23.12.2020

CMPE150.03 - Recursion practice questions

sources:

- https://practiceit.cs.washington.edu/
- https://www.cs.cornell.edu/courses/cs2110/2014sp/L07-Recursion/recursion_practice.pdf
- https://web.stanford.edu/class/cs9/lectures/06/Recursion%20Problems.pdf

Write a recursive function that, given two strings, returns whether the first string is asubsequence of the second. For example, given hac and cathartic, you should return true,but given bat and table, you should return false.

Compute the Factorial of a numberN. Fact(N) $=N\times(N-1)\cdots1$

Compute the sum of natural numbers until N.

Write a function for mutliply(a, b), where a and b are both positive integers, but you can only use the + or − operators.

Find Greatest Common Divisor (GCD) of 2 numbers using recursion.

Write a recursive function to reverse a string Write a recursive function to reverse the words in a string, i.e., “cat is running” becomes “running is cat”.

A word is considered elfish if it contains the letters: e, l, and f in it, in any order. For example, we would say that the following words are elfish: whiteleaf, tasteful, unfriendly,and waffles, because they each contain those letters.

- Write a function called elfish that, given a word, tells us if that word is elfish or not.

- Write a more generalized function called x_ish() that, given two words, returns true if all the letters of the first wordare contained in the second

---

Write a recursive method `digitMatch` that accepts two non-negative integers as parameters and that returns the number of digits that match between them. Two digits match if they are equal and have the same position relative to the end of the number (i.e. starting with the ones digit). In other words, the method should compare the last digits of each number, the second-to-last digits of each number, the third-to-last digits of each number, and so forth, counting how many pairs match. For example, for the call of `digitMatch(1072503891, 62530841)`, the method would compare as follows:

```
1 0 7 2 5 0 3 8 9 1
    | | | | | | | |
    6 2 5 3 0 8 4 1
```

The method should return `4` in this case because 4 of these pairs match (2-2, 5-5, 8-8, and 1-1). Below are more examples:

Call	Value Returned
`digitMatch(38, 34)`	`1`
`digitMatch(5, 5552)`	`0`
`digitMatch(892, 892)`	`3`
`digitMatch(298892, 7892)`	`3`
`digitMatch(380, 0)`	`1`
`digitMatch(123456, 654321)`	`0`
`digitMatch(1234567, 67)`	`2`

Write a recursive method `repeat` that accepts a string *s* and an integer *n* as parameters and that returns a `String` consisting of *n* copies of *s*. For example:

Call	Value Returned
`repeat("hello", 3)`	`"hellohellohello"`
`repeat("this is fun", 1)`	`"this is fun"`
`repeat("wow", 0)`	`""`
`repeat("hi ho! ", 5)`	`"hi ho! hi ho! hi ho! hi ho! hi ho! "`

You should solve this problem by concatenating using the `+` operator.

Write a recursive method `isReverse` that accepts two strings as a parameter and returns `true` if the two strings contain the same sequence of characters as each other but in the opposite order (ignoring capitalization), and `false` otherwise. For example, the string `"hello"` backwards is `"olleh"`, so a call of `isReverse("hello", "olleh")` would return true. Since the method is case-insensitive, you would also get a `true` result from a call of `isReverse("Hello", "oLLEh")`. The empty string, as well as any one-letter string, is considered to be its own reverse. The string could contain characters other than letters, such as numbers, spaces, or other punctuation; you should treat these like any other character. The key aspect is that the first string has the same sequence of characters as the second string, but in the opposite order, ignoring case. The table below shows more examples:

Call	Value Returned
`isReverse("CSE143", "341esc")`	`true`
`isReverse("Madam", "MaDAm")`	`true`
`isReverse("Q", "Q")`	`true`
`isReverse("", "")`	`true`
`isReverse("e via n", "N aIv E")`	`true`
`isReverse("Go! Go", "OG !OG")`	`true`
`isReverse("Obama", "McCain")`	`false`
`isReverse("banana", "nanaba")`	`false`
`isReverse("hello!!", "olleh")`	`false`
`isReverse("", "x")`	`false`
`isReverse("madam I", "i m adam")`	`false`
`isReverse("ok", "oko")`	`false`

Write a recursive method `indexOf` that accepts two `String`s as parameters and that returns the starting index of the first occurrence of the second `String` inside the first `String` (or `-1` if not found). The table below lists several calls to your method and their expected return values. Notice that case matters, as in the last example that returns `-1`.

Call	Value Returned
`indexOf("Barack Obama", "Bar")`	`0`
`indexOf("Barack Obama", "ck")`	`4`
`indexOf("Barack Obama", "a")`	`1`
`indexOf("Barack Obama", "McCain")`	`-1`
`indexOf("Barack Obama", "BAR")`	`-1`

Write a method `evenDigits` that accepts an integer parameter *n* and that returns the integer formed by removing the odd digits from *n*. The following table shows several calls and their expected return values:

Call	Valued Returned
`evenDigits(8342116);`	`8426`
`evenDigits(4109);`	`40`
`evenDigits(8);`	`8`
`evenDigits(-34512);`	`-42`
`evenDigits(-163505);`	`-60`
`evenDigits(3052);`	`2`
`evenDigits(7010496);`	`46`
`evenDigits(35179);`	`0`
`evenDigits(5307);`	`0`
`evenDigits(7);`	`0`

If a negative number with even digits other than 0 is passed to the method, the result should also be negative, as shown above when -34512 is passed. Leading zeros in the result should be ignored and if there are no even digits other than 0 in the number, the method should return 0, as shown in the last three outputs.

In this problem, the scenario we are evaluating is the following: You're standing at the base of a staircase and are heading to the top. A small stride will move up one stair, and a large stride advances two. You want to count the number of ways to climb the entire staircase based on different combinations of large and small strides. For example, a staircase of three steps can be climbed in three different ways: three small strides, one small stride followed by one large stride, or one large followed by one small.

Write a recursive method `waysToClimb` that takes a non-negative integer value representing a number of stairs and prints each unique way to climb a staircase of that height, taking strides of one or two stairs at a time. Your method should output each way to climb the stairs on its own line, using a 1 to indicate a small stride of 1 stair, and a 2 to indicate a large stride of 2 stairs. For example, the call of `waysToClimb(3)` should produce the following output:

```
[1, 1, 1]
[1, 2]
[2, 1]
```

The call of `waysToClimb(4)` should produce the following output:

```
[1, 1, 1, 1]
[1, 1, 2]
[1, 2, 1]
[2, 1, 1]
[2, 2]
```

The order in which you output the possible ways to climb the stairs is not important, so long as you list the right overall set of ways. There are no ways to climb zero stairs, so your method should produce no output if 0 is passed. Do not use any loops in solving this problem.

Write a method `countBinary` that accepts an integer n as a parameter and that prints all binary numbers that have n digits in ascending order, printing each value on a separate line. All n digits should be shown for all numbers, including leading zeros if necessary. You may assume that n is non-negative. If n is 0, a blank line of output should be produced. Do not use a loop in your solution; implement it recursively.

Call	Output
`countBinary(1);`	`0` `1`

`countBinary(2);`	`00`
	`01`
	`10`
	`11`

`countBinary(3);`	`000`
	`001`
	`010`
	`011`
	`100`
	`101`
	`110`
	`111`

Hint: It may help to define a private helper method that accepts different parameters than the original method. In particular, consider building up a set of characters as a String for eventual printing.

Write a method `subsets` that uses recursive backtracking to find every possible sub-list of a given list. A sub-list of a list *L* contains 0 or more of *L*'s elements. Your method should accept a `List` of strings as its parameter and print every sub-list that could be created from elements of that list, one per line. For example, suppose a variable called `list` stores the following elements:

```
[Janet, Robert, Morgan, Char]
```

The call of `subsets(list);` would produce output such as the following:

```
[Janet, Robert, Morgan, Char]
[Janet, Robert, Morgan]
[Janet, Robert, Char]
```

```
[Janet, Robert]
[Janet, Morgan, Char]
[Janet, Morgan]
[Janet, Char]
[Janet]
[Robert, Morgan, Char]
[Robert, Morgan]
[Robert, Char]
[Robert]
[Morgan, Char]
[Morgan]
[Char]
[]
```

The order in which you show the sub-lists does not matter, but the order of the elements of each sub-list DOES matter. The key thing is that your method should produce the correct overall set of sub-lists as its output. Notice that the empty list is considered one of these sub-lists. You may assume that the list passed to your method is not `None` and that the list contains no duplicates. Do not use any loops in solving this problem.

Hint: This problem is somewhat similar to the permutations problem. Consider each element and try to generate all sub-lists that do include it, as well as all sub-lists that do not include it.

It can be hard to see a pattern from looking at the lines of output. But notice that the first 8 of 16 total lines of output constitute all the sets that include Janet, and the last 8 lines are the sets that do not have her as a member. Within either of those groups of 8 lines, the first 4 of them are all the sets that include Robert, and the last 4 lines are the ones that do not include him. Within a clump of 4, the first 2 are the ones including Morgan, and the last 2 are the ones that do not include Morgan. And so on. Once again, you do not have to match this exact order, but looking at it can help with figuring out the patterns and the recursion.

Given a list of n distinct elements, write a function that lists all permutations of that list.

Given a list of n distinct elements and a number k, write a function that lists all k-element subsets of that list. Make sure not to output the same subset multiple times.

Given a list of n distinct elements and a number k, write a function that lists all k- element permutations of that list.

Given a nonnegative integer n, write a function that lists all strings formed from exactly npairs of balanced parentheses. For example, given n = 3, you'd list these five strings:

((())) (()()) (())() ()(()) ()()()

As a hint, given any string of n≥ 1 balanced parentheses, focus on the first open parentheses and the close parentheses it matches. What can you say about the string inside those parentheses? What about the string that follows those parentheses?

Given a number n, generate all n-character passwords, subject to the restriction that every password must have a number, a lower-case letter, an upper-case letter, and a symbol.

Given a number n, generate all distinct ways to write n as the sum of positive integers.For example, with n = 4, the options are 4, 3 + 1, 2 + 2, 2 + 1 + 1, and 1 + 1 + 1 + 1.

Given a list of n integers and a number U, write a function that returns whether there is asubset of those n integers that adds up to exactly U. (This is a famous problem called the“subset sum problem,” by the way.)

Write a recursive method `maxSum` that accepts a list of integers L and an integer limit n as its parameters and uses backtracking to find the maximum sum that can be generated by adding elements of L that does not exceed n. For example, if you are given the list of integers [7, 30, 8, 22, 6, 1, 14] and the limit of 19, the maximum sum that can be generated that does not exceed is 16, achieved by adding 7, 8, and 1. If the list L is empty, or if the limit is not a positive integer, or all of L's values exceed the limit, return 0.

Each index's element in the list can be added to the sum only once, but the same number value might occur more than once in a list, in which case each occurrence might be added to the sum. For example, if the list is [6, 2, 1] you may use up to one 6 in the sum, but if the list is [6, 2, 6, 1] you may use up to two sixes.

Here are several example calls to your method and their expected return values:

List L	Limit n	maxSum(L, n) returns
`[7, 30, 8, 22, 6, 1, 14]`	19	16
`[5, 30, 15, 13, 8]`	42	41
`[30, 15, 20]`	40	35
`[6, 2, 6, 9, 1]`	30	24
`[11, 5, 3, 7, 2]`	14	14
`[10, 20, 30]`	7	0
`[10, 20, 30]`	20	20
`[]`	10	0

You may assume that all values in the list are non-negative. Your method may alter the contents of the list L as it executes, but L should be restored to its original state before your method returns. Do not use any loops in solving this problem.

Write a method `printSquares` that uses recursive backtracking to find all ways to express an integer as a sum of squares of unique positive integers. For example, the call of `printSquares(200);` should produce the following output:

```
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2
1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2
1^2 + 2^2 + 5^2 + 7^2 + 11^2
1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2
1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2
2^2 + 4^2 + 6^2 + 12^2
2^2 + 14^2
3^2 + 5^2 + 6^2 + 7^2 + 9^2
6^2 + 8^2 + 10^2
```

Some numbers (such as 128 or 0) cannot be represented as a sum of squares, in which case your method should produce no output. Keep in mind that the sum has to be formed with unique integers. Otherwise you could always find a solution by adding `1^2` together until you got to whatever number you are working with.

As with any backtracking problem, this one amounts to a set of choices, one for each integer whose square might or might not be part of your sum. In many of our backtracking problems we store the choices in some kind of collection. In this problem you can instead generate the choices by doing a for loop over an appropriate range of numbers. Note that the maximum possible integer that can be part of a sum of squares for an integer *n* is the square root of *n*.

Like with other backtracking problems, you still need to keep track of which choices you have made at any given moment. In this case, the choices you have made consist of some group of integers whose squares may be part of a sum that will add up to *n*. Represent these chosen integers as an appropriate collection where you add the integer *i* to the collection to consider it as part of an answer. If you ever create such a collection whose values squared add up to *n*, you have found a sum that should be printed.

To help you solve this problem, assume there already exists a method printHelper that accepts any Java collection of integers (such as a list, set, stack, queue, etc.) and prints the collection's elements in order. For example, if a set `s` stores the elements `[1, 4, 8, 11]`, the call of `printHelper(s);` would produce the following output:

```
1^2 + 4^2 + 8^2 + 11^2
```