1. **See appendix**
2. Complexities
    a. Greedy Algorithm
        i. Time
            1. Basically O(0). Just kidding. It's $O(n^2)$. This is because it goes through all edges and finds the smallest edge. Then it repeats this until it has a route of length n, so it has to do it n times. It does O(1) work at each iteration, so that oesn't add anything. Technically it's a little better than $O(n^2)$ because each time it finds the shortest edge, it removes it from the next iteration so it won't be considered again. So the first loop is n edges, the second is n-1, then n-2 and so on. But close enough to $O(n^2)$
        ii. Space
            1. Space complexity is O(n). This is because it holds 2 arrays, that added together always have n elements. Because this algorithm is almost just sorting by edge length, it's essentially the same as just moving edges one at a time into a separate array and removing them from the first as you do so. Thus, O(n) space
    b. Priority Queue
        i. Time
            1. All operations are O(1). This is because my queue push() just compares the item being added to the top item, and then either places it at the beginning of the queue or the end of the queue. Pop() also just grabs the top item, which is O(1).
        ii. Space
            1. Space complexity can actually get pretty bad. WORST case scenario, $O(n!*n^2)$. This is because each city can lead to a max of n - 1 other cities, and each of those can lead to a max of n - 2 other cities, and so on. So assuming no states were pruned, the queue could hypothetically hold n! states. The really unfortunate thing is that each state has a matrix of size n^2 inside it, so the total complexity is $n!*n^2$
    c. Reduced Cost Matrix, including updating it
        i. Time
            1. Reduced cost matrix is pretty hefty. $O(n^2)$ Updating it is quite the procedure. First, you have to reduce the rows, then reduce the columns. They have the same complexity, so we'll think about them together as just 2 * whatever the complexity of one of them is. In order to reduce the rows,

you have to go down the row once and find the lowest number, then iterate over all the values again and subtract that value from every entry. So it’s 2n*n. Do it once for the rows and once for the columns and we’ve got O(n^2)

ii. Space

1. Each reduced cost matrix has a total complexity of O(n^2). Super simple, it’s a 2D array of size n^2, and each cell contains an integer. O(n^2)

d. BSSF Initialization

i. Time

1. See **Greedy Algorithm**, I used it for my BSSF

ii. Space

1. See **Greedy Algorithm**, I used it for my BSSF

e. Expanding one Search State into its children

i. Time

1. O(n^3) for this one unfortunately. Better than exponential though! To start off with, we iterate over all edges that the current state can lead to, which can be up to n - 1. So n to begin with. There are 3 other significant sections of code inside expand.

   a. First, we have to make a copy of the parent matrix. copy.deepcopy takes too long and seriously affected my overall time complexity, so I created a custom function to copy matrices so that you get a unique object disconnected from the object it’s copying, but only takes O(n) rather than the O(n^2) that deepcopy can take. Total complexity of expand is now O(n^2)

   b. Next, we have to set the entire row and entire column associated with the edge we’re considering to math.inf. We can do this in O(n) by doing *rcMatrix[fromIndex][i] = math.inf* and *rcMatrix[i][toIndex] = math.inf* for i in range(n). Total complexity of expand is still O(n^2)

   c. Last, we need to reduce the rows and columns. Unfortunately, as I’ve already discussed above, the time complexity of this section is O(n^2). This means the Total complexity of expand is now O(n^3)

ii. Space

1. O(n^3) as well I’m afraid. As stated before, we loop through the children of the current state, which can be up to n-1. For

each of those, we have to create an n*n matrix, which will mean our total complexity is n*n*n, or $O(n^3)$

f. The full Branch and Bound algorithm. You should be very exact on the complexities above. Time and Space complexity for the full branch and bound is harder to specify exactly but give your best effort to explain and discuss it.
    i. Time
        1. Well, to initialize the BSSF, it will take $O(n^2)$, and we'll need to do a certain number of explores. Assuming the worst case scenario as big O is supposed to be, if we could eliminate none of the paths, then the algorithm could take $O(n!*n^3 + n^2)$ since for each state we would have to expand each state, and up to n! states could be created. This is an absolutely worst case scenario and not at all realistic. I'm not sure what the more average case would be since I'm not exactly sure how to quantify how many states get pruned, or how that affects how many states get created in general. I would guess that it's a little closer to $O((n/2)! *n^3)$ since a large amount of states get trimmed
    ii. Space
        1. I'm a little foggy on state as well. It's hard to correlate number of states at any one time with the number of cities considered. But however many states that ends up being, it WILL be n^2 times the max number of states at any one time. This is because the only thing taking up space is the matrix inside of each of the states. So if you know the number of states, you know the number of rcMatrices, and therefore the total space complexity!

3. State data structure
    a. For my state data structure, I created a class with 5 attributes
        i. fromIndex: an integer represents the city directly before it in the route. Used in various calculations and exclusions in the rcMatrix
        ii. rcMatrix: a 2d array that contains the current reduced cost matrix at this state. The most important and most computationally expensive piece of the puzzle
        iii. LB: the lower bound at that state. The cost of the current route
        iv. depth: essentially the length of the route. Used as the primary method of determining which state to pop off the priority queue next. Could have used route length, but didn't want to add more time complexity in retrieving that

v. route: the current list of cities that have lead to the current city in the state

4. Priority queue
    a. I chose to create my own instead of using a different implementation! It's like halfway between a priority queue and a sorted list. When you push an item, the queue will compare the item's depth to the depth of the top item (if it exists), using lower LB to break ties. If the item being added is better than the top item, then it's added to the front of the array, otherwise, it's added to the back of the array. Popping the queue will pull the top item off the array.
    b. I might have made this queue better by allowing push() to search down, say, 25% of the array before placing the city at the end of the array, instead of only keeping the best at the top. For example, if I put the second best item at the bottom of a 500 item stack, and the best item ends up not working out, then I have to go through 500 items before I get to the next best item. That's a lot of wasted time
5. BSSF approach
    a. I chose to use the greedy algorithm result as my initial BSSF value. Not very creative, I'll give you that, but highly effective! Doing this gives a relatively accurate value for the shortest path, but always on the high side. Because of this, we explore less states because states that lead to long paths are trimmed sooner
6.

# Cities	Seed	Running time **Hard**	Cost of best **Hard** tour found (* = optimal)	Max # of stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned	Running time **Greedy**	Cost of best **Greedy** tour found
15	20	8.135137	*9397	328	4	12610	10831	0	11764
16	902	5.396911	*7988	217	7	9906	8815	0	12251
50	395	60	26479	1206	36	3726	3024	0.002991	26973
43	97	60	19967	1065	7	5411	4529	0.003506	22974
32	867	60	15364	1144	19	14655	12964	0.002991	17098
29	680	60	13848	988	18	14469	12380	0.003098	16556
25	825	17.09562	*10652	215	4	15009	14080	0.001026	15784
13	773	4.51406	*7896	250	11	6020	5110	0	8935
11	5856	0.954108	*8229	120	26	1599	1204	0	9488
10	282	0.247271	*8783	20	3	561	450	0	9651

7. Discuss the results in the table and why you think the numbers are what they are, including how states pruned and time and space complexity vary with problem size.
    a. The numbers just about fit with what you would expect! As you increase the number of cities, the **time** it takes to solve them goes up. It actually goes up quite quickly with each city, which is what you'd expect since the time complexity of solving the problem WITHOUT branch and bound is $n!*n^2$. Each and every extra city starts to take a toll, so even with a pretty good algorithm, it gets big fast.
    b. **Space** complexity operates in much the same sense as time complexity. More cities, more paths to look at, more states to consider at a time. If you

have 4 cities, you could only have max 3 states on the first explore, but if you had 50, then you'd have 49 max, which could each have 48. It gets real big real fast

c. **States pruned** is related to states created, where the more states you make, the more states are going to exceed the BSSF and require trimming. The more cities, the more states, the more trimming!!
d. I don't think the cost should be looked at too closely since I don't know how the edges are calculated and sometimes larger numbers of cities have smaller edges ¯_(ツ)_/¯
e. I'm honestly not seeing exactly how the # of BSSF updates relates to the size of problem though, they seem to be fairly sporadic
f. Also, greedy is just a powerhouse, time is almost always 0. Even if you have it solve 1500 cities, it only takes 4 seconds. NOT TO MENTION, the number it comes up isn't even that far off from the real thing! Definitely a move for larger problem sizes

8. Mechanisms tried
    a. Initially, I had a poor priority queue setup that took longer than it needed to. I iterated over the entire queue array every time I popped, and found the lowest value, and did all the comparisons there. Implementing my other priority queue dramatically sped up the search.
    b. I also tried using the lower bound as the main priority key, using depth as a tiebreaker. This led to longer times typically, because rather than diving to the end of routes, it would explore more like a BFS, going wide, and finding the best option.
    c. Prioritizing depth first was the most effective way I thought of to search deep instead of going wide, and once I went deep, finding low LBs was easy

Appendix (Code)

```
#!/usr/bin/python3
import copy
import math

import TSPClasses
from which_pyqt import PYQT_VER

if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF
```

```
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF
elif PYQT_VER == 'PYQT6':
    from PyQt6.QtCore import QLineF, QPointF
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time
import numpy as np
from TSPClasses import *
import heapq
import itertools


class TSPSolver:
    def __init__(self, gui_view):
        self._scenario = None

    def setupWithScenario(self, scenario):
        self._scenario = scenario

    ''' <summary>
        This is the entry point for the default solver
        which just finds a valid random tour.  Note this could be used to find
your
        initial BSSF.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
solution,
        time spent to find solution, number of permutations tried during search,
the
        solution found, and three null values for fields not used for this
        algorithm</returns>
    '''

    def defaultRandomTour(self, time_allowance=60.0):
        results = {}
        cities = self._scenario.getCities()
        ncities = len(cities)
        foundTour = False
        count = 0
```

```
        bssf = None
        start_time = time.time()
        while not foundTour and time.time() - start_time < time_allowance:
            # create a random permutation
            perm = np.random.permutation(ncities)
            route = []
            # Now build the route using the random permutation
            for i in range(ncities):
                route.append(cities[perm[i]])
            bssf = TSPSolution(route)
            count += 1
            if bssf.cost < np.inf:
                # Found a valid route
                foundTour = True
        end_time = time.time()
        results['cost'] = bssf.cost if foundTour else math.inf
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = None
        results['total'] = None
        results['pruned'] = None
        return results

    ''' <summary>
        This is the entry point for the greedy solver, which you must implement
for
        the group project (but it is probably a good idea to just do it for the
branch-and
        bound project as a way to get your feet wet).  Note this could be used to
find your
        initial BSSF.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
best solution,
        time spent to find best solution, total number of solutions found, the
best
        solution found, and three null values for fields not used for this
        algorithm</returns>
    '''
```

```python
def greedy(self, time_allowance=60.0):
    results = {}
    cities = copy.deepcopy(self._scenario.getCities())
    ncities = len(cities)
    foundTour = False
    count = 1
    bssf = None
    startCityIndex = random.randint(0, ncities)
    currentCity = cities.pop(startCityIndex)
    route = []
    route.append(currentCity)
    start_time = time.time()
    while not foundTour and time.time() - start_time < time_allowance:
        shortestPathIndex = None
        shortestPathLength = math.inf
        for index in range(ncities - len(route)):
            currentCost = currentCity.costTo(cities[index])
            if currentCost < shortestPathLength:
                shortestPathIndex = index
                shortestPathLength = currentCost

        if shortestPathIndex is not None:
            route.append(cities[shortestPathIndex])
            currentCity = cities.pop(shortestPathIndex)
        else:
            break

        bssf = TSPSolution(route)
        if len(route) == ncities and bssf.cost < math.inf:
            foundTour = True

    end_time = time.time()
    results['cost'] = bssf.cost if foundTour else math.inf
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = bssf
    results['max'] = None
    results['total'] = None
    results['pruned'] = None
    return results
```

```
    ''' <summary>
        This is the entry point for the branch-and-bound algorithm that you will
implement
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of
best solution,
        time spent to find best solution, total number solutions found during
search (does
        not include the initial BSSF), the best solution found, and three more
ints:
        max queue size, total number of states created, and number of pruned
states.</returns>
    '''

    def branchAndBound(self, time_allowance=60.0):
        numStatesCreated = 0
        numPruned = 0
        numSolutionsTested = 0
        maxPQSize = 0
        cities = self._scenario.getCities()
        ncities = len(cities)
        foundTour = False
        try:
            bssf = self.greedy()['soln']
        except:
            bssf = self.defaultRandomTour()['soln']
        initialRCMatrix = self.createAndPopulateRCMatrix(cities)
        initialState = State(0, initialRCMatrix,
self.reduceMatrix(initialRCMatrix, 0), 0, [cities[0]])
        pq = PQ()
        pq.push(initialState)
        start_time = time.time()

        while not pq.is_empty() and time.time() - start_time < time_allowance:
            parentState = pq.pop()
            fromIndex = parentState.getFromIndex()
            parentRCMatrix = parentState.getRCMatrix()
            parentDepth = parentState.getDepth()
            parentLB = parentState.getLB()
            parentRoute = parentState.getRoute()
```

```
            #  Expand
            for toIndex in range(len(parentRCMatrix)):
                if toIndex == fromIndex or self.isCityInRoute(parentRoute,
toIndex):
                    continue
                newState = State(toIndex, self.mediumDepthCopy(parentRCMatrix),
parentLB, parentDepth + 1, parentRoute.copy())
                numStatesCreated += 1
                newState.setLB(self.setCrosshairsToInfinity(fromIndex, toIndex,
newState.getRCMatrix(), newState.getLB()))
                newState.setLB(self.reduceMatrix(newState.getRCMatrix(),
newState.getLB()))
                newState.appendToRoute(cities[toIndex])

                if newState.getLB() < bssf.cost:
                    if newState.getRouteLength() == ncities:
                        bssf = TSPSolution(newState.getRoute())
                        numSolutionsTested += 1
                        foundTour = True
                    else:
                        pq.push(newState)
                else:
                    numPruned += 1

            if pq.get_size() > maxPQSize:
                maxPQSize = pq.get_size()

        end_time = time.time()
        results = {'cost': bssf.cost if foundTour else math.inf, 'time': end_time
- start_time,
                   'count': numSolutionsTested, 'soln': bssf, 'max': maxPQSize,
'total': numStatesCreated, 'pruned': numPruned + pq.get_size()}
        return results

    def createAndPopulateRCMatrix(self, cities):

        numberOfCities = len(cities)
        rcMatrix = [[math.inf for _ in range(numberOfCities)] for _ in
range(numberOfCities)]

        for fromIndex in range(numberOfCities):
```

```python
            fromCity = cities[fromIndex]
            for toIndex in range(numberOfCities):
                toCity = cities[toIndex]
                rcMatrix[fromIndex][toIndex] = fromCity.costTo(toCity)

        return rcMatrix

    def reduceMatrix(self, matrix, LB):
        matrixSize = len(matrix)
        changed = True

        while changed:
            changed = False
            LB, changed = self.reduceMatrixRows(matrix, matrixSize, LB, changed)
            LB, changed = self.reduceMatrixCols(matrix, matrixSize, LB, changed)
        return LB

    def reduceMatrixRows(self, matrix, matrixSize, LB, changed):
        for row in range(matrixSize):
            lowestVal = math.inf
            for col in range(matrixSize):
                currentVal = matrix[row][col]
                if currentVal <= lowestVal and currentVal != -1:
                    lowestVal = currentVal

            if lowestVal > 0 and lowestVal != math.inf:
                LB += lowestVal
                changed = True
                # Reduce all values in that row
                for col in range(matrixSize):
                    if matrix[row][col] != -1:
                        matrix[row][col] = matrix[row][col] - lowestVal
        return LB, changed

    def reduceMatrixCols(self, matrix, matrixSize, LB, changed):

        for col in range(matrixSize):
            lowestVal = math.inf
            for row in range(matrixSize):
                currentVal = matrix[row][col]
```

```python
                if currentVal <= lowestVal and currentVal != -1:
                    lowestVal = currentVal

            if lowestVal > 0 and lowestVal != math.inf:
                LB += lowestVal
                changed = True
                # Reduce all values in that row
                for row in range(matrixSize):
                    if matrix[row][col] != -1:
                        matrix[row][col] = matrix[row][col] - lowestVal
        return LB, changed

    def setCrosshairsToInfinity(self, fromIndex, toIndex, rcMatrix, LB):
        LB += rcMatrix[fromIndex][toIndex]

        #  Set row and column in crosshair pattern to visited
        for i in range(len(rcMatrix)):
            rcMatrix[fromIndex][i] = math.inf
            rcMatrix[i][toIndex] = math.inf

        return LB

    def isCityInRoute(self, route, index):
        for city in route:
            if city.getIndex() == index:
                return True

        return False

    #  copy.deepcopy() is super slow, and was making my time complexity a lot worse,
    #  so this function copies matrices without taking five-ever
    def mediumDepthCopy(self, matrix):
        copiedMatrix = []
        for row in range(len(matrix)):
            copiedMatrix.append(matrix[row].copy())

        return copiedMatrix

    ''' <summary>
```

```
		This is the entry point for the algorithm you'll write for your group project.
		</summary>
		<returns>results dictionary for GUI that contains three ints: cost of best solution,
		time spent to find best solution, total number of solutions found during search, the
		best solution found.  You may use the other three field however you like.
		algorithm</returns>
	'''

	def fancy(self, time_allowance=60.0):
		pass
```

```
#!/usr/bin/python3
import copy
import heapq
import math
import numpy as np
import random


class TSPSolution:
	def __init__(self, listOfCities):
		self.route = listOfCities
		self.cost = self._costOfRoute()

	def _costOfRoute(self):
		cost = 0
		last = self.route[0]
		for city in self.route[1:]:
			cost += last.costTo(city)
			last = city
		cost += self.route[-1].costTo(self.route[0])
		return cost

	def enumerateEdges(self):
		elist = []
		c1 = self.route[0]
		for c2 in self.route[1:]:
```

```python
            dist = c1.costTo(c2)
            if dist == np.inf:
                return None
            elist.append((c1, c2, int(math.ceil(dist))))
            c1 = c2
        dist = self.route[-1].costTo(self.route[0])
        if dist == np.inf:
            return None
        elist.append((self.route[-1], self.route[0], int(math.ceil(dist))))
        return elist


def nameForInt(num):
    if num == 0:
        return ''
    elif num <= 26:
        return chr(ord('A') + num - 1)
    else:
        return nameForInt((num - 1) // 26) + nameForInt((num - 1) % 26 + 1)


class Scenario:
    HARD_MODE_FRACTION_TO_REMOVE = 0.20  # Remove 20% of the edges

    def __init__(self, city_locations, difficulty, rand_seed):
        self._difficulty = difficulty

        if difficulty == "Normal" or difficulty == "Hard":
            self._cities = [City(pt.x(), pt.y(), \
                                 random.uniform(0.0, 1.0) \
                                 ) for pt in city_locations]
        elif difficulty == "Hard (Deterministic)":
            random.seed(rand_seed)
            self._cities = [City(pt.x(), pt.y(), \
                                 random.uniform(0.0, 1.0) \
                                 ) for pt in city_locations]
        else:
            self._cities = [City(pt.x(), pt.y()) for pt in city_locations]

        num = 0
```

```
        for city in self._cities:
            city.setScenario(self)
            city.setIndexAndName(num, nameForInt(num + 1))
            num += 1

        # Assume all edges exists except self-edges
        ncities = len(self._cities)
        self._edge_exists = (np.ones((ncities, ncities)) -
np.diag(np.ones((ncities)))) > 0

        if difficulty == "Hard":
            self.thinEdges()
        elif difficulty == "Hard (Deterministic)":
            self.thinEdges(deterministic=True)

    def getCities(self):
        return self._cities

    def randperm(self, n):
        perm = np.arange(n)
        for i in range(n):
            randind = random.randint(i, n - 1)
            save = perm[i]
            perm[i] = perm[randind]
            perm[randind] = save
        return perm

    def thinEdges(self, deterministic=False):
        ncities = len(self._cities)
        edge_count = ncities * (ncities - 1)  # can't have self-edge
        num_to_remove = np.floor(self.HARD_MODE_FRACTION_TO_REMOVE * edge_count)

        can_delete = self._edge_exists.copy()

        # Set aside a route to ensure at least one tour exists
        route_keep = np.random.permutation(ncities)
        if deterministic:
            route_keep = self.randperm(ncities)
        for i in range(ncities):
            can_delete[route_keep[i], route_keep[(i + 1) % ncities]] = False
```

```
      # Now remove edges until
      while num_to_remove > 0:
         if deterministic:
            src = random.randint(0, ncities - 1)
            dst = random.randint(0, ncities - 1)
         else:
            src = np.random.randint(ncities)
            dst = np.random.randint(ncities)
         if self._edge_exists[src, dst] and can_delete[src, dst]:
            self._edge_exists[src, dst] = False
            num_to_remove -= 1


class City:
   def __init__(self, x, y, elevation=0.0):
      self._x = x
      self._y = y
      self._elevation = elevation
      self._scenario = None
      self._index = -1
      self._name = None

   def setIndexAndName(self, index, name):
      self._index = index
      self._name = name

   def setScenario(self, scenario):
      self._scenario = scenario

   def getIndex(self):
      return self._index

   ''' <summary>
      How much does it cost to get from this city to the destination?
      Note that this is an asymmetric cost function.

      In advanced mode, it returns infinity when there is no connection.
      </summary> '''
   MAP_SCALE = 1000.0
```

```
    def costTo(self, other_city):

        assert (type(other_city) == City)

        # In hard mode, remove edges; this slows down the calculation...
        # Use this in all difficulties, it ensures INF for self-edge
        if not self._scenario._edge_exists[self._index, other_city._index]:
            return np.inf

        # Euclidean Distance
        cost = math.sqrt((other_city._x - self._x) ** 2 +
                         (other_city._y - self._y) ** 2)

        # For Medium and Hard modes, add in an asymmetric cost (in easy mode it
is zero).
        if not self._scenario._difficulty == 'Easy':
            cost += (other_city._elevation - self._elevation)
            if cost < 0.0:
                cost = 0.0

        return int(math.ceil(cost * self.MAP_SCALE))


class State:
    def __init__(self, fromIndex, rcMatrix, LB, depth, route):
        self._fromIndex = fromIndex
        self._rcMatrix = rcMatrix
        self._LB = LB
        self._depth = depth
        self._route = route

    def __lt__(self, other):
        if self._depth > other.getDepth():
            return True
        elif self._depth == other.getDepth():
            return self._LB < other.getLB()
        return False

    def getFromIndex(self):
```

```python
        return self._fromIndex

    def getDepth(self):
        return self._depth

    def getRCMatrix(self):
        return self._rcMatrix

    def getLB(self):
        return self._LB

    def setRCMatrix(self, value):
        self._rcMatrix = value

    def setLB(self, value):
        self._LB = value

    def getRoute(self):
        return self._route

    def getRouteLength(self):
        return len(self._route)

    def appendToRoute(self, item):
        self._route.append(item)


class PQ:
    def __init__(self):
        self._queue = []

    #  "Sorts" by depth, then by lower bound

    def push(self, item):
        if len(self._queue) > 0:
            firstItem = self._queue[0]
            lowestDepth = firstItem.getDepth()
            itemDepth = item.getDepth()

            if itemDepth > lowestDepth:
```

```
                self._queue.insert(0, item)
            elif itemDepth == lowestDepth and item.getLB() < firstItem.getLB():
                self._queue.insert(0, item)
            else:
                self._queue.append(item)
        else:
            self._queue.append(item)


    def pop(self):
        nextBestOption = copy.deepcopy(self._queue[0])
        self._queue.pop(0)
        return nextBestOption

    def get_size(self):
        return len(self._queue)

    def is_empty(self):
        return len(self._queue) == 0
```