Pseudocode:

```
def MST_Prim()
        PQ = priority queue using heap
        MST = graph with G.n vertex
        color[u] = white for all u∈G.V
        Visit(r, color, G, PQ)
        while PQ is not empty:
                <wt, pu→u> = PQ.Extract-min()

                if color[u] == white:
                        Visit(u, color, G, PQ)

                        MST.Addedge(pu, u, wt)
        return MST


def PreorderTraversal(node)
        if node is empty:
                return < >
        R = None
        for each child in node.children:
                R = R + Preorder_Traversal(child)
        return R


def ApproxTSPTour()
        root = Compute MST for G from root s using Prim's Algorithm as MST_Prim(G, w, s)
        H = Preorder_Traversal(root)
        return H


def TSPTour()
        Unvisited = get all permutations of nodes (n-1)! # graph is complete so we can do this
```

For node in unvisited:

For all paths in node:

Calculate the cost of every permutation and keep track of the minimum cost permutation.

Choose lowest cost to another node

Return cost

Analysis:

ApproxTSPTour() generates a minimum spanning tree using Prim's algorithm, and then performs a preorder traversal of the tree to get the optimal path through the matrix. With this path, it can calculate the total cost of the trip. The generation of the MST has a runtime of O(mlogn) while a preorder traversal of the MST has a runtime of O(n). The total runtime of this algorithm is O(mlogn + n).

TSPTour() solves the problem using brute force, therefore it is an NP-hard problem. The algorithm checks every possible combination of routes to provide the most optimal route possible for a given matrix. Since generating the permutations of possible paths will take O(n!) time, and the algorithm will loop every permutation of paths, the running time of this algorithm is O(n!).

Implementation:

See .py file

Test cases/Results:

See .py file for test cases

TEST CASE #	APPROXIMATED COST	COST
1	100	97
2	105	81
3	98	95
4	1007	979
5	267	241