**Forest Fire**

***Recursive backtracking*** is a problem-solving technique that utilizes recursive method calls to find solutions to a problem by exploring all possible combinations (subsets) of that problem.

When a backtracking algorithm reaches the "bounds" of a problem, it has two options: return the solution, or realize that the current attempt is not a valid solution and "backtrack" to a previous attempt (location).

1. Review the background info in the lab folder. Understanding how backtracking algorithms work will greatly increase your chances of success in writing the following methods.
2. You don't need the starter code yet (but you can import it for later if you want).
3. For the following problems, it may be necessary to make "helper" methods that have different parameter lists than the "client" methods. For example, the printBinary method below may have a helper method private void printBinaryHelper(int digits, String result).
4. Complete the method void printBinary(int digits) that prints the all the different binary numbers with digits number of digits, separated by a space.

printBinary(3) >>> 000 001 010 011 100 101 110 111

**Hint:** for one bit (one digit), the two possibilities are 0 and 1. For two bits: 00, 01, 10, and 11. Base case suggestion: your binary String's length is equal to the number of bits requested. Otherwise, try adding a zero, then try adding a one. Do your best to solve this problem on your own; if you're completely stuck, [this](https://youtu.be/KBHwDiNTmr8) video will walk you through it.

1. Complete the method void climbStairs(int steps) that prints the all the different ways you can climb a staircase with steps number of steps. Each step, you can take either a large stride or a small stride. A large stride will move up two stairs, a small stride will move up one stair.

climbStairs(4): [1, 1, 1, 1]

[1, 1, 2]

[1, 2, 1]

[2, 1, 1]

[2, 2]

1. If you're having problems with these exercises, it may be helpful to try the [CodingBat](http://www.codingbat.com) recursive backtracking problems (all the "Recursion-2" problems are backtracking). In classic CodingBat style, they are short, fairly easy, and the first few have hints / solutions available.
2. You're in the middle of the woods, and you know that your campsite is in the north-east corner of the woods. Complete the method void campsite(int x, int y) that prints the different routes you can take to get to your campsite at coordinates x, y. Your starting position will be 0, 0 and you can move (one mile at a time) east (E), north (N), or north-east (NE).

campsite(2, 1): E E N

E N E

E NE

N E E

NE E

1. Complete the method int getMax(List<Integer> nums, int limit) that returns the maximum sum that can be generated by adding elements of nums without going over limit.

getMax(Arrays.asList(7, 30, 8, 22, 6, 1, 14), 19) >>> 16

1. Complete the method int makeChange(int amount) that returns the number of different ways you can make change (given standard US coins of 1, 5, 10, 25) for amount. What is the time complexity (the Big-O) of this recursive algorithm?

makeChange(25) >>> 13

makeChange(100) >>> 242

1. Create a copy of the method above, and modify it such that it now prints out all the different ways change can be made, in this format:

 P  N  D  Q

------------

[1, 0, 1, 0]

[1, 2, 0, 0]

[6, 1, 0, 0]

[11, 0, 0, 0]

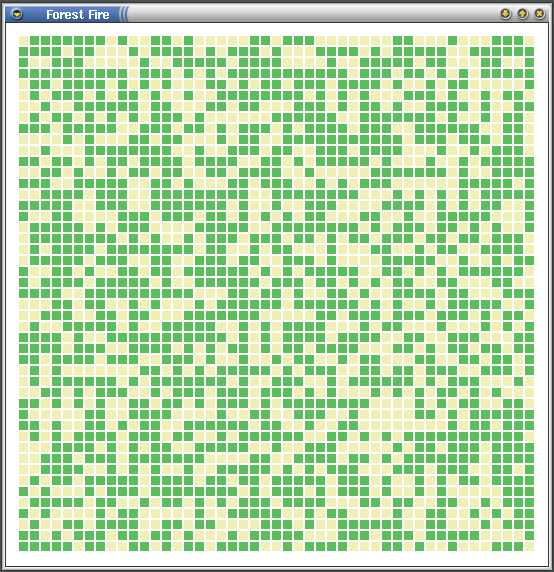
1. (Riddle) I can run but not walk. Wherever I go, thought follows close behind. What am I?
2. Complete the method String longestCommonSub(String a, String b) that returns the "longest common subsequence" that appears in both Strings. A common subsequence is a sequence of characters that appear in both Strings in the same relative order. This problem is somewhat less intuitive that the previous problems; try it on your own first, but if you get stuck, algorithm help can be found in the lab folder.

longestCommonSub("ABCDEFG", "BGCEHAF") >>> "BCEF"

longestCommonSub("12345", "54321 21 54321") >>> "123"

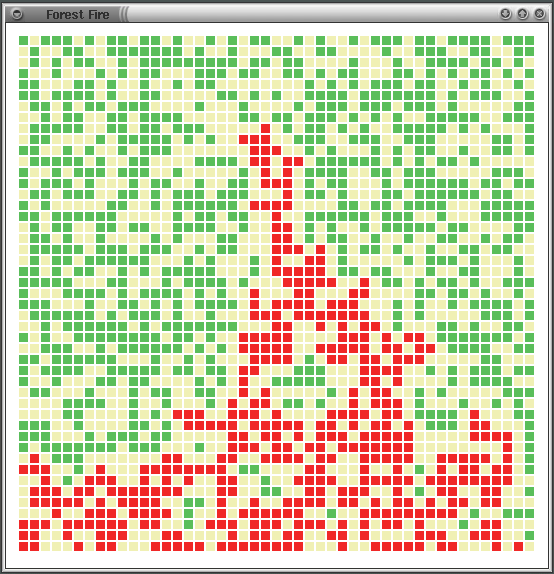
**Forest Fire**

You live in a beautiful mountain town called Onett in Colorado. Here is a map of one section of the forest just to the south of the little town, showing the trees in green and open spaces in tan:

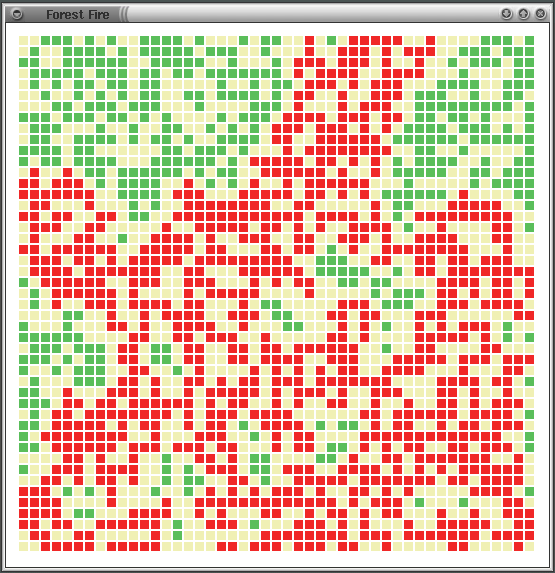


Occasionally in the summer, lightning strikes in the mountains start forest fires. The fire spreads from tree to tree; if a fire anywhere along the south side of this section of trees would make it to the northern border (where the town is), Onett will be in trouble.

Some forest configurations are too sparse, and the fire will not make it to the town:



Other forest configurations do provide a path for the fire to reach the northern edge of the forest:



If this is the case, the fire-fighters would like to know where to put their resources, so a program that maps the spread of the fire would be useful.

**The Assignment**

Given a "forest" with a 60% probability of having a tree in any location, write a program to determine if the fire will have a path from the bottom edge to the top edge. Fires can only go from tree to tree horizontally or vertically. If there is a path, display it in red, including all trees that would be destroyed.

Initially set each tree on the southern border on fire. Recursively call a method that sets adjacent trees on fire. When the recursion is complete, display the fire map. If a tree on the northern border is reported as on fire, print a message akin to "Onett is in trouble!", otherwise print "The town of Onett is safe."

You are given the following files, which follow the Model-View-Controller (MVC) pattern. Most of your code will go in the **FireModel** class, though you may need to modify the **FireCell** class as well.

|  |  |
| --- | --- |
| **Fire.java** | Controller (runner), contains a main method |
| **FireCell.java** | Individual location in a grid; could be dirt, a tree, or a burning tree |
| **FireModel.java** | Fire's path model class, contains the bulk of the logic |
| **FireView.java** | Contains a graphical view; displays fire map. |

*Forest Fire lab*

*http://www.rfrank.net/cslabs/1660-fire/1660.html*

**(Advanced) Number arrangements**

Suppose we have a collection (i.e. a set) of numbers: {1, 2, 3, 4} and we want to find all possible arrangements of these numbers. We'll define an arrangement of our set as a list of numbers consisting of every member of our set exactly once. So, {1, 2, 4, 3} and {2, 3, 4, 1} are valid arrangements.

However, {2, 2, 4, 3} and {5, 1, 2, 3} are not valid arrangements. For our purposes, we'll treat our set of numbers as a Set<Integer> and each specific arrangement as an int[].

Write the method Set<int[]> generateAllArrangements(Set<Integer> set) that returns the set of all possible arrangements of the given set of Integers.

**(Advanced) Knight's Tour**

A ***knight's tour*** is a series of moves by a knight on a chess board such that the knight visits every square on the board once (moving per the rules of chess). Wikipedia has a great visualization of a knight's tour [here](https://en.wikipedia.org/wiki/Knight's_tour#Brute-force_algorithms).

Write a program that will print a knight's tour for an n x n chess board. Your program should have the following:

* int[][] board – the chess board (all positions should initially start at 0)
* int size – size of the chess board, i.e. n x n
* void solve() – method that will print the knight's tour, starting at index location [0, 0]
* an overridden toString method for printing the tour

An example of a knight's tour for an 8 x 8 board, starting at [0, 0]:

1 60 39 34 31 18 9 64

38 35 32 61 10 63 30 17

59 2 37 40 33 28 19 8

36 49 42 27 62 11 16 29

43 58 3 50 41 24 7 20

48 51 46 55 26 21 12 15

57 44 53 4 23 14 25 6

52 47 56 45 54 5 22 13

Once you have it working, modify your code such that you can find a solution starting from any [row, col]. Example of a knight's tour of a 5 x 5 board, starting at [0, 2]:

25 14 1 8 19

4 9 18 13 2

15 24 3 20 7

10 5 22 17 12

23 16 11 6 21 //note that not all starting positions have solutions