Computer Science CS134 (Spring 2020)
Shikha Singh & Iris Howley
Laboratory 9
Fun with Recursion[1] *(due Thursday April 30 at 11pm EST)*

Objective. Writing recursive functions and develop "recursive thinking."

Lab overview video. The following videos provide an overview to this lab assignment:
`https://williams.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=fcb0e2bc-2b3d-4d3d-aa3b-aba6015dda8a`.

Recursion is a powerful design technique, but it can be a difficult concept to master. In this lab, we will concentrate on several isolated recursion problems that can be solved independently from one another. The goal of this lab is to practice writing recursive programs and *to train your brain to think recursively.* The recursive approach to problem solving is made up of the following three steps: (a) *reduce* to smaller version(s) of the same problem, (b) *delegate* the smaller subproblem(s) to the "recursion fairy"[2], (c) *combine* the solutions of the subproblem(s) and reach/return the result.

When writing a recursive function, always start with the *base case(s)*, which occur when the problem is so small that we can solve it directly. Make sure your base case is eventually reached lest you end up in an infinite recursion! Then write the recursive case, which involves taking the "leap of faith" in the form of the above three steps.

Getting Started. First, remember to connect to the Williams network with the VPN software. Now, clone the starter repository for this week's lab, using the process you followed for our previous remote labs. Go to `https://evolene.cs.williams.edu`, navigate to this week's repository, click `Clone` and select the `Copy URL to clipboard` button under the `Clone with HTTPS` text. Return to the `Terminal/Command Prompt`, navigate to your cs134 directory, and type `git clone` and paste the URL you just copied, followed by the name of the lab, e.g.:

```
git clone https://evolene.cs.williams.edu/cs134-s20/lab08/22xyz3.git lab09/
```

where your CS username replaces 22xyz3.

You will the find five Python files `sumDigits.py`, `hourglass.py`, `quilt.py`, `shrub.py` and `recursiveSquares.py`, each corresponding to Tasks 1 to 5 respectively.

This week's tasks. This week, you must attempt three out of the five tasks in this lab assignment. Take time to figure out how each problem is recursive in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the same problem. Make sure to watch Lectures 22-24 before starting as they cover similar examples and lay the foundation for this lab.

Task 1. (Fruitful Recursion with numbers) In this task, you will write the `sumDigits` function in the file `sumDigits.py`. The `sumDigits` function takes an integer `num` as input and computes and returns the sum of the digits of the absolute value of `num`.[3]

[1] This lab assignment has been partially adapted from Wellesley Fall 2018 course materials.

[2] The recursion fairy formally goes by the name *induction hypothesis*, but we won't talk about that in this course. Take CS136 or Math 200 to learn more!

[3] Recall that the absolute value of a number x, denoted $|x|$ is equal x to x if $x \geq 0$, and is equal to $-x$ if $x < 0$. In Python, you can use the in-built method `abs(x)` to compute the absolute value of a number x.

Your function must be recursive and should not use any loops. You are also not allowed to change the type of num from an integer to a string. Here are examples of how your function should behave:

```
>>> sumDigits(0)
0
>>> sumDigits(-7)
7
>>> sumDigits(90)
9
>>> sumDigits(-42)
6
>>> sumDigits(889832)
38
```

Recursive problem structure. Suppose `num = 1729`, you can think of `sumDigits(1729)` as `sumDigits(172)` + 9; and `sumDigits(172)` as `sumDigits(17)` + 2, and so on.

So, now the question is given an integer `num`, how do we separate it into two parts—(i) everything except its last digit and (ii) its last digit—using arithmetic operators. *Hint. Think back to the moon-age lab, where we extracted the last two digits of a year using the modulo (remainder) operator.*

For the base case, think about when is the number so small that we trivially know the sum of its digits and can return it directly.

Task 2. (Non-fruitful Recursion with Patterns) In this task, you must define a recursive function `hourglass` in the file `hourglass.py` that prints an hourglass shape of characters to the screen. You must use recursion (no loops are allowed). The function takes four arguments: `indent`, `width`, `char1` and `char2` (in that order), where

- `indent` is the number of spaces printed to the left of the topmost and bottommost line of characters
- `width` is the number of non-space characters in the topmost and bottommost line of the hourglass
- `char1` and `char2` are the two characters in the hourglass, and they alternate beginning with `char1`

The top line of the hourglass shape consists of `indent` number of spaces followed by width copies of `char1`. Each subsequent line has two fewer characters and begins with one more space as indent than the previous line, until a line with just one or two characters is reached. Then the lines start growing again in the opposite order. If width is less than or equal to zero, nothing should be printed.
The following invocations of the hourglass function should produce the output shown below (more doctests are provided in `hourglass.py`).

```
>>> hourglass(5, 7, '*', '-')
     *******
      -----
       ***
        -
       ***
      -----
     *******
```

```
>>> hourglass(0, 12, 'x', 'o')
xxxxxxxxxxxx
 oooooooooo
  xxxxxxxx
   oooooo
    xxxx
     oo
    xxxx
   oooooo
  xxxxxxxx
 oooooooooo
xxxxxxxxxxxx
>>> hourglass(0, 1, 'A', 'B')
A
>>> hourglass(4, 1, 'E', 'F')
    E
```

Recursive problem structure. Let's think about how we can break this problem up into a smaller version of itself. If you look at an example hourglass and forget the topmost and bottommost line, what can you say about the rest of the pattern? Can you determine how the arguments `indent`, `width`, `char1` and `char2` should change between the recursive calls? Think about how to order your print statements with respect to your recursive call. Finally, don't forget the base cases—there may be more than one!
Remark. It would be helpful to review the recursive patterns portion of Lecture 22 (`https://williams.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f139dc62-0e68-4b1b-9429-ab9f01819a13`).

Recursive Graphics with the `Turtle` module

The next three tasks make use of the `turtle` module in python. The module uses `tkinter` for the underlying graphics, so you need a version of Python installed with `Tk` support. Fortunately, most recent versions of Python on both Mac/Windows come with it pre-installed. To test whether your version can use the `turtle` package, go into interactive python (by typing `python3` in Terminal/Command Prompt) and run the following import command:

```
>>> from turtle import *
```

If this does not throw an error you should be good to go—to see the turtle graphics window, you can further type `fd(20)` which will pop up the window and move the turtle forward by 20 units. (By default the turtle's initial position is at the $(0, 0)$—the center of the screen.) If this does not work for you, please reach out to any of the CS134 instructors and we can help you get setup. Otherwise, you are ready to do some graphical recursion!

Remarks. You'll want to make sure you have watched Lecture 23 (`https://williams.hosted.panopto.com/Panopto/Pages/Viewer.aspx?pid=63110b7e-9602-465c-b9c1-aba3010c657e`) for an introduction to the Turtle module and Lecture 24 (`https://williams.hosted.panopto.com/Panopto/Pages/Viewer.aspx?pid=46d2a355-16a4-4cb9-a756-aba4016b3f11`) for examples of graphical recursion before attempting these tasks.

If you are using a Windows machine, you need an `epsviewer` to view the EPS files generated by the turtle graphics. You can download it for free from here: `https://epsviewer.org/`. In the coming tasks, if you want the graphics window to not close automatically you can use the command `exitonclick()`.

Task 3. (Draw Williams Quilt) In this task, you will create a beautiful recursive quilt with Williams colors to bring you a bit closer to the campus that you all miss.

In `quilt.py`, you will define the function `drawQuilt(size, level, color1=purple, color2=gold)` where the input parameters are described below:

- `size` denotes the side length of the whole quilt (that is, the largest square); the side length of each successive square goes down by half
- `level` determines how many recursive subpatterns the quilt will have—a quilt of level ℓ has a quilt of level $\ell - 1$ as its upper-left and lower-right subpattern. In particular, `level = 0` means nothing is drawn, `level = 1` means the entire quilt is one solid square, `level = 2` means the quilt has two level 1 quilts as subpatterns, and so on.
- `color1` and `color2` denote the colors in the quilt, set by default to `purple` and `gold`[4] respectively. The colors of the quilt alternate between `color1` and `color2`, starting with `color1`.

The following helper functions have been defined for you in `quilt.py`.

- `drawSquare(size, color)`. Draws a square of side length `size` filled with color `color`, with the turtle's starting position as one of the endpoints. Your function must call `drawSquare` only once.
- `initializeTurtle(size)`. Sets up a screen of size slightly bigger than the pattern, resets the turtle, and sets its position to be lower-left endpoint of the pattern, that is, `(-size/2, -size/2)`.
- `testDrawQuilt(size, level, color1 = purple, color2 = gold)`. Calls the `initializeTurtle` function followed by the `drawQuilt` function and saves the resulting figure as an `eps` file.

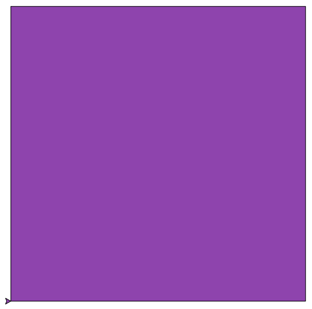
Aside from the single call to `drawSquare`, the remaining pattern must be drawn using recursion. The only turtle commands you can use in your function are `fd, bk, lt` and `rt` (forward, backward, left turn, and right turn, respectively).
Consider the following invocations of `drawQuilt`, which describe how your method should behave.

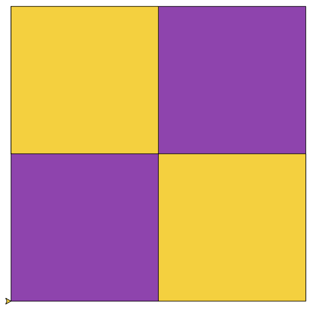
Turtle position invariant. Your `drawQuilt` function must be invariant with respect to the turtle's position and direction. The `initializeTurtle` function, which is called before `drawQuilt` in `testDrawQuilt`, positions the turtle at the lower-left corner of the quilt facing east. When `drawQuilt` is done drawing the quilt (via recursive calls) it should return the turtle to its starting configuration.

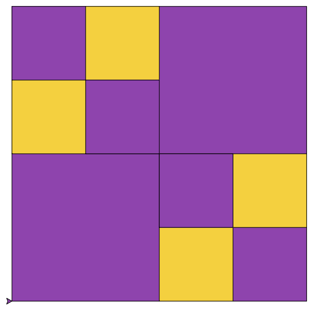
Recursive problem structure. Our quilts are recursively defined as their upper-left and lower-right quadrants are a smaller quilt. To get things right, all you need to do is set up your turtle in the right spot before and after making the recursive calls and let the recursion fairy take care of the rest. Consider how the parameters `size`, `length`, `color1`, and `color2` should change between the recursive calls. And of course, don't forget the base case—at what `level`, do we not have to draw anything at all?

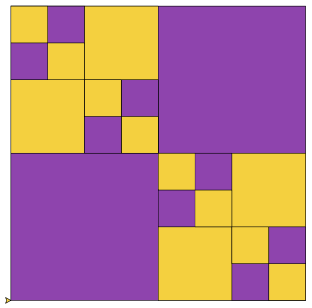
Testing your function. You can test your function by uncommenting the provided function calls to the `testDrawQuilt` function in the `if __name__ == '__main__':` block and comparing the output to the

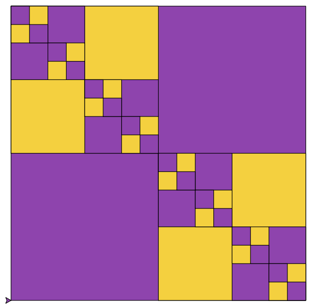
[4]Here purple and gold are global variables that have been predefined with the corresponding HEX color codes

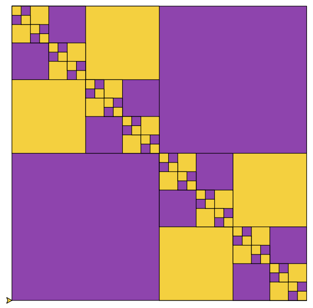

`drawQuilt(500, 1)`


`drawQuilt(500, 2)`


`drawQuilt(500, 3)`


`drawQuilt(500, 4)`


`drawQuilt(500, 5)`


`drawQuilt(500, 6)`

examples provided here. You may change the speed of the turtle in the `initializeTurtle` function to adjust the speed of the animation.

Task 4. (Recursive Shrubs) In this task, you will write a fruitful recursive function named `shrub` in the file `shrub.py` that draws a tree pattern and returns a tuple of values, described below.

The function `shrub(trunkLength, angle, shrinkFactor, minLength)` takes in four parameters:

- `trunkLength` is the length of the vertical branch at the base of the shrub
- `angle` is the angle between a trunk and its right and left branches
- `shrinkFactor` defines the length of the right and left branches relative to their trunk. Specifically, the length of the right branch is `shrinkFactor` times the `trunkLength`, and the trunk of the left branch is `shrinkFactor * shrinkFactor` times the `trunkLength`
- `minLength` is the minimum branch length in the shrub

The `shrub` function (in addition to drawing the shrub) must return a pair of items, where

- the first item is the total number of branches in the shrub, including the trunk
- the second item is the total length of branches in the shrub, including the trunk

The following helper functions have been defined for you in `shrub.py`.

- `initializeTurtle()`. Sets up the screen, resets the turtle, and positions it at an appropriate spot, and orients its pointer to face north.
- `testShrub(trunkLength, angle, shrinkFactor, minLength)`. Calls the `initializeTurtle` function followed by the `shrub` function, prints the tuple returned and saves the figure generated.

Your `shrub` function must only use turtle commands `fd`, `bk`, `lt` and `rt`.

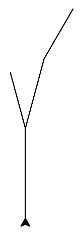
See the sample invocations of the shrub function on the next page, with the tuple after the function call -> indicating the value returned by that invocation.

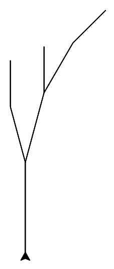
Turtle position invariant. Your shrub function should be invariant with respect to the turtle's position and direction. The `initializeTurtle` function, which is called before `shrub` in `testShrub`, positions the turtle at the base of the main trunk facing vertically up. When `shrub` is done drawing all its branches (via recursive calls) it should return the turtle to this starting configuration. Lecture 24 covers a similar example, reviewing which will be helpful in solving this problem.

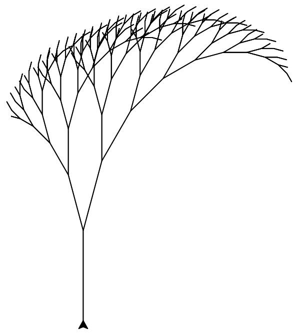
Recursive problem structure. A shrub is recursive in its structure as you can view its left and right branch as a shrub in itself (originating at an angle from its base trunk and having a shorter trunk, that is controlled by the `shrinkFactor` parameter).

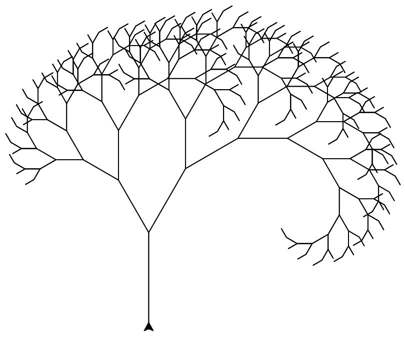
To get things right, you need to set up your turtle in the right spot before and after making the recursive calls and let the recursion fairy take care of the rest of it. Consider how the parameters `trunkLength` should change for the recursive call drawing the right and left branches. And of course, don't forget the base case—at what size of trunk length do we not have to draw anything at all?

Testing your function. You can test your function by uncommenting the provided function calls to the `testShrub` function in the `if __name__ == '__main__':` block and comparing the output to the examples here and the expected return values provided in comments. You may change the speed of the turtle in `initializeTurtle` function to adjust speed of the animation.


`shrub(100, 15, 0.8, 60) -> (4, 308.0)`


`shrub(100, 15, 0.8, 50) -> (7, 461.6)`


`shrub(100, 15, 0.8, 10) -> (12, 666.4000000000001)`


`shrub(100, 30, 0.82, 10) -> (376, 6386.440567704483)`

Task 5. (Challenging) In this task, you will combine the drawing of a colorful recursive pattern (as in Task 3) with the returning of tuples (as in Task 4). In particular, we will draw a recursive pattern of tri-colored squares and count the total number of squares of each color in the pattern. With three colors in the mix, this question really pushes you to understand fruitful recursion.

In `recursiveSquares.py`, you will define the function
`recursiveSqCount(patternSide, minSide, color1 = "red", color2 = "blue", color3 = "cyan")`
where the input parameters are described below:

- `patternSide` denotes the side length of the entire pattern, where the big solid square in the upper left quadrant has side length `patternSide/2`. Recursive subpatterns of side length `patternSide/2` fill the other three quadrants.
- `minSide` is the smallest value of `patternSide` for which a non-empty pattern is drawn. If the `patternSide` is strictly smaller than `minSide`, `recursiveSqCount` draws nothing
- `color1` is the color of the square in the upper left corner of the pattern
- `color2` and `color3` are the two colors that are used in the recursive subpatterns. To understand how they are used, consider the example invocations attached on the next page.

The `recursiveSqCount` function must return a triple (that is, a 3-item tuple) where:

- the first item is the total number of `color1` squares in the pattern
- the second item is the total number of `color2` squares in the pattern
- the third item is the total number of `color3` squares in the pattern

The following helper functions have been defined for you in `recursiveSquares.py`.

- `drawSquare(size, color)`. Draws a square of side length `size` filled with color `color`, with the turtle's starting position as one of the endpoints. Your function must call `drawSquare` only once.
- `initializeTurtle(patternSide)`. Sets up a screen of size slightly bigger than the pattern, resets the turtle, and sets its position to be lower-left endpoint of the pattern, that is, `(-patternSide/2, -patternSide/2)`.

- `testRecursiveSqCount(patternSide, minSide, color1 = "red", color2 = "blue", color3 = "cyan")`. Calls the `initializeTurtle` function followed by the `recursiveSqCount` function, prints the tuple returned and saves the resulting figure as an `eps` file.

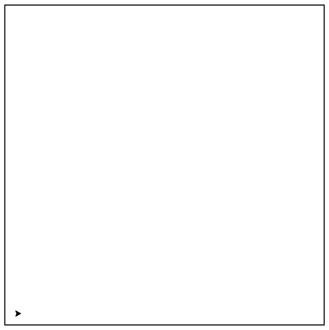
Aside from the single call to `drawSquare`, the remaining pattern must be drawn using recursion. The only turtle command you may use in your function are `fd`, `bk`, `lt` and `rt`.

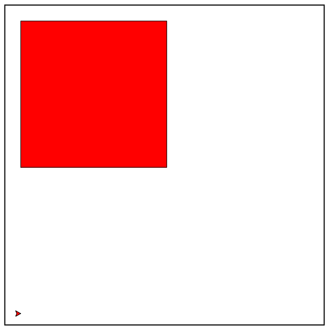
The sample invocations of the `recursiveSqCount` function, with the tuple after the function call `->` indicating the value returned by that invocation, are given on the next page. Note that `recursiveSqCount(512, 8)` takes a really long time to finish even with turtle speed set to 0 (fastest).

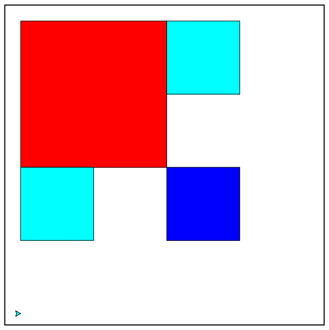
Turtle position invariant. Your `recursiveSqCount` function should be invariant with respect to the turtle's position and direction. The `initializeTurtle` function, which is called before `recursiveSqCount` in `testRecursiveSqCount`, positions the turtle at the bottom left corner of the pattern facing east. When `recursiveSqCount` is done it should return the turtle to this starting configuration.

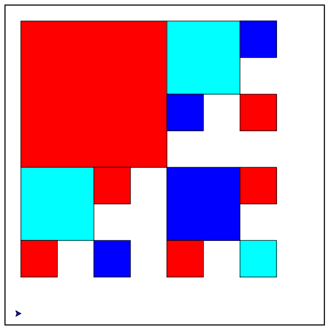
Recursive problem structure. The pattern is recursive in its structure as its upper-left, lower-right and lower-left quadrants are made up of a smaller version of the pattern itself, with `patternSide` value as half of the whole pattern. To get things right in this task, you need to not only maintain the position invariant and figure out how the colors change in each recursive call, you also need to determine how to combine the counts returned by each recursive call correctly—thinking about this is challenging!

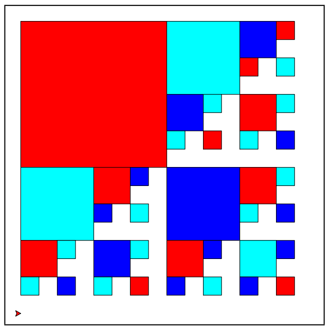
Testing your function. You can test your function by uncommenting the provided function calls to the `testRecursiveSqCount` function in the `if __name__ == '__main__':` block and comparing the output to the examples below. You may adjust the speed of the turtle in `initializeTurtle` function to adjust the speed of the animation.

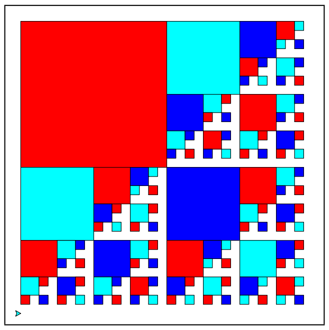

`recursiveSqCount(512, 600) -> (0, 0, 0)`

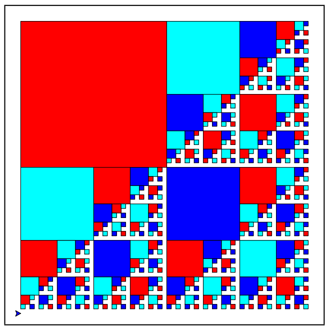

`recursiveSqCount(512, 512) -> (1, 0, 0)`

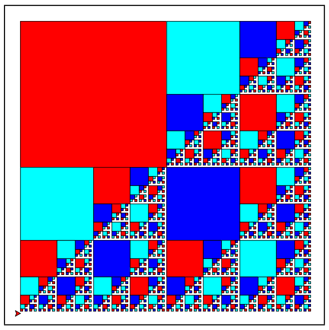

`recursiveSqCount(512, 256) -> (1, 1, 2)`


`recursiveSqCount(512, 128) -> (6, 4, 3)`


`recursiveSqCount(512, 64) -> (11, 13, 16)`


`recursiveSqCount(512, 32) -> (46, 40, 35)`


`recursiveSqCount(512, 16) -> (111, 121, 132)`


`recursiveSqCount(512, 8) -> (386, 364, 343)`

Submitting your work. When you're finished, stage, commit, and push your python files corresponding to the tasks you completed to the server as you did in previous labs. For tasks 3, 4, and 5, in addition to the python file, you must stage and push the figure generated by the following function calls:

- If attempting Task 3, submit the figure generated by `drawQuilt(500, 4)`
- If attempting Task 4, submit the figure generated by `shrub(100, 15, 0.8, 10)`
- If attempting Task 5, submit the figure generated by `recursiveSq(512, 64)`

Remember that you must certify that your work is your own, by typing out the Honor Code statement in the `honorcode.txt` file, committing and pushing it along with your work.

Late days. You are allowed a total of 3 late days over the semester, with at most 2 late days towards any one lab. You must request a late day in advance on the form, here: `http://bit.ly/s20late`.

Grading Guidelines. Both functionality and programming style are important when writing code, just as both the content and the writing style are important when writing an essay. In this program, some of the specific functional requirements we will test for include:

- You must attempt at least three out of the five tasks.
- Each of your functions for Tasks 1-5 must be recursive in nature (cannot use loops).
- Your base cases must be stated explicitly in your recursive functions.
- Your recursive function for `sumDigits` must not type cast input integer to `str` or use string slicing.
- For Tasks 3-5, your function must generate the figures as specified in the task description.
- You must commit and push one figure for each of graphical questions 3-5, as explained above.
- For any of the graphical recursion questions, your recursive function should not use any turtle command other than `fd, bk, lt` and `rt`.
- The `drawQuilt` and `recursiveSqCount` functions in Task 3 and 5 must call the helper function `drawSquare` only once and draw the rest of the pattern via recursion.
- Just like previous labs, we require that each of the functions defined follow our specifications. Do not modify the function names, their parameters, nor what is returned. The examples in task descriptions provide details about these constraints.

Stylistically, we expect to see programs that exhibit the following: meaningful names used in declarations, informative comments, docstrings for all methods and at the top-level, good and consistent formatting, and good choice of Python commands. There is some subjectivity to what makes good style, but the basic goal is to make your ideas as clear and easy to follow as possible.

Grading scale. Programming labs will be graded on the following scale:

A+	An absolutely fantastic submission of the sort that will only come along a few times during the semester.
A	A submission that exceeds our standard expectations for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.
A-	A submission that satisfies all the requirements for the assignment – a job well done.
B+	Submission meets the requirements for the assignment, possibly with a few small problems.
B	A submission that has problems serious enough to fall short of the requirements for the assignment.
C	A submission that has extremely serious problems, but nonetheless shows some effort & understanding.
D	A submission that shows little effort and does not represent passing work.

Extra credit—Challenge yourself! We're only requiring you solve 3 out of the 5 tasks, but if you'd like to dig deeper feel free to solve the rest as well!

⋆