Dynamic Programming

Bitonic Traveling Salesman

Given a set of points, you must find the shortest path that goes through all points once and ends where you began. Think of this problem in two halves: traveling forward and traveling backwards. If you order the points by increasing x value and start with the leftmost point, you will get to the rightmost point before turning around. This means that the forward path and backward path share a common startpoint and endpoint. You can asses both paths in the forward (left to right) direction since distance does not change. Now, you must determine which points should be on the forward path, and which points should be on the backward path.

Let distance(x, y) be a helper function that returns the euclidean distance between points x and y.

1. Create an $n * n$ matrix where n is the number of points. The points along the index will denote the end of the forward path. The points along the columns will denote the end of the backward path.
   -- e.g. M[3, 4] means that the forward path goes through points 1 -> 2 -> 3, and the backward path goes through 1 -> 4
2. M[1, 1] is initialized to zero, and M[1, 2] is initialized to distance(1, 2)
3. Since M[i, j] = M[j, i] we only need to calculate the values for the top half of the matrix above the M[i, i] diagonal. Fill in the values according to the following cases.
4. Case 1: $i < j - 1$
   M[i, j] = M[i, j-1] + distance(j-1, j)
   Remember, the forward path stops at i and the backwards path stops at j. Since $i < j - 1$, it means $j - 1$ cannot be on the forward path (since it stops before reaching $j - 1$). Therefore, we add the distance from $j - 1$ to j because $j - 1$ has to be on the backwards path.
5. Case 2: $i = j - 1$ or $i = j$
   M[i, j] = min(M[i, k] + distance(k, j) for k = [1, i))
   Here, we only know that the forward path is some path from 1 to i. What we can do is consider some point $1 <= k < j$ that divides the forward and backward paths. This makes the total cost of the path equal to the optimal path from i to k which is already calculated in M[i, k] added to the cost of the path from k to j which is the distance between the two points. Because we are minimizing distance, the optimal path is the point k that results in the shortest distance.
6. This means that the optimal path cost is found in M[n, n] because it finds the optimal k that divides the forward and backwards paths.

The progress of the algorithm is shown below. One path is in red, the other is in blue. All paths that were considered for case 2 are printed as well as the selected minimum value. If a path is printed twice but the k value is different, it is because the 'forward' and 'backward' paths were swapped but it does not change the total cost.

```
Distances between points
```

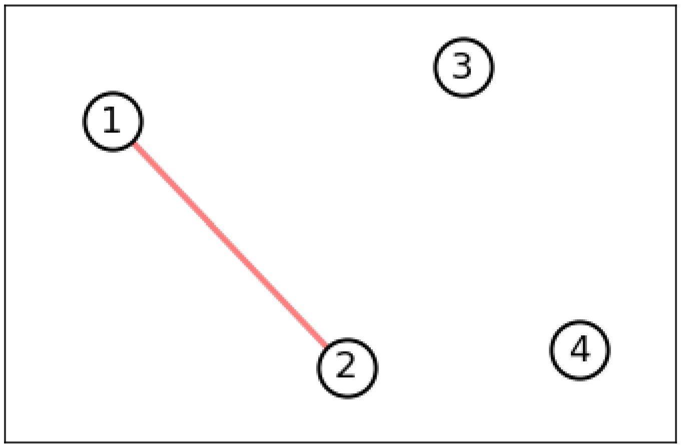
	1	2	3	4
1	0.00	14.14	4.24	13.60
2	14.14	0.00	17.03	2.24
3	4.24	17.03	0.00	16.03
4	13.60	2.24	16.03	0.00

```
Initialize Matrix
```

	1	2	3	4
1	0.00	0.00	0.00	0.00
2	-	0.00	0.00	0.00
3	-	-	0.00	0.00
4	-	-	-	0.00

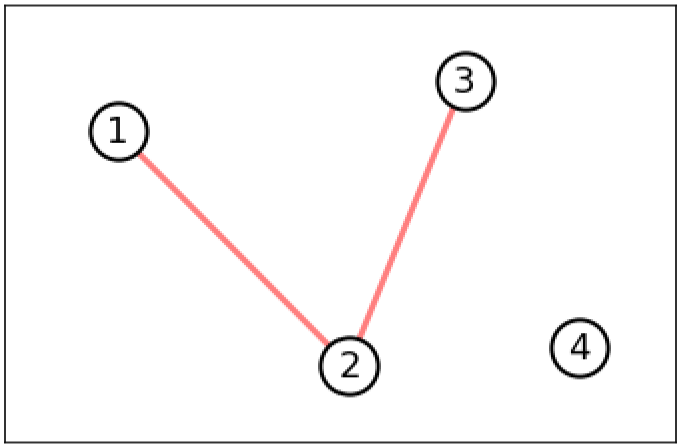
	1	2	3	4
1	0.00	14.14	0.00	0.00
2	-	0.00	0.00	0.00
3	-	-	0.00	0.00
4	-	-	-	0.00

```
i: 1, j: 2
```



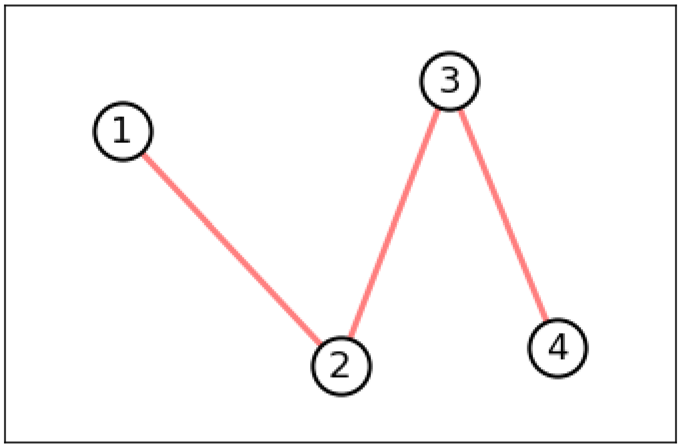
	1	2	3	4
1	0.00	14.14	31.17	0.00
2	-	0.00	0.00	0.00
3	-	-	0.00	0.00
4	-	-	-	0.00

```
i: 1, j: 3
```

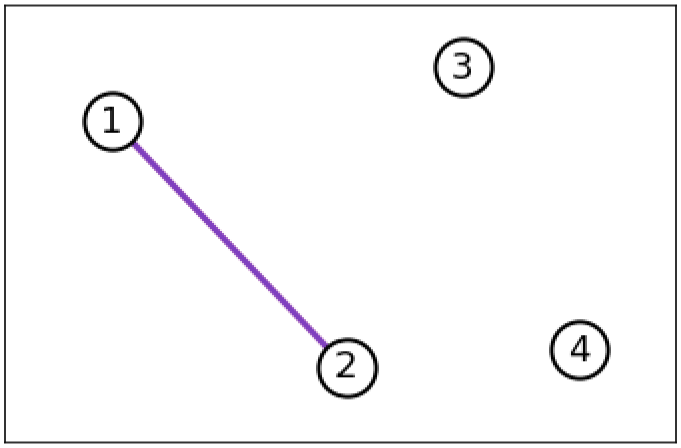


	1	2	3	4
1	0.00	14.14	31.17	47.20
2	-	0.00	0.00	0.00
3	-	-	0.00	0.00
4	-	-	-	0.00

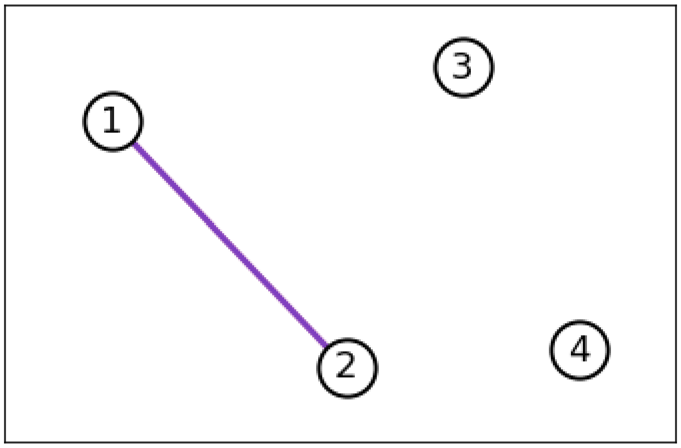
```
i: 1, j: 4
```



```
i: 2, j: 2, k: 1
```

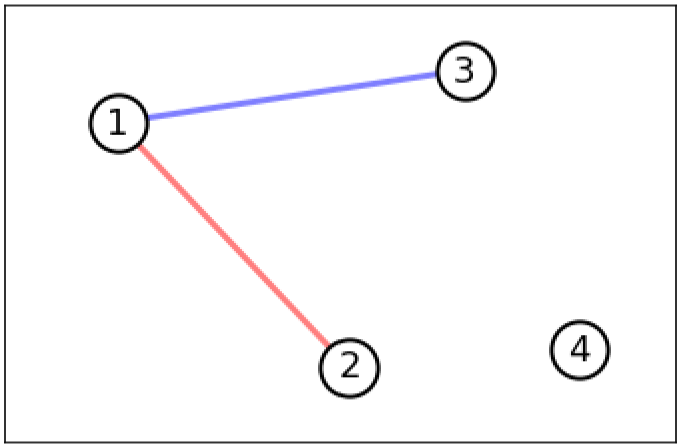


```
MIN i: 2, j: 2, k: 1
```

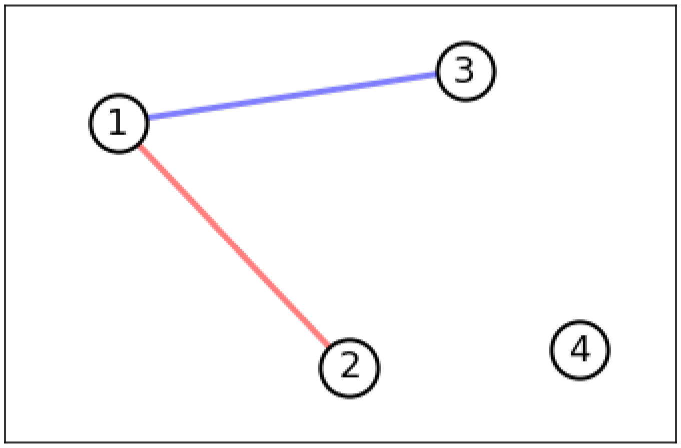


	1	2	3	4
1	0.00	14.14	31.17	47.20
2	-	28.28	0.00	0.00
3	-	-	0.00	0.00
4	-	-	-	0.00

```
i: 2, j: 3, k: 1
```

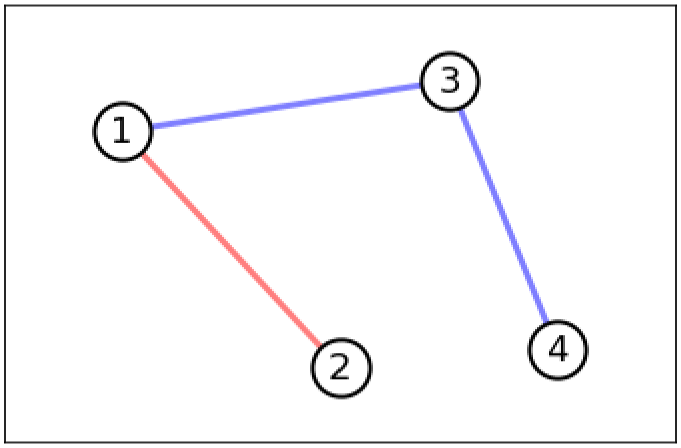


```
MIN i: 2, j: 3, k: 1
```



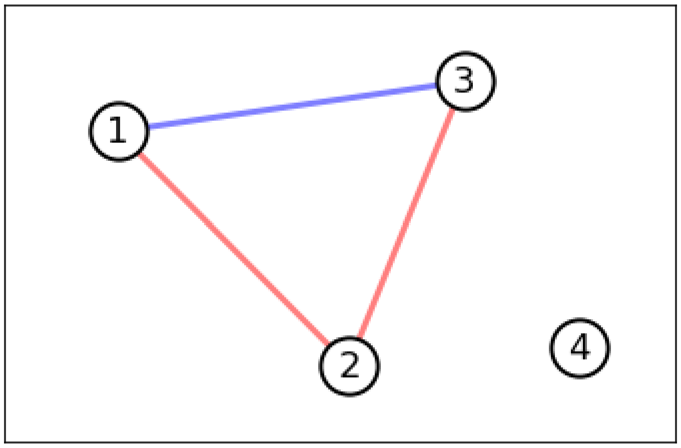
	1	2	3	4
1	0.00	14.14	31.17	47.20
2	-	28.28	18.38	0.00
3	-	-	0.00	0.00
4	-	-	-	0.00

```
i: 2, j: 4
```

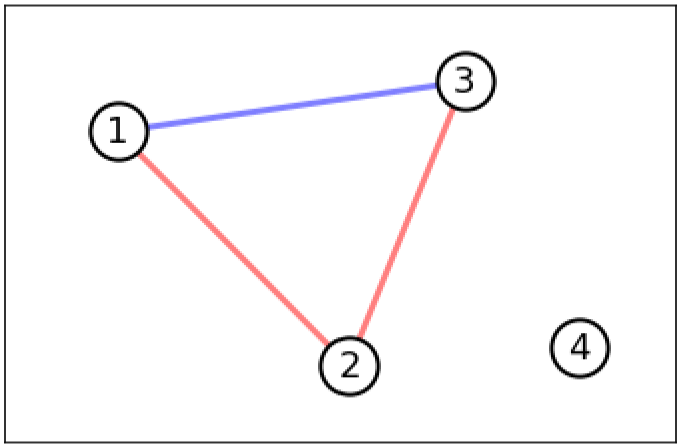


	1	2	3	4
1	0.00	14.14	31.17	47.20
2	-	28.28	18.38	34.42
3	-	-	0.00	0.00
4	-	-	-	0.00

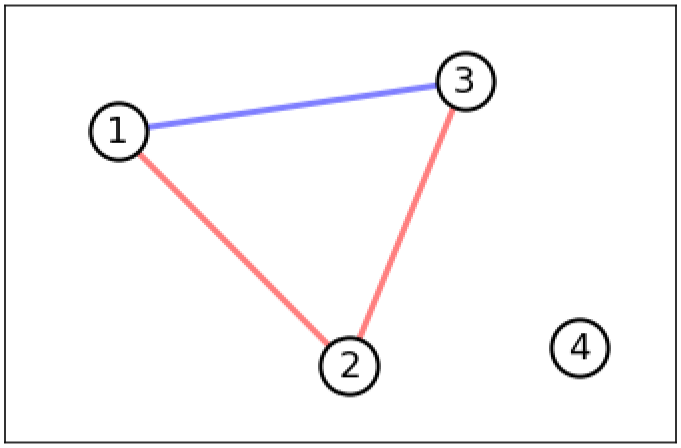
```
i: 3, j: 3, k: 1
```



```
i: 3, j: 3, k: 2
```

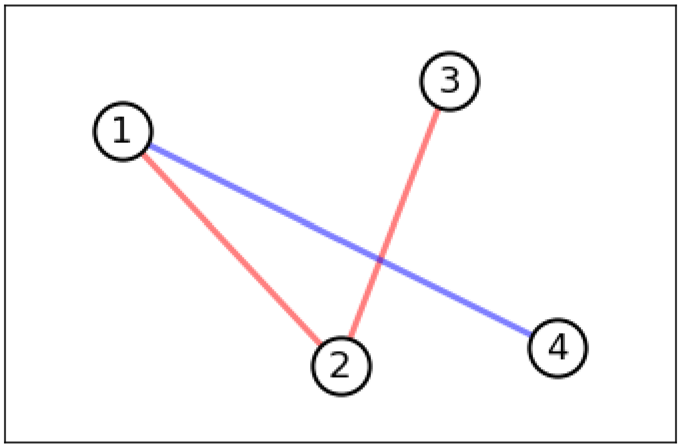


```
MIN i: 3, j: 3, k: 2
```

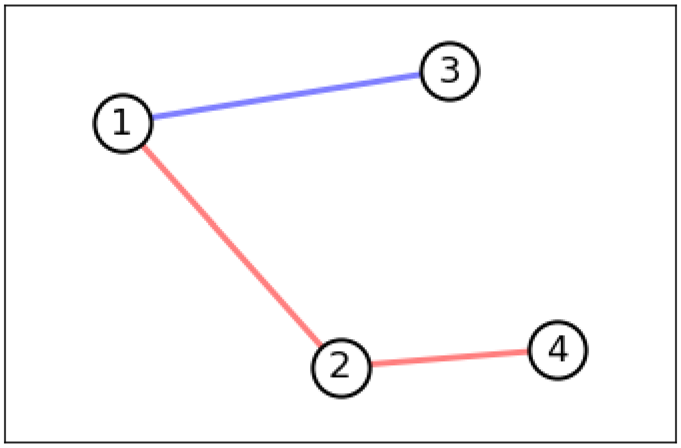


	1	2	3	4
1	0.00	14.14	31.17	47.20
2	-	28.28	18.38	34.42
3	-	-	35.41	0.00
4	-	-	-	0.00

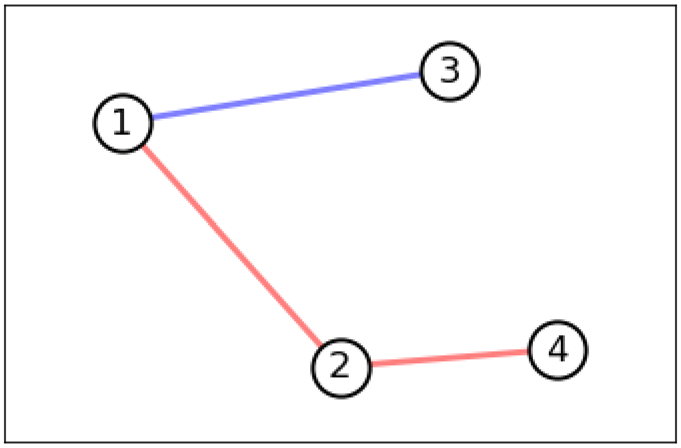
```
i: 3, j: 4, k: 1
```



```
i: 3, j: 4, k: 2
```

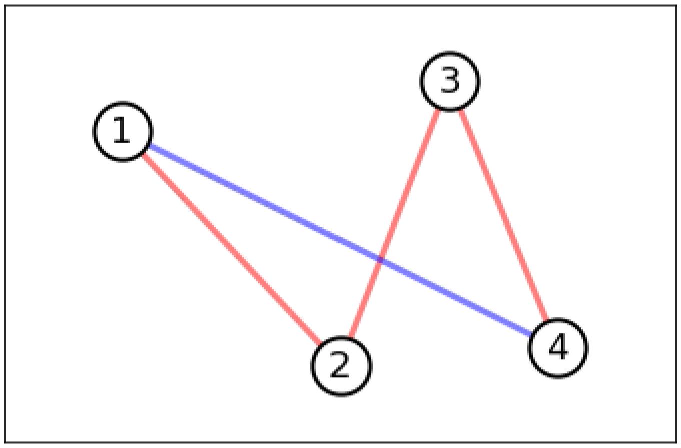


```
MIN i: 3, j: 4, k: 2
```

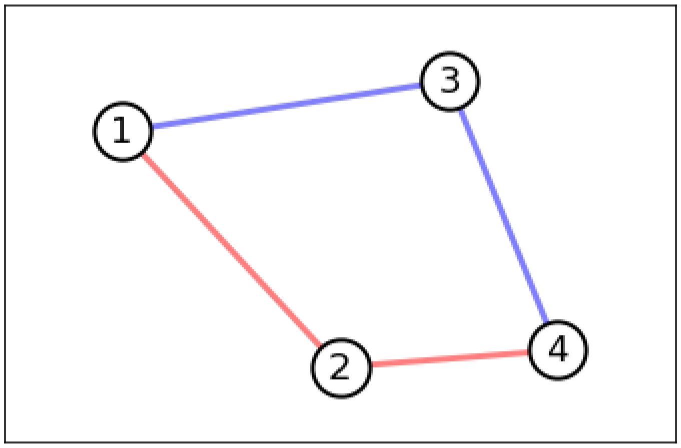


	1	2	3	4
1	0.00	14.14	31.17	47.20
2	-	28.28	18.38	34.42
3	-	-	35.41	20.62
4	-	-	-	0.00

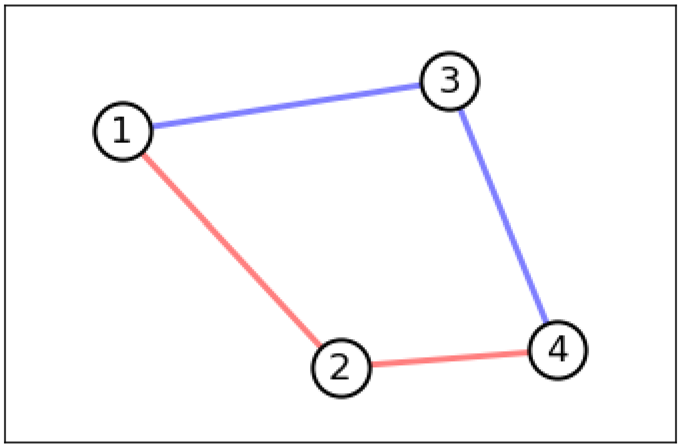
```
i: 4, j: 4, k: 1
```



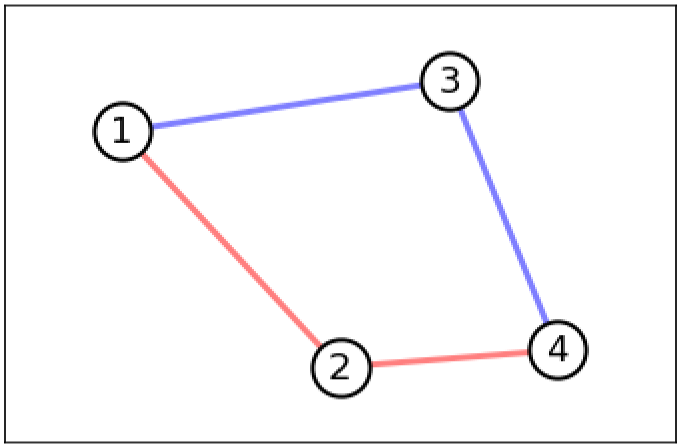
```
i: 4, j: 4, k: 2
```



```
i: 4, j: 4, k: 3
```



```
MIN i: 4, j: 4, k: 3
```



	1	2	3	4
1	0.00	14.14	31.17	47.20
2	-	28.28	18.38	34.42
3	-	-	35.41	20.62
4	-	-	-	36.65

```
Forward Path: [(1, 2), (2, 4)]
Backward Path: [(1, 3), (3, 4)]
```

Complexity

Time complexity: A naive upperbound is $O(n^3)$ where n is the number of points

- You must fill $\frac{n^2}{2}$ cells which can take $O(n)$ work per cell
- You can get a tighter bound by considering that it takes $O(n)$ work to fill a cell only under case 2. That is the case M[i, j] where $i = j - 1$ which only occurs $n - 1$ times. All other times it takes $O(1)$ work. Therefore, the time complexity is only $O(n^2)$

Space complexity: $O(n^2)$