Task 1:

My program supports the following options:

- -n <num>: set the number of cities
- -f <filename>: set the config file used (note: do not set n if using this)
- -s <num>: set the city to be used as a fixed starting point

Note that *-s* optional, and one of *-n* or *-f* must be set, but not both.

By default the specified number of cities is created and each is given random x and y coordinates. By default the program tries to find the shortest route by testing every city as a starting point. However as described above the *-s* option can be used to set a fixed starting point.

On each run that does not use the *-f* option, the city configuration (excluding the fixed starting point) is saved to a binary file called *config.bin*. I have included details of the file format at the end of this report. This config can then be reloaded using the *-f* option and specifying the file. As the fixed starting point is not part of the saved configuration, the same configuration can be run many times with different fixed starting points.

Task 2:

Because of the long running time need for even small values of *n* I only tested in the range of 5-11. I tested using only fixed starting points. Graphs for the results can be found in the *Task 2* directory.

With a fixed starting point I found that the time taken grew at approximately a rate of *n!*.
I used the actual running time for n=11 (1.28 seconds) to estimate the times for n=20, 50 and 100. I did by calculating 1.28*(20!/11!) for 20 as an example.

N=	**Fixed starting point estimate (seconds)**
11	1.28 (actual)
20	78015135744 [1.28*(20!/11!)]
50	9.75×10^{56} [1.28*(50!/11!)]
100	2.99×10^{150} [1.28*(100!/11!)]

This clearly shows that the brute force approach rapidly becomes unusable as *n* grows larger. This is true even if the estimate was wrong by several orders of magnitude.

Task 3:

This program supports the following options:

- -f <filename>: set the tsp file to be used.
- -s <num>: set the city to be used as a fixed starting point.

By default the program tries to find the shortest route by testing with every city as a starting point. However as described above the *-s* option can be used to set a fixed starting point.

A greedy nearest neighbour algorithm is used first. The results of using this solver are printed to the screen. Following this a 2opt solver takes the greedy results and tries to improve them. Once done the results are printed to the screen.

My program only works with the EUC_2D data sets - it does not work with the asymmetric data sets. I have included some data sets that it works well with in the *Task 3/data_sets/* directory.

The below table shows the improvement given by the 2opt solver for various data sets.
It shows that the 2opt solver can improve upon the greedy route sometimes, and fail to offer any noticeable improvement at other times. I would recommend that the 2opt solver be used when time permits as it can offer a noticeable improvement.
Times are in microseconds.

Data set	**Greedy route len**	**Greedy time taken**	**2opt route len**	**2opt time taken**	**% improvement**
Att48 (first city is fixed starting point)	~39964	273	32670	1616	22%
Berlin52 (no fixed starting point)	7310	13487	7305	1188	0.0006%
Att532 (first city is fixed starting point)	105495	125101	92573	2799021	14%

As can be seen the total time taken to run the greedy and 2opt solver for 532 cities was less than 3 seconds. This would take an impossibly long time using a brute force approach, but using these solvers it is perfectly acceptable.

The optimal solutions at the TSPLIB site seem to always use city 1 as a starting point. I forced my program to use this as the starting point too, so I could better compare the results. The optimal result can differ quite a lot from the route generated by my program, but I noticed that often several cities in a row are visited in the same order in both my generated route and the optimal route. This suggests to me that my program generates a route that is at least approaching the optimal one.

Task 4:

One thing to note is that using MPI in a brute force solver will be pointless most of the time. As the time taken grows at an *n!* rate the number of nodes needed to reduce the time taken to a reasonable level becomes impractical, no matter what way the work is distributed.

If fixed starting points are not used the greedy nearest neighbour algorithm can be parallelised with MPI quite easily. Each node simply takes a different starting point and works from there. The number of cities will need to be quite large to overcome the overhead involved in using MPI.

If fixed starting points are used then it is harder to divide the greedy algorithm's work between MPI nodes. Now the only part where work can be done in parallel is finding the closest neighbour. However this is a relatively fast operation, so the number of cities would need to be enormous before any benefit is seen. This is because the work is completed so fast serially that the number of cities needs to be large enough to avoid the overhead of communication between MPI nodes.

The 2opt solver can be parallelised using MPI relatively easily. Each node could do one or more iterations of the outer for loop. However with this approach the work will be a bit unevenly divided, as the inner loop uses $k = i + 1$ as a starting point. This means nodes that get a low value of i have more work to do. In a master slave model each node could just request a new value of i to work with as they finish their previous work, but this leads to a lot of communication overhead.

I believe a better approach would be to give a small number of small i values to some nodes, and a larger number of large i values to other nodes at the very start, greatly reducing communication overhead. Consider 10'000 cities and two nodes. The first node could take the first (say) 3'000 iterations of the outer loop, while the second node takes the remaining. While there's a large difference in the number of iterations because the inner loop uses $k = i + 1$ both nodes should be doing an even amount of work. Then at the end each node reports its shortest route and the shortest is chosen. Some experimentation would be needed to find the best ratio of iterations per node vs what range of i values that node is going to get.

Config.bin file format:

The config.bin file is in a binary format, not a text format.
The first 4 bytes is an integer specifying how many cities there is.
Following this there is an array of city structs - one for each city.
These structs are stored as a 4 byte id value, 4 byte float x value, 4 byte float y value, and then an array of 4 byte floats that store the distance from the current city to each other city.