Assignment: Backtracking & Greedy Algorithms

1. Implement a backtracking algorithm

Given a collection of amount values (A) and a target sum (S), find all unique combinations in A where the amount values sum up to S. Return these combinations in the form of a list.

Each amount value may be used only the number of times it occurs in list A. The solution set should not contain duplicate combinations. Amounts will be positive numbers.

Return an empty list if no possible solution exists.

Example: A = [11,1,3,2,6,1,5]; Target Sum = 8

Result = [[3, 5], [2, 6], [1, 2, 5], [1, 1, 6]]

1. **Describe a backtracking algorithm to solve this problem.**

The given code implements a backtracking algorithm to find all subsets of the input list A that sum up to the target value S. Here's a step-by-step breakdown of how the code works:

1. The function amount takes the input list A and the target sum S. It initializes an empty list result to store the subsets that add up to S.
2. The amount function calls the backtrack function with the initial parameters: A, S, an empty combination list, and the result list.
3. The backtrack function is a recursive function that performs the backtracking algorithm. It takes four parameters: the current list A, the remaining target sum target, the current combination of elements, and the result list.
4. The base cases in the backtrack function are checked.
   1. If the target becomes 0, it means we have found a valid combination that adds up to S.
      1. In this case, a copy of the current combination is added to the result list.
   2. If the target becomes negative, it means the current combination exceeds the target sum, so we backtrack and return.
5. If the base cases are not met, the function enters a loop that iterates through each element in the current list A.
6. Within the loop, the current element is appended to the combination list.
7. The backtrack function is recursively called with the updated parameters:
   1. the remaining list A starting from the next index (A[i+1:]),
   2. the updated target sum (target - A[i]),
   3. the updated combination,
   4. and the result list.
8. This recursive call explores the possibility of including the current element in the combination. The function will continue exploring other elements in the A list to find valid combinations that add up to the remaining target.
9. After the recursive call, the element is removed from the combination list to backtrack and explore other possibilities. This is done using the combination.pop() operation.
10. The loop continues to iterate through the remaining elements in A, and the process repeats until all possible combinations have been explored.
11. Once the loop finishes, the backtrack function returns, and the control returns to the amount function.
12. Finally, the amount function returns the result list, which contains all the subsets of A that sum up to S.
13. Implement the solution in a function amount(A, S). Name your file Amount.py
14. What is the time complexity of your implementation, you may find time complexity in detailed or state whether it is linear/polynomial/exponential. etc.?

The time complexity of this algorithm can be analyzed as follows:

The time complexity of the implementation using the backtracking algorithm is exponential. It has a time complexity of O(2^n), where n is the length of the input list.

The reason for the exponential time complexity is that the algorithm explores all possible combinations of subsets by including or excluding elements from the input list. This leads to a branching factor of 2 at each step, resulting in an exponential growth in the number of recursive calls.

As the size of the input list increases, the number of recursive calls and the overall runtime of the algorithm increase exponentially. Therefore, the time complexity is considered to be exponential.

2. Implement a Greedy algorithm

You are a pet store owner and you own few dogs. Each dog has a specific hunger level given by array hunger\_level [1..n] (ith dog has hunger level of hunger\_level [i]). You have couple of dog biscuits of size given by biscuit\_size [1…m]. Your goal to satisfy maximum number of hungry dogs. You need to find the number of dogs we can satisfy.

If a dog has hunger hunger\_level[i], it can be satisfied only by taking a biscuit of size biscuit\_size [j] >= hunger\_level [i] (i.e biscuit size should be greater than or equal to hunger level to satisfy a dog.)

If no dog can be satisfied return 0.

Conditions:

You cannot give same biscuit to two dogs.

Each dog can get only one biscuit.

Example 1:

Input: hunger\_level[1,2,3], biscuit\_size[1,1]

Output: 1

Explanation: Only one dog with hunger level of 1 can be satisfied with one cookie of size 1.

Example 2:

Input: hunger\_level[2, 1], biscuit\_size[1,3,2]

Output: 2

Explanation: Two dogs can be satisfied. The biscuit sizes are big enough to satisfy the hunger level of both the dogs.

1. **Describe a greedy algorithm to solve this problem**
2. Sort both the hunger\_level & biscuit\_size arrays in non-decreasing order.

Why?

* This allows us to prioritize feeding the dogs with lower hunger levels using smaller biscuits 1st.
* By iterating through the biscuit\_size array, which is also sorted, starting from the smallest biscuit, we have the opportunity to allocate smaller biscuits to dogs with lower hunger levels.
* This approach increases the chances of satisfying more dogs overall because smaller biscuits can only satisfy dogs with lower hunger levels, while larger biscuits can potentially satisfy dogs with both lower and higher hunger levels.
* If we were to allocate larger biscuits to dogs with lower hunger levels first, it would limit the available options for dogs with higher hunger levels. By prioritizing lower hunger levels, we optimize the allocation of biscuits and create more opportunities for satisfying dogs with varying hunger levels.

1. Initialize a variable happyDogs to keep track of the number of dogs that can be satisfied. Start with happyDogs = 0.
2. Initialize a pointer\* j to 0, which will be used to iterate through the hunger\_level array.
3. Iterate through each biscuit in the biscuit\_size array:
   1. Inside this loop, iterate through the hunger\_level array starting from index j:
      1. Check if the current biscuit size biscuit\_i is => or == to the hunger level hunger\_level[k].
         1. If it is:
            1. increment happyDogs by 1,
            2. update j to k + 1 to avoid using the same biscuit for multiple dogs,
            3. and break out of the inner loop.
         2. If the biscuit size is not sufficient to satisfy the current dog:
            1. continue to the next dog in the inner loop.

b. After the inner loop, check if j has reached the end of the hunger\_level array. If it has, return happyDogs as all dogs have been satisfied.

1. Return happyDogs as the final result.

This greedy algorithm ensures that dogs with lower hunger levels are prioritized and fed first using smaller biscuits, maximizing the number of dogs that can be satisfied.

1. Write an algorithm implementing the approach. Your function signature should be feedDog(hunger\_level, biscuit\_size); hunger\_level, biscuit\_size both are one dimension arrays . Name your file FeedDog.py
2. Analyze the time complexity of the approach.

For sorting array time complexity is O(nlogn**) +** O**(**mlogm) where n and m is the length of both array

And for mapping biscuit and hunger dog, it will take m+n where n and m is the length of both array

So time complexity will be O(nlogn) + O(mlogm) + O(m) + O(n)

Therefore:

Time complexity is O(N log N) where N is max{len(hunger\_index) , len( biscuit\_size) }