + 1, choices\_made, input, result)
choices\_made.pop()
powerset\_helper(pointer + 1, choices\_made, input, result)
```

The code above starts with a null subset, recurses to the largest subset, and memoizes it.
It breaks down the main array and recurses to a new subset, memoizing it and progressing.
There are two choices at each recursion level: add the current pointer item or don't.

The tree shows subsets at each subtree and the present indices.



Root

Depth 1: Add the first indice or don't.

Depth 2: Add the second indice or don't.

Depth 3: Add the third indice or don't.

The code generates 2^n subsets for input size n.