CSCI-SHU 210 Data Structures

Homework Assignment3 Recursion

About 84% of this assignment can be auto graded. Again, we have two submission options: You can submit everything to gradescope; (Which means you can see 84% of your grade earlier) You can also submit everything on NYU Classes.

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.          return -1
4.      elif (n < 10):
5.          return 1
6.      else:
7.          return 1 + recur(n // 10)
```

Question 3 (Element Uniqueness Problem):

Consider the Unique3 (Recursive Element Uniqueness Problem) program from the text book (Page Number 165 Code Fragment 4.6). Worst case runtime for this program could be O(2^n).

```
1  def unique3(S, start, stop):
2    """Return True if there are no duplicate elements in slice S[start:stop]."""
3    if stop − start <= 1: return True                  # at most one item
4    elif not unique(S, start, stop−1): return False    # first part has duplicate
5    elif not unique(S, start+1, stop): return False    # second part has duplicate
6    else: return S[start] != S[stop−1]                 # do first and last differ?
```

Code Fragment 4.6: Recursive unique3 for testing element uniqueness.

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 **`without using sorting`**. (8pts)

Example 1:

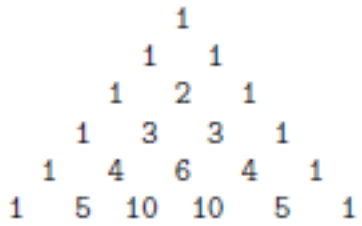
```
Input: unique([1, 7, 6, 5, 4, 3, 1])
Returns: False
```

Example 2:

```
Input: unique([9, 4, 'a', [], 0])     # All elements are unique
Returns: True
```

Question 4 (Pascal's Triangle):

In mathematics, Pascal's triangle is a triangular array of the binomial coefficients.
Implement a **recursive** function **`pascal(N)`** to generate a list of pascal values. Pascal triangle looks like the following:



Implement recursive function **`pascal(N)`**, which generates list of sublists, each sublist contains a level of Pascal values.

Example:

```
Input: pascal(5)     # five levels of pascal values
Returns: [[1], [1,1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

Problem 5 (Vowels and Consonants)

Given a string of letters representing a word(s), implement function, **`vc_count(word),`** that returns a **`list`** of 2 integers: the number of vowels, and the number of consonants in the word.

Example:

Input: vc_count("GoodMorningShanghai")

Returns: [7, 12] # 7 vowels, 12 consonants

Important:

- Your program must be recursive.
- Vowels are 'aeiouAEIOU'.
- You may assume that the input string contains only letters a-z and A-Z.
- You may need to declare new function(s) with additional parameters.

Problem 6 (All Possible Combinations)

The knapsack problem or rucksack problem is a classic in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

We're going to work on a simpler version of this problem. Given a set of items, each associated with a value, we're going to determine which combinations of items sum up to a given value: the capacity of the knapsack.

Complete function **`knapsack(capacity, weights)`**: this function enumerates and returns a list of sublists of all possible combinations of items whose values sum up to a given capacity.

Example:

Input: knapsack(14, [1, 2, 8, 4, 9, 1, 4, 5])

Returns: (Order for inner values, outer lists don't matter.)
[[9, 5], [9, 1, 4], [4, 1, 4, 5], [4, 9, 1], [8, 1, 5], [2, 8, 4], [2, 8, 4], [1, 9, 4], [1, 4, 4, 5], [1, 4, 9], [1, 8, 5], [1, 8, 1, 4], [1, 8, 4, 1]]

Important:

- You may need to declare new function(s) with additional parameters.
- Use the original function, to call your new function.