CSE240 – Assignment 4
Dynamic Arrays

50 points

Topics:

- Created and used variables
- Used printf and scanf to work with input and output
- Use simple File I/O
- Used control structures to control the flow of logic
- Work with math and random numbers
- Dynamic Allocation of multi-dimension arrays
- Create some basic void and value returning functions

Description

The aim of this assignment is to make sure you understand dynamic array allocation by creating dynamic arrays and performing algorithms on them.

You have **three** different options. Pick ONE.

If you choose to do more than one as a challenge, extra credit can be negotiated with the professor.

Use the following Guidelines:

- Give identifiers semantic meaning and make them easy to read (examples numStudents, grossPay, etc).
- Keep identifiers to a reasonably short length.
- Use upper case for constants. Use title case (first letter is upper case) for classes. Use lower case with uppercase word separators for all other identifiers (variables, methods, objects).
- Use tabs or spaces to indent code within blocks (code surrounded by braces). This includes classes, methods, and code associated with ifs, switches and loops. Be consistent with the number of spaces or tabs that you use to indent.
- Use white space to make your program more readable.

Important Note:

All submitted assignments must begin with the descriptive comment block. To avoid losing trivial points, make sure this comment header is included in every assignment you submit and that it is updated accordingly from assignment to assignment.

Programming Assignment:

Instructions:

In this assignment you will create your code from scratch. You are to create a C/C++ file named <lastname>_<firstname>_hw4.c (or .cpp)

Stay within bounds of what we've covered in class. You may look ahead a bit to things like arrays. You are to use functions in this assignment.

All of the algorithms should be coded by you. Code the algorithms from scratch. Don't just find a library to solve the problems for you.

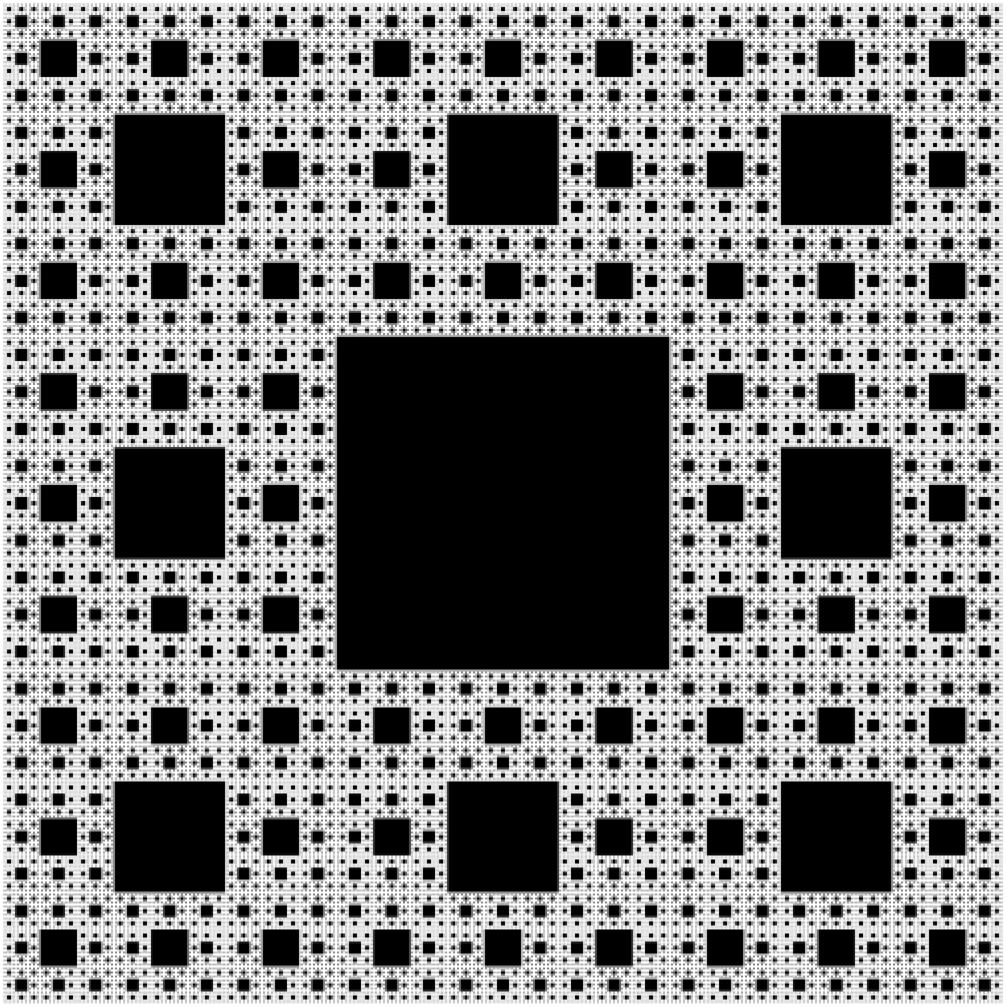
Library allowances:

If you are using C++ you may use: #include <stack> and std::stack or #include<queue> and std::queue for your algorithm needs.

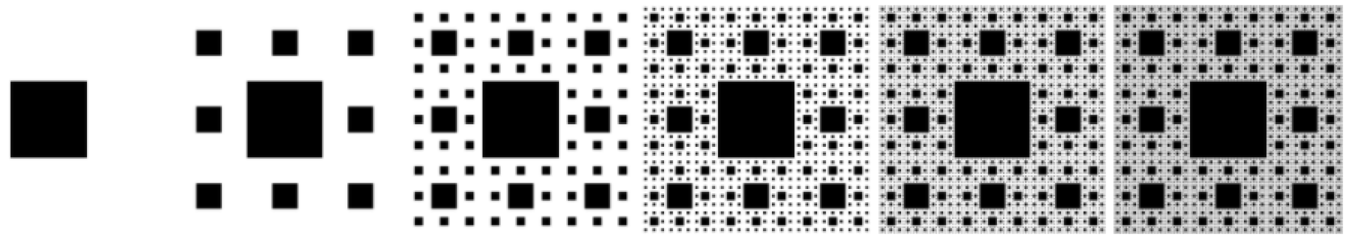
Note: C does not have standardized data structures libraries. If you need those data-structures for your algorithms, you should code in C++ or build your own functionality via an array.

Specifications:

Option #1 – Seirpinski Carpet Fractal



The Seirpinski Carpet is probably the ‘simplest’ fractal to represent in a 2D Array. The Carpet is always a square and is an *ad infinitum* repeating pattern of subdivisions. Each sub-square is representative of the whole and vice versa.



Directions:

Welcome the user to your Carpet Fractal Maker.

Prompt the user for how large a Carpet they want – or perhaps how many “layers” they want. *You might put a sanity check upper limit on the input* depending on the speed of the algorithm and size of the array. If you put such a limit on the input, you should inform the user of the upper limit.

Once the input is accepted, you should DYNAMICALLY ALLOCATE an appropriately sized 2D **char** array to fit your Carpet.

The size of your array is going to be $3^n \times 3^n$ where *n* is the number of “layers” given by the input. NOTE: if n = 0 then your size should be 1

Output:

Your carpet is going to get big in a hurry as you can tell from the samples given here →

```
N=0
#

N=1 (3 x 3)
###
# #
###

N=2 (9 x 9)
#########
# ## ## #
#########
###   ###
# #   # #
###   ###
#########
# ## ## #
#########

N=3 (27 x 27)
###########################
# ## ## ## ## ## ## ## ## #
###########################
###   ######   ######   ###
# #   # ## #   # ## #   # #
###   ######   ######   ###
###########################
# ## ## ## ## ## ## ## ## #
###########################
#########         #########
# ## ## #         # ## ## #
#########         #########
###   ###         ###   ###
# #   # #         # #   # #
###   ###         ###   ###
#########         #########
# ## ## #         # ## ## #
#########         #########
###########################
# ## ## ## ## ## ## ## ## #
###########################
###   ######   ######   ###
# #   # ## #   # ## #   # #
###   ######   ######   ###
###########################
# ## ## ## ## ## ## ## ## #
###########################
```

Output the carpet to the screen *and* a file if the number of layers asked for is less than or equal to three (*n* <= 3).

If the number of layers is more than that, then output only to a file.

Name the file **carpet.txt** (or prompt the user for a preferred name).

Help:

The carpet is a well repeated algorithm challenge. You can find many examples online. DO NOT JUST COPY THESE ALGORITHMS. You should learn and understand them and implement your own.

https://rosettacode.org/wiki/Sierpinski_carpet

https://mathworld.wolfram.com/SierpinskiCarpet.html

This $\left\{0\to\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}, 1\to\begin{bmatrix}1&1&1\\1&0&1\\1&1&1\end{bmatrix}\right\}.$ is super useful if you stare at it long enough.

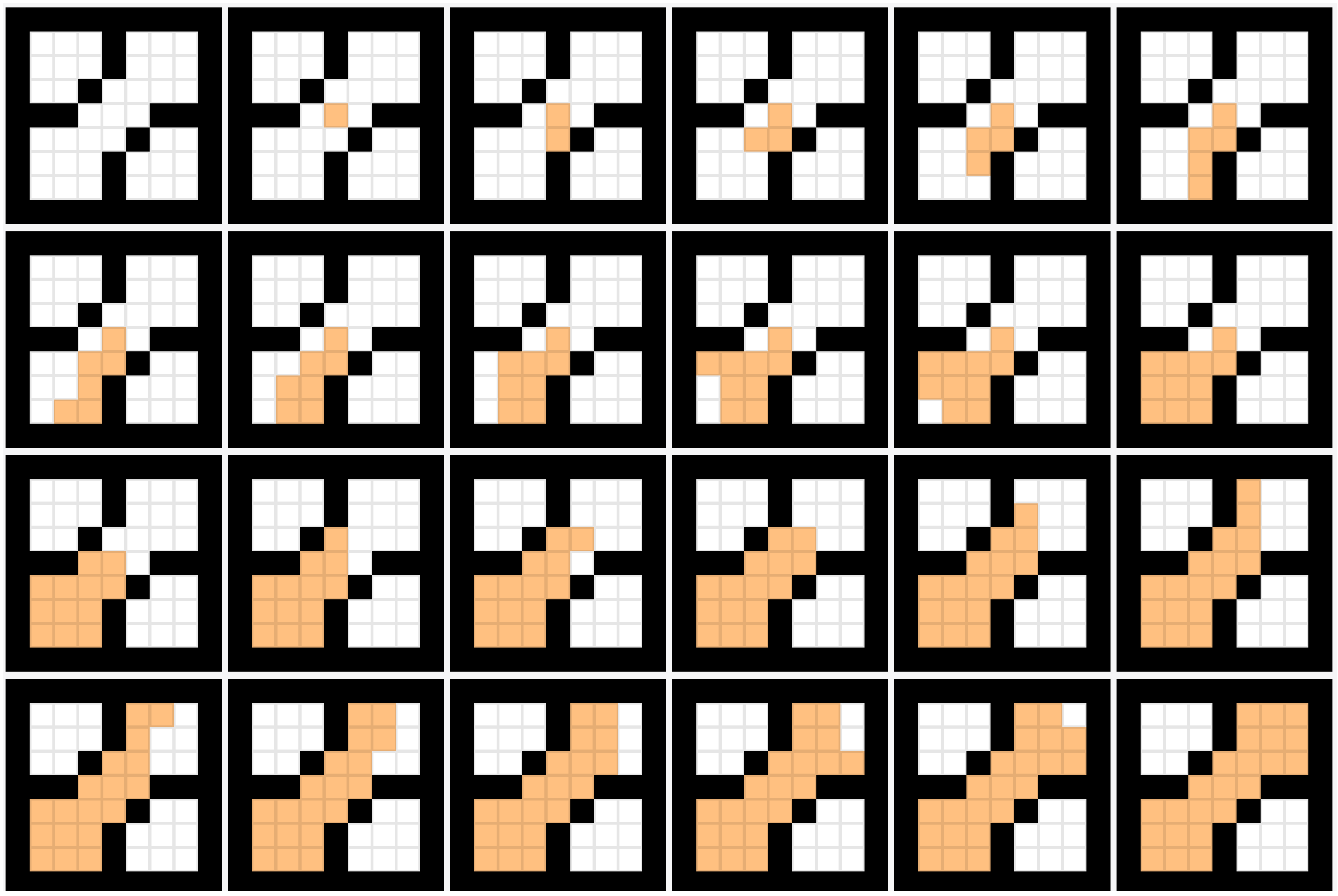
The 2D Dynamic Array is **required** in this assignment.

Extra Credit +5:

In addition to outputting to a txt file, create a bitmap without using a pre-built bitmap library (i.e. code the bitmap routines yourself).

Option #2 – Flood Fill Algorithm

Create a 2D Array and perform a flood fill based on user input.

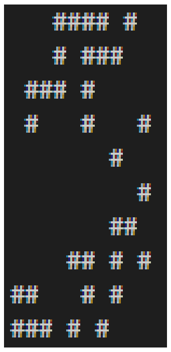


Directions:

You will ask the user for a width and a height for a Dynamically allocated 2D Character Array. You will create that array. Prompt the user for a "Percent of Impassible". Take that percentage as an integer. If the percentage is greater than 40 percent, warn the user that it might make for poor results and confirm the decision. If the user confirms, continue; otherwise, treat it as a bad input and re-prompt them.

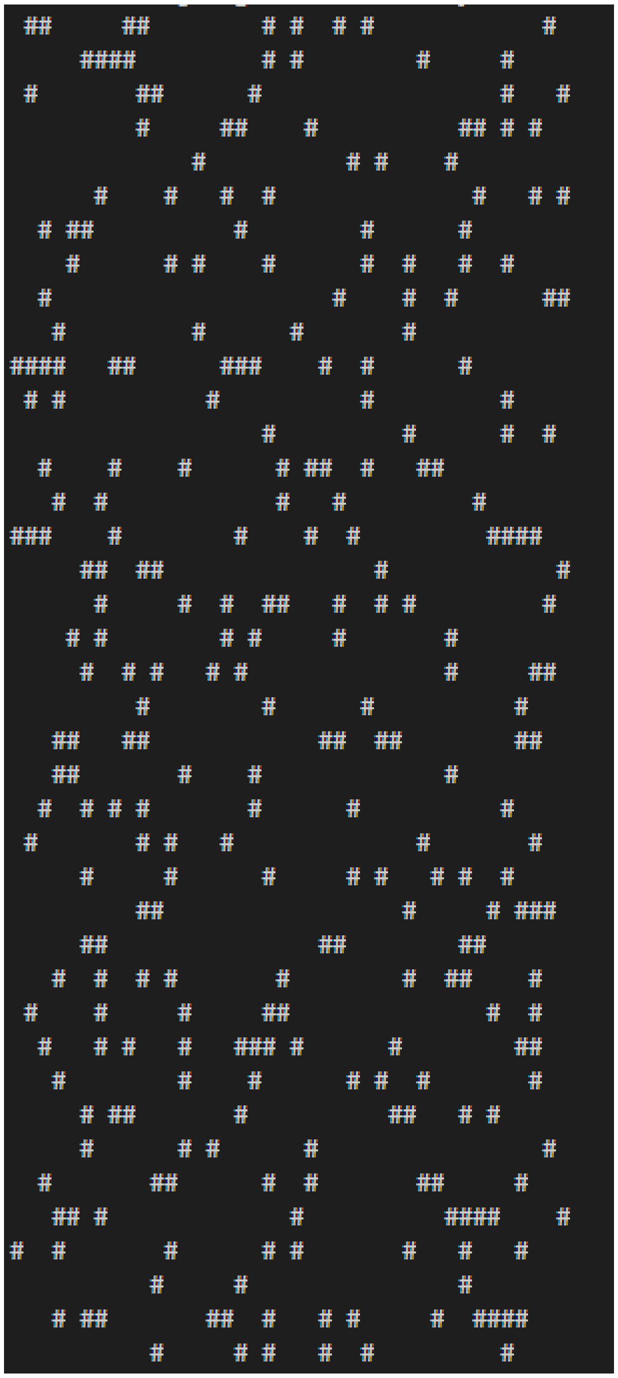
You will then populate the array with the "Percent of Impassible" of its size with "Impassible" items. You may choose what character represents "Impassible" but I recommend #.

Pre-Fill Grids



```
   #### #
   # ###
 ### #
 #   #   #
       #
         #
       ##
    ## # #
##   # #
### # #
```

```
10 x 10 Grid 30% Impassible

10*10 = 100

100 * 0.3 = 30 #'s
```



```
 ##     ##        # #  # #            #
     ####         # #        #     #
 #       ##      #                 #   #
         #     ##    #          ## # #
             #          # #    #
      #    #   #  #              #   # #
  # ##          #        #      #
    #      # #    #      #  #   #  #
  #                    #    #  #      ##
   #         #      #       #
####   ##      ###    #  #      #
 # #          #          #         #
                  #         #      #  #
  #    #    #      # ##  #   ##
   #  #            #   #         #
###    #        #    #  #         ####
     ##  ##               #            #
      #     #  #  ##   #  # #         #
    # #        # #     #       #
     #  # #   # #              #     ##
         #        #      #          #
   ##   ##            ##  ##        ##
   ##       #    #             #
  #  # # #       #      #          #
 #       # #   #             #       #
     #     #      #     # #   # #  #
         ##                 #     # ###
     ##               ##        ##
   #  #  # #       #        #  ##    #
 #    #     #     ##              #  #
  #   # #   #   ### #      #        ##
   #        #    #      # #  #       #
     # ##       #          ##   # #
     #      # #      #                #
  #       ##      #  #       ##     #
   ## #             #          ####    #
#  #       #      # #       #   #   #
          #     #               #
   # ##       ##  #   # #     #  ####
          #     # #   #  #         #
```

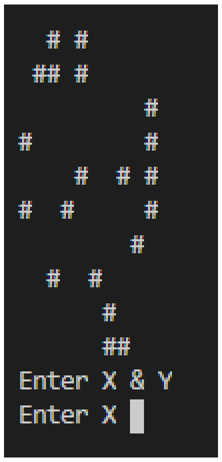
```
40 x 40 Grid 20% Impassible

40*40 = 1600

1600 * 0.2 = 320 #'s
```

Directions Continued:

After generating the grid and showing it to the user:

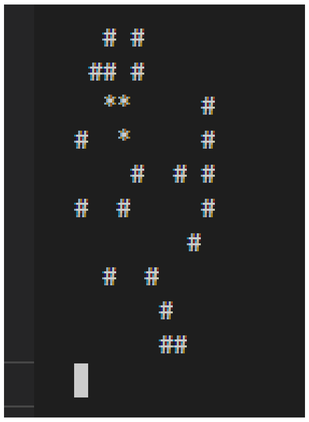
- Prompt the User for an X & Y Coordinate that would fall in the grid
- Prompt the User for the character they want to fill with

Perform the fill algorithm with that character from the specified index.



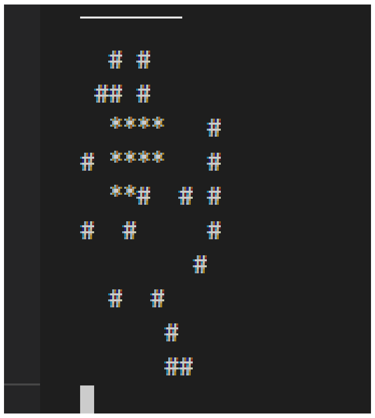
```
  # #
 ## #
         #
#        #
    #  # #
#  #     #
        #
  #  #
      #
      ##
Enter X & Y
Enter X
```

→



```
  # #
 ## #
  **     #
#  *     #
    #  # #
#  #     #
        #
  #  #
      #
      ##
```

→



```
_______

  # #
 ## #
  ****   #
# ****   #
  **#  # #
#  #     #
        #
  #  #
      #
      ##
```

You should show the fill happening. Make sure to have a separator between outputs or clear the screen.

Flood Fill:

There are multiple approaches to a flood-fill algorithm. Some are better than others. The very first time I ever tried my hand at this problem, my algorithm was O(n^3) … really bad.

Some major approaches:

- 4-Way (generally recursive) – O(n^2)
- Linescan flood fill – O(n^2)
- Breadth-First Search – O(|E| + |V|)
    - {|E| is the number of edges, |V| is the number of points}

Difference between the 4-Way and the BFS? The BFS basically spirals out from the point that the user selects and 4-Way “blobs out” in that direction.

Extra Credit Opportunity +5:

Now that you have a Flood Fill algorithm, you can use the same idea to generate a game of Mine Sweeper.

After generating the grid, you’ll have to do a pass to put the hint numbers in place.

Once the hint numbers are in place, you can play mine-sweeper. If the user guesses a grid space with a 0, then use your flood fill to reveal all the 0 places. Use any value that is non-zero as an Impassible.

Option #3 – Procedural Island Generator

This option has you creating an island/continent out of a method known as a "Particle Roll" Algorithm.

The idea is simple:

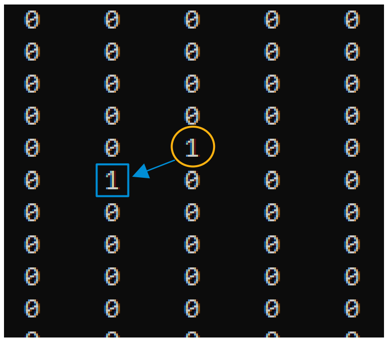
Set-Up:

1. Get a width and height from the user
2. Create a dynamically allocated 2D Array based on that width and height
3. Get an x-minimum, x-maximum, y-minimum, y-maximum from the user
    a. These values will define a 'drop window' over your 2D array
    b. Make sure:
        i. x-minimum & y-minimum >= 0
        ii. x-maximum > x-minimum
        iii. y-maximum > y-minimum
        iv. x-maximum & y-maximum < width & height
4. Get the number of particles to drop from the user
5. Get the max-life from the user
6. Get a value between 40 and 200 to use as the water-line

Drop-Roll-Generation:

1. "drop" a particle at a random x,y location within the window
2. increment that index by 1
3. confirm there is a valid move the particle can make
    a. the adjacent index value <= the current index value
    b. the adjacent index is >= 0 and < width/height
4. pick a random valid move and make it
    a. change the particle's x,y index to that new spot
5. reduce the particle's life
6. repeat 2-5 until the particle runs out of life or there are no valid moves

Visualization:



```
0    0    0    0    0
0    0    0    0    0
0    0    0    0    0
0    0    0    0    0
0    0    1    0    0
0    1    0    0    0
0    0    0    0    0
0    0    0    0    0
0    0    0    0    0
0    0    0    0    0
```

The Yellow Circle represents the original drop point

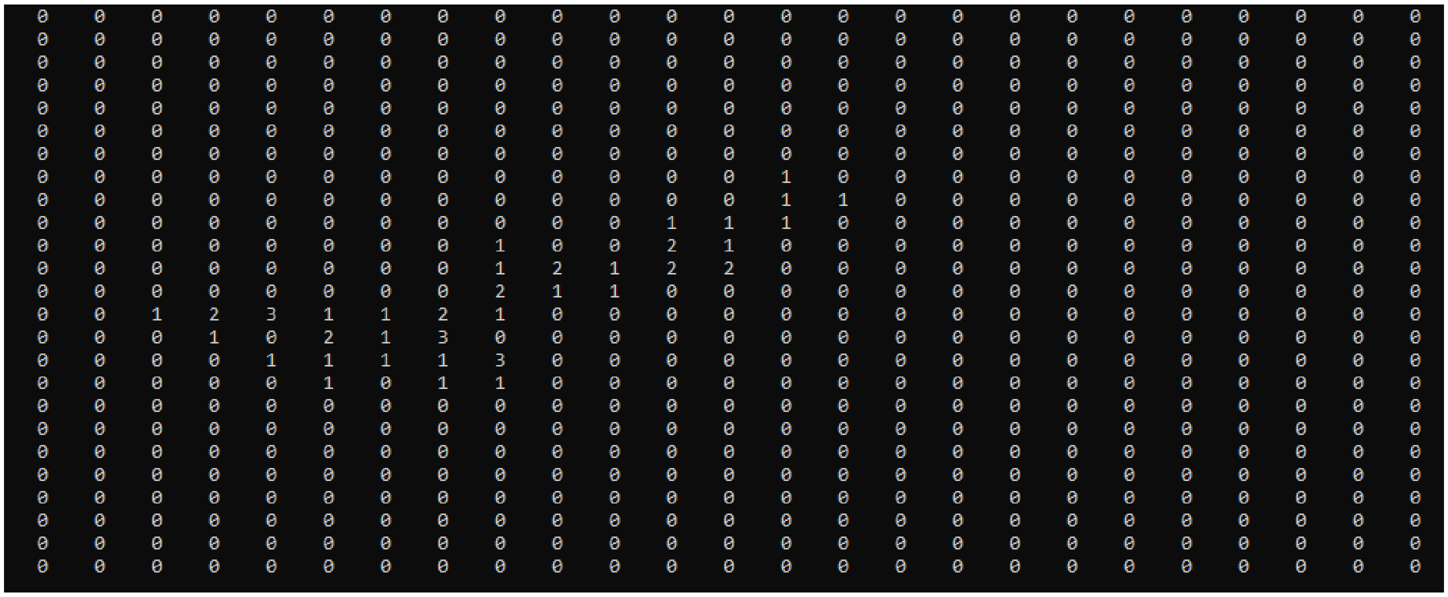
The Blue Arrow the move that was chosen

The Blue Box the new position (which is also incremented). The new position used to be 0 and is now 1 because the particle 'rolled into it.'

Visualization Part 2:

Here is an example of a single particle with a max-life of 50.

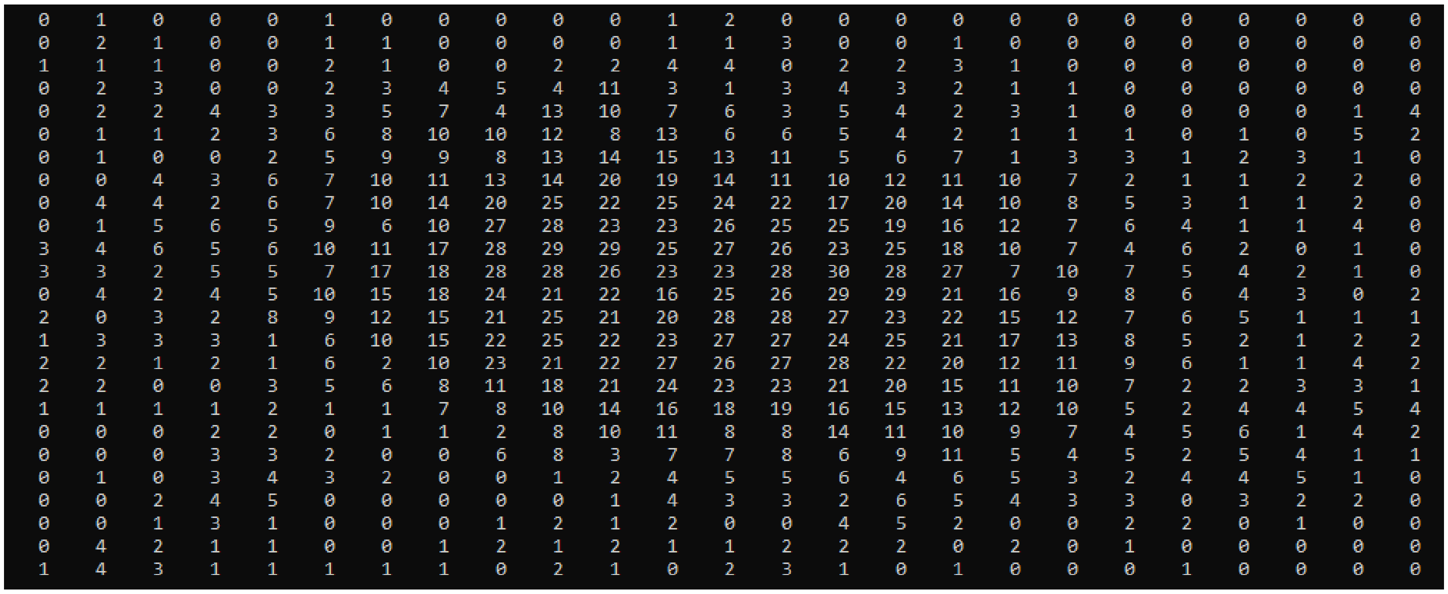
You can see the path it generally traveled. Some indexes became 2 or 3 because the particle was able to roll into those positions multiple times.



```
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    1    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    1    1    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    1    1    1    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    1    0    0    2    1    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    1    2    1    2    2    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    2    1    1    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    1    2    3    1    1    2    1    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    1    0    2    1    3    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    1    1    1    1    3    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    1    0    1    1    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
```

Visualization Part 3:

500 particles, max-life of 50.



```
    0    1    0    0    0    1    0    0    0    0    0    1    2    0    0    0    0    0    0    0    0    0    0    0    0
    0    2    1    0    0    1    1    0    0    0    0    1    1    3    0    0    1    0    0    0    0    0    0    0    0
    1    1    1    0    0    2    1    0    0    2    2    4    4    0    2    2    3    1    0    0    0    0    0    0    0
    0    2    3    0    0    2    3    4    5    4   11    3    1    3    4    3    2    1    1    0    0    0    0    0    0
    0    2    2    4    3    3    5    7    4   13   10    7    6    3    5    4    2    3    1    0    0    0    0    1    4
    0    1    1    2    3    6    8   10   10   12    8   13    6    6    5    4    2    1    1    1    0    1    0    5    2
    0    1    0    0    2    5    9    9    8   13   14   15   13   11    5    6    7    1    3    3    1    2    3    1    0
    0    0    4    3    6    7   10   11   13   14   20   19   14   11   10   12   11   10    7    2    1    1    2    2    0
    0    4    4    2    6    7   10   14   20   25   22   25   24   22   17   20   14   10    8    5    3    1    1    2    0
    0    1    5    6    5    9    6   10   27   28   23   23   26   25   25   19   16   12    7    6    4    1    1    4    0
    3    4    6    5    6   10   11   17   28   29   29   25   27   26   23   25   18   10    7    4    6    2    0    1    0
    3    3    2    5    5    7   17   18   28   28   26   23   23   28   30   28   27    7   10    7    5    4    2    1    0
    0    4    2    4    5   10   15   18   24   21   22   16   25   26   29   29   21   16    9    8    6    4    3    0    2
    2    0    3    2    8    9   12   15   21   25   21   20   28   28   27   23   22   15   12    7    6    5    1    1    1
    1    3    3    3    1    6   10   15   22   25   22   23   27   27   24   25   21   17   13    8    5    2    1    2    2
    2    2    1    2    1    6    2   10   23   21   22   27   26   27   28   22   20   12   11    9    6    1    1    4    2
    2    2    0    0    3    5    6    8   11   18   21   24   23   23   21   20   15   11   10    7    2    2    3    3    1
    1    1    1    1    2    1    1    7    8   10   14   16   18   19   16   15   13   12   10    5    2    4    4    5    4
    0    0    0    2    2    0    1    1    2    8   10   11    8    8   14   11   10    9    7    4    5    6    1    4    2
    0    0    0    3    3    2    0    0    6    8    3    7    7    8    6    9   11    5    4    5    2    5    4    1    1
    0    1    0    3    4    3    2    0    0    1    2    4    5    5    6    4    6    5    3    2    4    4    5    1    0
    0    0    2    4    5    0    0    0    0    0    1    4    3    3    2    6    5    4    3    3    0    3    2    2    0
    0    0    1    3    1    0    0    0    1    2    1    2    0    0    4    5    2    0    0    2    2    0    1    0    0
    0    4    2    1    1    0    0    1    2    1    2    1    1    2    2    2    0    2    0    1    0    0    0    0    0
    1    4    3    1    1    1    1    1    0    2    1    0    2    3    1    0    1    0    0    0    1    0    0    0    0
```

Polishing the Island

After you have generated the values for the landmass, do a quick analysis to find the highest value in the 2D array.

Normalize the values in the 2D array by dividing every value in the 2D array by that maximum then multiplying by 255. This will make every index a value between 0 and 255.

Create a 2D character array of the same size as your value array.

Process through the values in your array and set a character in your 2D character array based on the value's relationship to the water-line (from the set-up).

Output your polished island to the console AND to a file (island.txt is fine).

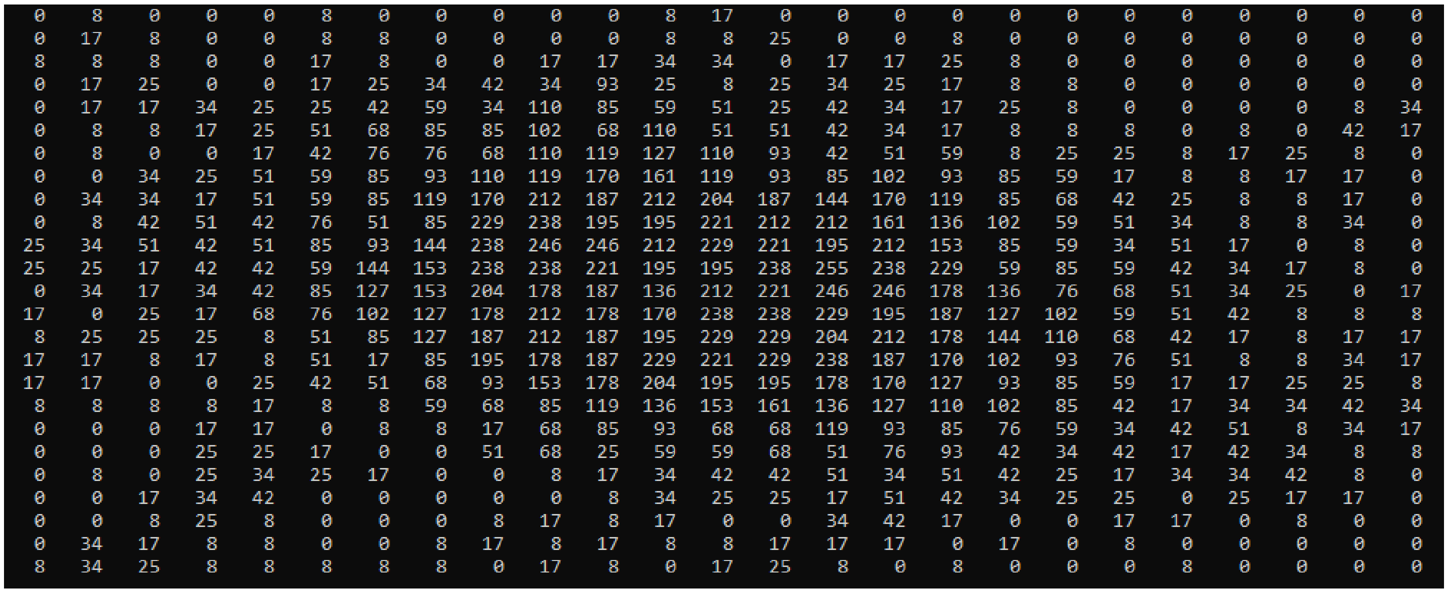
Calculate land-zone as 255 – water-line.

- value < 50% of water-line
    - o '#' – deep water
- value > 50% of water-line && <= water-line
    - o '~' – shallow water

Everything else is > water-line &&

- < (water-line + 15% of land-zone)
    - o '.' – coast/beach
- >= (water-line + 15% of land-zone) && < (water-line + 40% of land-zone)
    - o '-' – plains/grass
- >= (water-line + 40% of land-zone) && < (water-line + 80% of land-zone)
    - o '*' – forests
- else:
    - o '^' – mountains

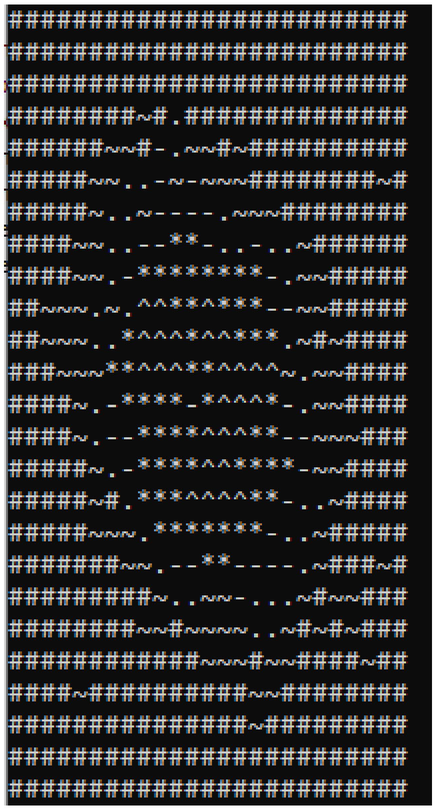
Island Normalized:



```
    0     8     0     0     0     8     0     0     0     0     0     8    17     0     0     0     0     0     0     0     0     0     0     0     0
    0    17     8     0     0     8     8     0     0     0     0     8     8    25     0     0     8     0     0     0     0     0     0     0     0
    8     8     8     0     0    17     8     0     0    17    17    34    34     0    17    17    25     8     0     0     0     0     0     0     0
    0    17    25     0     0    17    25    34    42    34    93    25     8    25    34    25    17     8     8     0     0     0     0     0     0
    0    17    17    34    25    25    42    59    34   110    85    59    51    25    42    34    17    25     8     0     0     0     0     8    34
    0     8     8    17    25    51    68    85    85   102    68   110    51    51    42    34    17     8     8     8     0     8     0    42    17
    0     8     0     0    17    42    76    76    68   110   119   127   110    93    42    51    59     8    25    25     8    17    25     8     0
    0     0    34    25    51    59    85    93   110   119   170   161   119    93    85   102    93    85    59    17     8     8    17    17     0
    0    34    34    17    51    59    85   119   170   212   187   212   204   187   144   170   119    85    68    42    25     8     8    17     0
    0     8    42    51    42    76    51    85   229   238   195   195   221   212   212   161   136   102    59    51    34     8     8    34     0
   25    34    51    42    51    85    93   144   238   246   246   212   229   221   195   212   153    85    59    34    51    17     0     8     0
   25    25    17    42    42    59   144   153   238   238   221   195   195   238   255   238   229    59    85    59    42    34    17     8     0
    0    34    17    34    42    85   127   153   204   178   187   136   212   221   246   246   178   136    76    68    51    34    25     0    17
   17     0    25    17    68    76   102   127   178   212   178   170   238   238   229   195   187   127   102    59    51    42     8     8     8
    8    25    25    25     8    51    85   127   187   212   187   195   229   229   204   212   178   144   110    68    42    17     8    17    17
   17    17     8    17     8    51    17    85   195   178   187   229   221   229   238   187   170   102    93    76    51     8     8    34    17
   17    17     0     0    25    42    51    68    93   153   178   204   195   195   178   170   127    93    85    59    17    17    25    25     8
    8     8     8     8    17     8     8    59    68    85   119   136   153   161   136   127   110   102    85    42    17    34    34    42    34
    0     0     0    17    17     0     8     8    17    68    85    93    68    68   119    93    85    76    59    34    42    51     8    34    17
    0     0     0    25    25    17     0     0    51    68    25    59    59    68    51    76    93    42    34    42    17    42    34     8     8
    0     8     0    25    34    25    17     0     0     8    17    34    42    42    51    34    51    42    25    17    34    34    42     8     0
    0     0    17    34    42     0     0     0     0     0     8    34    25    25    17    51    42    34    25    25     0    25    17    17     0
    0     0     8    25     8     0     0     0     8    17     8    17     0     0    34    42    17     0     0    17    17     0     8     0     0
    0    34    17     8     8     0     0     8    17     8    17     8     8    17    17    17     0    17     0     8     0     0     0     0     0
    8    34    25     8     8     8     8     8     0    17     8     0    17    25     8     0     8     0     0     0     8     0     0     0     0
```

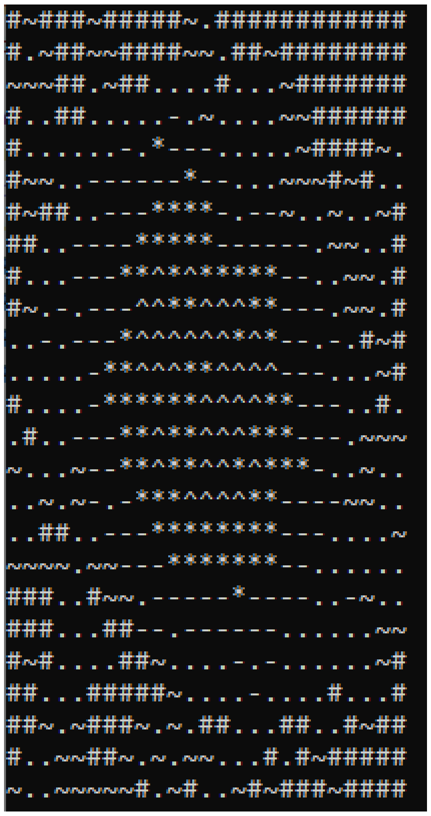
Polished

Figure 1 Waterline = 70



```
#########################
#########################
#########################
########~#.##############
######~~#-.~~#~##########
#####~~..-~-~~~########~#
#####~..~----.~~~########
####~~..--**-..-..~######
####~~.-********-.~~#####
##~~~.~.^^**^***--~~#####
##~~~..*^^^*^^***.~#~####
###~~~**^^^**^^^^~.~~####
####~.-****-*^^^*-.~~####
####~.--****^^^**--~~~###
#####~.-****^^****-~~####
#####~#.***^^^^**-..~####
#####~~~.*******-..~#####
#######~~.--**----.~###~#
#########~..~~-...~#~~###
########~~#~~~~..~#~#~###
############~~~#~~####~##
####~##########~~########
###############~#########
#########################
#########################
```

Figure 2 Waterline = 10



```
#~###~######~.###########
#.~##~~####~~.##~########
~~~##.~##....#...~#######
#..##.....-.~....~~######
#......-.*---.....~####~.
#~~..------*--...~~~#~#..
#~##..---****-.--~..~..~#
##..----*****------.~~..#
#...---**^*^*****--..~~.#
#~.-.---^^**^^^**---.~~.#
..-.---*^^^^^^*^*--.-.#~#
.....-**^^^**^^^^---...~#
#....-******^^^^**---..#.
.#..---**^**^^^***---.~~~
~...~--**^**^^*^***-..~..
..~.~-.-***^^^^**----~~..
..##..---********---....~
~~~~.~~---*******--......
###..#~~.-----*----..-~..
###...##--.------......~~
#~#....##~....-.-......~#
##...#####~....-....#...#
##~.~###~.~.##...##..#~##
#..~~##~.~.~~...#.#~#####
~..~~~~~#.~#..~#~###~####
```

Recommended Functions

Quality modularization of your code is part of your Code-Quality score in the rubric.

These function recommendations are for your benefit to make organizing this project easier …

makeParticleMap

```
int** makeParticleMap(int width, int height,
                      int minx, int maxX, int minY, int maxY,
                      int numParticles, int maxLife)
```

This function builds the array for the map data and performs particle roll algorithm to populate the array with map data.

You can modify this function to take the array as a parameter instead if you prefer to create the array externally to the particle roll algorithm. In which case, this function would become a void type.

NOTE: minX, maxX, minY, maxY are the parameters for the drop-window.

moveExists

```
bool moveExists(int** map, int width, int height, int x, int y)
```

This function looks a the 8 spots around x,y and determines if a valid move is possible. Return true if one is found. Note: you can return true as soon as the first one is found.

findMax

```
int findMax(int** map, int width, int height)
```

This function finds the maximum value in the map and returns it.

normalizeMap

```
void normalizeMap(int** map, int width, int height, int maxVal)
```

Performs the normalization operation on the map data. You could have this return a new array instead if you want to hold onto the original map data for some reason.

Extra Credit Opportunities:

Special Interface +3

Create a special interface to let the user repeatedly drop particles on a new location instead of a single drop.

Polish the map after they are done making drops.

Color +2

Color your output using a text coloring library. Find a console-color library that works across all Operating Systems and integrate it.

NOTE – this should be stand alone and not require any installation on the grader's part, just the library files in the proper location.

Any required compilation should be accounted for in a Makefile.

MAKE SURE you put any instruction for building/running your code in your submission notes.

I have used: https://github.com/ikalnytskyi/termcolor in the past to good effect.

Bugs and Debugging:

You should work within reasonable I/O standards. If the user gives you a number outside the parameters, you should correct the user and loop the input prompt again.

I will not hold you responsible for bizarre input like text into an integer at this point.

Notes and tips:

- Remember to write Functions. Since there are so many options and algorithms here it is impossible for me to write Required functions.
  - Modularize your code wherever and whenever possible!!
  - Remember code readability is a thing!
  - Don’t throw away coding standards for the sake of the algorithm.
    - If you are copying the algorithm in someone else’s coding style, that’s a big **red flag** that you need to redo the work until you understand the algorithm and can code it yourself in your style
- Code the algorithms yourself! Research each as necessary and accomplish the goals
  - The first 2 options are “old” and widely available on the internet, we will be watching for copied code
  - Study the algorithms and code your own
  - Study the scenario and try to puzzle it out yourself (even better)
- Generating a random number is easy in C, but you will want to include the time library to seed the random number generator. You might want to use this for testing your code.
  - #include <time.h>
  - srand(time(NULL)); //only needed once for your whole program usually in main
  - value = (rand() % range_value) + minimum_value;
- Remember, I’m allowing you to use some of the standard data-structures libraries for this assignment.
  - Make sure you look them up and understand how to use them properly

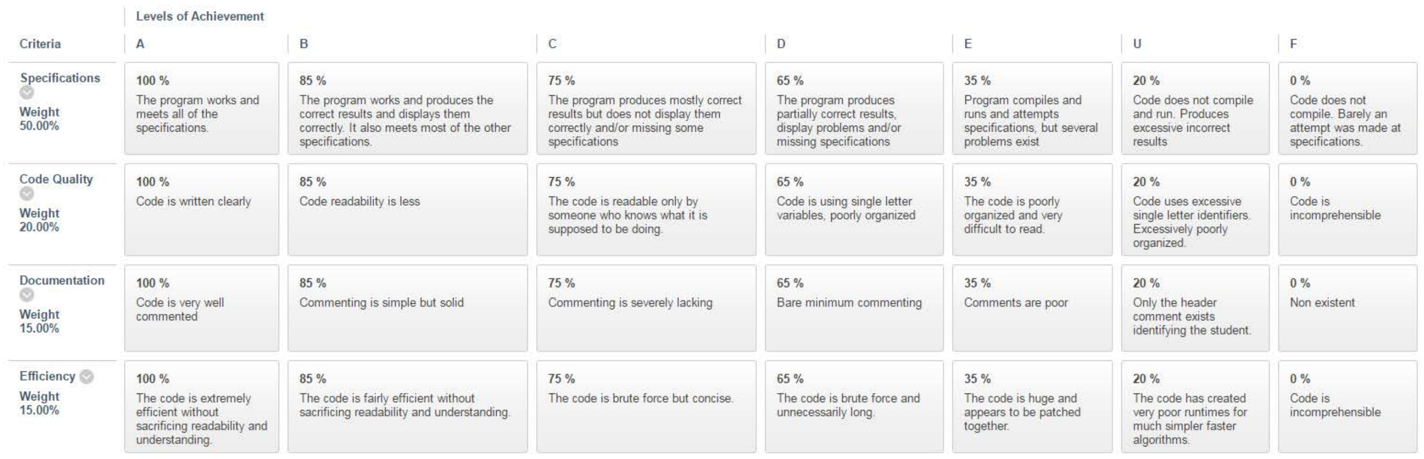
Grading of Programming Assignment

The Grader will grade your program following these steps:

(1) Compile the code. If it does not compile a U or F will be given in the Specifications section. This will probably also affect the Efficiency/Stability section.

(2) The Grader will read your program and give points based on the points allocated to each component, the readability of your code (organization of the code and comments), logic, inclusion of the required functions, and correctness of the implementations of each function.

Rubric:



	Levels of Achievement						
Criteria	A	B	C	D	E	U	F
Specifications Weight 50.00%	100 % The program works and meets all of the specifications.	85 % The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	75 % The program produces mostly correct results but does not display them correctly and/or missing some specifications	65 % The program produces partially correct results, display problems and/or missing specifications	35 % Program compiles and runs and attempts specifications, but several problems exist	20 % Code does not compile and run. Produces excessive incorrect results	0 % Code does not compile. Barely an attempt was made at specifications.
Code Quality Weight 20.00%	100 % Code is written clearly	85 % Code readability is less	75 % The code is readable only by someone who knows what it is supposed to be doing.	65 % Code is using single letter variables, poorly organized	35 % The code is poorly organized and very difficult to read.	20 % Code uses excessive single letter identifiers. Excessively poorly organized.	0 % Code is incomprehensible
Documentation Weight 15.00%	100 % Code is very well commented	85 % Commenting is simple but solid	75 % Commenting is severely lacking	65 % Bare minimum commenting	35 % Comments are poor	20 % Only the header comment exists identifying the student.	0 % Non existent
Efficiency Weight 15.00%	100 % The code is extremely efficient without sacrificing readability and understanding.	85 % The code is fairly efficient without sacrificing readability and understanding.	75 % The code is brute force but concise.	65 % The code is brute force and unnecessarily long.	35 % The code is huge and appears to be patched together.	20 % The code has created very poor runtimes for much simpler faster algorithms.	0 % Code is incomprehensible

What to Submit?

You are required to submit your solutions in a compressed format (.zip). Zip all files into a single zip file. Make sure your compressed file is labeled correctly - <lastname>_<firstname>_hw3.zip

The compressed file MUST contain the following:

- <lastname>_<firstname>_hw3.c

No other files should be in the compressed folder.

If multiple submissions are made, the most recent submission will be graded, even if the assignment is submitted late.

Where to Submit?

All submissions must be electronically submitted to the respected homework link in the course web page where you downloaded the assignment.

Academic Integrity and Honor Code.

You are encouraged to cooperate in study group on learning the course materials. However, you may not cooperate on preparing the individual assignments. Anything that you turn in must be your own work: You must write up your own solution with your own understanding. If you use an idea that is found in a book or from other sources, or that was developed by someone else or jointly with some group, make sure you acknowledge the source and/or the names of the persons in the write-up for each problem. When you help your peers, you should never show your work to them. All assignment questions must be asked in the course discussion board. Asking assignment questions or making your assignment available in the public websites before the assignment due will be considered cheating.

*The instructor and the TA will **CAREFULLY** check any possible proliferation or plagiarism. We will use the document/program comparison tools like MOSS (Measure Of Software Similarity: http://moss.stanford.edu/) to check any assignment that you submitted for grading. The Ira A. Fulton Schools of Engineering expect all students to adhere to ASU's policy on Academic Dishonesty. These policies can be found in the Code of Student Conduct:*

http://www.asu.edu/studentaffairs/studentlife/judicial/academic_integrity.htm

ALL cases of cheating or plagiarism will be handed to the Dean's office. Penalties include a failing grade in the class, a note on your official transcript that shows you were punished for cheating, suspension, expulsion and revocation of already awarded degrees.