# Bitonic Travelling Salesman

Mathew Clutter Adam Kraus

October 25, 2022

## 1 Problem Overview

The Euclidean Bitonic Traveling Salesman Problem is a simplified version of the traveling salesman problem. The traveling salesman problem asks what the shortest path that visits every node in a graph, and returns to the starting point is. This is an NP-hard problem in it's pure form. By only investigating bitonic paths, (paths where the route must strictly go from the leftmost point, to the rightmost point, and back to the leftmost point). Using this assumption, it becomes possible to solve this problem in a more tractable time.

## 2 Building and Running Solutions

The code to solve this problem was written in C++, and can be built using g++ with the standard g++ -o ¡outputFile¿ ¡sourceFile.cpp¿. To build, each solution (brute force and dynamic programming) only requires itself, there are no other header files to link and compile with. To run, simply execute the output file generated by g++ with ./¡outputFile¿. The time measurements were obtained through the Linux "time" command, from the "real" time output.

## 3 Brute Force Solution

### 3.1 Algorithm Explanation

The brute force solution works by:

1. Generate every permutation of the list of points to visit.
2. Check if the path is bitonic.
3. If it is, calculate the distance of the path.
4. If the path is the shortest path so far, save that distance and the path as the shortest path.
5. After every permutation has been checked, the shortest path and distance remain.

In order to generate every permutation of the list of points, the next_permutation function builtin to the C++ standard library is utilized. The path is checked to be bitonic by finding the minimum and maximum x values, and checking that the path starts at the minimum x, increases x values until reaching the maximum x value, and then decrease until returning to the starting point. The distance between points is calculated using standard Euclidean distance calculations.

This basic permutation algorithm provides a slow, but correct solution to the Euclidean Bitonic Travelling Salesman problem.

### 3.2 Run-time Analysis

This solution is $\mathcal{O}(n!)$, as it generates every permutation of a list of n points.

A table and graphs of the time for this algorithm to run follow:

| Input Size | Runtime (s) | n! | runtime / n! |
|---|---|---|---|
| 3 | 0.003 | 6 | 0.0005 |
| 4 | 0.003 | 24 | 0.000125 |
| 5 | 0.004 | 120 | 3.33333E-05 |
| 6 | 0.034 | 720 | 4.72222E-05 |
| 7 | 0.047 | 5040 | 9.3254E-06 |
| 8 | 0.56 | 40320 | 1.38889E-05 |
| 9 | 7.02 | 362880 | 1.93452E-05 |
| 10 | 89.755 | 3628800 | 2.47341E-05 |
| 11 | 1122.402 | 39916800 | 2.81185E-05 |
| 12 | 11705.569 | 479001600 | 2.44374E-05 |

Table 1: Brute Force Runtime Table

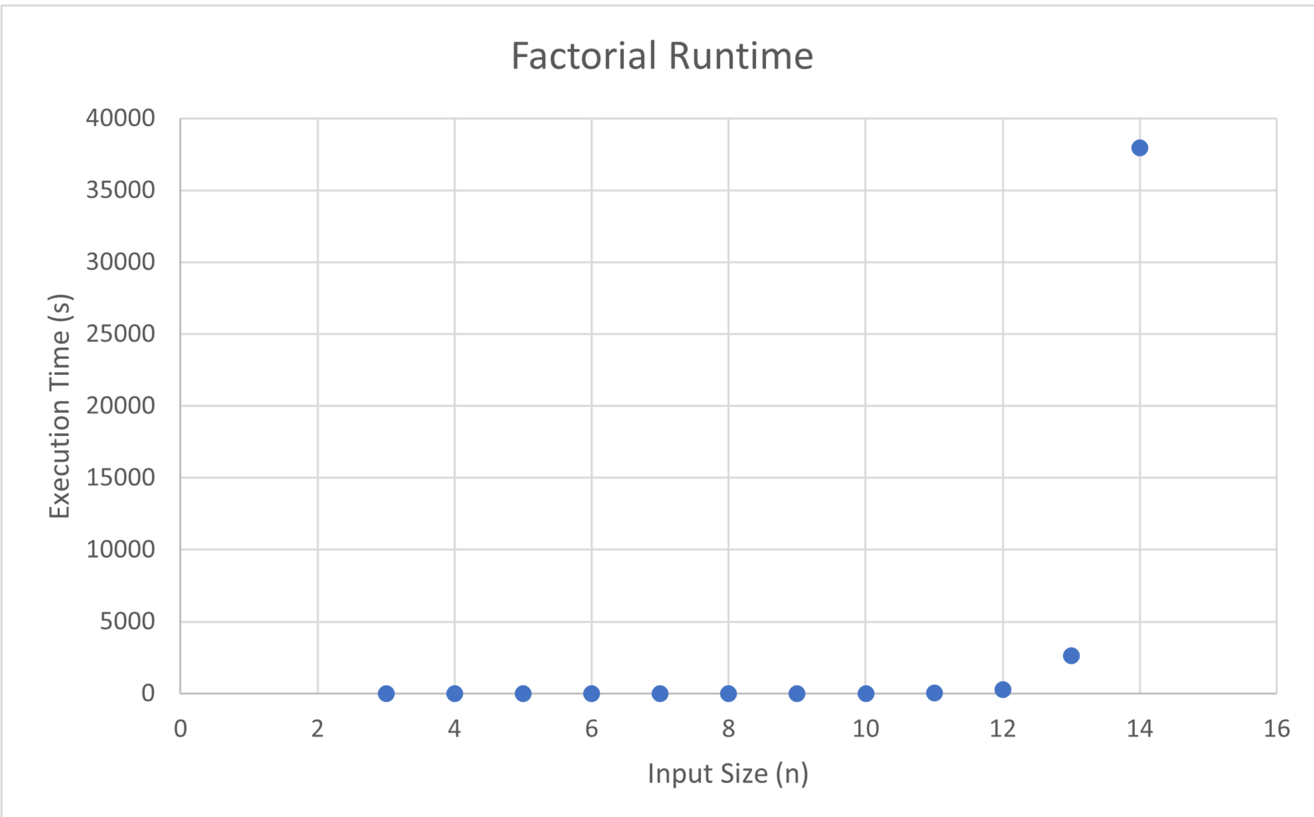


Figure 1: Brute Force Runtime

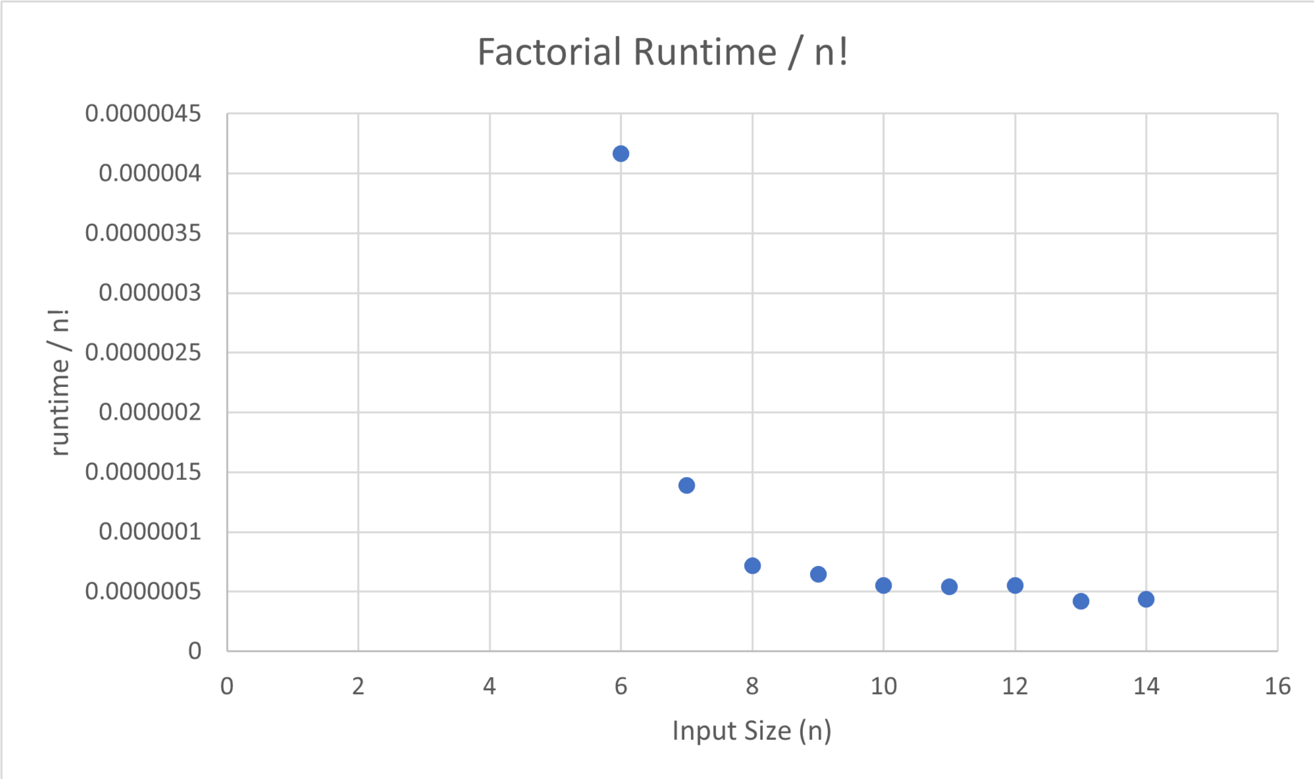


Figure 2: Brute Force Runtime / n!

Note that in these graphs, as the value of n increases, the runtime plot begins to look much more factorial, and the runtime / n! plot begins to linearize. This is expected, as $\mathcal{O}$ deals with input values as they grow larger.

# 4 Dynamic Programming Solution

## 4.1 Algorithm Explanation

The dynamic programming part for this problem is based on using previous bitonic tours. In our approach, the M array holds the optimal tour length at each index, so M[4] would be the optimal length for the first four points. M[1] and M[2] are trivial, since you either don't travel any distance, or only go back and forth to one point. The K array is used to hold key points that are found as each new tour step is made, more will be described on K shortly. The first loop takes i from 3 to n. In this case i represents tours consisting of i points. M[i] is set to infinity to help easily calculate the smallest tour. At this point, we only have an $\mathcal{O}(n)$ loop, and getting our next loop to bring our total to $\mathcal{O}(n^2)$ consists of looping through each point in the previous tour and calculating the new tour distance with that point. We add the distance between that point k a the new point i and between i and k+1. We also need to subtract the distance between k and k+1, since that edge wouldn't be in the new tour. If using the point k is best, we save it in K[i]. After all these steps, in the end M[n] is the optimal tour length.

With K, we can use the key points stored in it to create n segments that go between two points each to create our tour path. To convert these segments to the point order, we sorted them based on the first point in those segments. Noted is that for all the segments going to point A to point B, A < B. We just did a bubble sort, since using a better sort wouldn't improve out time complexity, but could be slightly faster. We then linearly went through the segment list, comparing against an initial point 0, to the point A in the segments. If they were equal, we set our variable to point B, and marked that segment as used. One loop got one half of the tour, going from 0 to the right most point. We then just needed to traverse the list backwards for unused points to get the other half of the tour.

## 4.2 Run-time Analysis

This solution appears to be $\mathcal{O}(n^2)$.

A table and graphs of the time for this algorithm to run follow:

| Input Size | Runtime | runtime / $n^2$ | $n^2$ |
|---|---|---|---|
| 4 | 0.005 | 0.0003125 | 16 |
| 5 | 0.005 | 0.0002 | 25 |
| 6 | 0.001 | 2.77778E-05 | 36 |
| 7 | 0.005 | 0.000102041 | 49 |
| 8 | 0.003 | 0.000046875 | 64 |
| 9 | 0.005 | 6.17284E-05 | 81 |
| 10 | 0.003 | 0.00003 | 100 |
| 11 | 0.003 | 2.47934E-05 | 121 |
| 12 | 0.003 | 2.08333E-05 | 144 |
| 13 | 0.004 | 2.36686E-05 | 169 |
| 14 | 0.001 | 5.10204E-06 | 196 |
| 20 | 0.004 | 0.00001 | 400 |
| 25 | 0.002 | 0.0000032 | 625 |
| 50 | 0.002 | 0.0000008 | 2500 |
| 75 | 0.003 | 5.33333E-07 | 5625 |
| 100 | 0.002 | 0.0000002 | 10000 |
| 200 | 0.005 | 0.000000125 | 40000 |
| 250 | 0.013 | 0.000000208 | 62500 |
| 500 | 0.043 | 0.000000172 | 250000 |
| 750 | 0.048 | 8.53333E-08 | 562500 |
| 1000 | 0.149 | 0.000000149 | 1000000 |
| 1500 | 0.179 | 7.95556E-08 | 2250000 |
| 2000 | 0.326 | 8.15E-08 | 4000000 |
| 2500 | 0.523 | 8.368E-08 | 6250000 |
| 3000 | 0.736 | 8.17778E-08 | 9000000 |
| 4000 | 1.265 | 7.90625E-08 | 16000000 |
| 5000 | 1.983 | 7.932E-08 | 25000000 |

Table 2: Brute Force Runtime Table

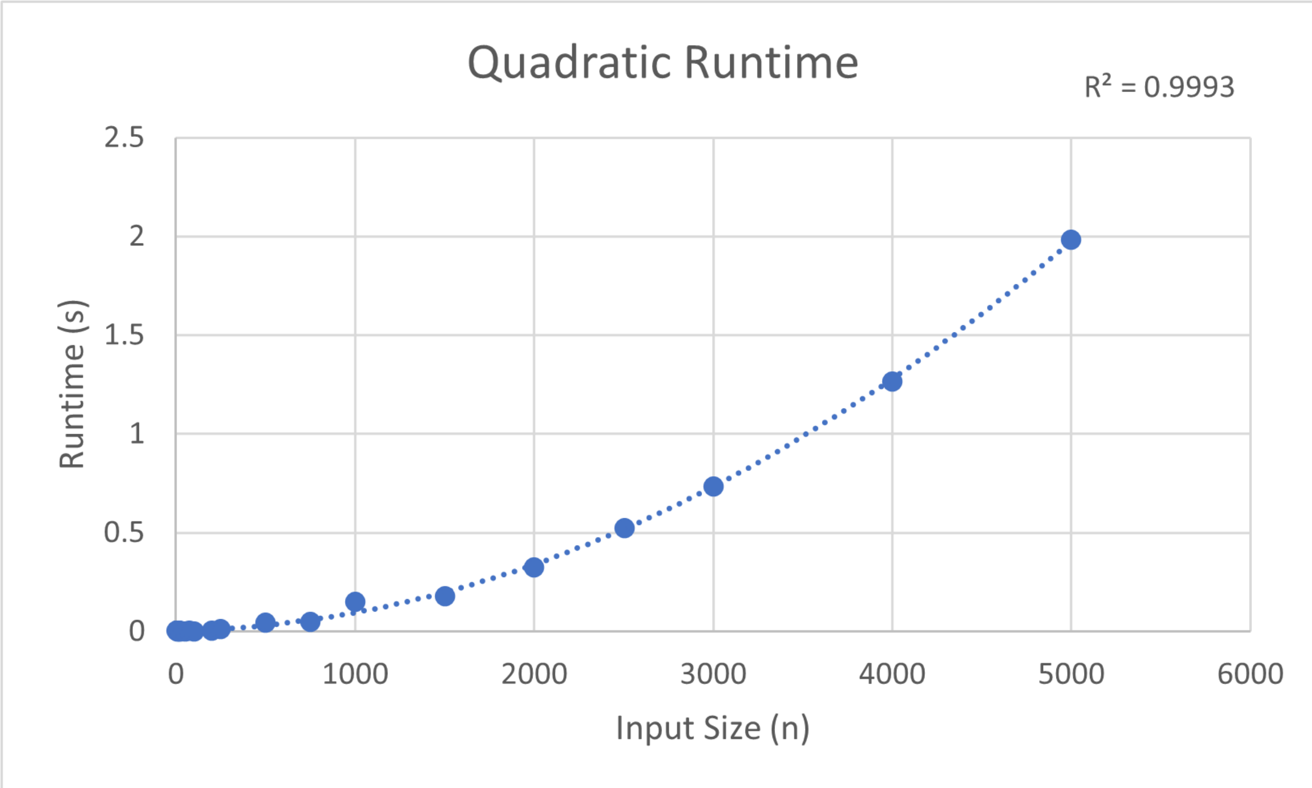


Figure 3: Dynamic Programming Runtime

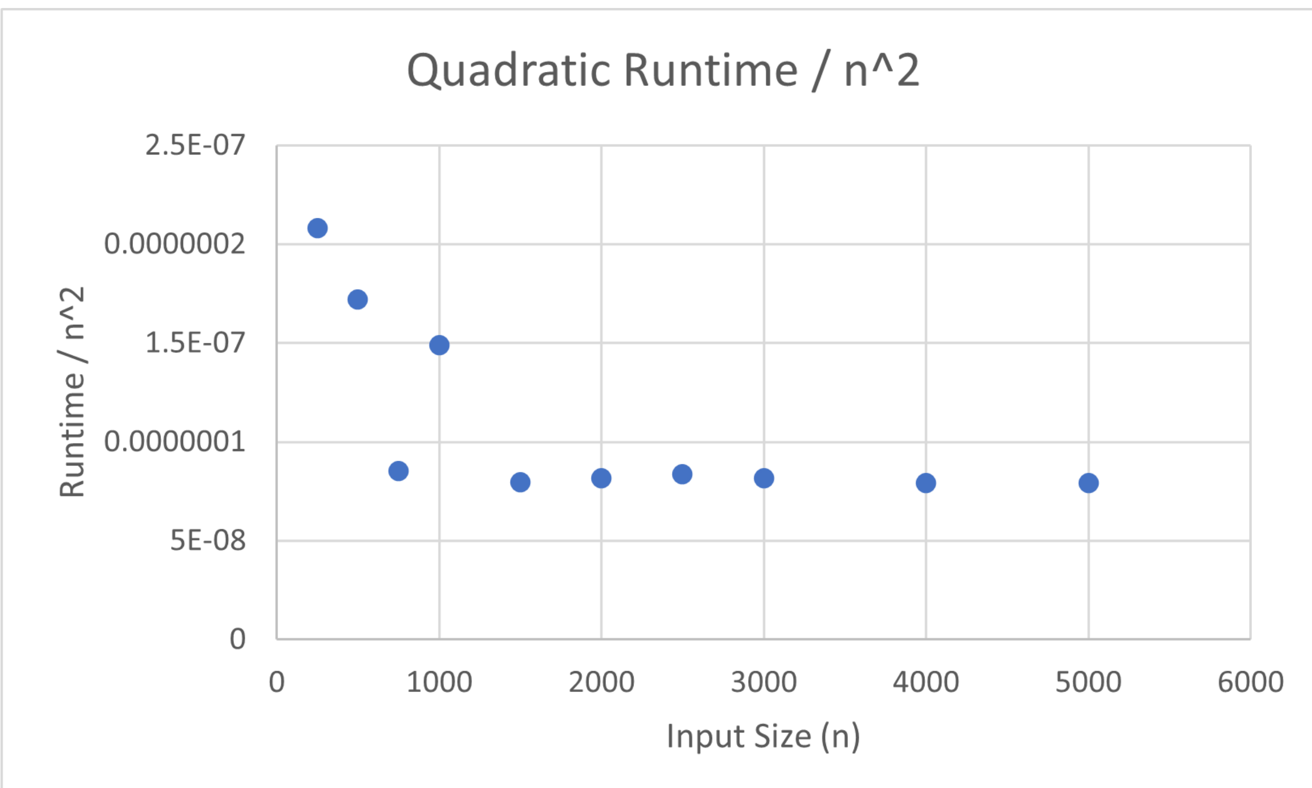


Figure 4: Dynamic Runtime / $n^2$