CSYE 7245
Big-Data Systems and Intelligence Analytics
Assignment 3

Professor: Nik Bear Brown
Due: November 5, 2018

Q1 (5 Points)

Give a brief definition for the following:

i. Linearity of expectation
ii. Z-score
iii. Chernoff bound
iv. Monte Carlo algorithm
v. Bias-variance tradeoff

Q2 (5 Points)

Arrange the following functions in increasing order of asymptotic growth:

- n^2
- 0.33^n
- $5n^5$
- $n^2 \sqrt{n}$
- 5^n
- $\log n$
- $\sqrt{n}$

Q3 (5 Points)

Master Theorem: For the following recurrence, give an expression for the runtime T(n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

a. $T(n) = 4T(n/2) + n$
b. $T(n) = 4T(n/2) + n^2$
c. $T(n) = 2T(n/2) + n^2 \log n$

Q4 (5 Points)

Stephen Curry hit 77 three-point shots in a row in practice. If his probability of hitting an unguarded three-point shot is 90%, what is the likelihood of Stephen Curry making at least 9 out of 10 three-point shots?

Q5 (5 Points)

A booth at the fair has 200 balloons, 5 of which contain \$10 and 1 of which contains \$20. The rest contain only air. If it costs \$1 to randomly break a balloon, what is the expected return of an individual making such an attempt?

Q6 (5 Points)

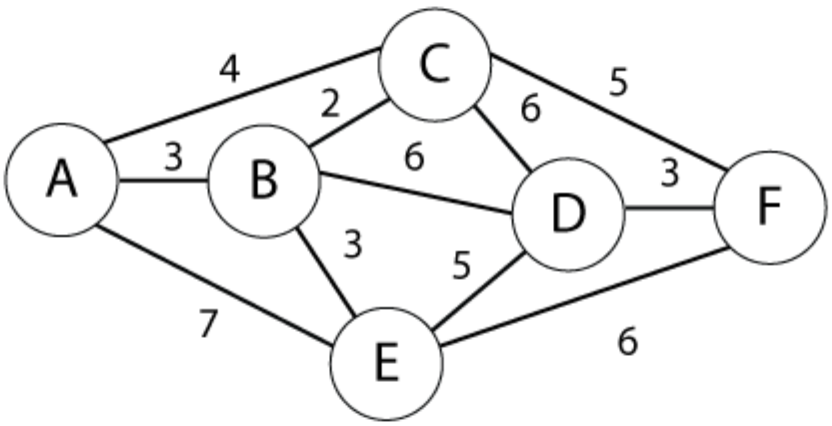
Sort the list of integers below using Merge sort. Show your work. Write a recurrence relation for Merge sort.

(22, 13, 26, 1, 12, 27, 33, 15)

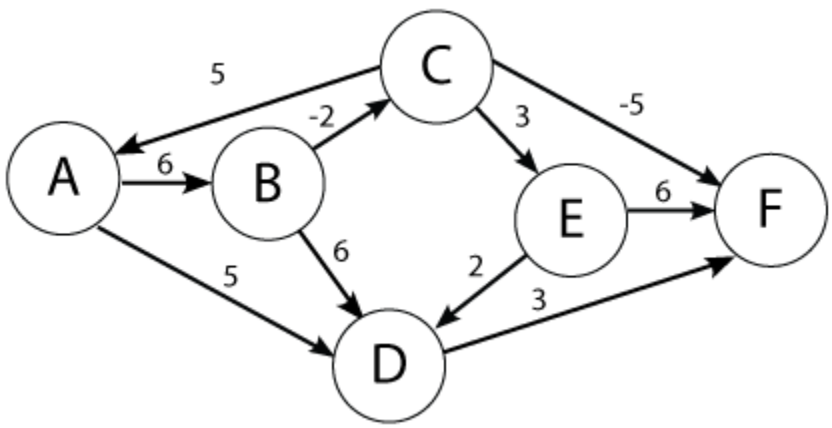
Use the Master Theorem to prove the complexity of your recurrence.

Q7 (5 Points)

Find shortest path from A to F in the graph below using Dijkstra's algorithm. *Show your steps.*



Q8 (5 Points)



Use the Bellman-Ford algorithm to find the shortest path from node A to F in the weighted directed graph above. *Show your work.*

Q9 (5 Points)

In a room of 23 people, what is the probability that someone has the same birthday as you?

Q10 (5 Points)

Suppose S is a very large set of real numbers, and you would like to estimate its median value by sampling. (It is too expensive to sort S to calculate the exact median). You may assume all the numbers in S are distinct. Let $n = |S|$; a number x is an *ε-approximate median* if at least $(0.5 - \varepsilon)n$ numbers in S are less than x and $(0.5 - \varepsilon)n$ numbers in S are greater than x.

Suppose you sample a subset uniformly at random (with replacement) and use that to estimate your median. Use a Chernoff-bound to calculate your confidence in your approximate median estimate. You can choose your confidence level (typical is 90%, 95% or 99% confidence) the distribution of the numbers and your sample size.

Q11 (5 Points)

Two linear regression models return t-statistics of 1 and 19 respectively. What is the null hypothesis in this case. Which t-statistic provides more evidence to reject the null hypothesis.

Q12 (5 Points)

Given the weights and values of the four items in the table below, select a subset of items with the maximum combined value that will fit in a knapsack with a weight limit, *W,* of 6. Use dynamic programming. *Show your work.*

Item $_i$	Value v_i	Weight w_i
1	3	4
2	2	3
3	4	2
4	4	3

Capacity of knapsack W=6

Q13 (40 Points) Search in Pacman

For this question, you'll be implementing search in Pacman (http://ai.berkeley.edu/search.html) from The Pac-Man projects were developed for UC Berkeley's introductory artificial intelligence course, CS 188 (http://ai.berkeley.edu/project_overview.html).

Your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

As in Project 0, this project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

See the autograder tutorial in Project 0 for more information about using the autograder.

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code and supporting files as a zip archive.

Q12-1: Depth First Search
Q12-2: Breadth First Search
Q12-3: Uniform Cost Search
Q12-4: A* Search
Q12-5: Corners Problem: Representation
Q12-6: Corners Problem: Heuristic
Q12-7: Eating All The Dots: Heuristic
Q12-8: Suboptimal Search

Question 1 (5 points): Finding a Fixed Food Dot using Depth First Search

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented -- that's your job. As you work through the following questions, you might find it useful to refer to the object glossary (the second to last tab in the navigation bar above).

First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

Important note: Make sure to **use** the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need *not* be of this form to receive full credit).

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. To make your algorithm *complete*, write the graph search version of DFS, which avoids expanding any already visited states.

Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent
```

```
python pacman.py -l mediumMaze -p SearchAgent
```

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a `Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 2 (5 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Does BFS find a least cost solution? If not, check your implementation.

Hint: If Pacman moves too slowly for you, try the option `--frameTime 0`.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

Question 3 (5 points): Varying the Cost Function

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Note: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details).

Question 4 (5 points): A* search

Implement A* graph search in the empty function `aStarSearch` in `search.py`. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `search.py` is a trivial example.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `manhattanHeuristic` in `searchAgents.py`).

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on `openMaze` for the various search strategies?

Question 5 (5 points): Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! *Hint*: the shortest path through `tinyCorners` takes 28 steps.

Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

Implement the `CornersProblem` search problem in `searchAgents.py`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

```
python pacman.py -l mediumCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Our implementation of `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners`. However, heuristics (used with A* search) can reduce the amount of searching required.

Question 6 (5 points): Corners Problem: Heuristic

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: `AStarCornersAgent` is a shortcut for

```
-p SearchAgent -a
fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic.
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost *c*, then taking that action can only cause a drop in heuristic of at most *c*.

Remember that admissibility isn't enough to guarantee correctness in graph search -- you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

Number of nodes expanded	Grade
more than 2000	0/5
at most 2000	1/5
at most 1600	3/5
at most 1200	5/5

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful!

Question 7 (4 points): Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, `A*` with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to `testSearch` with no code change on your part (total cost of 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note: `AStarFoodSearchAgent` is a shortcut for `-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic`.

You should find that UCS starts to slow down even for the seemingly simple `tinySearch`. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.

Note: Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

Fill in `foodHeuristic` in `searchAgents.py` with a consistent heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch`board:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.

Any non-trivial non-negative consistent heuristic will receive 1 point. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

Number of nodes expanded	Grade
more than 15000	1/4
at most 15000	2/4
at most 12000	3/4
at most 9000	4/4 (full credit; medium)
at most 7000	5/4 (optional extra credit; hard)

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

Question 8 (5 points): Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. `ClosestDotSearchAgent` is implemented for you in `searchAgents.py`, but it's missing a key function that finds a path to the closest dot.

Implement the function `findPathToClosestDot` in `searchAgents.py`. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Hint: The quickest way to complete `findPathToClosestDot` is to fill in the `AnyFoodSearchProblem`, which is missing its goal test. Then, solve that problem with an appropriate search function. The solution should be very short!

Your `ClosestDotSearchAgent` won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.