Backtracking

Assignment Solutions

PW SKILLS

1. Given n as input. Generate all strings that are palindromes with the number of digits as 'n'.
For example a palindrome of size 3 can be 313, 121, 030.
Note it can even contain leading zeros
Input : n = 2
Output : 00, 11, 22, 33, 44, 55, 66, 77, 88, 99

Code link: https://pastebin.com/it6rfLux

Explanation :

- Here we do a recursive function call, as the number is palindrome the second half will be the reverse of the first half. Then we only need to solve for the first half of the string and can simply reverse for the second half.
- We can do that using recursion. At each step just take the current string and keep on adding the elements. And when we revert back to the function call, just pop back that element.
- Keep on doing this till we reach half of the string and then simply calculate and print the answer.

Output:

```
2
00, 11, 22, 33, 44, 55, 66, 77, 88, 99,

...Program finished with exit code 0
Press ENTER to exit console.
```

2. Check if the product of some subset of an array is equal to the target value.
Where n is the size of the input array.
Note: Each index value can be used only once.
Input : n = 5 , target = 16
Array = [2 3 2 5 4]
Here the target will be equal to 2x2x4 = 16

Output : YES

Code link: https://pastebin.com/nKckmzYB

Explanation :

- Just make a recursive function and take two cases. In the first case we consider taking the element for product and multiply it with product
- In the second case, we do not take the element and leave that element.
- This way we reach the point which is the end of the array, simply check whether the product is equal to the target.

Output:

```
5 16
2 3 2 5 4
YES


...Program finished with exit code 0
Press ENTER to exit console.
```

3. **Given an integer array nums that may contain duplicates, return all possible subsets(the power set). The solution set must not contain duplicate subsets. Return the solution in any order.**
   Sample Input: **nums=[1,1,2]**
   Sample Output: **[],[1],[1,2],[1,1],[1,1,2],[2]**
   Sample Input: **nums=[1,2]**
   Sample Output: **[],[1],[2],[1,2]**

   Code Link: https://pastebin.com/wgvJPj51

   Explanation:
   - Sort the input array to make it easier to identify duplicates.
   - Define a recursive function that takes as input the current subset, the index of the next element to consider, and the set of all subsets found so far. The function should do the following:
     a. Add the current subset to the set of subsets found so far.
     b. Loop over the remaining elements in the input array starting from the next index. For each element:
     i. If the element is a duplicate of the previous element (i.e., nums[i] == nums[i-1]), continue to the next element to avoid duplicate subsets.
     ii. Add the current element to the current subset.
     iii. Recursively call the function with the updated subset and the next index.
     iv. Remove the current element from the current subset to prepare for the next iteration of the loop.
   - Call the recursive function with an empty subset and the starting index of 0.
   - Convert the set of subsets found so far to a list and return it.

   Output:

```
1
1 2
1 2 2
2
2 2


...Program finished with exit code 0
Press ENTER to exit console.
```

4. **Given a string s, you can transform every letter individually to be lowercase or uppercase to create another string.**
   Return a list of all possible strings we could create. Return the output in any order.
   Sample Input: **s="a1"**
   Sample Output : **["a1","A1"]**
   Sample Input: **s= "bc12"**
   Sample Output: **["bc12","bC12","Bc12","BC12"]**

   Code link: https://pastebin.com/CQsGk9S3

   Explanation:
   - Create an empty result list to store all possible strings.
   - Start a recursive function to generate all possible combinations of lowercase and uppercase letters for each character in the input string.
   - In the recursive function, iterate through each character in the input string and generate two possibilities for each character - either convert it to lowercase or uppercase.
   - Append the lowercase or uppercase character to the current result string and call the recursive function again with the remaining substring and the updated result string.
   - When we reach the end of the input string, add the current result string to the result list.
   - Return the result list.

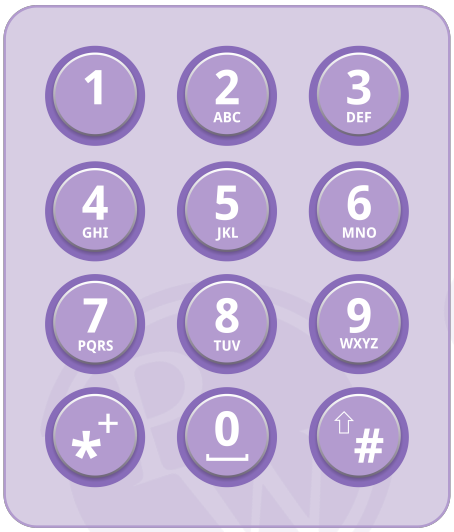
Output:

```
Enter the string : bc12
bc12
bC12
Bc12
BC12


...Program finished with exit code 0
Press ENTER to exit console.
```

5. **Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in any order.**

   A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



Sample Input: **"22"**
Sample Output: **["aa","ab","ac","ba","bb","bc","ca","cb","cc"]**
Sample Input: **"34"**
Sample Output: **["dg","dh","di","eg","eh","ei","fg","fh","fi"]**

Code link: https://pastebin.com/CHrKtKDL

Explanation:

- First, we can create a mapping of digits to corresponding letters. This mapping can be represented using an array of strings. For example, mapping[2] will contain "abc", mapping[3] will contain "def", and so on.
- We can use a recursive function that takes three parameters: the current index of the digit we are processing, the current string of letters generated so far, and the final result vector that stores all possible letter combinations.
- At each index, we can get the corresponding letters from the mapping array and append each letter to the current string. We then call the recursive function with the updated string and index+1.
- We keep doing this until we reach the end of the input string. At this point, we add the current string to the final result vector.
- Finally, we return the result vector containing all possible letter combinations.

Output:

```
dg dh di eg eh ei fg fh fi


...Program finished with exit code 0
Press ENTER to exit console.
```

6. **Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.**
   Sample Input: **n=1**
   Sample Output: ["()"]
   Sample Input: **n=3**
   Sample Output: ["((()))","(()())","(())()","()(())","()()()"]

   Code link: https://pastebin.com/6bjS0tm9

   Explanation:

- We start by calling backtrack with an empty string, and the number of remaining opening and closing parentheses set to n.
- In each recursive call, we add either an opening or closing parenthesis to the current string, and decrement the corresponding count of remaining parentheses.
- When there are no remaining parentheses left, we add the current string to the list of valid combinations.
- Note that we check if close > open before adding a closing parenthesis. This is because we cannot have more closing parentheses than opening parentheses at any point in the string.

   Output:

```
3
((()))
(()())
(())()
()(())
()()()


...Program finished with exit code 0
Press ENTER to exit console.
```

7. **You are given an integer array of matchsticks where matchsticks[i] is the length of the ith matchstick. You want to use all the matchsticks to make one square. You should not break any stick, but you can link them up, and each matchstick must be used exactly one time.**
   Return true if you can make this square and false otherwise.
   Sample Input: **[1,1,2,2,2]**
   Sample Output: **true**
   Explanation: The square formed will be of side 2
   Sample Input: **[1,1,2,3,4]**
   Sample Output: **false**
   Square cannot be formed.

   Code link: https://pastebin.com/6ZW4RfPp

   Explanation:

We can start by calculating the perimeter of the square that can be formed using the matchsticks. If the perimeter is not divisible by 4, then we cannot form a square. Otherwise, we can proceed with the backtracking approach.

We need to find a way to partition the matchsticks into 4 groups, each having the same length, such that each group's length is equal to the length of the side of the square. We can do this by trying out all possible combinations of the matchsticks, and checking if the length of each of the four groups is equal to the length of the side of the square.

The backtracking algorithm will look like this:

- Calculate the perimeter of the square using the sum of all the matchsticks. If the perimeter is not divisible by 4, return false.
- Sort the matchsticks in non-increasing order.
- Initialize an array of size 4, representing the four sides of the square.
- Call the recursive function canFormSquare(0, matchsticks, sides, sideLength), where sideLength is the length of each side of the square (i.e., the perimeter divided by 4).
- In the canFormSquare function:

a. If i == n, check if all the sides are equal to sideLength. If so, return true, otherwise return false.

b. For each of the four sides:

i. If matchsticks[i] is greater than the remaining length of the side, skip it.

ii. Otherwise, subtract matchsticks[i] from the remaining length of the side, and call canFormSquare(i+1, matchsticks, sides, sideLength). If this returns true, return true.

c. If none of the four sides can be formed using the remaining matchsticks, return false.

Output:

```
True
False


...Program finished with exit code 0
Press ENTER to exit console.
```

8. **Given two integers n and k, return all possible combinations of k numbers chosen from the range [1, n].**
   Note: The number should not be repeated in the combination.
   You may return the answer in any order.
   Sample Input: n=4, k=2
   Sample Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
   Sample Input: n=1, k=1
   Sample Output: [[1]]

   Code Link: https://pastebin.com/vi0fyaS2

   Explanation:

   We start with an empty list and consider each number from 1 to n. If the current list has less than k elements, we add the current number to the list and recursively call the function with the updated list and the remaining numbers. If the current list has k elements, we add the list to the result list. At each step, we remove the last added element from the list to backtrack and try other options.

Output:

```
4 2
1 2
1 3
1 4
2 3
2 4
3 4


...Program finished with exit code 0
```