TSP GROUP PROJECT 6

1. Greedy TSP algorithm:

```python
# total time complexity: O(n^3)
# total space: O(n) because our list cities stores n cities
def greedy( self,time_allowance=60.0 ):
    results = {}
    cities = self._scenario.getCities()
    ncities = len(cities)
    bssf = None
    count = 0
    start_time = time.time()

    # O(n) time
    # getPath is called n times and takes O(n^2) times resulting in total complexity of O(n^3)
    for city in cities:
        tempTuple = self.getPath(city,cities)
        tempCost = TSPSolution(tempTuple[0])
        if bssf == None or bssf.cost > tempCost.cost:
            bssf = tempCost
            count = tempTuple[1]

    end_time = time.time()
    results['cost'] = bssf.cost
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = bssf
    results['max'] = None
    results['total'] = None
    results['pruned'] = None

    return results

    pass
```

```python
    # runs in O(n^2) time (see while loop for more information)
    # space complexity: O(n) because unvisited, visited, and routes hold n objects. This results in
    # O(3n) space which simplifies to O(n).
    def getPath(self, startCity, cities):
        currCity = startCity
        unvisitied = []
        for unv in cities:
            unvisitied.append(unv)
        visited = []
        routes = []
        count = 0
        unvisitied.remove(currCity)
        visited.append(currCity)
        routes.append(currCity)

        #this loop runs n times
        while len(unvisitied) != 0:
            #  findClosestCity() runs n times, and each time takes n time.
            currCity = self.findClosestCity(currCity,unvisitied,routes,visited)
            count += 1

        return [routes, count]
```

```python
    # runs in O(n)
    # space complexity: O(n) because unvisited, visited, and routes hold n objects. This results in
    # O(3n) space which simplifies to O(n).
    def findClosestCity(self,currCity,unvisitied,routes,visited):
        if len(unvisitied) == 1:
            minCity = unvisitied[0]
            routes.append(minCity)
            visited.append(minCity)
            unvisitied.remove(minCity)
            return minCity

        #Find closest city
        minCost = 999999999999999
        minCity = None

        #This loop runs n times
        for city in unvisitied:
            if currCity.costTo(city) < minCost:
                minCity = city
                minCost = currCity.costTo(city)

        routes.append(minCity)
        visited.append(minCity)
        if len(unvisitied) == 1:
            unvisitied = []
        else:
            unvisitied.remove(minCity)
        return minCity
```

The greedy functions runs in O(N^3) time and takes N space. This is because of the need to store N cities and to call the function getPath, which takes O(N^2) time, N times.
The getPath algorithm takes O(N^2) time and N space. It uses 3N space to store the routes, cities that have been visited, and the cities that have not been visited. O(3N) when simplified is O(N), getPath takes O(N^20) time because findClosestCity takes N time and is called N time.

The findClosestCity uses 3N space to store the routes, cities that have been visited, and the cities that have not been visited. O(3N) when simplified is O(N). It only takes N time because of the singular for loop in the function.

2. Fancy algorithm implementation

```
# total time complexity: O(2n^2) which simplifies down to O(n^2)
# space complexity: O(2n^2) which simplifies down to O(n^2)
def fancy( self,time_allowance=60.0 ):
    results = {}
    cities = self._scenario.getCities()
    bssf = None
    count = 0
    start_time = time.time()

    #O(n) space (doesn't affect overall space)
    distances = []

    # this nested for loop runs in O(n^2) time
    for city in cities:
        tempDist = []
        for city2 in cities:
            if city == city2:
                tempDist.append(math.inf)
                continue
            tempDist.append(city.costTo(city2))
        distances.append(tempDist)

    # DPTSP is O(n^2) time and space
    optimalPath, optimalCost = self.DPTSP(distances)

    routes = []
    for city in optimalPath:
        routes.append(cities[city])

    bssf = TSPSolution(routes)

    end_time = time.time()
    results['cost'] = optimalCost
    results['time'] = end_time - start_time
    results['count'] = 5
    results['soln'] = bssf
    results['max'] = None
    results['total'] = None
    results['pruned'] = None
    return results
# total time complexity: O(2n^2) which simplifies down to O(n^2)
# space complexity: O(2n^2) which simplifies down to O(n^2)
def DPTSP(self,distances):
    n = len(distances)
    totalCities = set(range(n))

    # creating an nxn table below so space is O(n^2)
    dpTable = {(tuple([i]), i): tuple([0, None]) for i in range(n)}
    queue = [(tuple([i]), i) for i in range(n)]

    # for loop below runs O(n) times inside our while loop which runs O(n) times as well. Thus we
    # have a total time complexity of O(n^2)
    while queue: # Iterate through untile queue is empty
        prevVisited, prevLastPoint = queue.pop(0)
        prevDist, _ = dpTable[(prevVisited, prevLastPoint)]
        toVisit = totalCities.difference(set(prevVisited))
        for newLastPoint in toVisit:
            newVisited = tuple(sorted(list(prevVisited) + [newLastPoint]))
            newDist = (prevDist + distances[prevLastPoint][newLastPoint])
            if (newVisited, newLastPoint) not in dpTable:
                dpTable[(newVisited, newLastPoint)] = (newDist, prevLastPoint)
                queue += [(newVisited, newLastPoint)]
            else:
                if newDist < dpTable[(newVisited, newLastPoint)][0]:
                    dpTable[(newVisited, newLastPoint)] = (newDist, prevLastPoint)

    # retracing optimal path also costs O(n^2) time and space
    optimalPath, optimalCost = self.retracingOptimalPath(dpTable, n)
    return optimalPath, optimalCost
```

```python
# time complexity: O(n^2) because we iterate through the entire table, an nxn matrix
# space complexity: O(n^2) because we are storing an nxn table
def retracingOptimalPath(self, dpTable, n):

    # for loop below runs in O(n) time
    citiesToRetrace = tuple(range(n))
    fullPath = dict((k,v) for k,v in dpTable.items()
                if k[0] == citiesToRetrace)
    pathKey = min(fullPath.keys(), key=lambda x : fullPath[x][0])

    lastCity = pathKey[1]
    optimalCost, nextToLastCity = dpTable[pathKey]
    optimalPath = [lastCity]

    citiesToRetrace = tuple(sorted(set(citiesToRetrace).difference({lastCity})))

    # this while loop is O(n^2) because it goes through every element in our dynamic
    #programming table
    while nextToLastCity is not None:

        lastCity = nextToLastCity
        pathKey = (citiesToRetrace,lastCity)
        _, nextToLastCity = dpTable[pathKey]

        optimalPath = [lastCity] + optimalPath
        citiesToRetrace = tuple(sorted(set(citiesToRetrace).difference({lastCity})))

    return optimalPath, optimalCost
```

Complexity:
The whole dynamic programming table we implemented takes O(n^2) time and space. This is because it implements a 2d array table, which is O(n^2) space, and includes two functions called retracingOptimalPath() and DPTSP() which both run in O(n^2) time.

Reference:
We used in part a dynamic programming approach as shown in the following link:
https://towardsdatascience.com/solving-tsp-using-dynamic-programming-2c77da86610d
https://github.com/DalyaG/CodeSnippetsForPosterity/blob/master/SolvingTSPUsingDynamicProgramming/SolvingTSPUsingDynamicProgramming.ipynb

Personal addition:
We optimized and adapted this code for our program by adapting it to our given TSP interface and providing it with a correct data table. The original solution given ran in O(n^2) time, so we made sure that our additions did not exceed O(n^2) time.

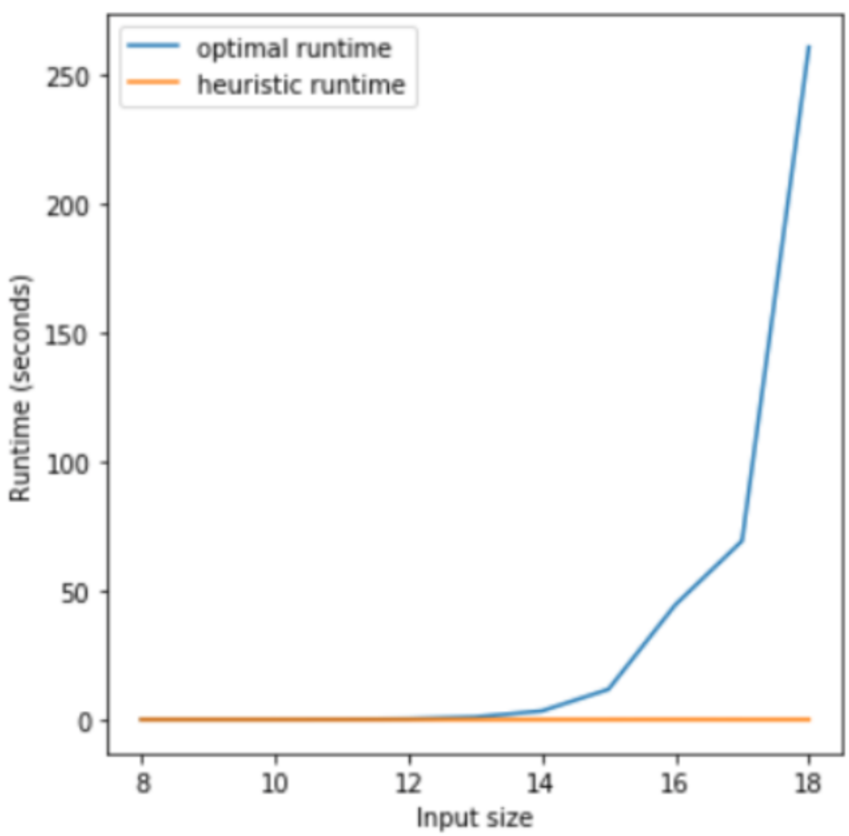
Overview:
The fancy algorithm we implemented uses a dynamic programming table to iterate through the possible paths our TSP could take, and then traces and returns the most optimal path found.

3. Programming table

	A	B	C	D	E	F	G	H	I	J	K	L
1		Random		Greedy			Branch and Bound			Our Algorithm		
2	# of Cities	Time (sec)	Path Length	Time (sec)	Path length	% of Random	Time (sec)	Path Length	% of Greedy	Time (sec)	Path Length	% of Greedy
3	15	0.001803	20468	0.003438	9926	48.50%	4.092465	9318	93.80%	4.17353	7217	72.70%
4	30	0.060042	39393	0.018567	18389	46.70%	60.00021	13580	73.80%	TB		
5	60	57.671891	83380	0.119673	26276	31.50%	60	26276	100%	TB		
6	100	60	inf	0.4737	33170	N/A	60	33170	100%	TB		
7	200	60	inf	TB			TB			TB		
8												
9	10	0.000281	15729	0.00073	8022	51%	0.042821	6952	86.60%	0.027633	5923	73.80%
10	17	0.001636	15214	0.005132	10458	68.70%	28.518594	9871	94.30%	51.1785	8592	82.15%
11												
12												

We implemented a few sizes of our own at city size 10 and 17 since our algorithm becomes inefficient after size 18. At larger sizes, the greedy and branch and bound algorithms seem to be more practical than the dynamic programming approach.

4. Graph



The blue line in the graph above shows our fancy algorithm's predicted runtime to input size ratio. It is a very effective algorithm until we reach about input size 18. This graph was taken from the website that our dynamic programming approach was referenced from.

We realized too late that our dynamic programming approach is ineffective for large input sizes, however it is very efficient at finding very optimal paths with smaller input sizes due to the need to create a dynamic programming table, the size of which increases exponentially. We also then have to back trace to find our path.