CSCI-SHU 210 Data Structures

Homework Assignment3 Recursion

About 84% of this assignment can be auto graded. Please submit your files to Gradescope.

Question 1 (x to the power of n):

Implement function, **power(x,n)** recursively. *n* could be any integer (positive, negative, or zero). In the text book, **power(x,n)** python program (Page Number 173 Code Fragment 4.12) only handles non-negative integers. Your **power(x,n)** function should be able to handle both positive and negative integers.

Example 1:

```
Input: power(-5, 2)      # (-5) ^ 2 = 25
Returns: 25
```

Example 2:

```
Input: power(4, -1)      # 4 ^ (-1) = 0.25
Returns: 0.25
```

Important:

- Of course, your function should not use ** operator!!
- The big O complexity of your program should be O(logN). (8pts)

Question 2 (Convert recursion to iteration):

Rewrite the following recursive function using iteration instead of recursion.

Your iterative function should do exactly the same task as the given recursive function.

```
1 def recur(n):
2     if n < 0:
3         n  = -n
4     if n < 10:
5         return n, n
6     a, b = recur(n//10)
7     return max(n%10, a), min(n%10, b)
```

Question 3 (Element Uniqueness Problem without Loops):

The e*lement uniqueness problem* is to dertermine whether there are duplicates in a given ***S*** of *n* elements. Implement an efficient recursive function **unique(S)** for solving the element uniqueness problem, which runs in time that is **at most O(n^2) in the worst case** (8pts). Besides, please:

- Do **not** use sorting: elements may not be comparable
- Do **not** use for-loops or while loops

Notice that although the input S is a Python List, we impose restrictions on its usage in your implementation: **we assume**

- There is no **S.__contain__(self, elem)** method defined (i.e., can not use **"x in S"**)
- **S.__getitem__(self, i)** (indexing) is supported, but slicing is not (i.e., **S[i]** works, but **S[i:j]** is not allowed)

Example 1:

```
Input: unique([1, 54, 3, 25, 39, 25, 2])
Returns: False
```

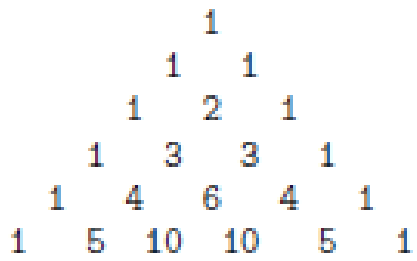
Example 2:

```
Input: unique( [ 9, 'a', [], [[35, 2], ['NYU']], (100,) ] )
Returns: True
```

Question 4 (Pascal's Triangle without Loops):

In mathematics, Pascal's triangle is a triangular array of the binomial coefficients. Implement a **recursive** function **pascal(k)** to generate a list of pascal values **at level k** (k starts from 0).

Pascal triangle looks like the following:



Example 1:

```
Input: pascal(0)
Returns: [1]
```

Example 2:

```
Input: pascal(5)
Returns: [1, 5, 10, 10, 5, 1]
```

Important:

- You should not use for/while loops
- You may introduce a helper function
- Hint: one way to solve this question is to implement two recursive parts: one is to calculate pascal values, and the other to "recursively proceed" from position *i* to *i+1* in the same level *k*

Question 5 (Stones Allocation):

Suppose you have $N \geq 1$ stones, and $M \geq 1$ days to throw at a target. Each day, you can throw 1 or 2 or 3 stones. In addition, on even number of days, you can only throw 1 or 3 stone(s) as a constraint. Implement a recursive function **throw_stones(N, M)** to return all possible valid sequences to throw *N* stones in *M* days under such constraint.

Example 1:

When N = 5, M = 3, valid sequences:

Day1	Day2	Day3
1	1	3
2	1	2
3	1	1
1	3	1

Invalid sequences:

Day1	Day2	Day3		
3	2	1		
3	1	2		
1	1	1	1	1

```
Input: throw_stones(5,3)
Returns: [ (1, 1, 3), (2, 1, 2), (3, 1, 1), (1, 3, 1) ]
```

Important:

- You should not use for/while loops
- You may introduce a helper function
- You cannot use Python dictionary to store temporary results during execution
- Your result could be returned in a list of lists, or a list of tuples. The order does not matter.

Question 6 (How to spend your money to decorate your kitchen?):

Dr. X invests you with *$X* and asks you to use the money to decorate kitchen with Y appliance types (e.g. [Fridge, Oven, Stove, Sink]). Each appliance has multiple brands of different costs.

Now, you want to spend your money wisely so that all types of appliances should be bought just once (i.e. just choose one brand for each appliance type), and yet you need to spend at least $Z < $X amount.

Write a recursive program **purchase_combination(Y_cost, X, Z)** to produce all possible purchase combinations. **Y_cost** is a list of costs of each appliance brand. E.g. . **Y_cost** = [[1, 5], [2, 4, 3, 6]], then:

- The number of brands for appliance type 0 is 2, which corresponds to cost [1, 5] from . **Y_cost[0]**
- The number of brands for appliance type 1 is 4, which corresponds to cost [2, 4, 3, 6] from **Y_cost[1]**
- Assuming all costs are > 0

Example 1:

```
Y_cost = [[1,5], [2,4,3,6]]
X = 12
Z = 9

Input: purchase_combination(Y_cost, X, Z)
Returns: [ (1,1), (1,3) ]
```

Important:

- No for/while loop is allowed in your recursive program.
- You may introduce a helper function to perform the actual recursion. Think about what additional inputs you would like to introduce in your solution.
- You should return a List of tuples.