**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 in a general way 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 soFar).
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 zero bits, there are no possible binary numbers. For one bit (one digit), the two possibilities are 0 and 1. For two bits: 00, 01, 10, and 11 (all combinations of 0 and 1). There will be two recursive calls, adding either a "0" or "1". Consider this image of all the possibilities:

Diagram

Description automatically generated

When the soFar String reaches the desired length, it *backtracks* to the previous call and tries other combinations of digits.

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

**Hint:** If you're done traversing the list, return the potential max sum or, if the current sum is over the limit, a very small number. If neither of the previous cases is true, return the larger of both possible courses of action (adding the current element and skipping the current element).

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 //see the FAQ if you have a much higher number!

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

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

**Hint:** Use a HashSet<String> object to prevent printing duplicate solutions. A HashSet is a data structure that is essentially a HashMap with only keys, i.e. it prevents duplicate values.

[1, 0, 1, 0]

[1, 2, 0, 0]

[6, 1, 0, 0]

[11, 0, 0, 0]

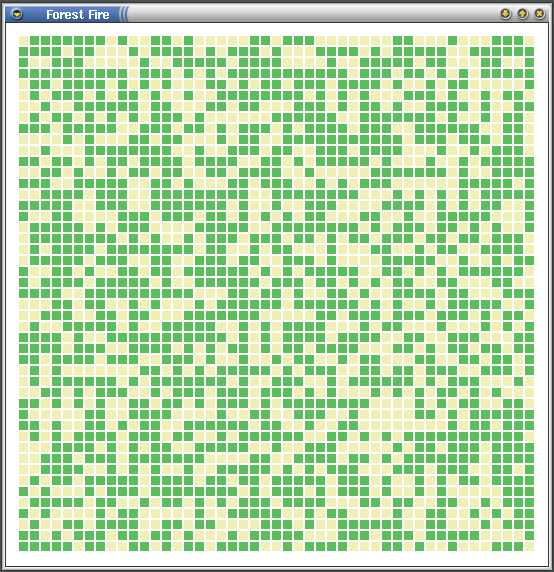
1. (Riddle) I can run but not walk. Wherever I go, thought follows close behind. What am I?
2. Complete the method String longestCommonSub(String a, String b) that returns the "longest common subsequence" that appears in both Strings. A common subsequence is a sequence of characters that appear in both 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"

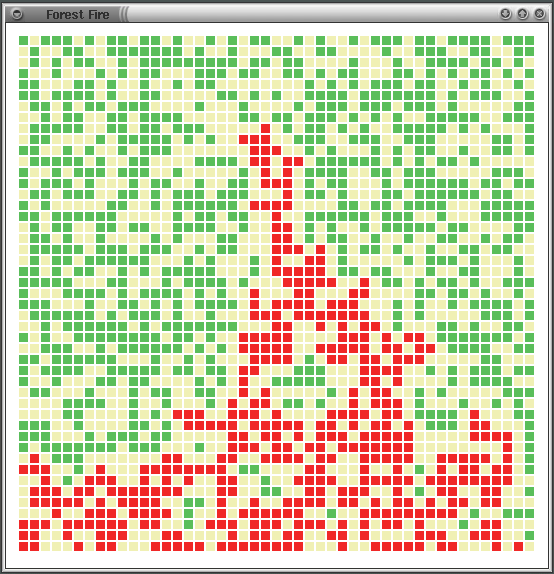
**Forest Fire**

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:

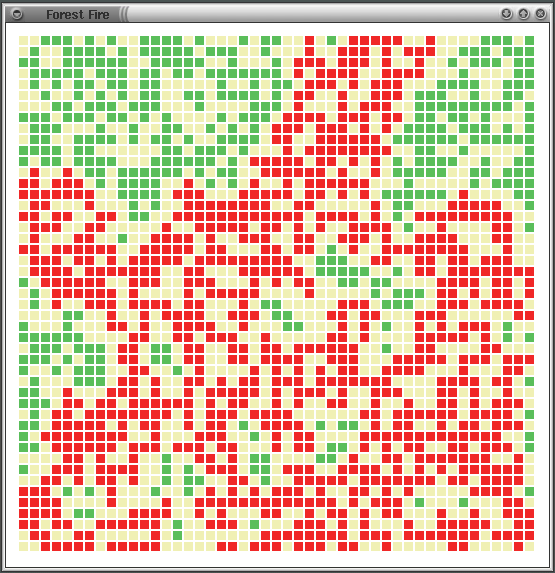


Occasionally in the summer, lightning strikes in the mountains start forest fires. The fire spreads from tree to tree; if a fire anywhere along the south side of this section of trees would make it to the northern border (where the town is), Onett will be in trouble.

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.

Use recursion to set adjacent trees on fire, starting with every tree on the southern border. 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 "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.

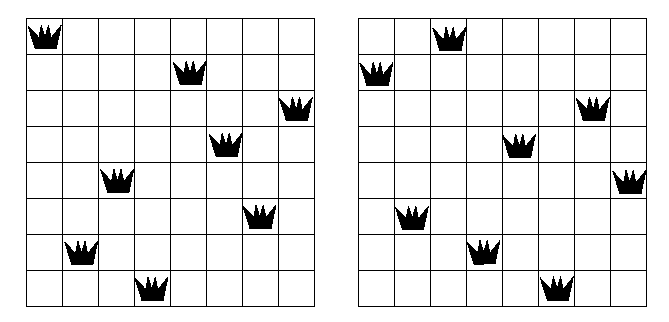
|  |  |
| --- | --- |
| **Fire.java** | Controller (runner), contains a main method |
| **FireCell.java** | Individual location in a grid; could be dirt, a tree, or a burning tree |
| **FireModel.java** | Fire's path model class, contains the bulk of the logic |
| **FireView.java** | Contains a graphical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

*Forest Fire lab*

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

**(Advanced) Eight Queens**

The eight queens problem attempts to place eight chess queens on an 8×8 board such that none of them attack one another (no two are in the same row, column, or diagonal). A couple solutions for an 8x8 board:



**Hints:**

* An abstraction of the board which handles much of the logic will simplify the recursive code.
* You can output the results to the console, but the program is more interesting with graphics. Build your own GUI (or use Greenfoot). If you get stuck, [here](https://www.dropbox.com/s/glrbh0ul28nw38e/BoardFrame.java?dl=0) is an example. Your Board class would need to be consistent with what the GUI expects.

**(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 nice 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