4100/5100 Assignment 1: A*, Heuristics, and the Fifteen Puzzle
Due Thursday May 17 9PM

In this assignment, you'll get some experience abstracting a problem into a search problem, implement the A* search algorithm (still the most common way of doing pathfinding in video games), and experiment with the effects of using different heuristics for the search.

Recall that the goal of a fifteen puzzle is to get all fifteen tiles in order from left to right, top to bottom, like so:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -

The only legal moves are to move a tile adjacent to the blank into the blank space, making the tile's previous space blank. Thus the maximum branching factor is 4, but the number of neighbors could be as small as 2 if the blank is in a corner.

1) Download the starter code `NumberPuzzle.java` from the same location you got this assignment.
2) Add the code that generates a list of neighbors for a particular puzzle configuration.
3) Implement A*, using a heuristic of "number of tiles in the wrong place" as the optimistic estimate of moves to go. Use a PriorityQueue generic to store your search nodes to ensure you have an efficient implementation. Your search nodes stored in the queue will need to implement the Comparable interface, using their distance traveled + heuristic values to decide the results of the compareTo method.
4) Submit your code to HackerRank and ensure it passes all our tests. You'll submit again later, but you should probably test now.

`https://www.hackerrank.com/contests/cs41005100-summer-1-hw1-fifteen-puzzle/challenges`

5) Time your implementation locally on the sixteen-move solution (test 3 on HackerRank). You can do this either using System.nanoTime() to get timestamps before and after your code runs on the hardcoded puzzle, or you can just type
`time java NumberPuzzle < sixteenMoves.txt`
at the command prompt if you're on a Mac/Linux machine (you want the user time). (If you're using timestamps, comment out the code where you print the time difference before submitting to HackerRank again so that you don't fail the tests.) Please report the time in reasonable units (not, say, "1232351241 nanoseconds") at the top of your java file in a comment.
6) Add a parameter to your A* code where, if it is set to false, it uses your old heuristic, and if it is set to true, it uses a Manhattan distance heuristic instead. (The Manhattan distance is the distance when you can only move up/down/left/right instead of diagonally

-- kind of like driving around a city environment like Manhattan, I guess -- and it will here just be the row difference plus column difference for the tile's actual vs. goal position.)
7) Hardcode the "true" value to set your heuristic to Manhattan distance and submit to HackerRank, again ensuring the code passes all our tests.
8) Time your code with this new heuristic on the sixteen move puzzle, like in step 5. Add this recorded time to the comments at the top of your java file.
9) At the top of your Java file, near the times, answer the following question: what would the likely effect be of using Euclidean distance (straight-line distance, allowing diagonals) instead of Manhattan distance or tiles displaced as the heuristic? Would this work? How is it likely to rank in speed, and why?
10) **Put your name in the comments** at the top of the file as well, so we can compile a list of student HackerRank usernames. **If you are auditing, please mention that as well.**
11) **Submit one last time to HackerRank**, so that we have your final comments. Do not leave in any code that causes you to fail the tests.

In the end, you will be submitting a working A* implementation that uses Manhattan distance, but for which it is easy to change one variable's value to use tiles displaced instead. **At the comments at the top will be your name, times for both heuristics you implemented, and your answer to the question about Euclidean distance.**

A* is one of the most successful algorithms in the history of AI, a champ at what it does (as long as you can come up with a good heuristic), and is probably the most important AI technology for a modern game AI programmer to understand (because of all the pathfinding that goes on). You're now well on your way to having some great AI skills!