1. To solve this problem using dynamic programming, I need to find the maximum sum of a subsequence where no 2 elements are next to each other. I started by creating DP table, where $dp[i]$ stores the best sum, we can get by considering elements up to index i.

At each step, there are 2 choices to consider whether to include the current element or to skip it. If I include the current element, I must skip the previous one, so the sum becomes $dp[i-2] + nums[i]$. If skipped, I just take $dp[i-1]$, which is the best sum without the current element. The recurrence formula will be $dp[i] = max(dp[i-1], dp[i-2] + nums[i])$. This ensures the highest possible sum is picked while following the rules of non-consecutive elements.

When the DP table is filled, backtrack is used to find which number is included. I start at the last index and check to see if it contributes to the maximum sum. An issue I ran into was handling the negative number because I kept getting -1 for the 2 negative test cases. I just added a conditional statement to return an empty list if the numbers are less than 0.

Time Complexity – O(n)

Pseudocode:

```
Function max_independent_set(nums):
    If nums is empty:
        Return empty list
    If all elements in nums are negative:
        Return empty list
    Set n = length of nums
    If n == 1:
        Return [nums[0]] if nums[0] > 0, else empty list
    # Initialize DP table
    Create array dp of size n, initialized to 0
    Set dp[0] = max(0, nums[0]) # First element or 0 if negative
    Set dp[1] = max(dp[0], nums[1]) if nums[1] > 0, else dp[0]
    # Fill DP table
    For i FROM 2 TO n-1:
        Set dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])
    # Backtrack to find elements contributing to max sum
    Create empty list result
    Set i = n - 1
    While i >= 0:
        If i == 0 OR dp[i] != dp[i - 1]:
            If nums[i] > 0:
                Append nums[i] to result
            Set i = i - 2 # Skip the previous element
    Else:
        Set i = i - 1 # Move to previous element
    Return result reversed
```

2. In this code, the powerset function starts the process by calling the backtrack function. The backtrack function builds the subsets recursively. At each step, it adds a copy of the current subset to the result list. It then loops through the elements of the input set, adding each element to the current subset and calling itself with the next index.

When it finishes one path, it backtracks by removing the last added element and continues with other possibilities. This process ensures that all possible subsets are generated without repeating or missing any.

Time Complexity – O(2^n)

Pseudocode:

```
Function powerset(inputSet):
    Create empty list result # Stores all subsets
    Call backtrack(0, inputSet, empty list, result) # Start recursion
    Return result
Function backtrack(start, inputSet, currentSet, result):
```

```
Append a copy of currentSet to result # Store the current subset
For i from start to length of inputSet - 1:
    Append inputSet[i] to currentSet # Include current element in subset
    Call backtrack(i + 1, inputSet, currentSet, result) # Recurse with next index
    Remove last element from currentSet # Backtrack to explore other subsets
```