Solution: Dynamic Programming & Backtracking

1. **Solve Dynamic Programming Problem and find its optimal solution.**
   Given a set of numbers, return a subset of non-consecutive numbers that would have the maximum sum.
   For example: Input: [7,2,5,8,6]
   Output: [7,5,6] (This will have sum of 18)
   Note: The numbers can include negative numbers as well.
   a. Write the recurrence formula to solve this problem using dynamic programming
   .

$$F(n) = \max \begin{cases} F(n-1) & \text{not choosing n-th element} \\ F(n-2) + nums[n] & \text{choosing n-th element} \end{cases}$$

$$F(0) = 0$$

   b. Write the pseudocode to solve the problem using dynamic Programming technique.

```
def max_independent_set(nums):
    memo = {-1:0, -2:0} #initialize memo
    n= len(nums)
    initialize ispicked = [True]*(n) to collect if a number is
picked or not

    for i in range(n):

        compare memo[i-1] , memo[i-2]+nums[i] and add bigger
value into memo[i]

        if(previous memo value from previous iteration = memo
value in current iteration):
            i-th number is not picked, set ispicked[i] = False

    return solution(ispicked, nums)


def solution(ispicked, nums):
    n = len(nums)
    revResult= []

    # check for the picked values from end as we started
collecting them from start
    i = n - 1
    while(i>=0):
        if i-th position is picked add it to revResult,
        this implies consecutive position is not picked so
decrement i two times

    return reversed revResult as we proccessed it in reverse
order
```

c. Implement the solution of this problem using dynamic Programming. Name your function **max_independent_set(nums)**. Name your file **MaxSet.py**

```
def max_independent_set(nums):
    memo = {-1:0, -2:0}
    n= len(nums)
    ispicked = [True]*(n)

    for i in range(n):
        memo[i] = max(memo[i-1] , memo[i-2]+nums[i])

        #if memo value is not moified mark ispicked[i] = false
        if(memo[i] == memo[i-1]):
            ispicked[i] = False

    # max possible sum in memo[n-1]
    return solution(ispicked, nums)


def solution(ispicked, nums):
    n = len(nums)
    revResult= []
    i = n-1

    # check for the picked values from end as we started
collecting them from start
    while(i>=0):
        #if i-th position is picked add it to revResult and
consecutive position is not picked so decrement i two times
        if(ispicked[i]):
            revResult.append(nums[i])
            i = i-1
        i = i-1

    return (list(reversed(revResult)))


#print(max_independent_set([1,1,7,2,-95,18,6]))
```

d. What is the time complexity of your implementation?
O(n)

2. Implement a backtracking algorithm

a. Write the implementation to solve the powerset problem discussed in the exercise. Name your function **powerset.py**. Name your file **PowerSet.py**

```
from copy import deepcopy

def powerset(input):
    result = []
    i= len(input)-1
    powerset_helper(i, [], input, result)
    return result
```

```
def powerset_helper(i, choices_made, input, result):
    if(i < 0):
        result.append(deepcopy(choices_made))  # make a deep copy
since we are working with objects
        return

    #consider i-th element
    choices_made.append(input[i])
    powerset_helper(i-1, choices_made, input, result)

    choices_made.pop() #backtrack
    powerset_helper(i-1, choices_made, input, result)
```

b. What is the time complexity of your implementation?
Program has exponential running time. $O(2^n)$; or $O(n * 2^n)$ if we consider the time taken to perform the deep copy.