Keep_Me

October 22, 2019

1 Table of Contents

2 Recursion

```
In [1]: def fact(n):
            if n == 0:
                return 1
            else:
                return n * fact(n-1)
```

```
In [2]: def rec_sum(n):
            if n == 0:
                return 0
            else:
                return n + rec_sum(n-1)
            pass
```

```
Out[2]: 120
```

```
In [ ]: def sum_func(n):
            if(n<10):
                return n
            else:
                return n%10 + sum_func(int(n/10))
            pass
```

```
In [ ]: def word_split(phrase,list_of_words, output = None):
            if output is None:
                output = []

            for word in list_of_words:
                if phrase.startswith(word):
                    output.append(word)
                    word_split(phrase[len(word):], list_of_words, output)
            return output
            pass

In [ ]: def reverse(s):

            # Base Case
            if len(s) <= 1:
                return s

            # Recursion
            return reverse(s[1:]) + s[0

In [ ]: def permute(s):
            out = []

            # Base Case
            if len(s) == 1:
                out = [s]

            else:
                # For every letter in string
                for i, let in enumerate(s):

                    # For every permutation resulting from Step 2 and 3 described above
                    for perm in permute(s[:i] + s[i+1:]):

                        # Add it to output
                        out += [let + perm]

            return out

In [ ]: def fib_rec(n):

            # Base Case
            if n == 0 or n == 1:
                return n

            # Recursion
            else:
                return fib_rec(n-1) + fib_rec(n-2)

In [ ]: # Instantiate Cache information
        n = 10
        cache = [None] * (n + 1)


        def fib_dyn(n):
            # Base Case
            if n == 0 or n == 1:
                return n

            # Check cache
            if cache[n] != None:
                return cache[n]

            # Keep setting cache
            cache[n] = fib_dyn(n-1) + fib_dyn(n-2)

            return cache[n]
```

```
In [ ]: def fib_iter(n):

            # Set starting point
            a = 0
            b = 1

            # Follow algorithm
            for i in range(n):

                a, b = b, a + b

            return a
```

```
In [1]: def rec_coin(target,coins):
            '''
            INPUT: Target change amount and list of coin values
            OUTPUT: Minimum coins needed to make change

            Note, this solution is not optimized.
            '''

            # Default to target value
            min_coins = target

            # Check to see if we have a single coin match (BASE CASE)
            if target in coins:
                return 1

            else:

                # for every coin value that is <= than target
                for i in [c for c in coins if c <= target]:

                    # Recursive Call (add a count coin and subtract from the target)
                    num_coins = 1 + rec_coin(target-i,coins)
                    # Reset Minimum if we have a new minimum
                    if num_coins < min_coins:

                        min_coins = num_coins

            return min_coins
```

```
In [2]: def rec_coin_dynam(target,coins,known_results):
            '''
            INPUT: This funciton takes in a target amount and a list of possible coins to use.
            It also takes a third parameter, known_results, indicating previously calculated res
            The known_results parameter shoud be started with [0] * (target+1)

            OUTPUT: Minimum number of coins needed to make the target.
            '''

            # Default output to target
            min_coins = target

            # Base Case
            if target in coins:
                known_results[target] = 1
                return 1

            # Return a known result if it happens to be greater than 1
            elif known_results[target] > 0:
                return known_results[target]

            else:
                # for every coin value that is <= than target
                for i in [c for c in coins if c <= target]:

                    # Recursive call, note how we include the known results!
                    num_coins = 1 + rec_coin_dynam(target-i,coins,known_results)

                    # Reset Minimum if we have a new minimum
                    if num_coins < min_coins:
                        min_coins = num_coins

                        # Reset the known result
                        known_results[target] = min_coins

            return min_coins
```

```
In [3]: !open .
```

```
In [ ]:
```