Solution Review: The Traveling Salesman Problem

In this lesson, we will review the solution to the famous traveling salesman problem.

We'll cover the following

Solution 1: Simple recursion

```
import numpy as np

def TSPrecursive(distances, check, index, start):
  minimum = np.inf
  for i in range(len(distances)):
    if i != index and i != start and i not in check:
      check[i] = 1
      minimum = min(minimum, distances[index][i]+TSPrecursive(distances, check, i, start))
      del check[i]
  if minimum == np.inf:
    return distances[index][start]
  return minimum

def TSP(distances):
  check = {}
  minimum = np.inf
  for i in range(len(distances)):
    minimum = min(minimum, TSPrecursive(distances, check, i, i))
  return minimum
```

```
print(TSP([
      [0, 10, 20],

      [12, 0, 10],
      [19, 11, 0],
]))
```

Explanation

Let's see what is going on here. In our main `TSP` function, we simply call the helper function `TSPrecursive` with different starting points. The starting point can play a huge role in finding the path with the minimum distance, that's why we check with every option for the starting point and then take the `min`. In the helper function `TSPrecursive`, the idea is very similar. We exhaust every option for the next city we have and then pick the option that results in the minimum distance. We keep track of visited cities by keeping a dictionary `check`. The base case in this algorithm is when we have visited every city, in this case, the for loop from *lines 5-9* won't be triggered and thus the `minimum` would still be infinity. Thus, we will know we have reached the last city and now we can only go to the start city.

Here is a high-level dry run of this algorithm.

```
TSP([
    [0, 10, 20],
    [12, 0, 10],
    [19, 11, 0],
])
```

TSP([[0, 10, 20], [12, 0, 10], [19, 11, 0],])

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

TSP([[0, 10, 20], [12, 0, 10], [19, 11, 0],])

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

For starting off, we have three options: we can start with either city 0, city 1 or city 2

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 10 & 20 \\ 1 & 12 & 0 & 10 \\ 2 & 19 & 11 & 0 \end{array}$$

From city 0, we have two options: either go to city 1 or city 2

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

Similarly from city 1 and city 2, we have two options again

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

From here onwards, we have one option to explore

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 10 & 20 \\ 1 & 12 & 0 & 10 \\ 2 & 19 & 11 & 0 \end{array}$$

Now we have explored all the city once, so returning to first city

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

39 42 42 39 39 42

39 is the distance of smallest path

8 of 8

Time complexity

The time complexity of this algorithm, as evident from the above visualization, is **O(n!)**. We are doing nothing but looking for all possible permutations of the cities

O(n!). We are doing nothing but looking for all possible permutations of the cities, and the time complexity is in order of factorial.

Moreover, since we are keeping the `check` dictionary to keep track of visited cities, our algorithm has a space complexity of **O(n)**.

Solution 2: Top-down dynamic programming

Let's see how this problem satisfies both pre-requisites of using dynamic programming

Optimal substructure

This problem can be best explained using some concepts of set theory. As the order in which cities are visited is not important as long as all the cities are visited, unordered sets are a perfect fit for the problem. To construct a set of size n, or to find an optimal answer to the problem of n cities, we can explore which subproblems of size $n-1$ leads to the optimal solution. Thus, if we have optimal answers to all the subproblems of size $n-1$, we can construct an optimal answer to the main problem of n.

Overlapping subproblems

Since we are finding permutations, we should expect to find a number of repetitions in the calculation of these permutations. Let's revisit the above visualization to find the overlapping subproblems.

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

Let's look at some overlapping subproblems.

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

Let's look at some overlapping subproblems.

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

Recalculations due to overlapping subproblems


```
import numpy as np

def TSPrecursive(distances, check, index, end, memo):
  keys = tuple(sorted(check.keys()))
  if (keys, index) in memo:
```

```
  if (keys, index) in memo:
    return memo[(keys, index)]
  minimum = np.inf

  for i in range(len(distances)):
    if i != index and i != end and i not in check:
      check[i] = 1
      minimum = min(minimum, distances[index][i]+TSPrecursive(distances, check, i, end, memo))
      del check[i]
  if minimum == np.inf:
    return distances[index][end]
  memo[(keys, index)] = minimum
  return memo[(keys, index)]

def TSP(distances):
  check = {}
  minimum = np.inf
  for i in range(len(distances)):
    minimum = min(minimum, TSPrecursive(distances, check, i, i, {}))
  return minimum

print(TSP([
      [0, 10, 20],
      [12, 0, 10],
      [19, 11, 0],
]))
```

Explanation

The important detail in this solution is what we are using for memorization. Let's rethink this in the context of the problem. How do we define a subproblem? For a problem of n cities, we need the optimal solution to all the problems of size $n-1$. If we had three cities, a, b, c then the subproblems of size 2 we can have are:

- a, b
- a, c
- b, c

The order in which these cities have been visited does not matter., What matters is the cities we have visited and the last city we visit because having a different last city could mean different distances. Thus, we use a tuple made of two things: a set of visited cities and the last city visited.

For the set of visited cities, we use a tuple again since tuples can index a dictionary.

Time and space complexity

In this algorithm, instead of exploring all the permutations, we end up evaluating for all the subsets. For n cities there can be 2^n subsets. For each of these subsets, we explore the solution with each possibility of the last city visited which can be in the order of *O(n)*. Thus, given a starting point, the time complexity of this algorithm is *O(n*2^n*)*. Since, there can be n starting points, the time complexity of our algorithm would be **O($n^2 2^n$)**. The space complexity would be bounded by **O($n2^n$)** because we have subproblems indexed by the number of subsets (2^n), and the last city visited (n).

Solution 3: Bottom-up dynamic programming

```
import numpy as np

def findSubsets(numbers, i, subsets):
        if len(numbers) == i:
                return subsets
        if len(subsets) == 0:
                return findSubsets(numbers, i+1, [(), tuple([numbers[i]])])
        temp_subsets = []
        for subset in subsets:
                temp_subsets += [tuple(list(subset) + [numbers[i]])]
        return findSubsets(numbers, i+1, subsets + temp_subsets)

# function to find shortest path starting from city `start` and back to it
def TSPbottomup(distances, start):
  dp = {} # dp table
  # subproblem of travelling to second city from start city
  for i in range(len(distances)):
    dp[(tuple([i]), i)] = distances[start][i]
  # find all possible subsets of the cities
  subsets = findSubsets(list(range(len(distances))), 0, [])
  # solve for subset of each size from 2 to n
  for subsetSize in range(2,len(distances)+1):
    for subset in subsets:
      if len(subset) == subsetSize:
        # evaluating minimum cost to travel `subsetSize` number of cities while ending up at each 
        for lastCity in subset:
          dp[(subset, lastCity)] = np.inf
          l = list(subset)
          l.remove(lastCity)
          subset2 = tuple(l)
          # to end up at city given by `lastCity`, it should be the last city to be traveled
          for city in subset2:
            dp[(subset, lastCity)] = min(dp[(subset, lastCity)], dp[(subset2, city)] + distances[c
  # return answer to the problem of travlling all cities while ending up at start city
  return dp[(subsets[-1], start)]

def TSP(distances):
  minimum = np.inf
  for i in range(len(distances)):
    minimum = min(minimum, TSPbottomup(distances, i))
  return minimum

print(TSP([
      [0, 10, 20],
      [12, 0, 10],
      [19, 11, 0],
```

```
]))
```

Explanation

We already saw in the last solution why we need a tuple of the set of visited cities and the last city visited to uniquely identify subproblems. We will build on this idea in our bottom-up solution. To solve a problem for n cities, we will require optimal solutions to all the subproblems. Each subproblem is concerned with solving how to visit a subset of the cities, for example, at k^{th} level we will have subsets of size k. We will find all the possible subsets of k and evaluate the optimal answer for them. Having all distinct subsets does not narrow down our subproblems exactly. For a set A of size k, we could have k different options of the last city visited. Thus, we need to evaluate subproblems for each subset and for each possibility of the last city visited. In our solution, we start building from the base case of going to every other city from the `start` city (*lines 17-18*). Next, we start solving for each possible subset, starting from the subsets of size two up until n (*lines 22-24*). For each of these subsets, we find the optimal answer with every possibility of the last city visited (*line 26*).

Now let's see how we find the optimal answer for a subproblem with an example of a subset A of size k with the last city visited being i. We have already solved the subproblems of size $k-1$, so we can use them in the evaluation of the current subproblem. By removing the city i from the set A, we get a new set A' of size $k-1$ (*lines 28-30*). From this set A', we check each option of the last city visited and ultimately choose the one which incurs the least cost to take us to the city i (*lines 32-33*).

At the conclusion of this function, we return the answer to the problem of size n, ending at the `start` city.

Let's take a look at this algorithm's visualization as well.

```
TSP([
    [0, 10, 20],
    [12, 0, 10],
    [19, 11, 0],
])
```

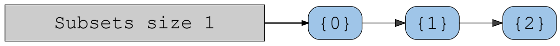
TSP([[0, 10, 20], [12, 0, 10], [19, 11, 0],])

1 of 17

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

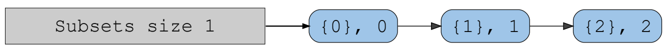
TSP([[0, 10, 20], [12, 0, 10], [19, 11, 0],])

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0



Let our start city be 0, Let's start with base case i.e. subsets of size 1

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0



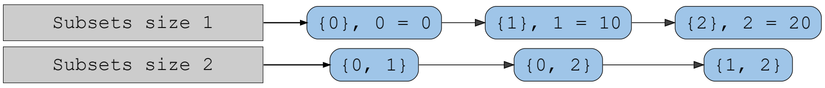
Let's also add the last visited city for each set, in this case it can only be these cities themselves.

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

Subsets size 1 → {0}, 0 = 0 → {1}, 1 = 10 → {2}, 2 = 20

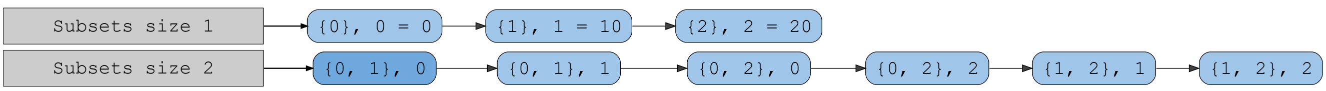
These we know will evaluate to the distance from start city i.e. city 0

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0



Now let's move on to the subsets of size 2

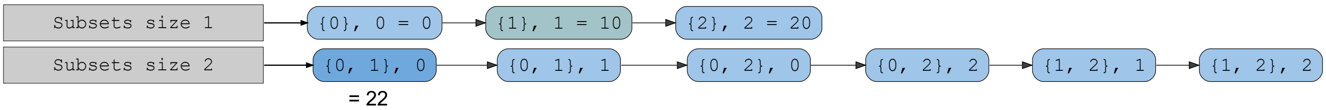
	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0



TSP({0, 1}, 0) =

Let's evaluate each of these subproblems starting with {0, 1}, 0

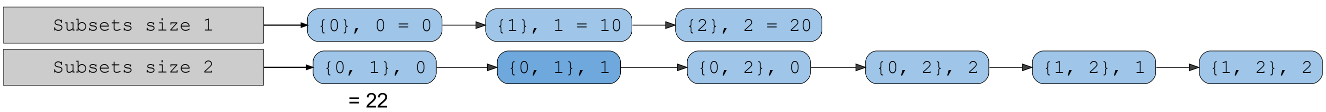
	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0



TSP({0, 1}, 0) = TSP({1} , 1) + distance[1][0]
TSP({0, 1}, 0) = 10 + 12
TSP({0, 1}, 0) = 22

It will be equal to the distance for set {1} and then distance from city 1 to city 0

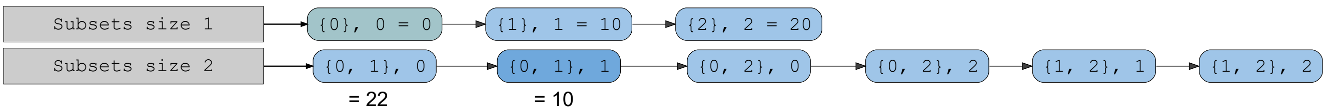
	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0



TSP({0, 1}, 1) =

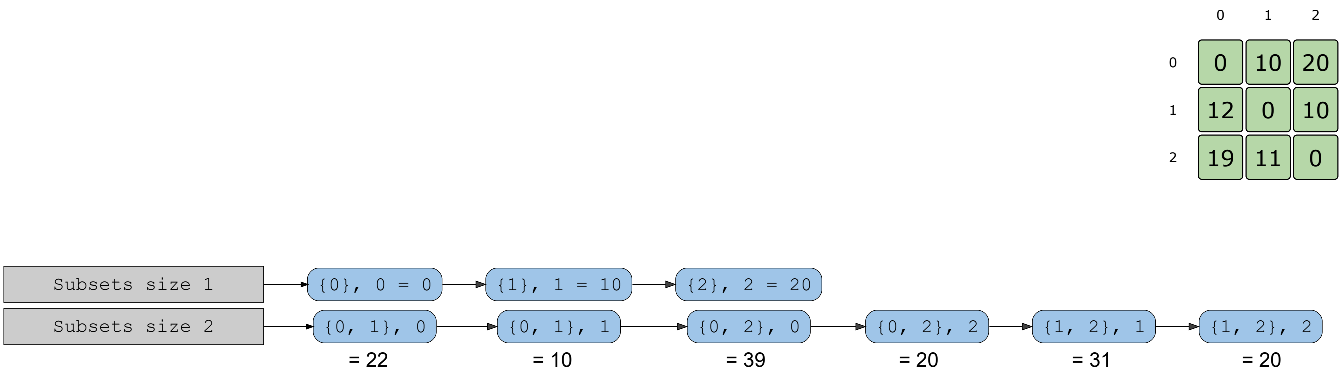
Moving on to {0, 1}, 1

	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0

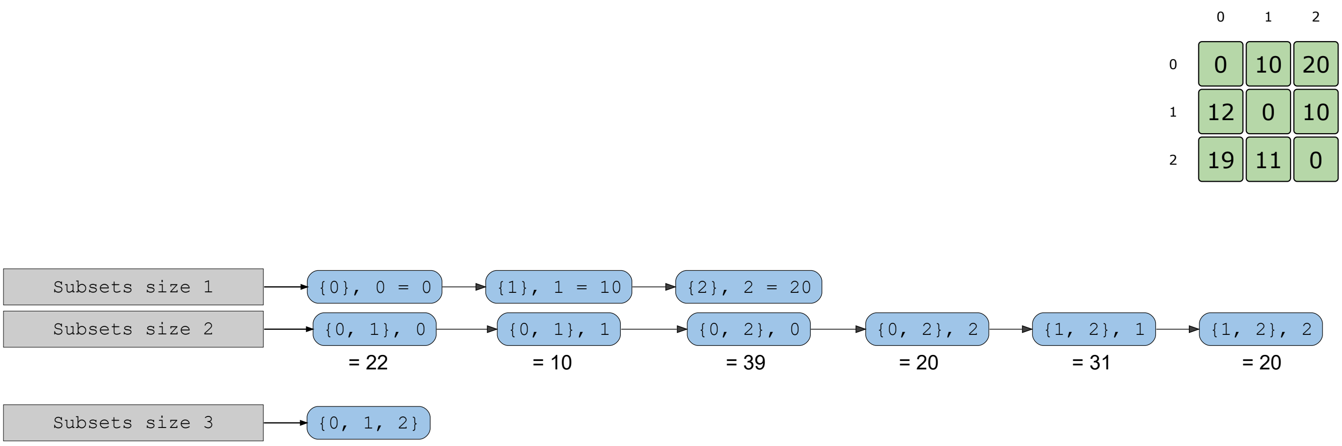


TSP({0, 1}, 1) = TSP({0} , 0) + distance[0][1]
TSP({0, 1}, 1) = 0 + 10
TSP({0, 1}, 1) = 10

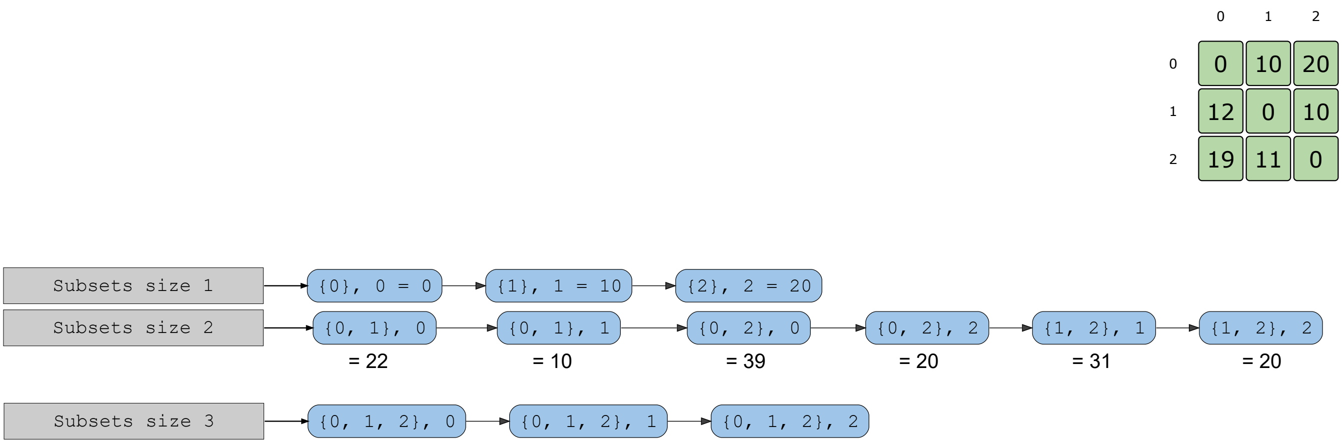
It will be equal to the distance for set {0} and then distance from city 0 to city 1



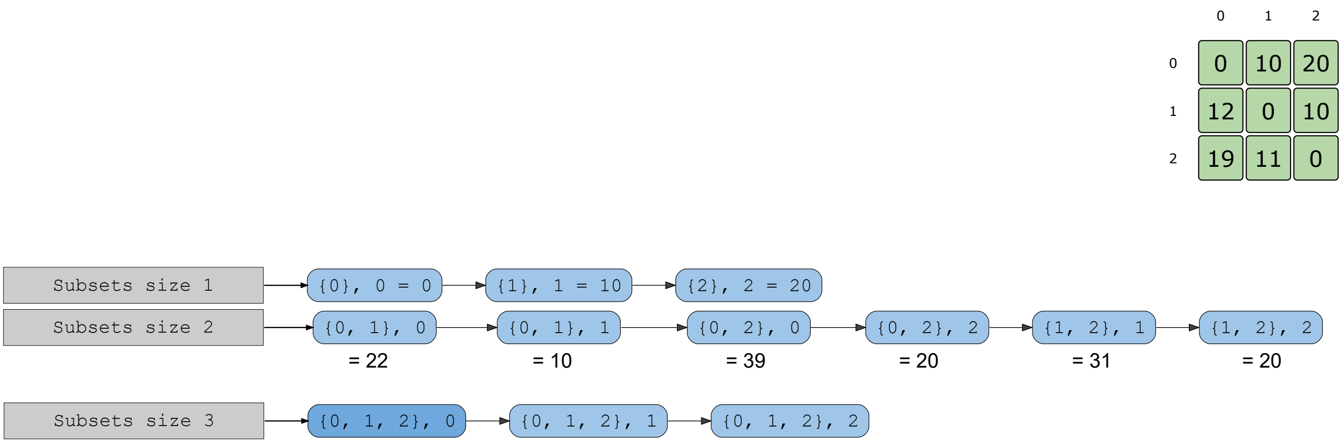
Similarly solving for the rest of this level



Moving on to the subsets of size 3



We can have three options for last city visited

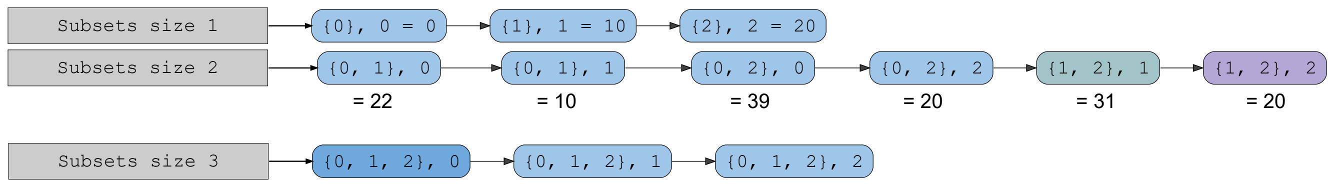


TSP({0, 1, 2}, 0) =

Solving for {0, 1, 2}, 0 (required answer)

14 of 17

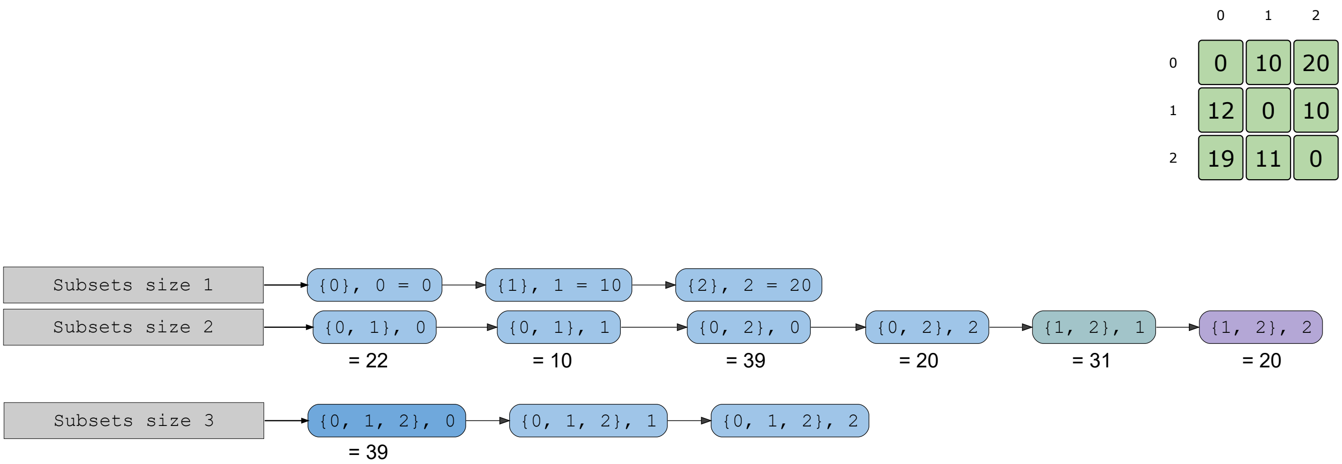
	0	1	2
0	0	10	20
1	12	0	10
2	19	11	0



TSP({0, 1, 2}, 0) = min (TSP({1,2}, 1) + distances[1][0],
TSP({1,2}, 2) + distances[2][0])

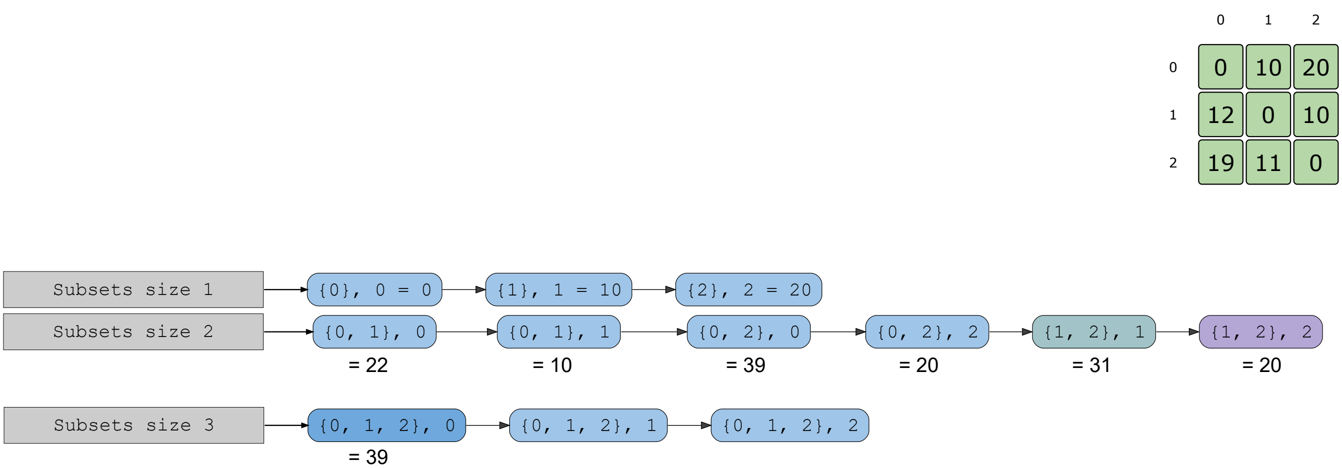
We need to look for subproblems of set {1,2}, we have two such subproblems one with last city being 1 and other being 2

15 of 17



TSP({0, 1, 2}, 0) = min (TSP({1,2}, 1) + distances[1][0],
TSP({1,2}, 2) + distances[2][0])
TSP({0, 1, 2}, 0) = min(31 + 12,
20 + 19)
TSP({0, 1, 2}, 0) = min(43,
39)
TSP({0, 1, 2}, 0) = 39

By evaluating we get the answer to be 39



This was our required answer.

Time and space complexity

As we saw in the previous solution, we have a total of $n2^n$ problems and each of them takes $O(n)$ to evaluate. Thus, the time complexity comes out to be **O(n^2 2^n)**. The space complexity would be **O(n 2^n)** because that is the number of unique subproblems we have.

Room for improvement

You can see in the visualization while solving the subproblems for subsets of size k, we only require the subproblems of size $k - 1$ and none before that. So, instead of storing all the results, we just store the results of subsets of size one unit smaller. Since the highest number of subsets of the same size is $\binom{n}{\frac{n}{2}}$. Thus the time complexity would be **O(n $\binom{n}{\frac{n}{2}}$)** for such an algorithm.

In the next lesson, you will work on another dynamic programming coding challenge.