Q. I don't have much time to spare for this project...

A. This project can easily eat up all of your time, so writing bug-free code incrementally is the key to minimize development time. In particular, implementing minimax (and then alpha-beta pruning) correctly is most important, because i**f you implement your alpha-beta pruning correctly, you can already get at least [256, 256, 256, 256, 256] (38/100 points) with a very simple heuristic such as available number of tiles**. **Please make sure that your minimax/alpha-beta are correct before working on heuristics and save the correct version of minimax in case if you need to rollback.** Then you can reach five 512 (63/100 points) or more in the next two or three hours by tweaking heuristics.
You might find **this website** useful to understand minimax search and alpha-beta pruning deeply.

Q. How can we enforce time limit?

In the **GameManager** class, you will see that we enforce the time limit strictly. That is, if the getMove() function takes longer than 0.2 seconds to return a move, the game will **immediately terminate**, and the maximum tile value will be assessed pre-maturely.
Think about how to enforce this rule strictly on your end. In particular, **please do not rely on a depth-limit as a basis for limiting the running time.** Some past submissions used fixed max depth such as **if depth => 4: return** for adjusting search time. However, while such an algorithm may manage to abide by the time limit via a depth limit on your own computer, there is no guarantee that it will on other machines, in particular on the grading machine. Your assignments will all be executed on the grading platform, which evens the efficiency playing field for all submissions, and holds everyone accountable for the same time limit. Some of the past submissions were forced to a halt by **GameManager** class' time limit logic, **typically before the very first PlayerAI move**. This is because the first move has the nearly-maximum branching factor, and **the run will get the maximumTile of 2 or 4...**
There are multiple tools out there to help you clock time. However, make sure you are relying on **CPU time**, instead of **wall time**, as your clocking mechanism. If you take a look at GameManager.py, you will see that we are using time.clock() on our end to enforce the limit. If you want to be safe, then use exactly that for your own implementation-side limit as well.

Q. Are there more grading examples?

A. The followings are the examples of 10 maximum tile values and corresponding grades (top 5 run values are shown):

- [1024, 1024, **2048**, **2048**, 4096] => 100 / 100 points (upper-bounded by 100)
- [1024, 1024, **2048**, **2048**, **2048**] => 100 points (most typical full points)
- [1024, 1024, 1024, **2048**, 4096] => 100
- [1024, 1024, 1024, **2048**, **2048**] => 98
- [1024, 1024, 1024, 1024, **2048**] => 93
- [1024, 1024, 1024, 1024, 1024] => 88
- [512, 1024, 1024, 1024, **2048**] => 88
- [512, 1024, 1024, 1024, 1024] => 83
- [512, 512, 512, 512, 512] => 63

As shown in the above examples, maximum credit is capped by 100 (thus [1024, 1024, **2048**, **2048**, **2048**] is not 103 but 100). As one of five runs receives one-level smaller/higher values, the credit will decrease/increase by 5 points.

Note that if you employ time limit by max depth, it is possible that you would get all runs with 2 or 4 maximum tile values and would get zero point.

Q. How can we come up with good heuristics?

A. You might consider watching some of pro 2048 players' videos. You would observe a few patterns to survive long in this game, and these observations are useful for understanding whether your AI is playing well or not.

Your heuristics will likely make or break your player. Consider both qualitative and quantitative measures, such as:

- the absolute value of tiles,
- the difference in value between adjacent tiles,

- the potential for merging of similar tiles,
- the ordering of tiles across rows, columns, and diagonals,
- etc. There are endless possibilities...

You are also encouraged to research on 2048-specific heuristics. **Here** is a basic AI written in JavaScript. You are supposed to do better! Another hint is this interesting StackOverflow **discussion**. Especially, you might find "Monotonicity" and "Smoothness" heuristics interesting. There are lots of great ideas everywhere, but remember that you are required to use **the minimax algorithm** (not expectimax in the StackOverflow discussion) and alpha-beta pruning in this assignment.

Q. How can we combine multiple heuristics?

A. As mentioned in project instruction page, one simple but strong approach is to add weights for each heuristic value. For example, if you consider using two heuristics, averageTileNumber and MedianTileNumber, the return value looks like this:

```
return weight1 * averageTileNumber + weight2 * MedianTileNumber
```

You could decide what weight values are the best for your AI's performance by trying different weights. There are more systematic ways to tune parameters (and that is the important part of the history of game AI), as we said "If you feel adventurous, you can also simply write an optimization meta-algorithm to iterate over the space of weight vectors, until you arrive at results that you are happy enough with." Perhaps you could set some "intermediate" states as initial states for faster experimentation. Note that grading script will override read-only files before grading.

Having said that, guessing would be enough effective in the initial design phase.

Q. How can we combine multiple heuristics with different units?

A. Suppose that you are going to combine two heuristics: **availableNumberOfCells** and **averageTileNumbers**. If we naively combine these two heuristics, the best heuristic value is almost governed by averageTileNumbers because it can be significantly larger than **availableNumberOfCells**.

One possible quick fix is to apply **math.log2** for the **averageTileNumbers** heuristic. However, what transformation is the best is not necessarily obvious.

Q. How many heuristics should be combined?

A. Basically, 2 - 5 heuristics are enough to reach 2048.

Q. How long does the grading script take to complete?

The grading time varies by the impletementation of the algorithm. Typical runtime of a full score algorithm should take a few hours to complete. A algorithm with max tile 64 ~ 128 should only take around 10 to 20 mins to complete.

Q. Is skeleton code optimized?

A. As the astute student might observe, the skeleton code provided may not be the most efficient. In particular, the operations: move(), getAvailableCells(), insertTile(), and clone() are by no means the most efficient implementations available for achieving the desired functionality. However, the purpose of the assignment is to gain practice in adversarial search and heuristic evaluation, and not to optimize object representation with bit-arrays and low-level operations.
Everyone will be using the same skeleton code. If you wish, you are free to write your own helper methods in a separate file (remember, Grid.py is read-only). However, this is by no means necessary; in fact, students have written player AIs that have beaten the game handily by employing the given methods. **You should focus your efforts on developing a smart algorithm and heuristics first.** If you have additional time, or as an extra boost (or last resort...), you are certainly encouraged to improve these methods by writing your own efficient ones.