Karen Shay West
9 Shannon Marie Way, North Easton, MA 02356

508-844-9776 http://www.linkedin.com/in/karenshaywest KarenWest15@gmail.com

SUMMARY OF RECENT ONLINE TECHNICAL – Applied Python Examples – see my Linked In Project Section for details and github links to files:

- **Applied Python Algorithms:**
  - We used Python to do:
    - **financial calculations demonstrating various speeds of paying off debt:** We first used the formula: Any payment you make at beginning of month is deducted from balance = p0. At beginning of month 1, credit company charges interest on balance. So if your annual interest rate is r, at the beginning of month 1, your new balance is your previous balance b0 less the payment p0 plus interest on the new balance for the month. b1 = (b0-p0) * (1 +r/12). In month 1, we make another payment p1 to cover interest costs too, thus at beginning of month 2, balance is: b2 = (b1-p1) * (1 + r/12). We saw that if you choose to pay off the minimum monthly payment each month, compound interest dramatically reduces your ability to lower your debt. Next, we calculated the minimum fixed monthly payment needed in order to pay off a credit card balance within 12 months. So in this part, we are not dealing with a minimum monthly payment rate. We computed the lowest monthly payment that would pay off the debt in under 1 year. We assumed interest was compounded monthly according to the balance at the end of the month after payment was made. Monthly Interest Rate = Annual Interest Rate / 12. Updated balance each month = (Previous balance - Minimum monthly payment) * (1 + Monthly Interest Rate). We then used bisection search to make our program faster. Lower bound = 1/12th of original balance. Upper bound = 1/12th of the balance, after it has been compounded monthly for an entire year = (Balance * (1 + Monthly interest rate) ** 12) / 12. You should get the same answer as you did for part 2 in this part 3 only faster with bisection search.
    - **write computer games such as hangman and word find:** we used successive approximation to try to guess the answer to a problem and then check our guess. If guess is wrong, keep improving it in small increments and checking it, until you determine the answer is good enough. We also looked at Newton's Method, which uses successive approximation to to find the roots of a function. Polynomials were represented as lists. The index of the list was the power, and the value of the index to represent the coefficient. We also had to compute the derivative of polynomials. Newton's Method (also known as Newton-Raphson Method) is a successive approximation method for finding the roots of a function, or the values of x such that f(x) = 0. (1) iteration zero - guess some x0. (2) check to see if a root by calcuating f(x0) within some small epsilon of 0. (3) If f(x0) not good enough, come up with better guess x1. x1 = x0 - f(x0) / (deriv. of f(xo)), (4) check to see if x1 close enough to a root. If not, make better guess x2 and check and so on. For every x(n), that is not close enough to root, replace x(n+1) = x(n) - f(x(n)) / (deriv. of f(x(n))) and check. Repeat until you have a value close enough to root. Polynomial functions were used for simplicity. The last part of this problem set was a hang man word game. The computer selected from a list of available words read in from a list file. The player knows the number of letters at start of game. Player guesses one letter per round. Player receives immediate feedback

whether letter is in word. Each round the game displays partially guessed word so far, as well as letters player has not used yet. Player is allowed 8 guesses. Player loses a guess only when guessing incorrectly. Error message displayed if same letter guessed twice, but not counted. Game ends when player either guesses word or runs out of guesses.

- **Scrabble game:** Rules of the Game: Player is dealt hand of n letters chosen at random. Player arranges hand into as many words as they want out of letters, using each letter at most once. Some letters remain unused and are not scored. The score for the hand is the sum of the scores for each word formed. The score for a word is the sum of the points for letters in the word, multiplied by the length of the word, plus 50 points if all n letters are used on the first word created. Letters are scored as in Scrabble. A=1, B=3, C=3, D=2, E=1, and so on. A dictionary was used (key,value pair entries) to associate letter to its value for this game. A function was used to get the frequency of a letter in a word. When given a string of letters as input, it returns a dictionary (key,value pairs) of a letter as key, and the value was the number of occurrences of that letter in the word.
- **ciper decryptions and secret messages:** Encryption is the process of obscuring information to make it unreadable without special knowledge. Encryption lets you share information with other trusted people, without fear of disclosure. A cipher is an algorithm for performing encryption. The original information is called plaintext. After encryption, it is called ciphertext. A cipher depends on a piece of information called a key. The same plaintext encrypted with 2 different keys should have 2 different ciphertexts. Without the key, it should be difficult to decrypt the resulting ciphertext into readable plaintext. Caesar Cipher's basic idea is to choose an integer key, shift every letter of your message by the integer key. "h" becomes "j" for integer 2, "e" becomes "g", etc, shifting the letter 2 spots to the right. For this cipher's simplicity, punctuation and other such letters and numbers were ignored, and upper case was mapped to upper case, lower case to lower case. We then took someone else's encrypted text, and had to find the shift key that would decrypt the text, by finding if all words were valid from the shift key chosen. Next, we were asked to do string reversal using only string operations: indexing, slicing, concatenation with a recursive function, not using any loops. Next we had to write an Erician function, meaning if a word contained the letters "eric" in that order, but there could be other letters in between, as long as in that order. The last part of this assignment was to emulate an old fashioned type writer, in that when ever it is in the middle of a word, and reaches the desired line length, the internal bell rings, signifying to the typist that after finishing the current word, a newline must be inserted manually. This process was emulated such that a newline was inserted as required after each word that exceeded the desired line length, and this newline insert function had to be recursive.
- **parsing web pages such as Yahoo and Google to create triggers when certain phrases were found within the web pages:** We built a program to monitor news feeds over the internet. It filtered the news, alerting the user when it notices a news story that matches the user's interests. Python concepts: implement classes and their attributes, understanding class methods, understanding inheritance, telling the difference between a class and an instance of that class, and utilizing libraries as black boxes. The project purpose was as follows. Many web sites have content that is updated on an unpredictable schedule. News sites such as Google News, are a good example of this. One tedious way to keep track of this changing content is to load a website up in your browser, and periodically hit the refresh button. This process can be streamlined and

automated by connecting to the web site's RSS feed, using an RSS feed reader instead of a web browser. An RSS (Really Simple Syndication) reader will periodically collect and draw your attention to updated content. An RSS feed consists of (periodically changing) data stored in an XML-format file residing on a web server. For this project we did not need to know XML or how to access these files over the network. We used a special Python module to deal with these lower level details. The higher level details of the structure of the Google News RSS feed was our focus of this problem set. If we loaded the URL into a browser, we would see the browser's interpretation of the XML code generated by the feed. When you connect to Google News RSS feed, you receive a list of items. Each entry in this list represents a single news item. In a Google News feed, every entry has the following fields: guid = globally unique identifier for this news story, title, subject, summary, link. The goal was to create an application that aggregates several RSS feeds from various sources and can act on all of them in the exact same way all in one place.

- **used a graphic of a tiled floor and various types of Python made robots to demonstrate how best and efficiently it could be cleaned by these:** We used Python library packages Matplotlib (Pylab) and NumPy for this assignment. In this assignment we coded a simulation to compare how much time a group of Roomba like robots take to clean a floor using 2 different strategies. The robot starts out in a random position in the room and with a random direction of motion. A given floor had N > 0 robots. The room was rectangular. We had to keep track of which tiles of the floor had already been cleaned by the robots. Each robot had a position (x,y) in the room and a direction of motion (0 <= dir < 360) degrees. All robots moved at the same speed. Every time step, a robot moves in its direction by speed "s" units. If a robot detects it will hit a wall within the time step, that time step is instead spent picking a new direction at random. We did not put all the robots methods in one class, since we also used alternate moving strategy robots that used the same robot interface otherwise. In each trial, the objective is to determine how many time steps on average are needed before a fraction of the room has been cleaned. The professors provided us with some animation software so we could more easy with visualizations debug our code while our robots cleaned the room. However, it slows down running your code. We tried out robots that change direction only when they hits walls, and other robots that randomly change direction every time step. Using Pylab, we plotted the percentage of room that a robot cleans, time it takes 1-10 robots to clean 70% of a room, percentage of a room that 1-10 robots clean, time for robots to clean varying percentages of a room, and area of a room that 1-10 robots clean, and similarly these statistics for various sized rooms.
- **simulations for the drug and disease industries where you have a number of diseased cells, antibiotics, resistances to antibiotics, and a percentage you need to achieve to remove the illness, and calculation to compare using and not using a drug for it:**
  - **Stochastic Simulation of Patient Virus Population Dynamics And Treatment Regimens:** In this assignment, we used Python and Pylab to design and implement a stochastic simulation of patient and virus population dynamics, and reach conclusions about treatment regimens based on the simulation results. In biology class, you learn that traits of an organism are determined by its genetic code. When organisms reproduce, their offspring will inherit genetic info. from their parent. Their genetic info. will be modified due to mixing or mutations in the genome replication process, thus introducing diversity into a population. Viruses are no

exception. Two characteristics of viruses make them particularly difficult to treat. The first is that their replication mechanism often lacks the error checking mechanisms that are present in more complex organisms. This speeds up the rate of mutation. Secondly, viruses replicate extremely quickly (orders of magnitude faster than humans) - thus while we may be used to thinking of evolution as a process which occurs over long time scales, populations of viruses can undergo substantial evolutionary changes within a single patient over the course of treatment. These 2 characteristics allow a virus population to acquire genetic resistance to therapy quickly. In this problem set, we make use of simulations to explore the effect of drugs on the virus population and determine how best to address the treatment challenges in a simplified model.

- **Stochastic Simulation of Patient Virus Population Dynamics - Effect of Delaying Drug Treatment on Patient Disease, Using 2 Drugs, Patient non-Compliance:** we used the classes implemented in problem set 8 described in another entry in this project list (ResistantVirus, TreatedPatient - see document in link for problem set 8 for details). In this problem set, we examined the effect of delaying treatment on the ability of the drug to eradicate the virus population. We needed multiple simulations to observe trends in the distributions of patient outcomes. In the last part of problem set 8, we ran a simulation that consisted of 150 time steps before adding drug guttagonol, followed by another 150 time steps. Here we run the same simulation but this time for 300, 150, 75 and 0 time steps before adding guttagonol to the patient, and then ran another 150 time steps. We used the same init params as problem set 8 (refer to that description). For each case, we repeated the experiment for enough trials to gain reasonable insight into the expected result. Rather than averaging the final virus population across different trials as in probSet8, we used pylab's hist() function to plot a histogram of the final virus populations under each case for each trial. The x-axis of the histogram should be the final virus population values, and the y-axis the number of trials belonging to each histogram bin, We had to justify the number of trials we ran for each case in order to obtain a reasonable distribution. We filled in the function simulationDelayedTreatment(numTrials) to perform this simulation. With a delay of 300 and 150 cases, 0-5% were cured, for 75, 6-15%, for 0, 86-100%, so the earlier the drug was applied the more likely for this virus it was to be cured. We then examined these same things with 2 drugs and found it cured the patient more quickly when administered early for this virus. We also discussed how to model a patient that refuses to take medication. You make a TreatedPatient class that stochastically does not take its medication, and predicted less patients would be cured.

▪ **comparing the Breadth First Search, Depth First Search, tree and graph algorithms, puzzles, cliques and power sets for our class projects**

- **Shortest Route on Map of MIT Optimizing Outdoor Time Using Brute Force and Depth First Search Algorithms:** we found the solution to an optimization problem on how to find the shortest route from one building to another on the MIT campus given you wish to constrain the amount of time you spend walking outdoors. A graph consists of a set of nodes and edges where an edge connects 2 nodes. Each node has children if there is an edge connecting it to a node below it. There are directed and undirected nodes. Directed graphs are digraphs. Undirected graphs are bidirectional. An edge can have a weight. The objective function for graph theory is either minimized or maximized. So for this problem, we read in a map of the MIT campus and built a graph from it: start building, destination building, distance between them in meters, and distance between spent outdoors. Not every route was bi-directional. So we first created a data structure for a weighted digraph, node and edge. We then modeled the MIT campus as a graph. We then found the shortest route using Brute Force, a variation of Depth First Search algorithm, spending minimal time outside.

- **BFS, DFS, graphs, optimization:** A graph - set of nodes connected by edges. Edges - unidirectional if directed, or digraph. Weight or cost added to an edge, then weighted graph. Graph optimizations: shortest path (SP) - shortest set of edges from node A to B, shortest weighted path - same as SP but with a min. function on weights of edges in seq., cliques - set of nodes such that there is a path with max. length in graph between each pair of nodes in set, Min. Cut - 2 sets of nodes, a cut is a set of edges whose removal eliminates all paths from every node in one set to each node in the other. The min. cut - smallest set of cuts. DFS (Depth First Search) starts at root and if not goal node, extend current path by adding each child of current node to path, unless already in path. New paths added to set of paths at front (a stack - last in, first out). Select next path - recursively repeat. No children, go to next, and stop when goal node reached or no more paths. SP - track best path found so far - when you find a new path - keep going if path still shorter than best. BFS (Breadth First First) - do not go down first branch of existing tree - examine all children of current node first before going deeper in tree. BFS, no weights, stop when you find solution, guaranteed SP. BFS starts at root node, and if not goal, extends current path unless child already in path. Add new paths to potential set of paths - put at end of set (uses a queue - First In, First Out, push to end, pop at front). Select next path and recursively repeat. If no children - go to next - stop at goal node or when no more paths. Weighted graphs - DFS can sum weights rather than just check length of path. BFS - 1st found solution may not be best.
- **Puzzles, Cliques and Power Sets:** We explored graphs to find paths that represent physical networks. We use graphs to search to explore changes to state of physical system, with nodes representing states of system, edges the actions that cause the change of state, and find a sequence of actions to convert the system to desired state. We did an 8-puzzle example and kept track of legal shifts in a dictionary (graph edges), and the puzzle used BFS and DFS, checking for loops, and BFS was quicker for this, with DFS taking a long time and running out of memory. We then explored max-cliques - making subgraphs, such as all people connected to each other in social network, infected populations that had contact, and you make collections of cliques. They are used for communication networks, circuits, genes, social networks, disease populations, etc. The max clique can use brute force for small problems - find all subgraphs and test if complete (all connected), keep track of largest clique, and extend to recursively find other large cliques by removing nodes of largest cliques and repeating. The last example was a power set, the set of all subsets. To find cliques, generate power sets, all subgraphs, test if all connected/complete, keep track of largest found. DFS and BFS and optimization were used for graph searches of related objects.

▪ **optimization problems, dynamic programming, fitting data to linear and quadratic curves, normal distributions, measuring goodness of fit of data to a curve, uniform distributions, estimating Pi, Monty Hall Problems, number of samples needed for a coin flip experiment, variance and standard deviation, plotted histograms, causal**

and predictive nondeterminism, stochastic processes, monte carlo simulations, hash tables, random walks and simulation models, examined different algorithms for searching and sorting and their Big O Notation performances.

- **Optimization Problems:** This lecture example Python code demonstrated code optimization. You start with an objective function and a set of constraints the solution must satisfy. Examples were the n-queens problem, where you place n-queens on an NxN board so that no 2 queens attack each other (no 2 queens share the same row, column or diagonal). Other examples were bin packing, cutting stock and min cut networks and the traveling salesmen problem. The challenge shown was that these problems are "hard" to solve. Often finding optimal solutions requires examining all possible combinations of items. The time to examine all combinations grows exponentially with the number of items. "Real World" problems have a large number of items. The next set of examples were the burglar's dilemma, 0/1 knapsack problem, and greedy algorithms. Which metric works best? Max. value, min. weight or max-value / weight. Greedy Algorithm is a heuristic for a 0/1 knapsack problem. Given materials of different value/weight ratios, find the most valuable combination of materials which fit in a knapsack of fixed capacity W. Greedy algorithm with max value/weight metric finds optimal solution for continuous knapsack problem. Optimal solution for 0/1 knapsack problem - formal problem statement: Given vectors of weight wi, and values vi, where 0 <= i <= N-1 items; knapsack holds W. Find 0/1 vector ti. Technique exhaustive search enumerates all possible combinations of items and choose the best one that satisfies all constraints. Generate the power set combinations of all items. Greedy was O(N log N) and Optimal was O(N 2**N).
- **Dynamic Programming:** The dynamic programming using optimal substructure and overlapping subproblems. In the Fibonacci example the memoize function was used to demonstrate using function arguments as a sequence, which works for immutable / hashable arguments. The memoize function remembers previously computed answers, much like a cache does in hardware. It showed knapsack subproblems where you try different ways to optimize a long search, such as to not take the first item in the knapsack, then take the 1st item in the knapsack, combine the 2 solutions and see if there are overlaps. Code examples (in link for this entry) applied these optimizations to find where to place line breaks in a paragraph to minimize "badness". The line break subproblems first tried a solution with no line breaks after 1st word, then another with a line break after 1st word, and combined the solutions to find overlaps. Another code example demonstrated the minimum cost of alignment between 2 sequences.