Complexity:

- Greedy:

    I essentially loop through every point in the edges, and while I am looking at every row, I perform a while loop that checks the each neighbor and sees if I have visited it yet and checks if it the smallest edge cost. If I haven't visited and it is less than the min_cost for that row, I set that edge value as the lowest for that row. I move onto the next neighbor to see if another neighbor has the smallest.

    After checking all the neighbors, I check if it is a dead_end (meaning I didn't get back to the original city). If it's not a dead end, I take the smallest value and add that associated neighbor to our visited set and total all the edges to create our stats object.

    The complexity occurs while we iterate over every row and column. We have a n x n matrix, so our complexity would be Big O = O(n^2). We are literally checking every point in our algorithm with no pruning.

- DFS:

    I have a stack that will track our current path and current visited set. If our current path is the size of edges and its score is better than our current best score, I create and append a stat object and adjust that score to be the new best score.

    Otherwise I will loop through all the cities that have valid paths and add all of those onto the stack. A valid path is one that is not visited and its edge is not infinity. If a child path from the current path is valid, we append that city to the end of the path and add it to the visited set. Once we have a current path that is as long as the size of edges, we do that check that is in the previous paragraph.

    The complexity of the algorithm is that it assumes any of the child paths might be legitimately good. There is no way to check and see if a potential path is valid after just checking one child, we assume all paths are good options.

- Branch and Bound:

    I start by running greedy to get an initial answer. I take the score from the initial answer and set my best score so far (bssf) to be that initial score. I then create a reduced matrix from our edges and calculate the initial lower bound for our algorithm.

    I create an initial path and initial visited set that only contain the index of the first city in the edges matrix (i.e. 0). I then put the initial path and initial visited set on a stack.

I then perform a similar process to DFS, but instead of just adding all the child cities to a the current path and pushing each potential new path onto the stack, we set the row of the current city to infinity and the column for the child to infinity (on a copied matrix) and we then perform another matrix reduction to see if there is any additional edges we can remove. If that additional cost is infinity or the current lower bound plus the current edge on our reduced matrix plus the additional cost is not less than the current bssf , we prune that edge to that child city. Otherwise we add that child city to the path and the visited set.

Once the path is the same size as edges, we do the check as described in DFS. If everything checks out, I add the new stats object with all the relevant information.

The complexity is reduced from DFS because we have started to prune edges from our tree. We no longer will consider paths that will not give us an optimal chance of finding the best answer. The issue remains that we are not ranking those paths that are still just added in order of discovery, not importance.

- Smart Branch and Bound:

For this algorithm, I created a Node class that stores the current path, current visited set, reduced matrix, and lower bound.

I start by running greedy to get an initial answer. I take the score from the initial answer and set my best score so far (bssf) to be that initial score. I then create a reduced matrix from our edges and calculate the initial lower bound for our algorithm.

What makes this algorithm smart is that I replace the stack with a priority queue. Every time we find a valid path for a child (see how a valid path is defined in Branch and Bound), we calculate where the priority of this path should be place in our priority queue. I used a heap queue as my priority, so I wanted to place the highest priority paths at the top of my tree (assuming my tree “grows” down). I did this by multiplying that current node’s lower bound by 10000 and then subtracting that length of the path multiplied by 500. This calculation rewards the low lower bounds while hurting low path lengths. This effective allows us to focus on the paths that are growing with low lower bounds.

I want to reiterate, the process is nearly identical to Branch and Bound, meaning we are still pruning and getting rid of unnecessary paths.

The complexity is better than the regular brand and bound because it allows us to focus on the promising paths, rather than looking at everything on a stack before we find a solution.

Empirical Results:

Seed	N	Random	Greedy	DFS	B&B	Smart B&B
312	10	3.376	inf	3.376	3.376	3.411
1	15	5.277	5.131	5.181	3.98	3.98
2	20	6.968	4.419	7.268	4.419	3.91
3	30	12.091	inf	11.269	10.302	5.271
4	50	26.35	8.715	24.763	8.715	Na
10	25	9.74	5.881	11.71	5.881	Na
200	35	14.882	inf	15.934	15.81	5.46

Discussion:

- Strengths:
  - Random:
    - It got an answer every time on all my tests
    - Does pretty good for low n quantities
  - Greedy:
    - When it found an answer, it found a low quantity for n!
  - DFS:
    - It got an answer every time on all my tests
    - It is pretty is better than random most of the time
  - B&B:
    - It does really well with low n quantities
    - Not bad for big n quantities
    - Always got an answer with my tests
  - B&B Smart:
    - It always finds the best answer available, nothing finds a better answer.

Strengths:

- Random:
  - Typically bad scores
- Greedy:
  - Prone to find inf on tough problems

- DFS:
    - Scores aren’t very impressive
- B&B:
    - Not as good as the smart B&B when they both find an answer
- B&B Smart:
    - Doesn’t always find an answer