CS 106B Section 3 Handout (Week 4)

1. Out of Space

A buggy program you wrote for CS106B has filled up your CS106B folder with thousands of large files full of nothing but junk data! It would take too long to delete all of the junk files one by one and you don't want to delete your source code. You realize that your only option is to write a program to clean out the junk data without removing your source code.

Lucky for you, the buggy program would always use the ".data" file extension for the data files it created. You decide to write a recursive function which removes all .data files in a given directory and its subdirectories. You write out the following function prototype and wonder what to do next:

```
void cleanDirectory(string startingDirectory, string extension);
```

Fill out the body of this function and reclaim your hard drive space! You may find the following methods from the Stanford C++ Library useful:

`deleteFile(name)`	Removes file from disk
`getExtension(name)`	Return the file's extension (e.g. "foo.cpp" → ".cpp")
`listDirectory(name, v)`	Fills the given Vector<string> with the names of all the files contained in the given directory
`isDirectory(name)`	Returns whether this file name represents a directory
`isFile(name)`	Returns whether this file name represents a regular file
`getCurrentDirectory()`	Returns directory the current C++ program runs in
`fileExists(name)`	Returns whether this file exists on the disk
`renameFile(old, new)`	Changes a file's name

2. Out of Space (again!)

Oh no! It looks like your solution to the last problem didn't work. You still have lots of large data files in our CS106B folder, taking up your precious disk space! Careful investigation reveals that the program which created these obnoxious data files was so buggy that it didn't even give the files the correct file extension!

Undeterred, you put on your recursion hat again and start to think up a new way to solve this problem. Instead of searching for a common file extension, you observe that the data files are much larger than your source code files. This observation leads you to try a new algorithm: remove all of the files in the folder who have a larger than average filesize (averaged over all files in the originally specified directory and its subdirectories).

Write a recursive function which implements this clean-up. Your function should take a single string as a parameter, representing the starting directory to look for files and subdirectories in. You may assume you have a function which returns the size of a file:

```
int filesize(string filename); // returns the size of the given file
```

3. waysToClimb

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 function `waysToClimb` that takes a positive 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. Output each way to climb the stairs on its own line, using a 1 to indicate a stride of 1 stair, and a 2 to indicate a stride of 2 stairs. For example, here are two calls to your function and their corresponding output:

`waysToClimb(3);`	`waysToClimb(4);`
`{1, 1, 1}`	`{1, 1, 1, 1}`
`{1, 2}`	`{1, 1, 2}`
`{2, 1}`	`{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. You may assume that the number of stairs passed is non-negative.

4. maxSum

Write a recursive function `maxSum` that accepts a reference to a vector of integers *V* and an integer limit *n* as parameters and uses backtracking to find the maximum sum that can be generated by adding elements of *V* that does not exceed *n*. For example, if you are given the vector {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 vector is empty, or if the limit is not a positive integer, or all of *V*'s values exceed the limit, return 0.

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

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

Vector *V*	**Limit *n***	**maxSum(*V*, *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 vector are non-negative. Your function may alter the contents of the vector *V* as it executes, but *V* should be restored to its original state before your function returns.

5. **partitionable:**

Write a function partitionable that accepts a reference to a vector of integers as a parameter and uses recursive backtracking to discover whether the vector can be partitioned into two sub-lists of equal sum. You should return true if the given vector can be partitioned equally, and false if not. The table below indicates various possible values for a variable named v and the value that would be returned by the call of partitionable(v):

Vector Contents	**Value Returned**	**Example Partition**
{}	true	{} and {}
{42}	false	
{1, 2, 3}	true	{1, 2} and {3}
{1, 2, 3, 4, 6}	true	{1, 3, 4} and {2, 6}
{2, 1, 8, 3}	false	
{8, 8}	true	{8}, {8}
{-3, 3, 14, 8}	true	{-3, 14} and {3, 8}
{-4, 5, 7, 2, 9}	false	

For example, the vector `{1, 2, 3, 4, 6}` can be split into `{1, 3, 4}` and `{2, 6}`, both of which have a sum of 8. For the vector `{2, 1, 8, 3}`, there is no way to split it into two sub-lists whose sum is equal.

You are allowed to modify the vector passed in as the parameter as you compute the answer, as long as you restore it to its original form by the time the overall call is finished. Note that the vector might be empty.

Hints: If the overall vector is partitionable, every integer from the vector must become part of one of the two sub-lists. The challenge lies in trying every possible combination of each number in one sub-list or the other. As with all recursive problems, ask yourself, in what way is this task self-similar? How is partitioning this vector related to the task of partitioning some other, smaller vector?