Project 5_TSP Report
Group 4: Emma Paul, Ian Paul, Abdulhalim Bambang

Detailed descriptions of the ideas behind your algorithm:

We originally did an algorithm that didn't use brute force and had a fast time complexity that we thought of ourselves, but after working hard on it and finding the outputs to be insufficient to get a grade that accurately justified the work put in, we scrapped it and moved on to a more brute force algorithm. This original program has been included in the zip file as rough_draft_algorithm.c, as well as an explanation ("Project_5_rough_draft_explanation_write_up.pdf") which explains our theories behind that algorithm, what it does, and why it fell short. Needing to have an algorithm that would produce a sufficient grade, we decided to use a brute force algorithm. However, to make it faster, we implemented a quadrant idea where cities are split up into quadrants and the algorithm only looks at cities within the quadrant instead of all cities. Quadrants are split into SW(q1), NW(q2), NE(q3), SE(q4). This caused our tour length to go down because the distance between the start and end city was reduced by using the quadrants, which forced the tour to travel in a circle. Our brute function is recursively called to find a shortest path in each quadrant. Though the time complexity is O(n!), we are able to reduce the overall time complexity by making it so the search is only on $1/4^{th}$ of the cities at any time. We know that for really big city lists, splitting the problem into fourths won't make it any faster, however, splitting the problem into smaller sub-problems is incorporating the idea behind dynamic programming.

Pseudocode for our algorithm:

1) Put the given information from the txt file into a multidimensional city_list array where 1st column is city id, 2nd is x coordinate, and 3rd is y coordinate (of that city id). 2) Using a for loop from 0 to number of cities, copy city list into an ordered_x_array and ordered_y_array with same values and columns from city_list, as well as a 4th column initialized to 0 for every id to indicate false, that city has not been visited. 3) Sort ordered_x_array using bubble sort, ordering them by the value of the x coordinate (also sort the ordered_y_array in the same manner). 4) Split the x-y plane into 4 quadrants (SW, NW, NE, SE). Using the sorted arrays, go to the middle position (called half_cities) in the array to find the half_x_point and half_y_point values. half_cities = num_cities/2; half_x_point = orered_x_array[half_cities][1]; //find value of x coordinate that is half way (on the x plane of //the graph) Note: same is done for half_y_point

```
for(i = 0; i < num_cities; i++) //from 0 to the number of cities
        if(ordered_x_array[i][1] <= half_x_point ) //if the city's x coordinate is <= to half_x_point
                if(ordered_x_array[i][2] <= half_y_point //and the city's y is <= to half_y_point
                        q1++; //that city is in the first quadrant, so add to q1 count (SW)
                else //else x point is <= to half_x_point and y point is > half_y_point
                        q2++; //so id is in quadrant 2, add to count of q2 (NW)
        else// else x coordinate is greater than half_x_point
                if(ordered_x_array[i][2] <= half_y_point)//if y coord is <= to half_y_point
                        q4++; //id is in q4, add to count (SE)
                else // y coord is > half_y_point
                        q3++; //id is in q3 (NE)
```

5) Dynamically create quad1, quad2, quad3, and quad4 arrays with size based on count of each quadrant (q1, q2, q3, q4). Each quad array is essentially the city list array for each quadrant (1^{st} column is city id, 2^{nd} is x coordinate, and 3^{rd} is y coordinate, 4^{th} is set to 0 as a visited flag – 0 being false). Using the same for loop as above, fill each quad array:

```
for(i = 0; i < num_cities; i++)
        if(ordered_x_array[i][1] <= half_x_point)
                if(ordered_x_array[i][2] <= half_y_point)
                        quad1[q1_spot][0] = ordered_x_array[i][0]; //1st column is id
                        quad1[q1_spot][1] = ordered_x_array[i][1];//2nd is x coordinate
                        quad1[q1_spot][2] = ordered_x_array[i][2];//3rd is y coordinate
                        quad1[q1_spot][3] = ordered_x_array[i][3];//4th is 0 for not visited
                        q1_spot++; //maintain position to appropriately fill quad1
```

//Note: pattern is repeated for filling the rest of the quad arrays using for loop above

6) Brute force function below finds the next city id with the lowest distance from the current city id. When the first distance is calculated in the function, it needs to compare to some distance to see if it is lower, therefore we create a tempDis that is INT_MAX.

7) Call function passing in values: &quad_count is how many cities have been visited in the quadrant so far, &tour_count is the location to place the next city visited in the tour array, tour is the tour array (that holds the final tour), quad1 is the multi-dimensional quadrant 1 array, final_pos_start(id of the current city), q1is size of quad 1 array. If a quadrant only has 1 city, populate tour array and increment tour_count.

/*Initialize values before passing into the function: quad_count = 0 to start which is the count of how many cities the function finds for the tour, which is used to call the function recursively the appropriate amount of times, lpNextCity = quad1[q1-1][0] (this is the city id of the last city in quad1, causing it to start from the right most location in quad1, which essentially is the starting point of the circle that the tour will travel in (going from q1 to q2 to q3 to q4)), tour[0] = lpNextCity which is the first city in the tour, final_pos_start = lpNextCity (start at that id). For quadrant 1, the function is called like so:

```
bruteFunc(&quad_count, &tour_count, tour, quad1, final_pos_start, tempDis, &tourPlace, q1);*/
```

```
void bruteFunc(int *quad_count, int *tour_count, int *tour, int **city_list, int final_pos_start, int tempDis, int *tourPlace, int num_cities)

int local_tour_count = *quad_count;//number of cities left for visiting
int closestCity; //will be the next closest city id the function finds
int local_tour_place = final_pos_start; //current city id we are looking at
int x1, x2, y1, y2; //variables for computing distance
int final_pos_end; //will be the city id that the distance is computed between current city id
//find where the current id's position is within the city list (city list is the quad 1 array)
for(i=0; i<num_cities; i++)
        if(local_tour_place == city_list[i][0])// if the id we are looking at is at the i position in city list
                final_pos_start = i; //position in city list we use for first coordinate to calculate distance

for(i = 0; i < num_cities; i++) //from 0 to the number of cities (in quad1)
        if(city_list[i][3] == 0) //if that city has not yet been visited
                final_pos_end = i; //position in city list we use for the 2nd coordinate to calculate distance
                        x1 = city_list[final_pos_start][1] ; //get x coordinate of "from" city, do same for y coord
                        x2 = city_list[final_pos_end][1]; //get x coordinate of "to" city, so same for y coord

                        distance = ((x2-x1)*(x2-x1)) + ((y2-y1)*(y2-y1));//calculate distance
                        total_length = lrint(sqrt(distance)); //calculate rounded distance

                        if(total_length < tempDis) //if total_length is lowest found distance so far
                                tempDis = total_length; //update tempDis
                                tempCityID = city_list[i][0]; //get the id of that city that is closest

for(i = 0; i < num_cities; i++) //from 0 to num cities in quadrant
        if(city_list[i][0] == tempCityID) //if the id in the list is the id of the closest city
                clostestCity = i; //save position of that id in the array

        set tempDis to INT_MAX to reset to a big value to check against for when the function is called again.

int tour_placer = *tour_count; //this gets the position we need to place the next city we found into tour array
tour[tour_placer] = tempCityID; //populate that position with the city id that has the closest distance
city_list[closestCity][3]; //go to that city id's position in the city_list array and mark as 1 for visited
locat_tour_place++; //increment to go to next position for filling when function is called recursively
local_tour_count++; //increment variable (used to see how many cities have been visited by function)
*tour_count = *tour_count+1; //update position to populate with next city in the tour we find
*quad_count = *quad_count+1; //inc variable (used to see how many cities have been visited by function)

if(local_tour_count < num_cities) //if the function hasn't found a tour for all the cities
        bruteFunc(quad_count, tour_count, tour, city_list, final_pos_start, tempDis, &local_tour_place, num_cities);
        //recursively call function in order to find tour of quad 1 array
```

Note: the function above is called on all 4 quadrants, which fills the final tour array with all the cities in the full city list. Once a tour array is found, we calculate the total length using the distance code shown above.
Finally we output the total length and the city id list of the tour to a txt file [input].txt.tour

Possible improvements we would make:

If we had more time, we would incorporate an edge swap aspect to improve the tour our algorithm finds. Knowing an edge swap would take up a lot of time, we would only do a low number of edge swaps to ensure not adding too much time complexity. However, if we had time to completely redo the assignment, we might simply implement an edge swap algorithm. We avoided it in the first place due to it's theoretical time complexity, however after observing how fast the servers seemed to perform on our brute force algorithm for even 15,000 cities, we believe it would have been sufficient for the competition. Due to the aspect of the completion, or original focus was to creatively come up with an algorithm that had a fast theoretical time complexity. Given that this problem is an NP, and our original algorithm didn't find a sufficient length, we implemented a brute force algorithm knowing it would find a sufficient length. Had we not gone down a path trying to create our own quick algorithm, we probably would have just tried to implement one that has already been created, and proven to be effective. However, that felt lazy, so we created our own which we are proud of (all things considered).