Rationale

The aim of this Math IA is to explore the probabilities behind the game *2048* in order to answer questions I had upon first playing the game, such as optimum strategy, whether it is even possible to win, and if so, how probable it would be, as well as using probabilities to establish a generalization for the winning tile and looking into different algorithms that may be used to solve the game. This topic interests me because it is a game that I like to play, and because it allows me to delve into the subject of Computer Science, which I plan on studying. My purpose then is to clear up questions others may still have about this famous game, and bring the subject of game strategy and programming a bit closer to readers of this paper, while looking at what kind of probabilities make *2048* so addicting.

In order to achieve these goals, I created proofs by mathematical induction, calculated probabilities, and made generalizations by using geometric sequences and sums, exponent rules, laws of probability, and frequency tables and graphs. I also wrote my own programs in Python for the purpose of this IA, and tried my hand at manipulating source codes (Gabrielecirulli & Ovolve) and existing websites to be able to have the exact tools that I needed for my investigation.

Three years ago, there was a craze around the game *2048* (Cirulli). Like everyone else, I tried to win by getting the "2048" tile, but never succeeded. I will now investigate probabilities of winning by executing random moves, optimum strategies, and successful algorithms that can win the game 100% of the time.

2048 is a simple game played using the arrow keys on a 4x4 board. At the beginning, two tiles are present, and upon every move (press of an arrow), a new tile is spawned, with a probability of it being a 2 of 90% and a chance of it being a 4 of 10% (Gabrielecirulli). If two tiles of the same value are moved to be on top of each other, they merge to the sum of the two numbers. Therefore, all the numbers created will be a power of 2.

Probability of winning the game:

In order to reach the maximum sum (where the largest number on the field is 2^{17}, and the field is filled with a decreasing sequence of powers of 2), we need to spawn a 4 in order to have the most efficient decomposition (meaning, there is a monotonic sequence which takes up as little space as possible by having less terms - we have one 4 on the field instead of two 2s). However, this is only 10% likely. And, as it turns out, 2^{17} is not the only number where this requirement holds true. Using logic, we can establish that this will always be the case with numbers that, with maximum efficiency in their decomposition (i.e., the smallest number is a 4 and not two 2s), use up all 16 tiles. Using maths to prove this:

The geometric sum of any of these configurations will be $(2^1 + 2^2 + 2^3 ... + 2^{17}) - 2^k$ where k is between 1 and 17, as for any of these numbers, we will have exactly 16 terms. That means that on the playing field, we will have a configuration of the above geometric sequence, but it will be incomplete as the entire sequence contains 17 terms, but we only have 16 tile spaces. We are therefore missing one term of the sequence, and for each of these configurations we will need to spawn a 4. When $k = 1$, that is the mathematical endcase as the resulting configuration will be 2^{17} with all other tiles filled with decreasing values. However, for $2 \leq k \leq 17$, all 16 tiles will be filled as these numbers need 16 powers of 2 to decompose. As k can therefore take up 16 values, we need to spawn a 4 16 times throughout the game in order to reach the mathematical endcase. We thus need to spawn a 4 at least 16 times throughout the game in order to reach the absolute end configuration.

When k=17: 

Fig. 1. Lê Nguyen Hoang. *Screenshot of Completed Game.* 05/2014. Image modified by candidate Instead of a 2, a 4 should have been spawned.

When k=16: 

Fig. 2. Lê Nguyen Hoang. *Screenshot of Completed Game.* 05/2014. Image modified by candidate Instead of a 2, a 4 should have been spawned.

When k=15: 

Fig. 3. Lê Nguyen Hoang. *Screenshot of Completed Game.* 05/2014. Image modified by candidate Instead of a 2, a 4 should have been spawned.

The probability of spawning a 4 is $\frac{1}{10}$, so to spawn it 16 times, we take $(\frac{1}{10})^{16}$. The probability is therefore $\frac{1}{10^{16}}$. This does not take into account other problems faced within the game, and is based on the player executing optimal moves. One player would need to play 10^{16} games in order to reach this endcase at least once. The endcase is made up of $1+2+4+8+16+32$ $2048s$. My personal average time taken to reach 2048 once is 9 minutes; taking this, we would need the sum of the above sequence, which is $S_6 = \frac{1(1-2^6)}{1-2} = 63$. Then $63 \times 9 = 567$ minutes for reaching $131072...2048$. However, we still have $1024...2$, so we will add another 9 minutes for that. This gives $567 + 9 = 576$ minutes to reach the mathematical endcase. If we calculate using this time (not every game will last this long, and some may be longer due to suboptimal moves being taken - this serves as an average), $576 \times 10^{16} = 5.76 \times 10^{18}$ minutes of non-stop playing would be needed to play that many games. That is equivalent to 9.6×10^{16}

hours, 4×10^{15} days, or over 1.1×10^{13} years. That means it would take 1.46×10^{11} lifetimes. As a comparison, there are currently 7.5×10^{9} people on earth. That means that even if everyone currently alive spent their whole lives playing the game with optimum strategy, we might still never reach the mathematical endcase even once.

However, this only takes into account one impossibility of the game; others include configurations in which a non-optimal move must be taken, which may ruin the player's strategy, and lead to tiles being spawned in worst-case positions (which is a case of robust -- worst-case -- probability). It should be noted, however, that while the possibilities considered are very limited, the calculation assumes stochastic optimality. In other words, the optimality of newly spawned tiles and executed moves is neither best- nor worst-case, but rather average. This means that assuming equal probability of the new tile spawning in any empty tile space, using $P(A) = \frac{n(A)}{n(U)}$, there is a probability of $\frac{1}{n}$ (where n is the number of empty tile spaces) of the tile being spawned in the optimal position (assuming there is 1 optimal position) and a probability of $\frac{1}{n}$ of the tile being spawned in the worst possible location. Assuming the game configuration in Fig. 4, 11 spaces are filled, so that there was a $\frac{1}{5}$ chance of the new tile being spawned in any one position. A 4 would have been spawned in the "worst-case" position (3rd row, 2nd column), with a chance of $\frac{1}{10} \times \frac{1}{5} = \frac{1}{50}$, or 2%. A 2 being spawned in the same position would actually result in the optimal scenario, as merging the tiles into a 2048 could be initiated. The chance of this happening is $\frac{9}{10} \times \frac{1}{5} = \frac{9}{50}$, or 18%. Of course, this is assuming that the screenshot was taken after a move was made, but before a new tile is spawned; in reality, another move must first be taken for another tile to be spawned, resulting in a different situation.

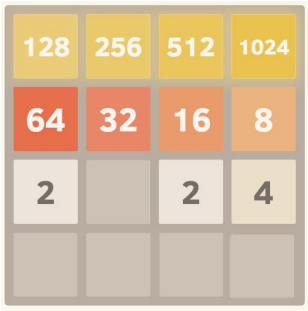

Fig. 4. Image by candidate

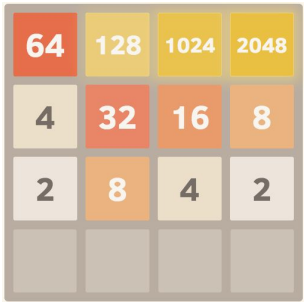

Fig. 5. Image by candidate
Non-optimal move (downward) must be taken to break the draw, exposing the top row to smaller, uncombinable tiles.

In order to reach 2048:

Although I initially thought that I could use the same principle as for the mathematical endcase, I soon realized that in order to make a 2048 tile, one does not need to spawn a 4, as $2048 = 2^{11}$, meaning that its decomposition will take up 12 tiles (including the last 2 required to start merging) or 11 tiles if one spawns a 4. The playing field is therefore (assuming monotonicity in a sequential optimal pattern) only $\frac{3}{4}$ full, and we have more than enough space to spawn 2s and merge them. Thus, it is not essential to spawn a 4 at any point in the game up until one is very close to the mathematical endcase, and the pattern we derived earlier does not apply. As the probability of winning is neither best-case nor robust, but rather stochastic, and involves many different variables and components, it is nearly impossible to compute it. I therefore had the idea to write a short program which executes random (or rather, pseudorandom) moves, and then take its success rate as the experimental probability of reaching 2048. This is, of course, not very accurate, as a human player can never execute completely random moves, and shouldn't, as it is not a very good strategy. However, it is an approximation, and is useful for finding a relation between the winning tile and the size of the field.

To prove the impossibility of calculating the probability of reaching *2048*;
The complete game tree of possibilities for *2048* is huge. We can see how huge by taking the average number of moves required in a game that reaches the 2048 tile (calculated on page 8), which is 1438 moves. As we usually have 4 different moves we can make - up, down, left, right - our game tree would have approximately 4^{1438} game states we would need to take into consideration to calculate the probability.

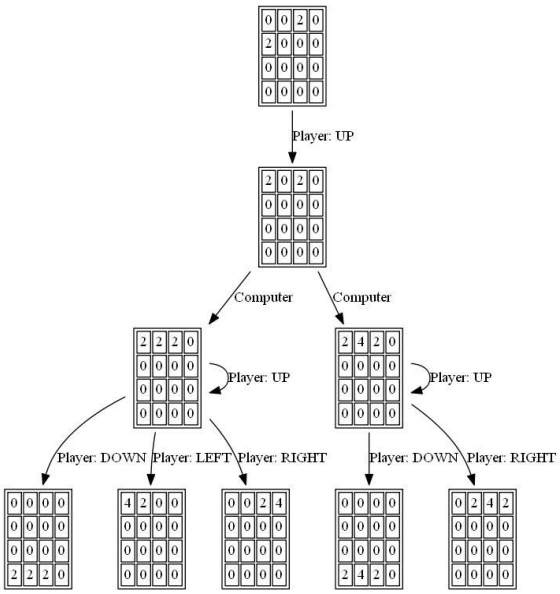

Fig. 6. Sandipan Dey. *Game Tree*. 03/2017. Nitish712, et al.
Sample Game Tree

To make a generalization relating the field size and the maximum possible number creatable:
If we assume that the two games are proportionally isomorphic, that is, essentially the same but in different sizes (examples for perfectly isomorphic games might be that the base is replaced -- 3 instead of 2 -- or the tiles can be replaced by the powers themselves, where merging two powers k results in a tile $k+1$ just like merging two tiles 2^k results in 2^{k+1}). The games would have the same rules, so that the most efficient decomposition involves a tile of q^2 being the smallest present tile rather than q^1. We could then use the same principles as above for our calculations.

Taking n to be the total number of tile spaces in the playing field:
We previously established that the starting power will be 2, and to get the maximum exponent present before the last merges, we add the number of fields containing smaller tiles ($n-1$, as one tile space is occupied by the largest tile), and then subtract 1 as we need an empty tile space in order to spawn the last tile to then be able to start merging. We therefore have $2+(n-2)$ for the biggest tile on the field before starting to merge the remaining tiles into the biggest possible one; in other words, before we start merging, the biggest present tile is q^n. When we have merged all the tiles, we obtain two tiles of q^n, so that the new tile, which will be the biggest possible tile, is q^{n+1}. q can be any positive number bigger than 1 (if $q=1$, all resulting tiles will simply be 1).

However, for 2048, the winning tile is not the biggest possible tile; instead of 2^{17}, it is 2^{11}. My hypothesis then was that this is due to the aforementioned fact that we need to spawn a 4 in order to reach certain tiles, and due to the unlikelihood of that, 2^{11} is hard enough to reach. However, as mentioned before, I realized that $(2^1+2^2+2^3...+2^{17})-2^k$ is only valid for the last few steps of the game when one is trying to achieve the mathematical end configuration and the whole board is filled. As we therefore do not need to spawn a 4 in order to reach 2048, I wondered why else this number specifically was chosen to be the winning tile. Perhaps it was nothing mathematical, but rather the probability of winning was low enough for people to keep trying but rarely succeeding, yet enticing them to keep playing. I therefore used a self-written program to experimentally gain an approximation of the probability of getting to 2048 with random moves. Then, even though this does not realistically represent a human player, we can assume that for other field sizes, this measured probability stays proportional to the actual probability of a human player with some sort of constant strategy of achieving the winning tile. If we then find the random probability, we can make a generalization about the winning tile on different board sizes.

Finding the experimental probability of reaching 2048 using random moves:

For this purpose, I wrote a short program in Python which executes random moves using Pyautogui (Bhandari). For simplicity, I programmed it to open the official game in Google Chrome in order to make sure that all original rules apply. However, that limited me as it made it considerably more difficult to extract information regarding whether the game is over and the maximum tile achieved. Still, I decided against writing an emulation of the game in order to have access to this information. Although I tried to extract the information from the source code using the developer options, it did not work in an optimal way and I thus decided to log the information by hand.

I therefore needed to specify the number of random moves the program should execute; as it would not know when to stop, that number should be enough to be able to achieve 2048, but not too far above it as that would waste time with unnecessary moves when the game was already over. In the program above, the program would execute 50 moves. I therefore calculated the minimum number of moves required to achieve 2048:

For the minimum number of moves, we assume that a 4 is spawned every time, as that reduces the number of moves needed. We would then need $2048\div 4=512$ 4s. However, considering that we start the game with two tiles, and we assume that both of these are 4s, we can subtract 2 moves, so $512-2=510$ is the number of 4s we need to spawn after the initial configuration. In other words, we need to execute at least 510 moves in order to spawn enough 4s to be able to create a 2048. This is assuming that we merge all possible tiles with each move, so that we achieve maximum efficiency. When we spawn the last required 4 on our 510th move, we still need to merge all tiles as we will have the following board configuration (again, assuming best-case scenario, where the 4 spawns in the optimal position):

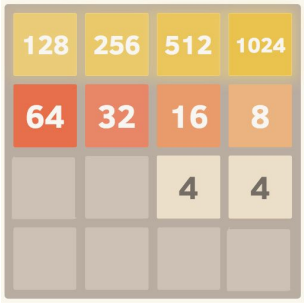

Fig. 7. Image by candidate

As one can see, we still need 9 moves to merge all of these tiles into a 2048. Assuming the best-case scenario, we therefore need $510 + 9 = 519$ moves minimum (Goel).

For the maximum number of tiles, we assume that only tiles of 2 are spawned, but still assume best-case scenario and maximum efficiency. We would need $2048 \div 2 = 1024$ spawned 2 s. As we start with two tiles, assuming these are 2 s, we need $1024 - 2 = 1022$ moves just to spawn all required tiles needed to be merged into a 2048. Using the same principle as above, we then have this configuration:

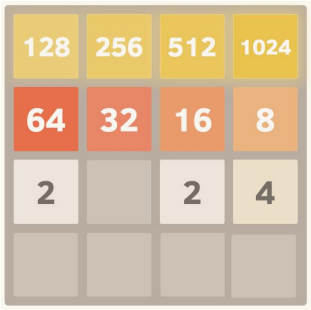

Fig. 8. Image by candidate

Here, we need 10 more moves to merge all remaining tiles into a 2048. Therefore, we need $1022 + 10 = 1032$ moves to reach 2048. However, this is a simplified approximation; it does not account for ineffective or illegal moves, nor for mistakes and suboptimal game strategy. In reality, the maximum number of moves may be infinite as long as the game is not over yet.

However, using these calculations, the number of moves (executing only optimal moves) lies within the range of $[519, 1032]$. We can use the probabilities of spawning 2 s and 4 s to determine the likely number of moves needed. The likelihood of spawning 512 4 s (taking into consideration the initial two 4 s, but not the tiles that will spawn as a result of the merging moves) is $0.1^{512} = 10^{-512}$. The probability of spawning 1024 2 s is $0.9^{1024} = 6 \times 10^{-48}$. If we want to find the likeliest number of moves needed, we need to keep the ratio of $9 : 1$. We can therefore weigh the number of moves that we calculated if only 2 s spawn with 0.9, and the number of moves if only 4 s spawn with 0.1. Therefore $(0.1 \times 519) + (0.9 \times 1032) = 51.9 + 928.8 \approx 981$ moves.

However, this calculation does not account for merges executed during the game; it solely takes into account the number of tiles we need to spawn and the final few merges. So, I recalculated the moves needed in order to make a formula that accounts for all of these things.

If we always spawn 4 s:

We start with two 4 s. Depending on their position, we need either one or two moves to merge them. The first 4 can be in 16 different places. The second 4 can be in 15 different places. Since it does not matter where the first 4 is spawned, we can ignore that one. The second 4 must then spawn in either the same

row or column as the first one in order to be merged within one move. This means that the second 4 may be spawned in 6 different spaces in order to be in the same row or column. The probability of this is $\frac{6}{15} = \frac{2}{5}$. So, for the first 8, with a chance of $\frac{2}{5}$ we need 1 move, and with a chance of $\frac{3}{5}$ we need 2 moves. However, this is only valid for the first 8, as later on the game board with fill up and it is impossible to account for all possible configurations. We can then say that in order to make an 8, if only 4s are spawned, we need 2 moves. For a 16, we then need 5 moves (2 for each 8, and another move to merge them). Continuing this pattern, for a 32 we need 11 moves, etc.

So if n is the power of the tile, we have $f(n) = moves \rightarrow f(3) = 2$, $f(4) = 5$ …

Also, $f(n+1) = 2f(n) + 1 \; for \; n > 3$. Therefore, $f(n) = 2^{n-1} - 2^{n-3} - 1 \; for \; n \geq 3$. (Volko)

For 2048, $n = 11$, so $f(11) = 767$ if only 4s are spawned.

If we always spawn 2s:

The same rules apply as above; we can ignore the instances where only 1 move is required to make a 4, and say that $g(2) = 2$. Again, $g(n+1) = 2g(n) + 1 \; for \; n > 2$. Therefore $g(n) = 2^n - 2^{n-2} - 1 \; for \; n \geq 2$. $g(11) = 1535$ if only $2s$ are spawned. Weighing these again with 0.9 and 0.1 gives $(0.1 \times 767) + (0.9 \times 1535) \approx 1458$ moves on average.

This is still not very accurate for our purposes, of course, as this is assuming optimal moves and no "wasted" or illegal moves where no tiles are moved; with random moves, this number would be much higher. On the other hand, using random moves would make it so unlikely to reach 2048, that we can dismiss this number completely and just see it as a somewhat helpful approximation in that it gives us an idea of how many moves a game might last.

I therefore programmed the bot to execute 200 moves and started to run it and collect my experimental data, which I could repeatedly run in case more moves were needed.

```
import pyautogui
import time
import random


pyautogui.hotkey('command', 'space');
pyautogui.typewrite('goog');
pyautogui.hotkey('enter');

time.sleep(2)
x = 0
for x in range (0,200):
    x+= 1
    rand = random.randint(0,3)
    if rand == 0:
          rand = 'left'
    elif rand == 1:
          rand = 'right'
    elif rand == 2:
           rand = 'up'
    elif rand == 3:
          rand = 'down'
    print(rand)

    pyautogui.press([rand])
```

Fig. 9. Image by candidate

Python program written by candidate

```
down
right
left
right
up
down
left
right
up
left
down
up
up
right
down
left
left
```

Fig. 10. Image by candidate
Sample data log, as created by a pseudo-random function

I soon realized that it would be extremely unlikely to reach *2048* with random moves, and that the number of trials I would have to execute in order to see it happen even once would far exceed my planned 100 trials. I therefore changed my plan so that I would continue my experiment as planned, collect the data, and then calculate the probability of reaching the highest tile it can achieve, which I already saw would likely be 256, as a fraction of 2048. Perhaps I could also repeat the trials with an AI that doesn't have a too high success rate - as long as the AI stays constant, I can easily apply it to larger field sizes as well.

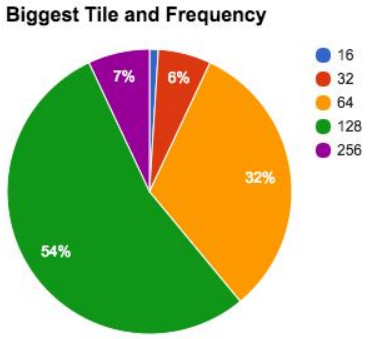

Fig. 11. Image by candidate

So, for 100 trials, the random-move bot reached 256 7% of the time. Although the max tile was 128 54% of the time according to the data, in reality, the bot reached 128 more often, as it reached it at least twice for each 256, and sometimes there were two 128 in one field but could not be merged before the game was over, so the maximum tile value recorded was still 128, even though the frequency of 128s achieved for such a game should be 2 instead of 1. However, the random algorithm never achieved two 256 tiles within one game, and the data for that value is therefore more accurate. Calculating with 256; $2048 \div 256 = 8$, therefore 256 is $\frac{1}{8}$ of 2048. Using random moves, the bot achieved a 256 tile 7% of the time. We can therefore say that using random moves, the tile that is $\frac{1}{8}$ of the size of the winning tile should be achieved 7% of the time.
This is obviously quite limited and not very accurate; however, it serves as a starting point.

The next step was to repeat the same trials with a 5x5 field (Xiao). I soon realized that the 5x5 field required many more moves; I therefore input 800 moves into the random-moves bot, and then repeatedly launched it as needed.

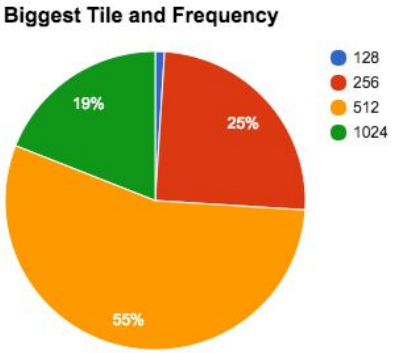

Fig. 12. Image by candidate

We can see that compared to the frequency of the biggest tile achieved with random moves in the 4x4 field, in the 5x5 field the biggest tile achieved was reached 12% more often. The results are therefore not very comparable. I then theorized that it may have something to do with the amount of tile spaces that need to be filled in order to make the tile; this would affect the probability, but is easier to calculate.

For 2048 on a $4x4$ field;

To achieve a $1024 + 512 + 256... + 2 + 2$ sequence (assuming that two 2s are spawned rather than a 4), we need to fill up 11 of the 16 available spaces ($2^{11} = 2048$).

For the first try: $16 - 11 = 5$, so perhaps the winning tile has a power of $n - 5$. Then for a $5x5$ field, $n = 25$, so that the winning tile would be 2^{20}. However, that number is much too large.

For the second try: $\frac{11}{16}$ as a ratio, so $\frac{11}{16} \times 25 \approx 17$, making the winning tile on a $5x5$ field 2^{17}. This number is much more plausible, but still quite big. Still, there is no way to find out which of the two would have a similar probability as 2048 on a $4x4$ field. So, I made a graph of the frequency vs the biggest tile achieved for the $5x5$ field, and then found a point with a frequency of 7 (for a 7% probability).

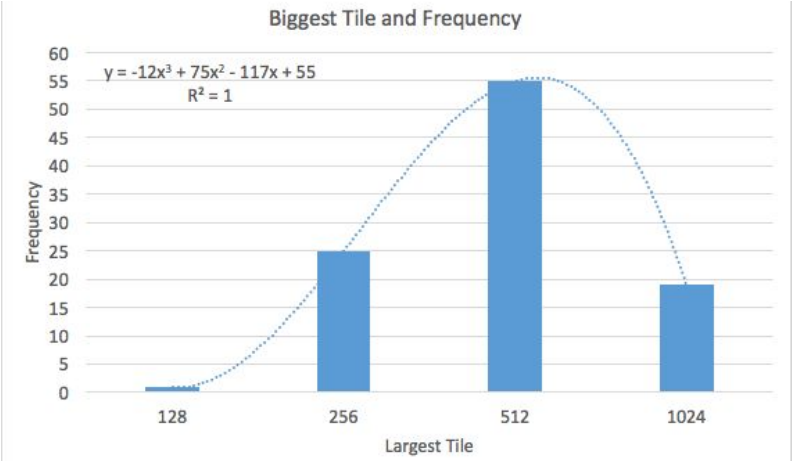

Fig. 13. Image by candidate

This graph looks cubic ($y = -12x^3 + 75x^2 - 117x + 55$, with an R^2 value of 1); however, the graph does not account for the fact that in order to achieve bigger tiles, the smaller tiles have been created several times, as only the biggest tile currently on the board was recorded. The part of the graph to the left of the maximum therefore should have a negative gradient as well. We can therefore discard the Tile value for the Frequency of 7 to the left of the maximum, and only take into account the value for Tile at Frequency 7 on the right of the maximum. This gives a Tile value of ≈ 1024. So if on a $4x4$ field, there is a 7% probability of achieving 2^8 and the equivalent tile to that on a $5x5$ field is 2^{10}, we could look at ratios.

Calculating using only the powers:

$\frac{8}{10} = \frac{11}{x}$, as on a $4x4$ field, the winning tile has the power 11, and on a $5x5$ field the power is unknown. Solving for x gives $x \approx 14$. As I now had three different answers, I decided to repeat 100 trials with random moves on a 3x3 field (Cyberhzg).

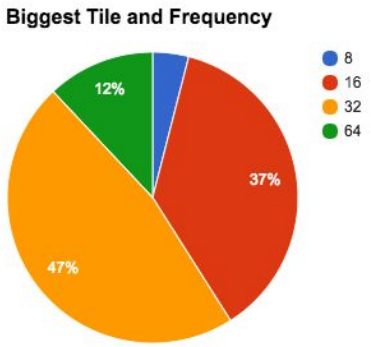

Fig. 14. Image by candidate

As we can see, the highest tile achieved with random moves was 64, and it was achieved 12% of the time. This probability fits into the range of 7%-19%

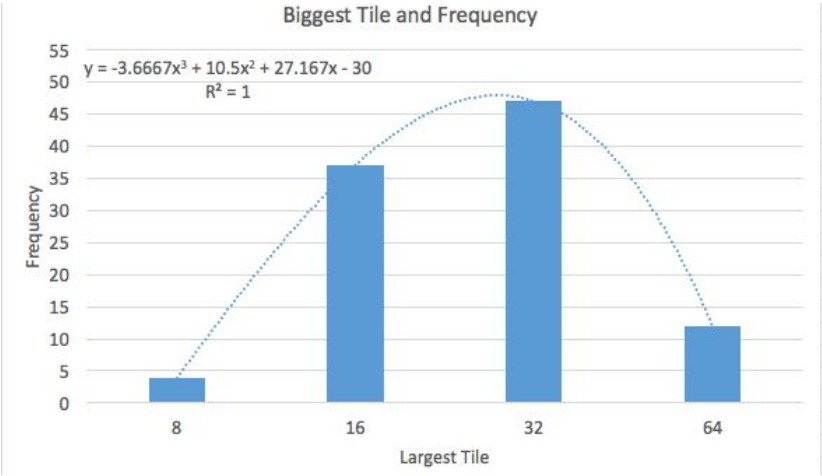

Fig. 15. Image by candidate

Using the same logic as above, when frequency is 7, Tile ≈ 64.
$64 = 2^6$

We now have powers of 6, 8, and 10; it seems that for each next largest board size, we simply add 2 to the power. In other words, we have the power $2p$, where p is the one-dimensional square root of n. The winning power for $p = 4$ ($4x4$ field) is 11; if we say that $8 + 3 = 11$, then $2p + 3 = 11$. Due to the proportional isomorphism of the game, we may then take $2p + 3$ to be the winning power on any board, making the winning tile q^{2n+3}.

Algorithm	Median Score	Mean Score	Std. Dev.	Max. Tile	Median Tile
random	1028	1075	512	256	128
simple greedy	3326	3620	1708	1024	256
simple Monte Carlo (depth 3)	14586	14295	6592	2048	1024
expectimax (depth 2)	14398	14241	6717	2048	1024
expectimax (depth 3)	27132	25270	10429	4096	2048

Fig. 16. Todd W. Neller. *Performance Results*. 04/2017

Upon researching different algorithms that could be used to solve the game, I found Figure 16, which confirmed my findings using a random algorithm. To compare, I calculated the standard deviation of my scores:

$$s_n = \sqrt{\frac{\sum_{i=1}^{k} f_i(x_i - \overline{x})^2}{n}} = \sqrt{\frac{\sum_{i=1}^{k} f_i x_i^2}{n} - \overline{x}^2}$$

$\overline{x} = 1133.7$, $n = 100$

$s_n = 512.2$

As one can see, although my mean score was a bit higher, my standard deviation is the same, as are the maximum and median tiles for a random algorithm. Out of curiosity, I repeated 100 trials with an AI bot (Overlan) that plays the game several times for each possible move using random moves to figure out the average score resultant of each possible next move and execute the move that gives the highest average end score (this is called a pure Monte Carlo algorithm, as it is based only on random moves). The number of games run per move is specifiable, so I input 10 games per move in order to have a not too high success rate. Indeed, even with such a low number, it performed quite a bit better than my own random bot that simply executed a random move rather than calculating the average score that may result from it. Increasing the number of games played per move would obviously increase the success rate.

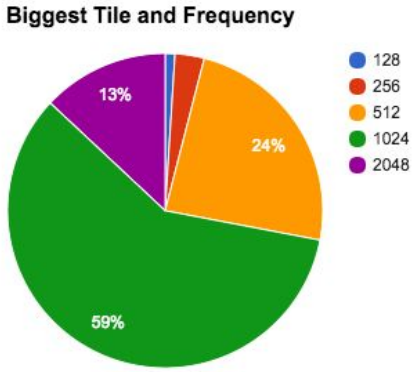

Fig. 17. Image by candidate

Comparing this to the algorithms explored in a table, I found that the median tile is 1024, the maximum tile is 2048, the mean score is 13701.4, and the standard deviation of the scores is 6679.4. We can thus see that this AI performs similarly to either a simple Monte Carlo algorithm with search depth 3 or an expectimax of depth 2.

As another way of comparison, I calculated the Expectation $E(X)$ for both of the $4x4$ trials (random vs. AI). As the theoretical probability is impossible to calculate, I took an experimental approach. Due to the lack of theoretical probability, $\overline{x} = \frac{\sum fx}{\sum f} == E(X) = \mu = \sum xP(X = x)$ when calculated using experimental data.

Frequency Distribution for the $4x4$ Random Move Trials

Highest Tile Achieved, x	16	32	64	128	256	
Frequency, f	1	6	32	54	7	Total 100

Fig. 18. Table & data by candidate

$$E(X) = \bar{x} = \frac{\sum fx}{\sum f} = \frac{(16)(1)+(32)(6)+(64)(32)+(128)(54)+(256)(7)}{100} = 109.6$$

Frequency Distribution for the $4x4$ AI Trials

Highest Tile Achieved, x	128	256	512	1024	2048	
Frequency, f	1	3	24	59	13	Total 100

Fig. 19. Table & data by candidate

$$E(X) = \bar{x} = \frac{\sum fx}{\sum f} = \frac{(128)(1)+(256)(3)+(512)(24)+(1024)(59)+(2048)(13)}{100} = 1002.24$$

Therefore, we can see that the AI's expected largest tile was about 9 times bigger than the expected largest tile for the random moves.

Probability of Winning

As previously seen, there is no way to calculate the probability of reaching 2048; there are simply too many variables and unknown factors, including the player himself. With pseudorandom moves, my program had a 0% chance of reaching 2048. Of course, with enough trials, there may be an outlier or two in which it may reach 2048 by pure luck. However, the number of trials needed for this to happen would be exponentially huge. The pure Monte Carlo algorithm that played only 10 games per move reached 2048 with a 13% probability; yet, this is not representative of a human player either, as playing even one game per move to the end would be unfeasible and impossible. Still, using the optimum strategy, a human player is able to achieve a quite high success rate. Naturally, this varies from player to player, and more experience can help increase this rate. Concluding, it it not possible to calculate a probability of winning, as in the end, the optimum strategy coupled with logic and luck may result in various success rates depending on the player.

Works Cited

Bhandari, Bhishan. "How Do You Build Bots for Games by Python?" *Quora*. Quora, 1 July 2016. Web. 8 Apr. 2017. <https://www.quora.com/How-do-you-build-bots-for-games-by-Python>.

Cirulli, Gabriele. "2048." *htp://gabrielecirulli.github.io/*. GitHub, Inc., n.d. Web. 15 Feb. 2017. <http://gabrielecirulli.github.io/2048/>.

Cyberzhg. "2147483648." *2147483648*. Http://cyberzhg.github.io/, n.d. Web. 14 Apr. 2017. <http://cyberzhg.github.io/2048/index.html?size=3>.

Gabrielecirulli. "Gabrielecirulli/2048." *GitHub*. GitHub, Inc., 21 Oct. 2015. Web. 4 Apr. 2017. <https://github.com/gabrielecirulli/2048>.

Goel, Bhargavi. *Mathematical Analysis of 2048, The Game* Advances in Applied Mathematical Analysis 12.1 (2017): 1-7. *www.ripublication.com*. Research India Publications, 2017. Web. 12 Apr. 2017. <https://www.ripublication.com/aama17/aamav12n1_01.pdf>.

Hoang, Lê Nguyen. "The Addictive Mathematics of the 2048 Tile Game." *Science4All*, Science4All, 29 Jan. 2016, www.science4all.org/article/2048-game/. Accessed 13 Feb. 2017.

Mathtuition88. "2048 Math Game Free Strategy Guide / Walkthrough." *Singapore Maths Tuition*. WordPress.com, 19 May 2014. Web. 20 Feb. 2017. <https://mathtuition88.com/2014/04/14/2048-math-game-free-strategy-guide-walkthrough/>.

Neller, Todd W. "PEDAGOGICAL POSSIBILITIES FOR THE 2048 PUZZLE GAME." *The Journal of Computing Sciences in Colleges* 30.3 (2015): 38-46.*Http://cs.gettysburg.edu/*. Gettysburg College, 2014. Web. 8 Apr. 2017. <http://cs.gettysburg.edu/~tneller/papers/ccsce14-2048.pdf>.

Nitish712, et al. "What Is the Optimal Algorithm for the Game 2048?" *Stackoverflow*, Stackoverflow, 12 Mar. 2014, stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048. Accessed 18 Feb. 2017.

Overlan, Matt, and Ronen Zilberman. "2048 - AI." *Http://ronzil.github.io/*. GitHub, Inc., n.d. Web. 4 Apr. 2017. <http://ronzil.github.io/2048-AI/>.

Ovolve. "Ovolve/2048-AI." *GitHub*. GitHub, Inc., 24 Jan. 2016. Web. 6 Apr. 2017. <https://github.com/ovolve/2048-AI>.

Volko, Claus D. "Mathematical Analysis of the 2048 Game." *New Math*, vol. 32, pp. 1–2., www.hugi.scene.org/adok/math/new_math32.pdf. Accessed 14 Apr. 2017.

Xiao, Ray. "2048-5." *Http://rayxiao92.github.io/*. GitHub, Inc., n.d. Web. 10 Apr. 2017. <http://rayxiao92.github.io/2048quintile/>.

Works Consulted

Bfontaine. "Bfontaine/term2048." *GitHub*. GitHub, Inc., 06 Mar. 2017. Web. 12 Apr. 2017. <https://github.com/bfontaine/term2048>.

Galebach, Brian. "Solving the 3x3 Variant of 2048." *ProbabilitySports*. Probability Sports, 7 May 2014. Web. 9 Apr. 2017. <http://probabilitysports.com/2048.html>.

Keshav, Malay. "AI Script for 2048 to Run on Chrome Console." *Malay Keshav*. N.p., 08 June 2014. Web. 14 Apr. 2017. <http://www.malaykeshav.me/ai-script-2048-run-chrome-console/>.

Lu, Kevin. "Game Theory of 2048." *Math 89s: Game Theory and Democracy* (2014): n. pag.*Services.math.duke.edu*. Duke University, 10 Dec. 2014. Web. 9 Apr. 2017. <https://services.math.duke.edu/~bray/Courses/49s/StudentSurveys/Fall2014/KL_Paper3_GameTheoryOf2048.pdf>.

Nicola17. "Nicola17/term2048-AI." *GitHub*. GitHub, Inc., 27 Mar. 2014. Web. 12 Apr. 2017. <https://github.com/Nicola17/term2048-AI>.

Salomon, Paul. "2048, 2584, and Variations on a Theme." *Math Munch*. WordPress.com, 24 Mar. 2014. Web. 11 Apr. 2017. <https://mathmunch.org/2014/03/24/2048-2584-and-variations-on-a-theme/>.

Singh, Anhad Jai. "What Is the Logic for Cracking the '2048 Puzzle'?" *Quora*. Quora, 29 Oct. 2014. Web. 11 Apr. 2017. <https://www.quora.com/What-is-the-logic-for-cracking-the-2048-puzzle>.

Singh, Kushagra. "An Artificial Intelligence for the 2048 Game." *HI, I AM KUSH*. Ghost, 21 Dec. 2015. Web. 14 Apr. 2017. <http://iamkush.me/an-artificial-intelligence-for-the-2048-game/>.

Yangshun. "Yangshun/2048-python." *GitHub*. GitHub, Inc., 20 Mar. 2014. Web. 27 Apr. 2017. <https://github.com/yangshun/2048-python>.

Appendix 1: Experimental Probability of Random Moves on a 4x4 field

Biggest Tile	Score
128	1356
64	648
128	1440
32	288
128	988
128	1332
64	616
64	424
256	2112
256	2036
32	404
64	624
64	612
64	528
128	1016
128	1592
128	1268
128	1116
128	1816
64	496
64	548
32	360
128	1364
64	1044
64	648
64	588
256	2256
128	1460
64	812
64	728
64	564
128	1784
128	1548
128	1144
128	1332
64	616
128	1448
128	1812
128	1888
128	1484
64	772
64	984
64	824
128	1356
64	628
64	612
128	1320
128	1408
128	996
64	832
128	1452
128	1636
128	1356
16	164
128	1352
128	1300
128	1100
128	1780
128	1408
64	596
128	1552
128	1776
128	1412
64	556
128	1316
128	1456
128	1020
64	628
32	216
64	628
64	652
64	532
128	1168
64	824
128	2164
128	1028
128	1292
64	832
32	444
128	1048
64	648
128	1360
128	1316
64	724
128	1292
128	1320
128	1440
128	1300
128	1280
128	1292
256	2120
128	1108
64	816
256	2048
256	2064
256	2448
128	996
128	1440
128	1408
32	388

Biggest Tile	Frequency	Cum. Frequency
16	1	1
32	6	7
64	32	39
128	54	93
256	7	100

Appendix 2: Experimental Probability of Random Moves on a 5x5 field

Biggest Tile	Score
512	7644
1024	15680
512	8064
512	5920
512	8904
256	5140
512	8792
256	3552
512	8236
512	7988
1024	14200
1024	14712
512	7440
512	6456
512	7492
512	6548
512	6720
512	5200
512	7588
512	7424
1024	16648
512	7484
1024	16316
512	6292
256	3620
1024	16756
512	6572
512	8204
256	4528
256	3688
512	7368
512	7768
512	5636
512	6460
1024	12224
1024	12320
512	8536
256	4364
512	5972
1024	16876
256	2872
1024	16352
512	7148
512	6592
1024	12260
512	8212
1024	15744
512	7232
256	4648
256	3376
1024	12932
256	4584
512	8180
256	3820
512	8076
256	4388
512	8320
256	3876
512	7660
1024	10900
512	6876
1024	12488
256	4400
1024	12372
128	1692
1024	12996
256	4408
256	4272
512	5976
256	3204
512	5432
256	4572
256	4012
512	7276
512	9636
512	10040
512	9244
512	5712
512	5996
256	2936
256	4112
512	7972
512	10220
512	7964
512	7500
512	5508
256	3572
256	3735
256	3688
512	9024
256	4076
512	5880
512	7092
512	7064
512	6796
1024	13008
512	9020
1024	12644
512	7332
512	6892

Biggest Tile	Frequency	Cum. Frequency
128	1	1
256	25	26
512	55	81
1024	19	100

Appendix 3: Experimental Probability of Random Moves on a 3x3 field

Biggest Tile	Score
32	160
32	216
16	124
32	168
16	92
32	212
32	176
32	196
16	100
32	216
16	72
16	88
32	192
32	248
32	232
32	168
16	76
64	472
32	160
16	72
16	72
32	196
32	312
32	156
8	56
16	96
32	176
16	112
16	76
32	216
32	272
16	76
32	176
64	360
16	96
32	208
16	88
16	64
32	228
16	128
32	204
8	52
64	380
32	204
32	204
16	92
32	212
16	108
32	212
16	96
16	84
16	104
32	200
16	128
64	380
16	104
32	200
32	220
32	200
64	496
32	272
32	252
16	88
32	176
32	168
16	140
32	232
64	432
32	208
16	108
8	28
64	376
8	56
16	64
32	152
16	92
64	360
16	92
32	188
16	88
32	188
32	196
16	76
64	360
16	100
32	164
16	112
64	388
16	96
32	172
16	116
64	468
16	88
32	216
64	408
32	200
32	160
32	168
16	136
32	180

Biggest Tile	Frequency	Cum. Frequency
8	4	4
16	37	41
32	47	88
64	12	100

Appendix 4: Experimental Probability of an AI with 10 runs per move on a 4x4 field

Biggest Tile	Score
512	5984
1024	10152
256	3100
512	7100
512	6932
1024	15636
1024	10060
1024	15480
1024	16232
512	7324
1024	12008
1024	16080
1024	15784
1024	15880
1024	16112
1024	12060
1024	11724
2048	32292
2048	32180
512	7272
512	7068
512	7044
1024	11924
1024	14424
1024	13944
512	6696
1024	16108
1024	14668
1024	16308
1024	16168
2048	21640
1024	14416
1024	13756
1024	16096
1024	16256
1024	12340
2048	32036
1024	11756
1024	16204
512	7212
2048	34304
1024	16268
1024	16060
1024	15312
512	6728
1024	16252
2048	23132
2048	27100
1024	12924
1024	15352
1024	14444
512	7032
512	7448
1024	11884
1024	11992
1024	16544
1024	15432
1024	15724
1024	14252
2048	26232
1024	16160
1024	15568
2048	25668
2048	23040
512	7224
512	6872
512	7116
512	7360
512	7064
1024	10324
1024	12112
1024	12508
1024	12212
1024	14108
1024	12172
512	5336
1024	14028
1024	14352
1024	14360
1024	15912
1024	16396
512	5316
512	5200
256	3144
512	5048
2048	23456
1024	16476
1024	14428
256	2176
2048	23528
512	7144
1024	13968
2048	26952
512	6052
128	1332
1024	12168
512	7368
1024	16084
1024	14296
1024	16240

Biggest Tile	Frequency	Cum. Frequency
128	1	1
256	3	4
512	24	28
1024	59	87
2048	13	100

Appendix 5: Python code written by candidate and sample data log

```
import pyautogui
import time
import random


pyautogui.hotkey('command', 'space');
pyautogui.typewrite('goog');
pyautogui.hotkey('enter');

time.sleep(2)
x = 0
for x in range (0,200):
    x+= 1
    rand = random.randint(0,3)
    if rand == 0:
         rand = 'left'
    elif rand == 1:
         rand = 'right'
    elif rand == 2:
           rand = 'up'
    elif rand == 3:
          rand = 'down'
    print(rand)

    pyautogui.press([rand])
```

```
down
right
left
right
up
down
left
right
up
left
down
up
up
right
down
left
left
up
right
left
up
left
up
up
left
left
left
down
right
up
up
right
up
right
right
right
right
left
```