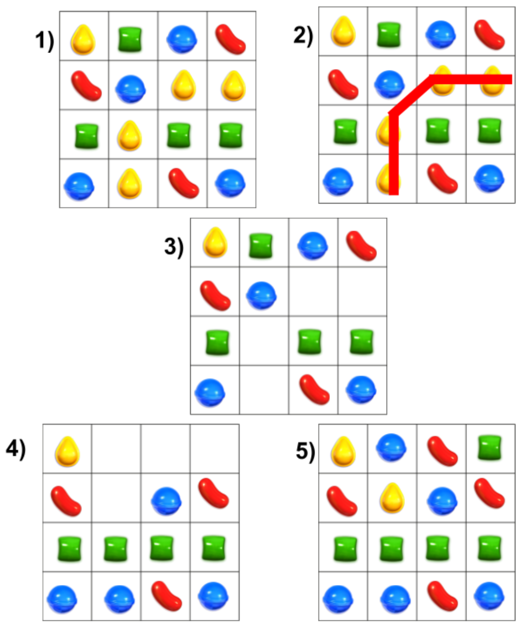
Regenerating Candy Crush Grid

I) Problem description

Almost everyone has at one point played the game Candy Crush, it's a classic after all. The fundamentals of the game are to swipe candies in any direction to create sets of 3 or more matching candies that become crushed. In the process you gain points for matching candies and the board is refilled with more candies to match. Although have you ever thought about the underlying operations that have allowed the game to function as you create and match sets of candies?

Your goal for this assignment is to implement the function in the game that would remove items from a given path of "crushed candies", move all the candy pieces down on the grid according to gravity, and automatically regenerate new candies to fill the grid.



Let's break this down a little further looking at the figure to the left. In Step 1, you see a fully randomized grid of colored candies, which is the starting point of a Candy Crush game. Matching candies are going to be any candies of the same color. In Step 2, we see a set of 4 matching candies have been chosen to be crushed. As a result in Step 3 we see that these crushed candies have been removed from the game board. In order to continue and follow the mechanism of the game, gravity has to be applied to push any candies down that are above open spaces. That's why we see in Step 4, The blue and green candy in the second column have been moved down and then the blue and red candy in columns 3 and 4 have also been moved down. Lastly we now have these open spaces which limits the possibilities of matching more candies. Thus we are going to set all the open spaces to one of the four candies randomly to refill the grid in step 5. And we're done!

We have provided you with the class "CandyCrushStarter" that generates the grid of candies for you. This class is a simplified version of candy crush so there are only 4 candy colors, "red", "blue", "green", and "yellow", and the grids only come in regular rectangle/square shapes.

The provided function **retrieveBoardState()** returns a grid datatype representation of the board.

The other provided function **printBoard()** prints out a 2D representation of the candy board to the console.

You are tasked with writing the function:

```
void regenerateGrid(Set<GridLocation> path)
```

This accepts input of **Set<GridLocation>** which is the path of crushed candies to be removed. Nothing is returned from the function as you have to make all edits to the grid by updating the **Grid<string> colorBoard** variable provided in the starter header file.

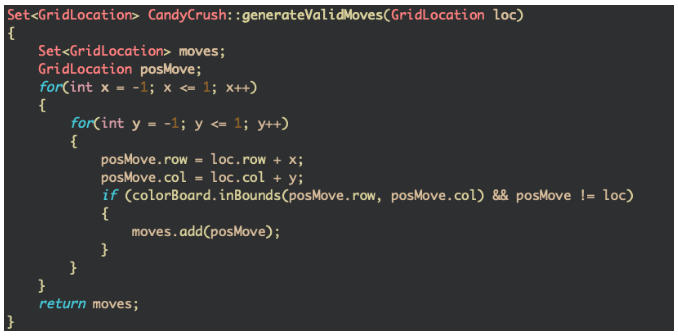
Your Function has to fill the following requirements:

- Validating that the inputted path is only of the same color, at least three candies in length, all have in bound grid locations, and that all the grid locations are connected such that they are in the 3 x 3 radius of another matching candy in a single path (meaning diagonals are allowed, but stray candies not adjacent to the path aren't). If any of these are invalid an error should be thrown.
- Removing all the Grid Location provided in the set to crush them (Specifically by setting the value of the grid locations to an empty string).
- Apply gravity to the board to push down candies that were above any removed in the set.
- Replace any empty Grid Locations in the board randomly with one of the four candies.

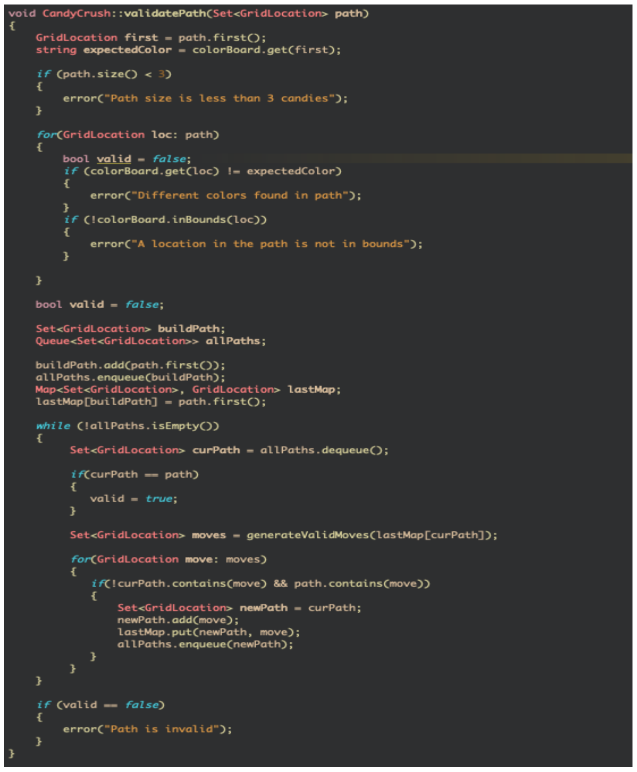
We suggest breaking some or all of these steps into helper functions to assist you in testing and creating your solution.

2) Solutions

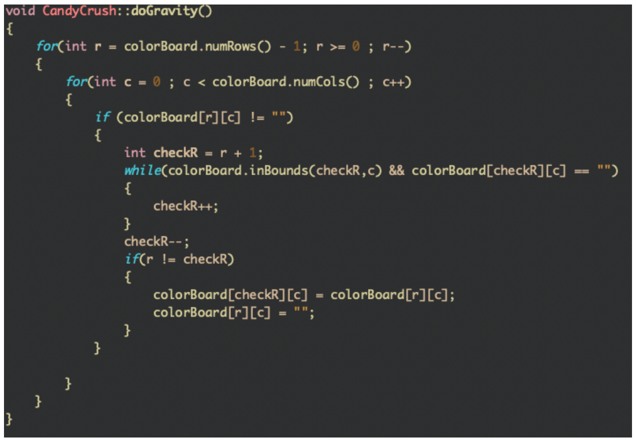
Solution 1:



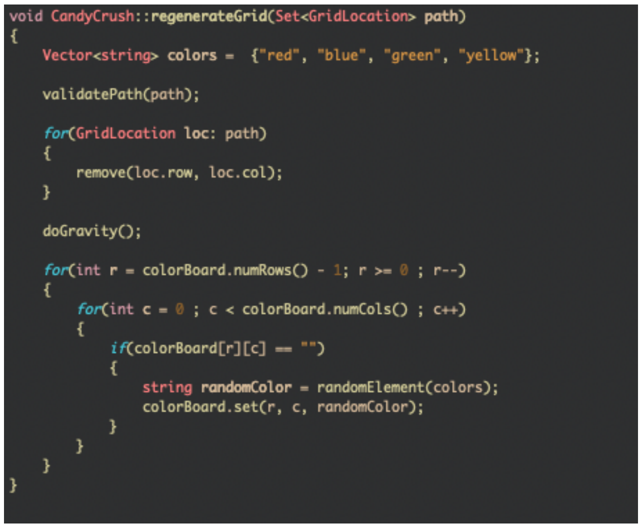
```cpp
Set<GridLocation> CandyCrush::generateValidMoves(GridLocation loc)
{
    Set<GridLocation> moves;
    GridLocation posMove;
    for(int x = -1; x <= 1; x++)
    {
        for(int y = -1; y <= 1; y++)
        {
            posMove.row = loc.row + x;
            posMove.col = loc.col + y;
            if (colorBoard.inBounds(posMove.row, posMove.col) && posMove != loc)
            {
                moves.add(posMove);
            }
        }
    }
    return moves;
}
```



```cpp
void CandyCrush::validatePath(Set<GridLocation> path)
{
    GridLocation first = path.first();
    string expectedColor = colorBoard.get(first);

    if (path.size() < 3)
    {
        error("Path size is less than 3 candies");
    }

    for(GridLocation loc: path)
    {
        bool valid = false;
        if (colorBoard.get(loc) != expectedColor)
        {
            error("Different colors found in path");
        }
        if (!colorBoard.inBounds(loc))
        {
            error("A location in the path is not in bounds");
        }

    }

    bool valid = false;

    Set<GridLocation> buildPath;
    Queue<Set<GridLocation>> allPaths;

    buildPath.add(path.first());
    allPaths.enqueue(buildPath);
    Map<Set<GridLocation>, GridLocation> lastMap;
    lastMap[buildPath] = path.first();

    while (!allPaths.isEmpty())
    {
         Set<GridLocation> curPath = allPaths.dequeue();

         if(curPath == path)
         {
            valid = true;
         }

         Set<GridLocation> moves = generateValidMoves(lastMap[curPath]);

         for(GridLocation move: moves)
         {
            if(!curPath.contains(move) && path.contains(move))
            {
                Set<GridLocation> newPath = curPath;
                newPath.add(move);
                lastMap.put(newPath, move);
                allPaths.enqueue(newPath);
            }
         }
    }

    if (valid == false)
    {
        error("Path is invalid");
    }
}
```



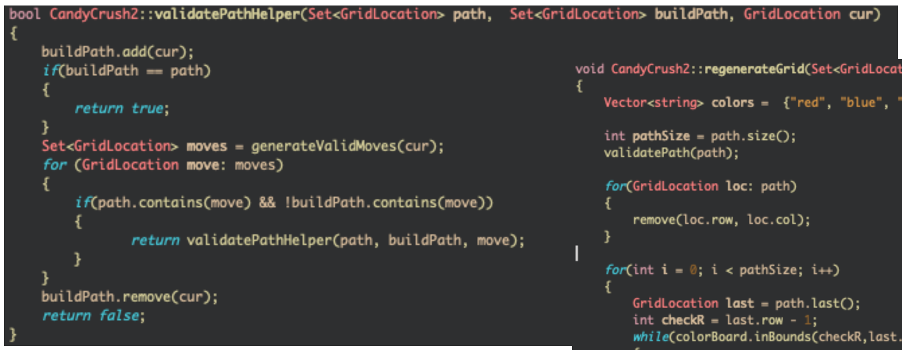
```cpp
void CandyCrush::doGravity()
{
    for(int r = colorBoard.numRows() - 1; r >= 0 ; r--)
    {
        for(int c = 0 ; c < colorBoard.numCols() ; c++)
        {
            if (colorBoard[r][c] != "")
            {
                int checkR = r + 1;
                while(colorBoard.inBounds(checkR,c) && colorBoard[checkR][c] == "")
                {
                    checkR++;
                }
                checkR--;
                if(r != checkR)
                {
                    colorBoard[checkR][c] = colorBoard[r][c];
                    colorBoard[r][c] = "";
                }
            }

        }
    }
}
```



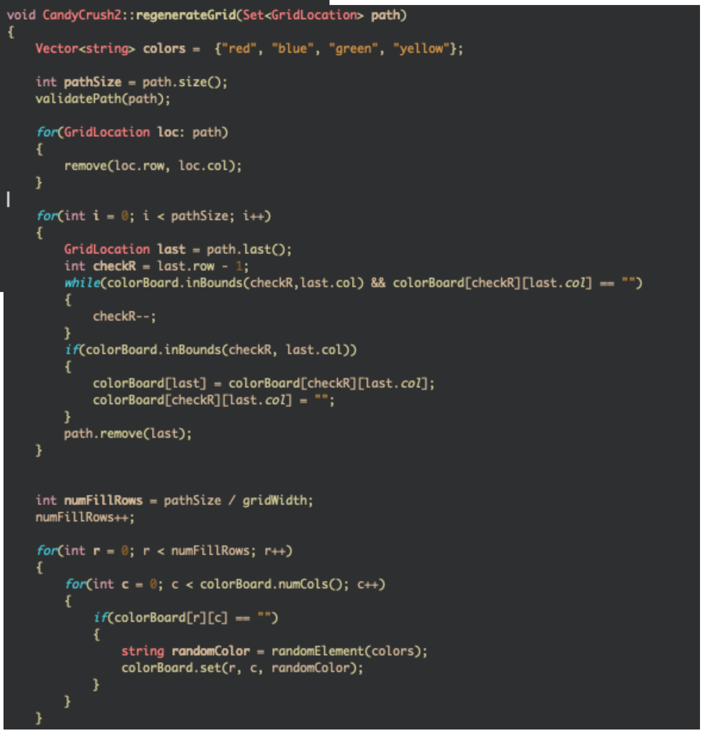
```cpp
void CandyCrush::regenerateGrid(Set<GridLocation> path)
{
    Vector<string> colors =  {"red", "blue", "green", "yellow"};

    validatePath(path);

    for(GridLocation loc: path)
    {
        remove(loc.row, loc.col);
    }

    doGravity();

    for(int r = colorBoard.numRows() - 1; r >= 0 ; r--)
    {
        for(int c = 0 ; c < colorBoard.numCols() ; c++)
        {
            if(colorBoard[r][c] == "")
            {
                string randomColor = randomElement(colors);
                colorBoard.set(r, c, randomColor);
            }
        }
    }
}
```

Solution 1 is decomposed into 3 helper functions to help concise the code in the regenerateGrid function. “GenerateValidMoves” consists of a double for loop in a 3 by 3 radius of a grid location to return a set of grid locations that a specific grid location could be in contact with, this is useful for later validating the path of crushed candies. Then the “validatePath” function checks the length, bound, and color requirements using an if statement and a for loop through the path inputted as parameter. For the last requirement of checking that the path is all connected, this solution uniquely uses a form of breadth first search in the validatePath function. This search involves iterating through a queue of paths until empty, while dequeuing paths and checking if they match with the originally passed in path. If they don't then all the generated moves are iterated through, and any new valid moves are added to a path and enqueued to the overlapping queue. I had to use a map function to force the computer to remember the last inputted Set location as well. Subsequently the “doGravity” function uses a double for loop to check which locations in the board are not empty strings after being removed. The for loop starts on the last row first so that gravity is implemented from the bottom-up avoiding locations being moved multiple times. A while loop is then used to check where the last open spot below each location is, if any at all, and replaces the values at the locations. The “regenerate” grid function calls validatePath, removes the path grid locations, calls doGravity, and then uses a nested for loop to fill any open spaces with a random color. With the accompanied for loops all put together in regenerateGrid, there are two nested for loops spanning over the size of the grid and breadth first search iterator iterating through the path, placing the Big O to be O(p * n^2) where n is the width or height of the grid and p is the number of elements in the path. This solution is clearly not the fastest but I think would be the most common because it is the most easy to conceptually understand, abiding straight by the requirements.

Solution 2:

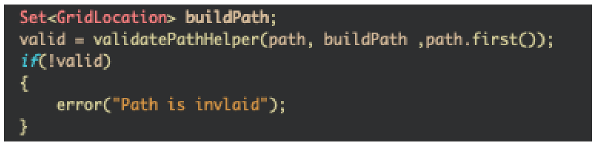


```
bool CandyCrush2::validatePathHelper(Set<GridLocation> path,  Set<GridLocation> buildPath, GridLocation cur)
{
    buildPath.add(cur);
    if(buildPath == path)
    {
        return true;
    }
    Set<GridLocation> moves = generateValidMoves(cur);
    for (GridLocation move: moves)
    {
        if(path.contains(move) && !buildPath.contains(move))
        {
                return validatePathHelper(path, buildPath, move);
        }
    }
    buildPath.remove(cur);
    return false;
}
```



```
void CandyCrush2::regenerateGrid(Set<GridLocation> path)
{
    Vector<string> colors =  {"red", "blue", "green", "yellow"};

    int pathSize = path.size();
    validatePath(path);

    for(GridLocation loc: path)
    {
        remove(loc.row, loc.col);
    }

    for(int i = 0; i < pathSize; i++)
    {
        GridLocation last = path.last();
        int checkR = last.row - 1;
        while(colorBoard.inBounds(checkR,last.col) && colorBoard[checkR][last.col] == "")
        {
            checkR--;
        }
        if(colorBoard.inBounds(checkR, last.col))
        {
            colorBoard[last] = colorBoard[checkR][last.col];
            colorBoard[checkR][last.col] = "";
        }
        path.remove(last);
    }


    int numFillRows = pathSize / gridWidth;
    numFillRows++;

    for(int r = 0; r < numFillRows; r++)
    {
        for(int c = 0; c < colorBoard.numCols(); c++)
        {
            if(colorBoard[r][c] == "")
            {
                string randomColor = randomElement(colors);
                colorBoard.set(r, c, randomColor);
            }
        }
    }
```

Call to validatePathHelper in
validPath function



```
Set<GridLocation> buildPath;
valid = validatePathHelper(path, buildPath ,path.first());
if(!valid)
{
    error("Path is invlaid");
}
```

Solution 2 has the same generateValidMoves function, removes the doGravity function and puts it in the regenrateGrid function, and essentially has the same validatePath function but instead calls validatePathHelper. Starting with the differences in validating the Path, solution 2 adds the helper function ValidatePathHelper to conduct depth first search. The base case is if the path being built recursively equals the original path. The Recursive case involves exploring a move from generateValidMoves and adding it to the path being built up, returning the function until a base case is reached. For the unchoose step the build path removes the current grid location, and a return false statement is added for if the path is not valid. In comparison, the BFS strategy seems to work faster with smaller paths because it operates by looking at paths of shorter lengths first, but with longer paths DFS seems to work better and has a smaller memory footprint, as it starts over from each new candy it looks at. In the regenerated grid function, gravity is implemented using one for loop instead of two. This is implemented by iterating through and adding gravity only to grid locations above the paths that were removed, as they are the only ones with empty strings. For the operation of refilling the grid, the regenrateGrid counts the number of items in the path to delete and divides it by the width of the grid, because that calculates how many rows are possible to have empty strings after gravity is done. Thus when regenerating, it only loops through the ros needed. These two adjustments in this solution help for time complexity as the whole grid does not have to be iterated through anymore when checking conditions such as empty strings. The Big O notation of this solution is still (p * n^2) in the worst case scenario, but given optimal conditions can be reduced to O(p * n) if refilling the grid has a limited amount of rows, and this solution has better space complexity with DFS.

For testing, I isolated testing the helper functions as a testing strategy to build up to the solution. For generating valid moves the test cases included checking the edges and centers of grids. For validating the path this consisted of test cases of diagonals and boundaries of the grid to make sure checking the path connection was accurate, as well as using EXPECT_ERROR to ensure paths that arent the same color or less than three in size were immediately stopped in the program. To test the doGravity function I made copies of the original grid and checked the positions after the doGravity function was called to see that the colors and empty spaces were in the right locations. Additionally edge cases included grids that were completely empty and full to make sure gravity wasn't incorrectly being applied to locations. Lastly, for regenerating the grid, I did normal test cases to make sure all the functions were interacting properly together.

3) Problem Motivation

Conceptual Motivation

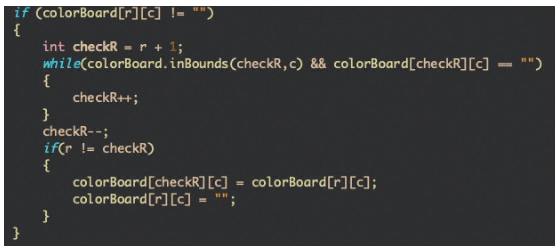
This question allows students to become familiar with the Grid data structure specifically as it is one of the hardest to adapt to. In the process other containers could also have been used in the implementation including Vectors to store color lists or grid locations, as well as Queues and Maps for searching algorithms. Techniques that are strengthened by this problem include dynamically traversing and updating friends, learning how to maximize efficiency with data structures that can contain a lot of data, as well as a refresher of iteration and recursion for search algorithms to validate and find a path given by the user. The problem can be broken down into many different functions or methods as well so choosing how to approach the problem is a good skill that students will learn. In my solutions breaking everything down into helper functions was a good example of this technique for conceptual understanding as well as storing different data values in parameters. Lastly, given that the problem was phrased in the form of a class, students could have a little bit of practice analyzing how to use other programmers' classes and add more functionality to them.

Personal motivation

Coming from Java and Python where the ADT's have very different implementations, I wanted to focus on data structures to boost my understanding in C++ as well. Additionally I also felt that I was struggling with analyzing Big O in the big scope of a program so with all the helper functions and loops that were integrated into this project, the difference between the notations started to become more clear. I also didn't fully understand the difference between depth first search and breadth first search and being able to apply it to a topic was very intriguing to me. The candy crush inspiration was brought up from my interest in gaming and the fact that I could make part of a real product that millions of people use.

4) Conceptual mastery, common misconceptions

The Gravity and regenerating functions of the program uniquely focus on mastering traversing grids of any size. I assume that when doing gravity most students would presume to just do a standard nested loop to iterate over the grid but this mistake will cause many errors. A reversed loop that starts on the bottom row and goes up is needed so that gravity is applied to objects on the bottom first and a certain object in a grid location is not moved twice. Another area in relation to the grid that a student would really have to think about is an off by one error when using grids as the width could be 10, but iterations would stop at 9. For example in the code snippet to the left when checking if a grid location had open spaces below it, I had to start at a location with a row right below the current location and then increment until there was no open space left. A student may forget when trying to replace the locations here they have to decrement the row integer variable by one since if they found a position where there was no longer an open space they have to move down the current location to the location right above this. A conceptual problem that also may confuse a few students is the fact that y values increase as you go down the grid which is contrary to math versions of grids, and the fact that rows relate to the height of the grid and columns relate to the width. When making the problem I got a little bit confused about what to make private and public in the class, since to test functions they need to be public. One misconception that could occur is trying not to use the private variable of the grid that was given to the student, without doing so the code could get very confusing by trying to pass in more variables into functions than actually needed. Lastly the solutions for BFS and DFS could trip up a few students as they may not even think that a searching algorithm is necessary or get tripped up on how to implement it. With depth first search I had the base case be if the path I was building matched up to the original path whereas



```
if (colorBoard[r][c] != "")
{
    int checkR = r + 1;
    while(colorBoard.inBounds(checkR,c) && colorBoard[checkR][c] == "")
    {
        checkR++;
    }
    checkR--;
    if(r != checkR)
    {
        colorBoard[checkR][c] = colorBoard[r][c];
        colorBoard[r][c] = "";
    }
}
```

at first I was trying to remove parts of the path as I continued to recurse. This ended in problems as I was removing parts that I needed to explore, before the designated unchoose step. In BFS I think there are many methods that a student could come up with. Although I noticed by my decision to store GridLocation paths as sets it was hard to access the last ordered element that was added when generating valid moves. Not noticing that this causes the loop to end early as it cycles between the structuring of how sets order GridLocations, and doesn't keep adding to potential sets. To solve this I implemented a Map to store a certain set of GridLocations to its last respective GridLocation added, but not thinking of this could confuse students or they could have thought of a completely different solution overall.

5) Ethics Reflection

My project

One ethical problem associated with my project is copyright/stealing other people's ideas. Technically by recreating candy crush I was replicating their game and this could infringe on some policies that they have around the protection of their game development rights. Although because of the fact that I am not selling this product nor allowing other people to use this, I do not think it would actually raise any problems. Whereas if Candy Crush does have some form of protection and someone sold a copy of their game by just recoding it, I do think it could present a problem on a level that could approach legal action.

Article + ethics question

This article from wired discusses how a bug on the messaging program, Slack, exposed a small amount of their users hashed passwords when they created or revoked a shared invite link. This directly relates to Lecture 24 on Hashing technology, which is most seen through passwords in the real world. Slack most likely uses a form hashing that takes in the user's password as a string and then returns a corresponding hash value in their database.

Sample Question**:**

A big tech company has hired you to build a hashing algorithm to store passwords for their company, what ethical and security considerations would you take account to implement the most secure hashing for this extremely private information:

Sample Answer:

My first approach to this would be trying to make multiple hashing algorithms instead of just one. This would ensure that even if malicious actors gained control of the hashed database they would see inconsistencies between the different hash patterns and be unable to decode it. For ethical reasons I would also try to be as communicative to the consumers using the product. This includes sending warning messages if there is ever a data brief, actively telling them about security protocol, and notifying them if someone tries to sign in to their account. To expand on the hashing algorithm, I would also try to incorporate cryptography that expanded past just numbers, such as letters or even symbols for further encrypted hashing.