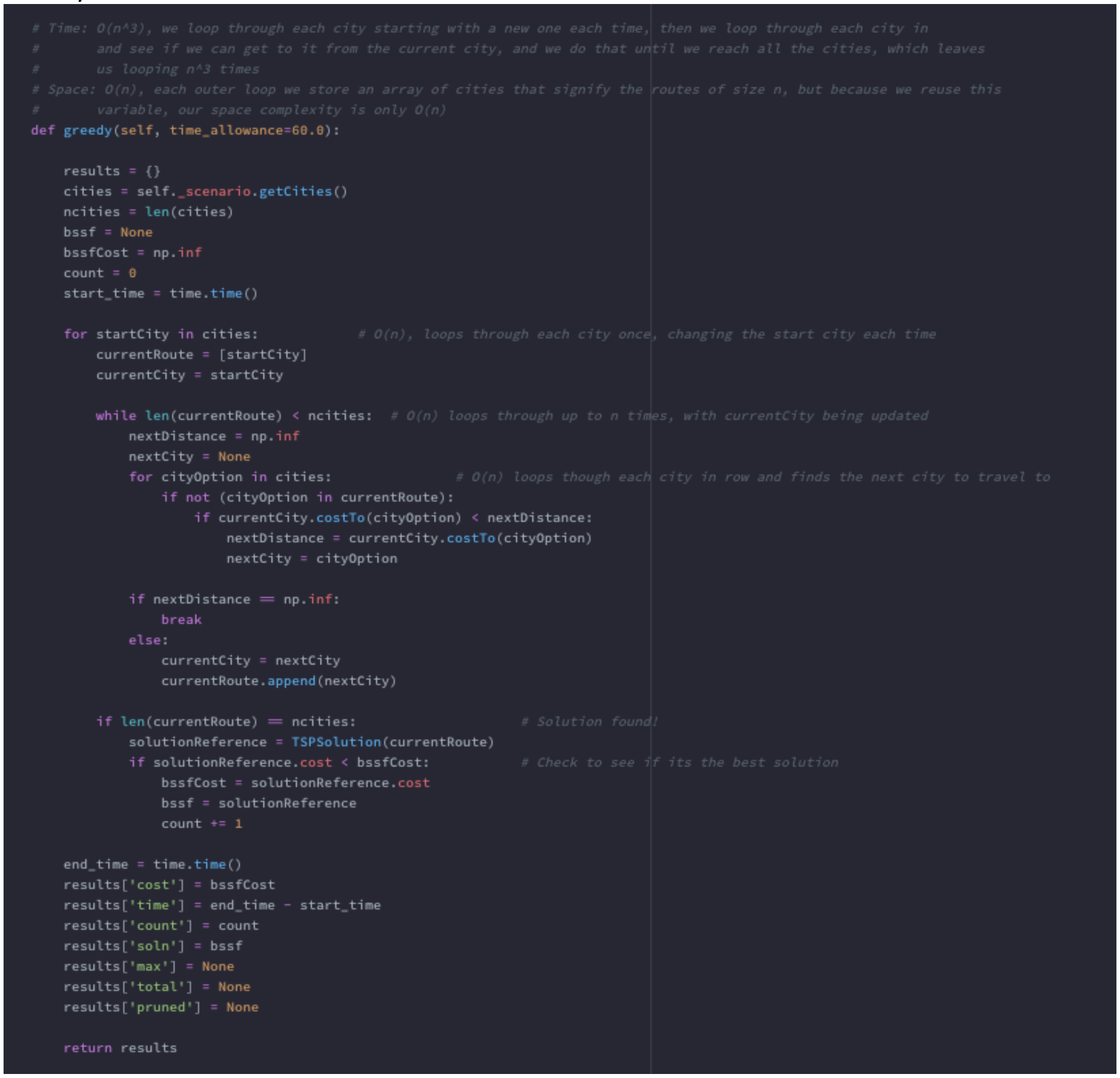
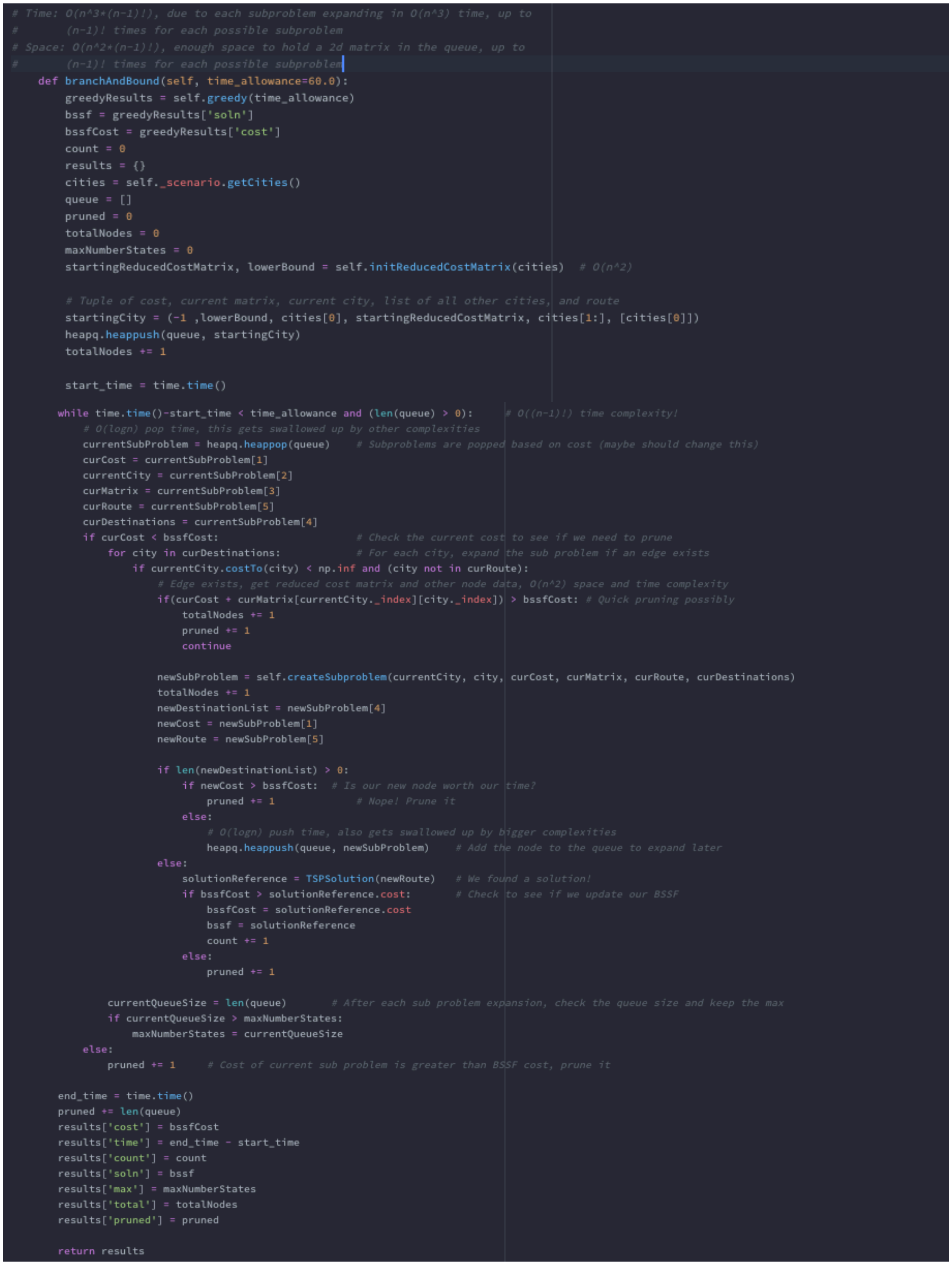
Code:

Greedy:



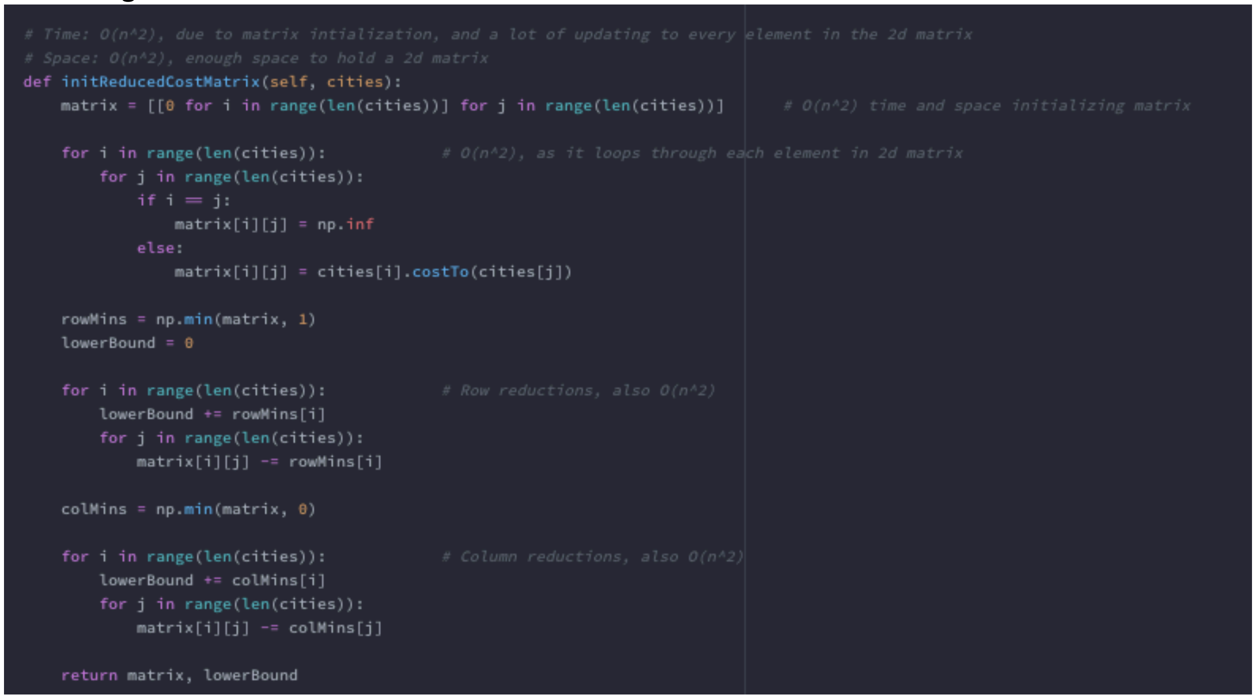
```python
# Time: O(n^3), we loop through each city starting with a new one each time, then we loop through each city in
#       and see if we can get to it from the current city, and we do that until we reach all the cities, which leaves
#       us looping n^3 times
# Space: O(n), each outer loop we store an array of cities that signify the routes of size n, but because we reuse this
#       variable, our space complexity is only O(n)
def greedy(self, time_allowance=60.0):

    results = {}
    cities = self._scenario.getCities()
    ncities = len(cities)
    bssf = None
    bssfCost = np.inf
    count = 0
    start_time = time.time()

    for startCity in cities:            # O(n), loops through each city once, changing the start city each time
        currentRoute = [startCity]
        currentCity = startCity

        while len(currentRoute) < ncities:  # O(n) loops through up to n times, with currentCity being updated
            nextDistance = np.inf
            nextCity = None
            for cityOption in cities:               # O(n) loops though each city in row and finds the next city to travel to
                if not (cityOption in currentRoute):
                    if currentCity.costTo(cityOption) < nextDistance:
                        nextDistance = currentCity.costTo(cityOption)
                        nextCity = cityOption

            if nextDistance == np.inf:
                break
            else:
                currentCity = nextCity
                currentRoute.append(nextCity)

        if len(currentRoute) == ncities:                    # Solution found!
            solutionReference = TSPSolution(currentRoute)
            if solutionReference.cost < bssfCost:           # Check to see if its the best solution
                bssfCost = solutionReference.cost
                bssf = solutionReference
                count += 1

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

    return results
```

Branch and Bound:



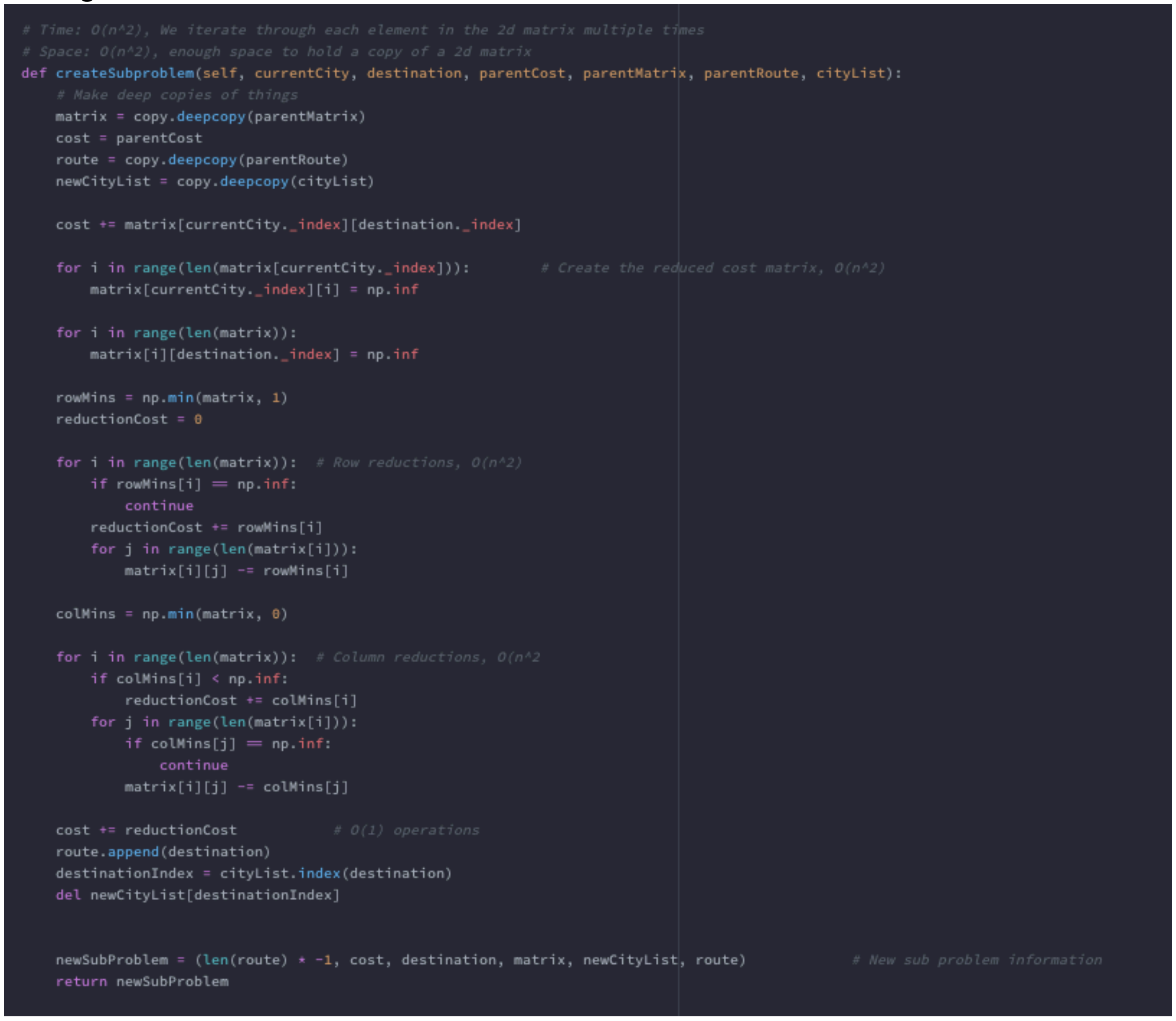
```python
# Time: O(n^3*(n-1)!), due to each subproblem expanding in O(n^3) time, up to
#       (n-1)! times for each possible subproblem
# Space: O(n^2*(n-1)!), enough space to hold a 2d matrix in the queue, up to
#       (n-1)! times for each possible subproblem
    def branchAndBound(self, time_allowance=60.0):
        greedyResults = self.greedy(time_allowance)
        bssf = greedyResults['soln']
        bssfCost = greedyResults['cost']
        count = 0
        results = {}
        cities = self._scenario.getCities()
        queue = []
        pruned = 0
        totalNodes = 0
        maxNumberStates = 0
        startingReducedCostMatrix, lowerBound = self.initReducedCostMatrix(cities)  # O(n^2)

        # Tuple of cost, current matrix, current city, list of all other cities, and route
        startingCity = (-1 ,lowerBound, cities[0], startingReducedCostMatrix, cities[1:], [cities[0]])
        heapq.heappush(queue, startingCity)
        totalNodes += 1

        start_time = time.time()

        while time.time()-start_time < time_allowance and (len(queue) > 0):     # O((n-1)!) time complexity!
            # O(logn) pop time, this gets swallowed up by other complexities
            currentSubProblem = heapq.heappop(queue)    # Subproblems are popped based on cost (maybe should change this)
            curCost = currentSubProblem[1]
            currentCity = currentSubProblem[2]
            curMatrix = currentSubProblem[3]
            curRoute = currentSubProblem[5]
            curDestinations = currentSubProblem[4]
            if curCost < bssfCost:                      # Check the current cost to see if we need to prune
                for city in curDestinations:            # For each city, expand the sub problem if an edge exists
                    if currentCity.costTo(city) < np.inf and (city not in curRoute):
                        # Edge exists, get reduced cost matrix and other node data, O(n^2) space and time complexity
                        if(curCost + curMatrix[currentCity._index][city._index]) > bssfCost: # Quick pruning possibly
                            totalNodes += 1
                            pruned += 1
                            continue

                        newSubProblem = self.createSubproblem(currentCity, city, curCost, curMatrix, curRoute, curDestinations)
                        totalNodes += 1
                        newDestinationList = newSubProblem[4]
                        newCost = newSubProblem[1]
                        newRoute = newSubProblem[5]

                        if len(newDestinationList) > 0:
                            if newCost > bssfCost:  # Is our new node worth our time?
                                pruned += 1             # Nope! Prune it
                            else:
                                # O(logn) push time, also gets swallowed up by bigger complexities
                                heapq.heappush(queue, newSubProblem)    # Add the node to the queue to expand later
                        else:
                            solutionReference = TSPSolution(newRoute)   # We found a solution!
                            if bssfCost > solutionReference.cost:       # Check to see if we update our BSSF
                                bssfCost = solutionReference.cost
                                bssf = solutionReference
                                count += 1
                            else:
                                pruned += 1

                currentQueueSize = len(queue)       # After each sub problem expansion, check the queue size and keep the max
                if currentQueueSize > maxNumberStates:
                    maxNumberStates = currentQueueSize
            else:
                pruned += 1     # Cost of current sub problem is greater than BSSF cost, prune it

        end_time = time.time()
        pruned += len(queue)
        results['cost'] = bssfCost
        results['time'] = end_time - start_time
        results['count'] = count
        results['soln'] = bssf
        results['max'] = maxNumberStates
        results['total'] = totalNodes
        results['pruned'] = pruned

        return results
```

Initializing initial matrix:



```
# Time: O(n^2), due to matrix intialization, and a lot of updating to every element in the 2d matrix
# Space: O(n^2), enough space to hold a 2d matrix
def initReducedCostMatrix(self, cities):
    matrix = [[0 for i in range(len(cities))] for j in range(len(cities))]      # O(n^2) time and space initializing matrix

    for i in range(len(cities)):            # O(n^2), as it loops through each element in 2d matrix
        for j in range(len(cities)):
            if i == j:
                matrix[i][j] = np.inf
            else:
                matrix[i][j] = cities[i].costTo(cities[j])

    rowMins = np.min(matrix, 1)
    lowerBound = 0

    for i in range(len(cities)):            # Row reductions, also O(n^2)
        lowerBound += rowMins[i]
        for j in range(len(cities)):
            matrix[i][j] -= rowMins[i]

    colMins = np.min(matrix, 0)

    for i in range(len(cities)):            # Column reductions, also O(n^2)
        lowerBound += colMins[i]
        for j in range(len(cities)):
            matrix[i][j] -= colMins[j]

    return matrix, lowerBound
```

Making a new state:



```python
# Time: O(n^2), We iterate through each element in the 2d matrix multiple times
# Space: O(n^2), enough space to hold a copy of a 2d matrix
def createSubproblem(self, currentCity, destination, parentCost, parentMatrix, parentRoute, cityList):
    # Make deep copies of things
    matrix = copy.deepcopy(parentMatrix)
    cost = parentCost
    route = copy.deepcopy(parentRoute)
    newCityList = copy.deepcopy(cityList)

    cost += matrix[currentCity._index][destination._index]

    for i in range(len(matrix[currentCity._index])):        # Create the reduced cost matrix, O(n^2)
        matrix[currentCity._index][i] = np.inf

    for i in range(len(matrix)):
        matrix[i][destination._index] = np.inf

    rowMins = np.min(matrix, 1)
    reductionCost = 0

    for i in range(len(matrix)):  # Row reductions, O(n^2)
        if rowMins[i] == np.inf:
            continue
        reductionCost += rowMins[i]
        for j in range(len(matrix[i])):
            matrix[i][j] -= rowMins[i]

    colMins = np.min(matrix, 0)

    for i in range(len(matrix)):  # Column reductions, O(n^2
        if colMins[i] < np.inf:
            reductionCost += colMins[i]
        for j in range(len(matrix[i])):
            if colMins[j] == np.inf:
                continue
            matrix[i][j] -= colMins[j]

    cost += reductionCost            # O(1) operations
    route.append(destination)
    destinationIndex = cityList.index(destination)
    del newCityList[destinationIndex]


    newSubProblem = (len(route) * -1, cost, destination, matrix, newCityList, route)            # New sub problem information
    return newSubProblem
```

Time and Space Complexity:

Priority Queue:

The priority queue that I used had a time complexity of O(logn) for both it's push and pop functions. Its space complexity is dependent on how many tuples are stored in it. At most, (n-1)! Solutions are possible, and so with each state being O(n^2) space, we could have as bad a complexity as O(n^2 * (n-1)!).

Reduced Cost Matrix:

The time complexity for creating the reduced cost matrix for each subproblem (as well as the initial subproblem) is O(n^2) as we loop through each element in a 2d matrix array of size n^2. Each operation of updating/comparing each element is complexity O(1), so O(1) operations * n^2 times leaves us with a total complexity of O(n^2). Although we alter every element multiple times, they are separate loops of n^2 complexity, so we are left with O(n^2)

The space complexity for creating both the initial and the new reduced cost matrix is O(n^2) as well, simply because in each iteration we are creating a new 2d array matrix of n^2 elements, which leads to O(n^2) space complexity to hold the entire array.

BSSF Initialization:

The time complexity for our BSSF initialization (greedy algorithm) is O(n^3). We loop through each city [O(n)], keeping that city as the starting city. From there, we loop through each route, first checking if the edge exists and then we find the minimum route out of our city O(n). We repeat this for each city, which is n times. This leaves us with two inner for loops, giving us a complexity of O(n) * O(n) * O(n), which simplifies to O(n^3) time complexity.

The space complexity for our BSSF initialization (greedy algorithm) is O(n). The only thing we store in our function is the array of cities, and the array of routes, each having a maximum size of n. Being as they are not stored in a 2d array, our maximum space is around O(2n) which simplifies to O(n). These arrays are populated with at most n elements (there are n cities).

Expanding one Search State:

Expanding one search state takes O(n^2) space and O(n^3) time, because for each state, you need to loop through each city and make a new reduced matrix for each one, costing O(n^2) time and space for each possible path. So, for each state expansion, we loop n times and make a 2d matrix that takes n^2 times, resulting in a O(n^3) time complexity for each state. For each city we have to store that matrix of size n^2, so we have a space complexity of O(n^2)

Full Branch and Bound Algorithm:

The time complexity for my full algorithm is O(n^3 * (n-1)!). Because we loop through every possible state that could be a solution, we could end up with (n-1)! States. Each state takes O(n^3) time to expand, so expanding that many states would take O(n^3*(n-1)!) time. Other things such as time complexity of pushing and popping from the queue do not alter the time complexity when it is as big as what we have.

The space complexity for my algorithm is O(n^2*(n-1)!) because, as stated above, we could have up to (n-1)! States in our queue, and each one holds a 2d array of size n^2, so we could at worst case end up with a queue of size O(n^2*(n-1)!) space complexity.

Data Structure:

The data structure I used to store state information is a simple tuple containing the priority sorting number, current state's cost, its current city, its reduced matrix, the list of cities it still needs to visit, and its current route that it has already taken. This way the priority queue will prioritize the length of its route first, and the cost second, and when we pop the state, we have all the information we need to either expand it or prune it. Expanded states that already have a higher cost than the BSSF are automatically pruned and not added to the queue.

Priority Queue Structure:

The priority queue that I used to hold my states was simply heapq, an imported python library (it was already imported in the project). I used heapq to push and pop my states, and they were sorted by cost. For both functions of this queue I used (push and pop), both ran in O(logn) time. Heapq sorted the tuples I gave containing all the sub problem data first by route length, then if the lengths were the same, it sorted the tuples based on the cost, alphabetically ordered. For example a tuple that is further along in it's route will get a higher priority over one that isn't, and one that is just beginning won't have a high priority.

Initial BSSF:

My initial BSSF solution came from simply running my greedy algorithm on the given input. The greedy algorithm returns solution data, including a BSSF cost and a BSSF route, which becomes my initial BSSF to compare my Branch and Bound solution to, and helps with pruning nodes early on so I can quickly return the optimal solution. The greedy algorithm iterates through each city (as the starting city) and then through each route option, picking the lowest valued edge if that corresponding city is not included in the city's route so far. This way, we will get the minimum route of each edge from each city, which gives us the minimum solution for a greedy implementation. This solution gets updated if we find a different route with a lower cost. This information is returned to my branch and bound algorithm to give us a pretty good solution that we can compare for pruning.

Results Table:

# Cities	seed	Difficulty	Time (greedy)	Tour Distance (greedy)	Time	Cost of best Tour	Max # of Stored States	# of BSSF updates	Total # of states	Total # of pruned states
15	20	Hard	0.00967	14806	6.997	9987*	73	6	7845	6495
16	902	Hard	0.0114	8672	5.1865	7856*	59	3	7663	6685
10	700	Hard	0.0039	10371	0.793	8312*	31	2	841	640
12	554	Hard	0.0059	9255	0.908	8495*	33	1	1784	1445
13	170	Hard	0.00799	10425	37.71	8905*	51	10	37216	29072
18	542	Hard	0.01537	12316	60	10745	103	5	68285	59671
20	295	Hard	0.0228	11468	60	10180	115	3	44547	38919
30	626	Hard	0.08618	21616	60	16057	336	1	10337	8028
40	233	Hard	0.2428	18393	60	16206	592	1	3098	2441
50	557	Hard	0.5956	23855	60	20327	953	6	1926	1543

Results Analysis:

The data shows that the time complexity is generally greater than O(n^3), and although I haven't calculated it, it does show a way bigger number than n^3, such as (n-1)! as I suggested earlier in my analysis. The number of states, the time, and the best tour depends greatly on the seed, and the number of states as well as the number of pruned states greatly rely on the number of cities. As the number of cities increase higher and higher, the number of states does show a decline, mainly due to the limited time that the algorithm has to create the reduction matrices. The time complexity that comes from creating the reduced matrix after each possible state is discovered is a huge factor in my program. With 50 cities we see this, as a 2d matrix of size 50^2 causes the program to run a lot slower between each subproblem discovery. This being said, if given a longer amount of time to run, the program would have created and pruned far more states than any of the previous problems. As the number of states declined, so did the number of pruned states, which makes sense as well. I was surprised at how many total states there were that were considered unpruned, which only means that there were so many parents that had so many different possible states.

It was fascinating to me how quickly my program was prone to time out with just an increase of 2 cities from 16 to 18. One thing I found was that there were more variables that depended on the seed. While the number of cities did affect things like time and the max number of cities stored, I found that the seed affected how quickly my algorithm could come up with best solutions so far, and how quickly and how often they were updated. With the seeds having such a big effect on our data, it is hard to find very consistent findings, and running the same number of cities twice with a different seed is bound to give very different answers.