CCPS 109, Recursion Exercises

You might recognize some of the following problems from the earlier labs and class exercises. However, now these methods are required to be **fully recursive** in that they contain **no loops at all;** neither for, while or do-while.

1. *void listNumbers(int start, int end)* that outputs the numbers from *start* to *end* to console. Write one version that outputs the numbers in ascending order, and another that outputs them in descending order.
2. *String repeat(String s, int n)* that creates and returns a new string that is made by concatenating *n* copies of the parameter string *s*. For example, calling this method with the parameters "Hello" and 3 would return the string "HelloHelloHello". If *n* equals zero, the method should return the empty string.
3. *int min(int[] a, int start, int end)* that returns the smallest element between the indices *start* and *end* in the parameter array a.
4. *int mul(int a, int b)* that computes the product of two integers *a* and *b*. The only arithmetic operation that you are allowed to use in this problem is addition +.
5. *double power(double a, int b)* that calculates the power a^b. You are allowed to use the multiplication operator *.
6. *double harmonicSum(int n)* that calculates and returns the sum $1 + ½ + 1/3 + ... + 1/n$.
7. *String mySubstring(String s, int start, int end)* that works like the *substring* method of the *String* class. You can only use the *String* methods *charAt* and +.
8. *int count(String s, char c)* that counts how many times the character *c* occurs inside string *s*. You can use the *String* methods *charAt* and *substring* to extract information from string *s*.
9. *String disemvowel(String s)* that creates and returns a string that is otherwise the same as the parameter string *s*, except that all vowels have been removed. (For simplicity, assume that you already have the method *boolean isVowel(char c)* that tells whether the character *c* is a vowel.)
10. *int sumOfDigits(int n)* that computes and returns the sum of digits of the positive integer *n*. For example, when called with the parameter 12345, this method would return 15.
11. *int reverseDigits(int n)* that returns the positive integer that you get when you reverse the digits of parameter *n*. For example, when called with the parameter 12345, this method would return 54321. (Do this with proper integer arithmetic instead of just simply converting to *String*, reversing that and then using *parseInt* to extract the result.)
12. *boolean subsetSum(int[] a, int n, int goal)* that checks if it is possible to select some subset of the first *n* elements of the array *a* so that the selected elements add up exactly to *goal*.
13. *boolean binarySearch(int[] a, int start, int end, int x)* that uses the binary search algorithm to check whether the sorted array *a* contains the element *x* anywhere between indices *start* and *end*, inclusive.
14. *void shuffle(int[] a, int start, int end, Random rng)* that shuffles the elements in the subarray from *start* to *end*, getting the random numbers it needs from the *rng* provided as parameter.
15. The algorithms for binary search trees can often naturally be written recursively, since a binary search tree itself is a recursively defined structure: a BST is either empty (this is the base case of structural recursion), or it consists of a root node whose left and right children are themselves BST's. Assume that the class *Node* has fields *int key*, *Node left* and *Node right*. Write a recursive method *Node maximum(Node root)* that returns a reference to the node that contains the maximum value in the tree that starts from the given *root* node.
16. Write a recursive method *Node contains(Node root, int key)* that returns a reference to the node

that contains the given *key*, and returns *null* if there is no such node.

17. Write a recursive method *void add(Node root, int key)*. You can assume here that the tree is initially nonempty. However, if the tree already contains the given *key*, this method should not add another one (so our tree is a *set*, not a *multiset*).
18. Write a recursive method *void rangeSearch(Node root, int min, int max)* that outputs all the keys in the tree whose value is between *min* and *max*, inclusive. Don't just recursively walk through the whole tree, but examine only those branches of the tree that you really need to examine.
19. The empty tree has a *height* of zero, and otherwise the tree height is one plus the length of the path from the root to the farthest leaf node. Write a recursive method *int height(Node root)* to compute the height of the given tree.
20. Write a method *Node createTree(int[] a, int start, int end)* that creates and returns a BST that contains the elements of the array *a* between the indices *start* and *end*, inclusive. You can assume that the parameter array *a* is already sorted in ascending order.
21. When you have a set of n people, how many ways are there to choose a committee of k members, that is, a k-element subset? The order in which you choose the people who end up in the committee doesn't matter: that is, first choosing Alice, then Bob and then Carol is the same as first choosing Carol, then Bob and then Alice. The answer is given by the recursive equation with base cases $C(0,k) = 0$ and $C(n,1) = n$, and recursion $C(n,k) = C(n\text{-}1,k\text{-}1) + C(n\text{-}1,k)$. Write this function as a recursive method *int choose(int n, int k)*.
22. In a two-player game played with an array of numbers, the player whose turn it is to move can take a number from either the beginning or the end of the array, and gets that many points. The game is over when the array becomes empty, and the player whose point total is greater wins. Write a method *int gameValue(int[] a, int start, int end)* that computes the result for the game for the player whose turn it is to move, where the subarray of *a* from *start* to *end* contains the numbers that remain.
23. You are standing at the point (n,m) of the grid of positive integers and want to go to origin (0,0) by taking steps either to left or down: that is, from each point (n,m) you are allowed to move either to the point $(n\text{-}1,m)$ or the point $(n,m\text{-}1)$. Write a method *int countPaths(int n, int m)* that counts the number of different paths from the point (n,m) to the origin.
24. In the famous *knapsack problem*, you have in front of you a bunch of items, each with a *size* and a *value*. Your task is to select a subset of elements so that the total value is maximized and their total size is at most equal to the *capacity* of your knapsack. Write a recursive method *int knapsack(int[] s, int[] v, int n, int capacity)* that returns the maximum value that you can get by choosing from n items whose sizes and values are given in arrays s and v, respectively.