Travelling Salesman Problem (TSP) with Dynamic Programming

William Fiset

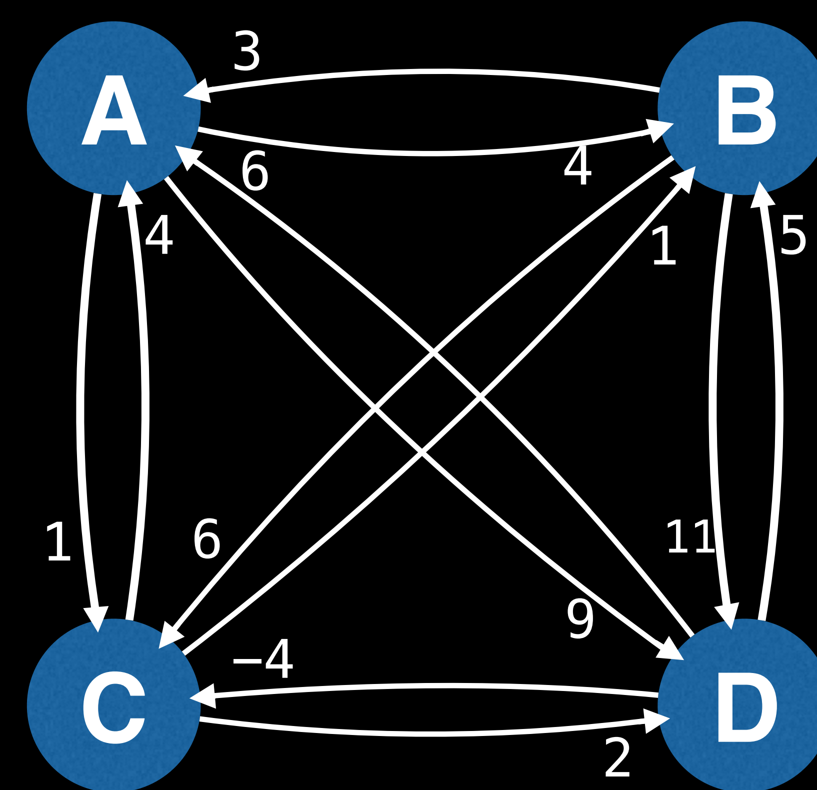
What is the TSP?

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” — Wiki

What is the TSP?

In other words, the problem is: given a **complete graph** with weighted edges (as an adjacency matrix) what is the **Hamiltonian cycle** (path that visits every node once) of minimum cost?

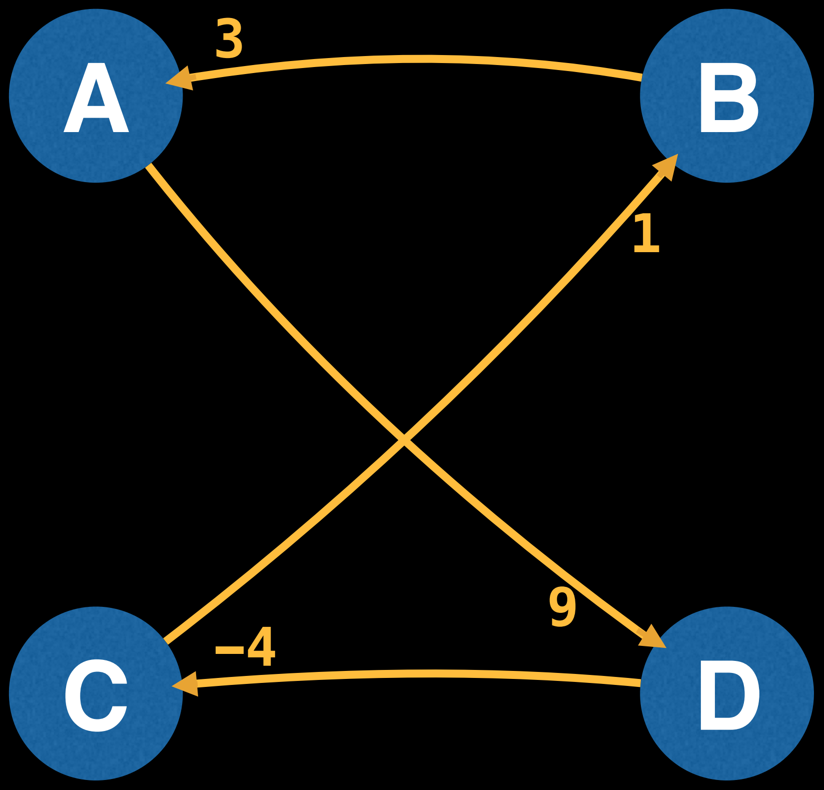
$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 4 & 1 & 9 \\ 3 & 0 & 6 & 11 \\ 4 & 1 & 0 & 2 \\ 6 & 5 & -4 & 0 \end{bmatrix}
\end{array}
$$



What is the TSP?

In other words, the problem is: given a **complete graph** with weighted edges (as an adjacency matrix) what is the **Hamiltonian cycle** (path that visits every node once) of minimum cost?

$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 4 & 1 & 9 \\ 3 & 0 & 6 & 11 \\ 4 & 1 & 0 & 2 \\ 6 & 5 & -4 & 0 \end{bmatrix} \end{array}$$

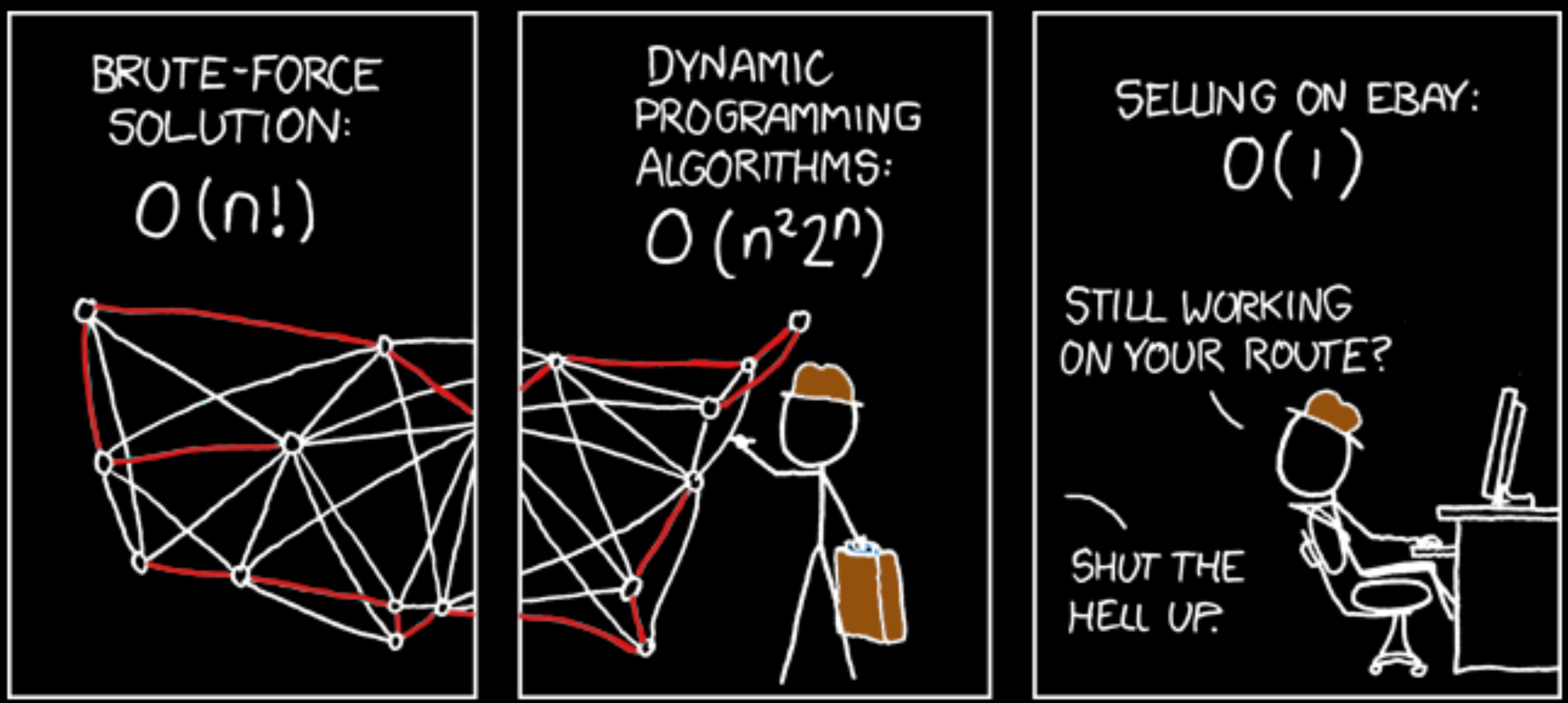


Full tour: A -> D -> C -> B -> A

Tour cost: 9 + −4 + 1 + 3 = 9

What is the TSP?

Finding the optimal solution to the TSP problem is **very hard**; in fact, the problem is known to be **NP-Complete**.



Brute force solution

The brute force way to solve the TSP is to compute the cost of every possible tour. This means we have to try all possible permutations of node orderings which takes **O(n!)** time.

$$
\begin{array}{c|cccc|}
 & A & B & C & D \\
A & 0 & 4 & 1 & 9 \\
B & 3 & 0 & 6 & 11 \\
C & 4 & 1 & 0 & 2 \\
D & 6 & 5 & -4 & 0
\end{array}
$$

Tour	Cost	Tour	Cost
A B C D	18	C A B D	15
A B D C	15	C A D B	24
A C B D	19	**C B A D**	**9**
A C D B	11	C B D A	19
A D B C	24	C D A B	18
A D C B	**9**	C D B A	11
B A C D	11	D A B C	18
B A D C	**9**	D A C B	19
B C A D	24	D B A C	11
B C D A	18	D B C A	24
B D A C	19	D C A B	15
B D C A	15	**D C B A**	**9**

TSP with DP

The dynamic programming solution to the TSP problem significantly improves on the time complexity, taking it from **$O(n!)$** to **$O(n^2 2^n)$**.

At first glance, this may not seem like a substantial improvement, however, it now makes solving this problem feasible on graphs with up to roughly 23 nodes on a typical computer.

TSP with DP

n	n!	$n^2 2^n$
1	1	2
2	2	16
3	6	72
4	24	256
5	120	800
6	720	2304
...	...	...
15	1307674368000	7372800
16	20922789888000	16777216
17	355687428096000	37879808

TSP with DP

The main idea will be to compute the optimal solution for all the subpaths of length **N** while using information from the already known optimal partial tours of length **N**–1.

TSP with DP

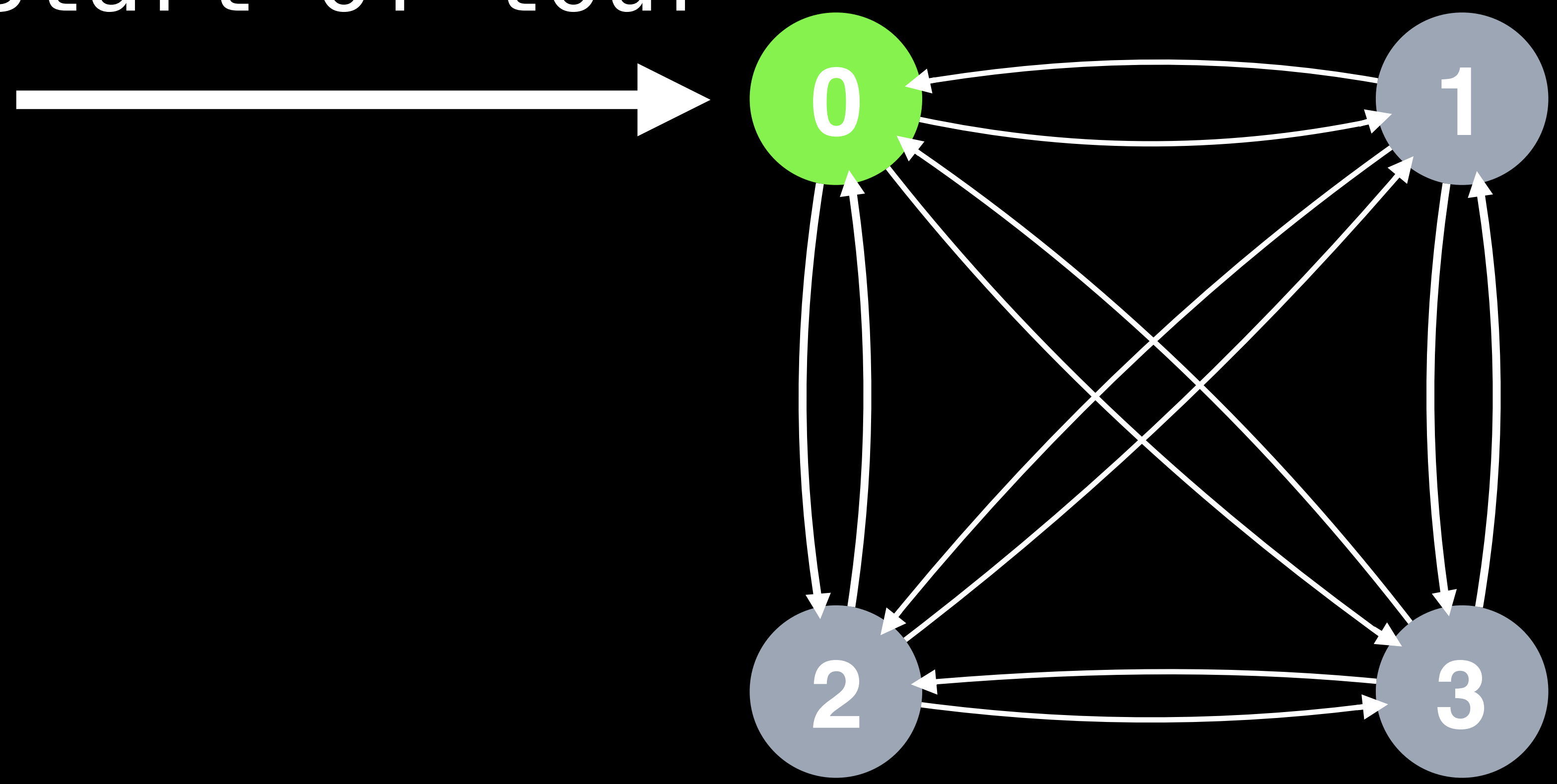
Before starting, make sure to **select a node $0 \leq S < N$ to be the designated starting node for the tour**.

TSP with DP

Before starting, make sure to **select a node 0 ≤ S < N to be the designated starting node for the tour**.

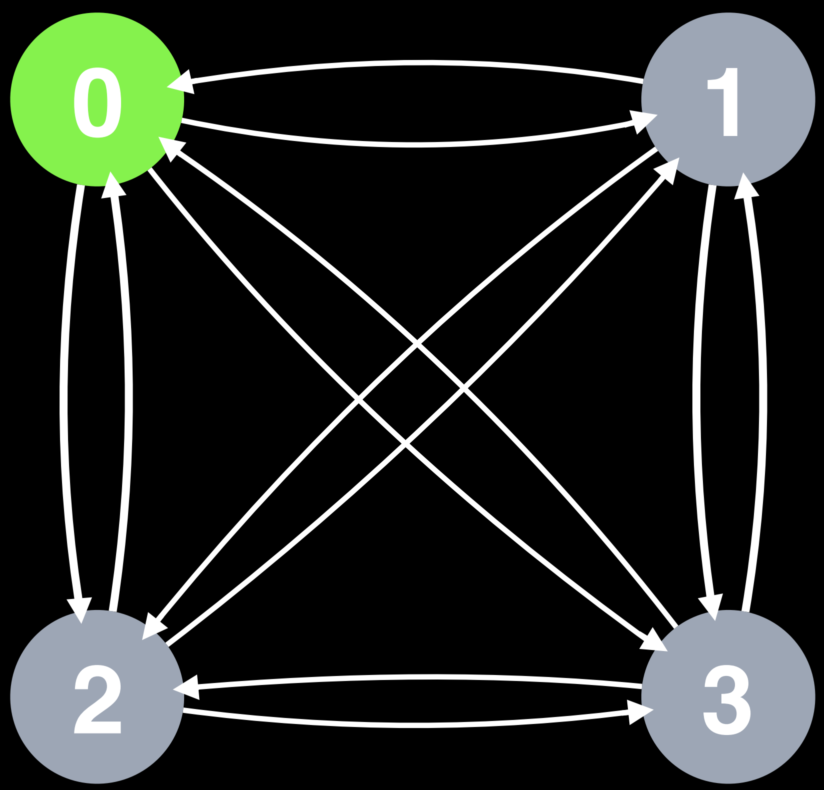
For this example let **S** = node 0

Start of tour



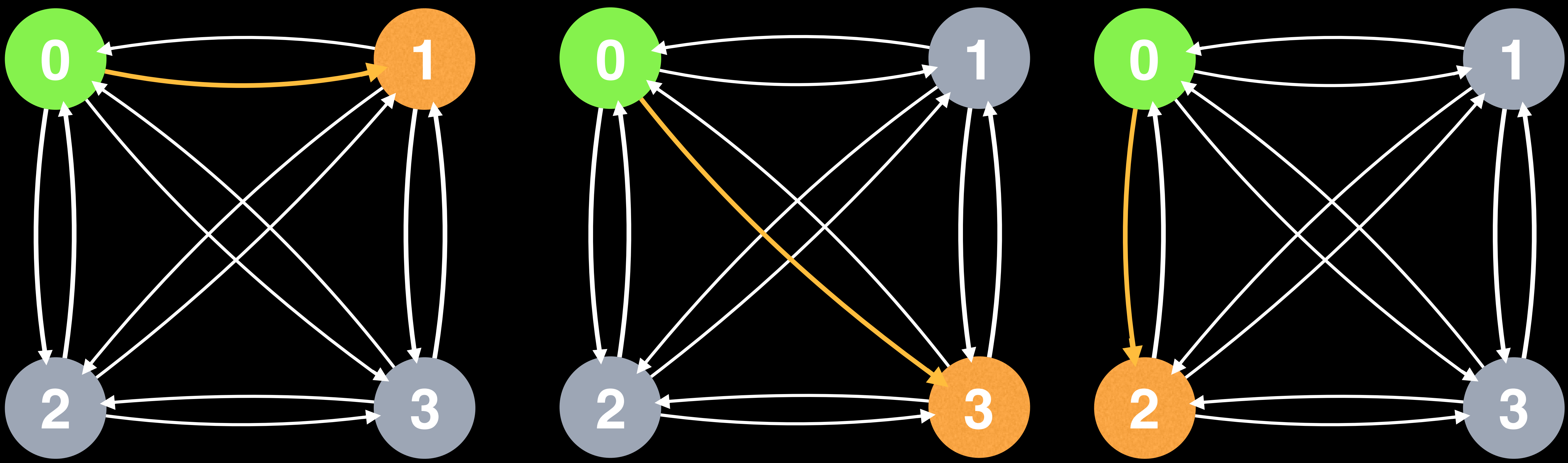
TSP with DP

Next, compute and store the optimal value from **S** to each node **X** (≠ **S**). This will solve TSP problem for all paths of length n = 2.



TSP with DP

Next, compute and store the optimal value from **S** to each node **X** (≠ **S**). This will solve TSP problem for all paths of length n = 2.



TSP with DP

To compute the optimal solution for paths of length 3, we need to remember (store) two things from each of the n = 2 cases:

1) The **set of visited nodes** in the subpath

2) The **index of the last visited node** in the path

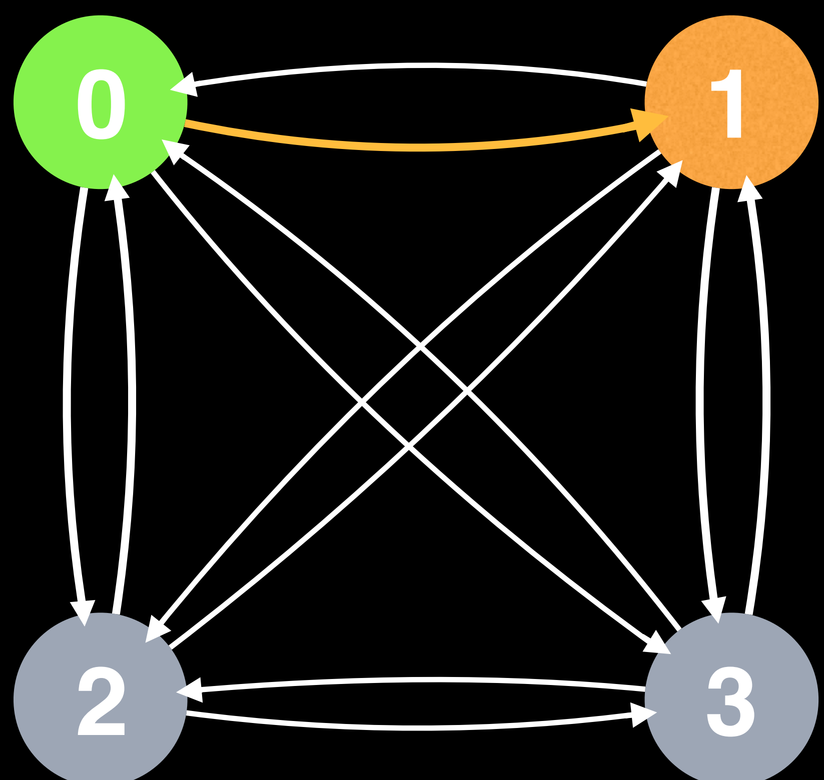
Together these two things form our dynamic programming state. There are N possible nodes that we could have visited last and 2^N possible subsets of visited nodes. Therefore the space needed to store the answer to each subproblem is bounded by **$O(N2^N)$**.

Visited Nodes as a Bit Field

The best way to represent the set of visited nodes is to use **a single 32-bit integer**. A 32-bit int is compact, quick and allows for easy caching in a memo table.

Visited Nodes as a Bit Field

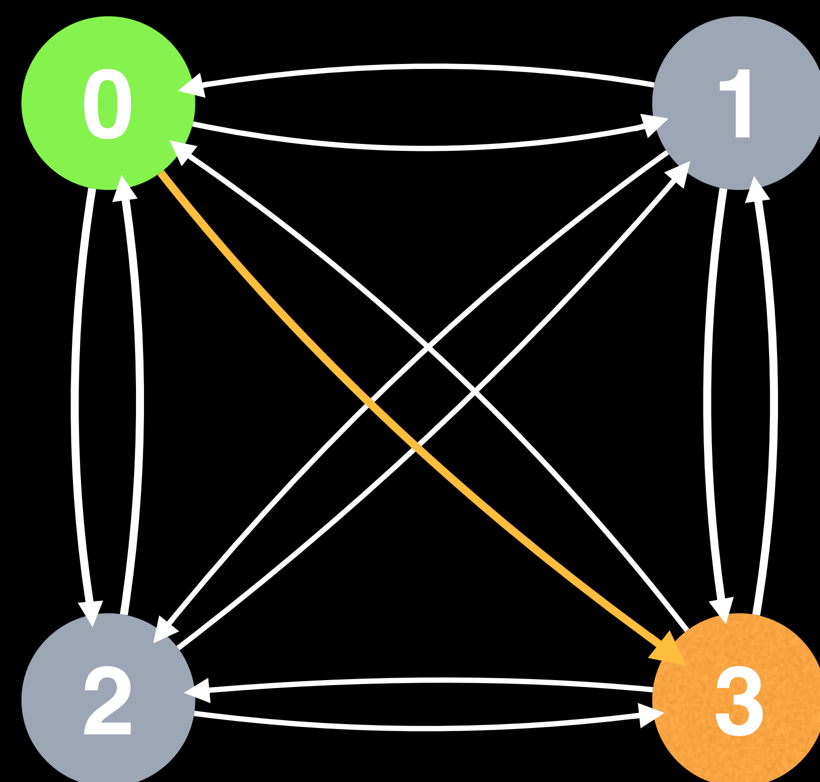
The best way to represent the set of visited nodes is to use **a single 32-bit integer**. A 32-bit int is compact, quick and allows for easy caching in a memo table.



State

Binary rep: $0011_2 = 3$

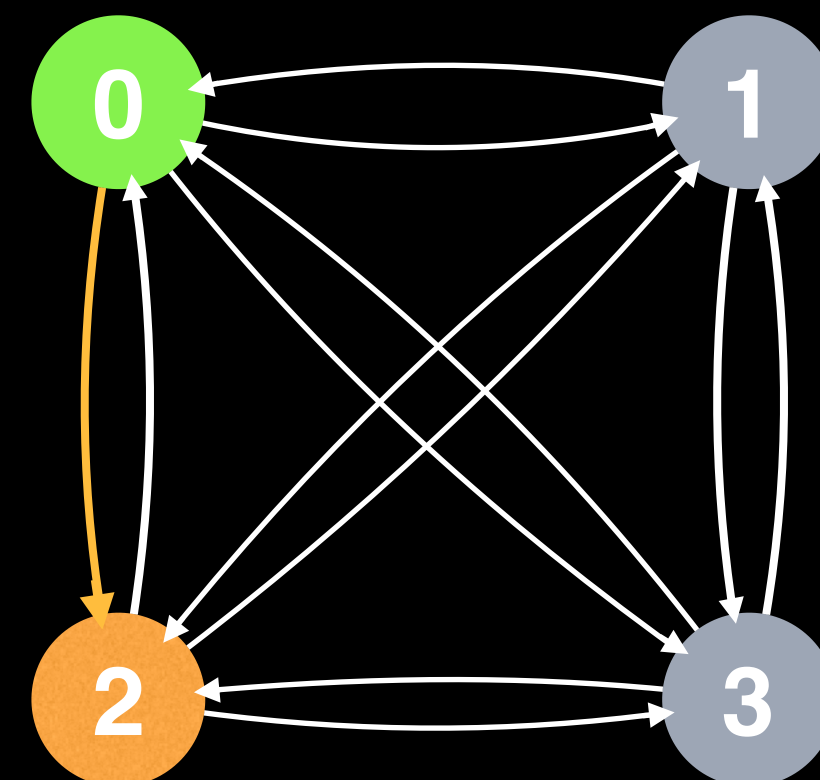
Last node: 1



State

Binary rep: $1001_2 = 9$

Last node: 3



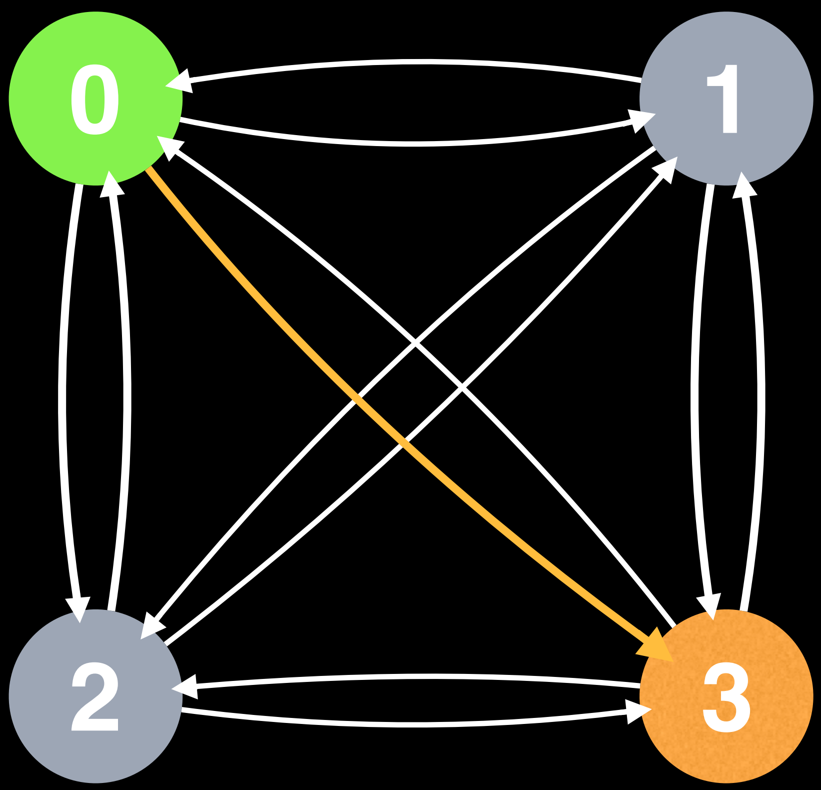
State

Binary rep: $0101_2 = 5$

Last node: 2

TSP with DP

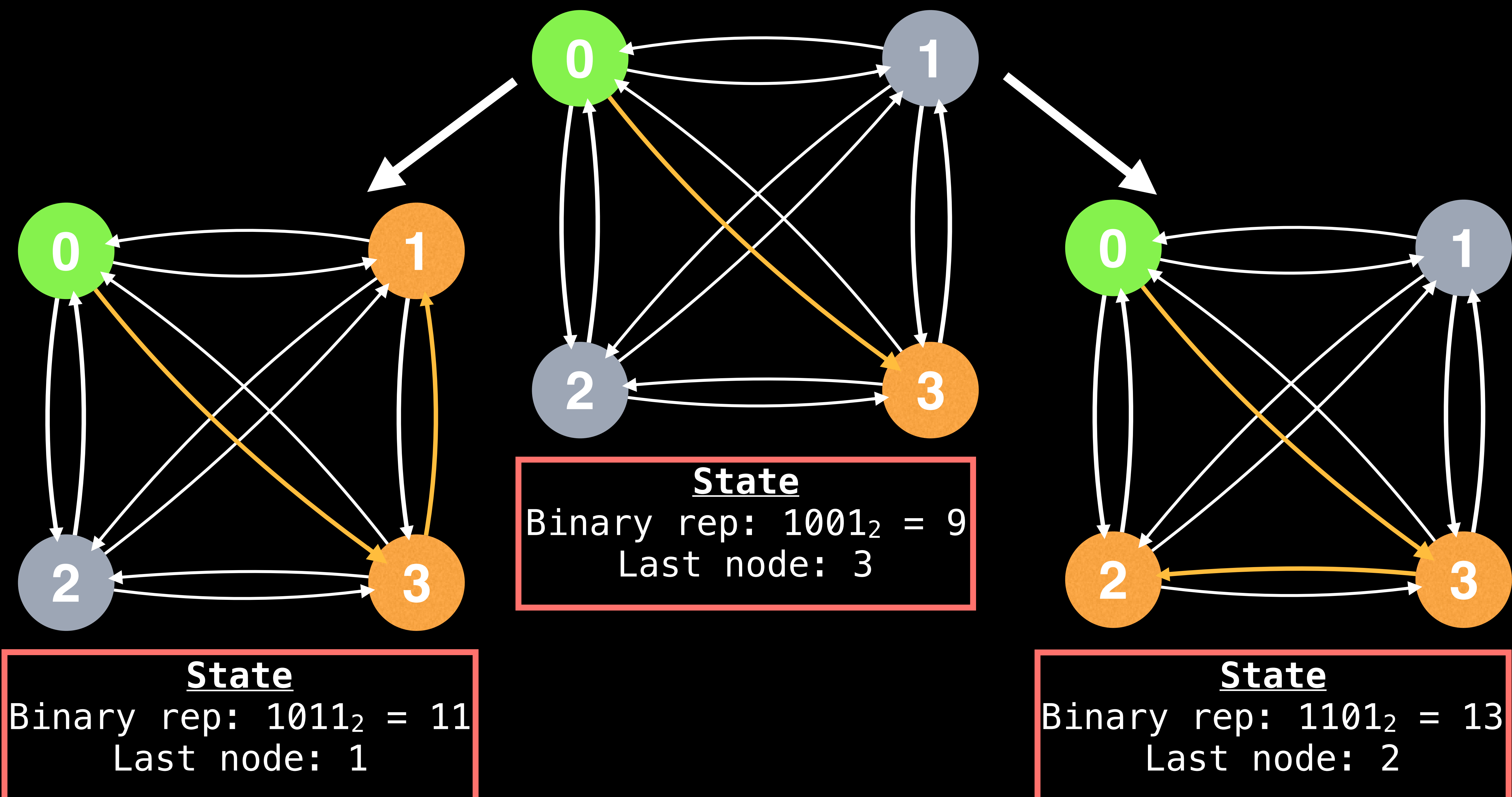
To solve $3 \leq n \leq \mathbf{N}$, we're going to take the solved subpaths from n−1 and add another edge extending to a node which has not already been visited from the last visited node (which has been saved).



State
Binary rep: $1001_2 = 9$
Last node: 3

TSP with DP

To solve $3 \leq n \leq$ **N**, we're going to take the solved subpaths from n−1 and add another edge extending to a node which has not already been visited from the last visited node (which has been saved).



TSP with DP

To complete the TSP tour, we need to connect our tour back to the start node **S**.

Loop over the end state* in the memo table for every possible end position and minimize the lookup value plus the cost of going back to **S**.

* *The end state is the state where the binary representation is composed of* ***N*** *1's*

TSP Pseudocode

In the next few slides we'll look at some pseudocode for the TSP problem.

WARNING: The following slides make use of advanced bit manipulation techniques. Make sure you're very comfortable with the binary operators <<, &, | and ^

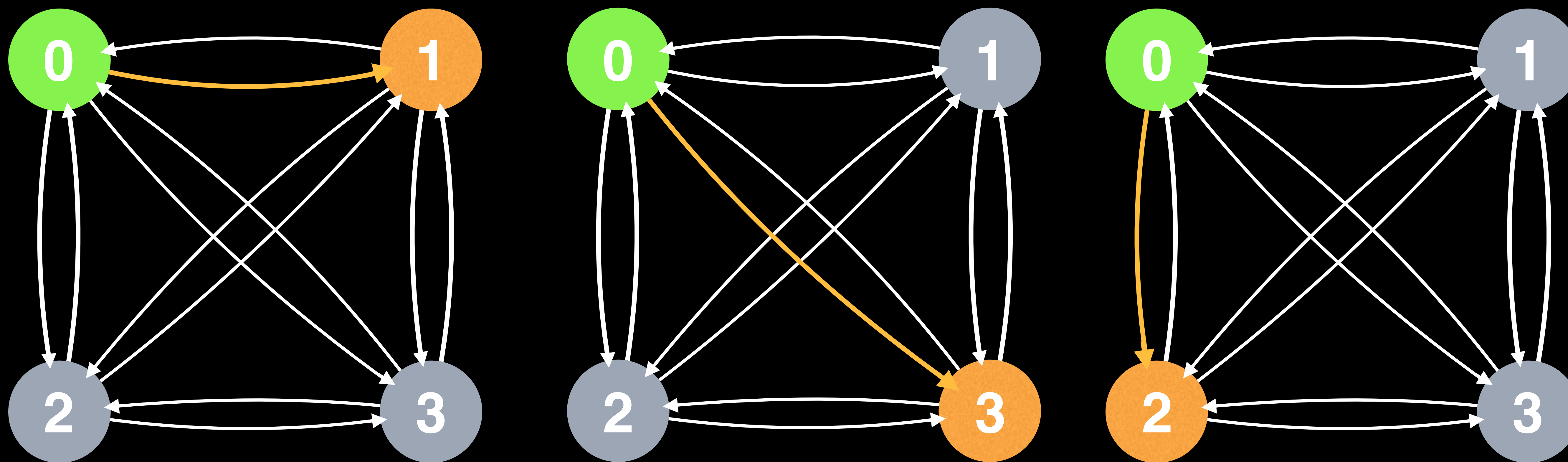
```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
# Initializes the memo table by caching
# the optimal solution from the start
# node to every other node.
function setup(m, memo, S, N):

  for (i = 0; i < N; i = i + 1):

    if i == S: continue

    # Store the optimal value from node S
    # to each node i (this is given as input
    # in the adjacency matrix m).
    memo[i][1 << S | 1 << i] = m[S][i]
```

Node S

Node i



State
Binary rep: $0011_2 = 3$
Last node: 1

State
Binary rep: $1001_2 = 9$
Last node: 3

State
Binary rep: $0101_2 = 5$
Last node: 2

```
# Initializes the memo table by caching
# the optimal solution from the start
# node to every other node.
function setup(m, memo, S, N):

  for (i = 0; i < N; i = i + 1):

    if i == S: continue

    # Store the optimal value from node S
    # to each node i (this is given as input
    # in the adjacency matrix m).
    memo[i][1 << S | 1 << i] = m[S][i]
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
function solve(m, memo, S, N):
 for (r = 3; r <= N; r++):
  # The combinations function generates all bit sets
  # of size N with r bits set to 1. For example,
  # combinations(3, 4) = {0111₂, 1011₂, 1101₂, 1110₂}
  for subset in combinations(r, N):
   if notIn(S, subset): continue
   for (next = 0; next < N; next = next + 1):
    if next == S || notIn(next, subset): continue
    # The subset state without the next node
    state = subset ^ (1 << next)
    minDist = +∞
    # 'e' is short for end node.
    for (e = 0; e < N; e = e + 1):
     if e == S || e == next || notIn(e, subset)):
      continue
     newDistance = memo[e][state] + m[e][next]
     if (newDistance < minDist): minDist = newDistance
    memo[next][subset] = minDist
# Returns true if the ith bit in 'subset' is not set
function notIn(i, subset):
 return ((1 << i) & subset) == 0
```

```
function solve(m, memo, S, N):
 for (r = 3; r <= N; r++):
  # The combinations function generates all bit sets
  # of size N with r bits set to 1. For example,
  # combinations(3, 4) = {0111₂, 1011₂, 1101₂, 1110₂}
  for subset in combinations(r, N):
   if notIn(S, subset): continue
   for (next = 0; next < N; next = next + 1):
    if next == S || notIn(next, subset): continue
    # The subset state without the next node
    state = subset ^ (1 << next)
    minDist = +∞
    # 'e' is short for end node.
    for (e = 0; e < N; e = e + 1):
     if e == S || e == next || notIn(e, subset)):
      continue
     newDistance = memo[e][state] + m[e][next]
     if (newDistance < minDist): minDist = newDistance
    memo[next][subset] = minDist
# Returns true if the ith bit in 'subset' is not set
function notIn(i, subset):
 return ((1 << i) & subset) == 0
```

```
# Generate all bit sets of size n with r bits set to 1.
function combinations(r, n):
  subsets = []
  combinations(0, 0, r, n, subsets)
  return subsets

# Recursive method to generate bit sets.
function combinations(set, at, r, n, subsets):
  if r == 0:
    subsets.add(set)
  else:
    for (i = at; i < n; i = i + 1):
      # Flip on ith bit
      set = set | (1 << i)

      combinations(set, i + 1, r - 1, n, subsets)

      # Backtrack and flip off ith bit
      set = set & ~(1 << i)
```

NOTE: For a more detailed explanation on generating combinations see video “Backtracking tutorial: power set”

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
function findMinCost(m, memo, S, N):

  # The end state is the bit mask with
  # N bits set to 1 (equivalently 2^N - 1)
  END_STATE = (1 << N) - 1

  minTourCost = +∞

  for (e = 0; e < N; e = e + 1):

    if e == S: continue

    tourCost = memo[e][END_STATE] + m[e][S]
    if tourCost < minTourCost:
      minTourCost = tourCost

  return minTourCost
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```

```
function findOptimalTour(m, memo, S, N):
  lastIndex = S
  state = (1 << N) - 1; # End state
  tour = array of size N+1

  for (i = N-1; i >= 1; i--):
    index = -1
    for (j = 0; j < N; j++):
      if j == S || notIn(j, state): continue
      if (index == -1) index = j
      prevDist = memo[index][state] + m[index]lastIndex]
      newDist  = memo[j][state] + m[j][lastIndex];
      if (newDist < prevDist) index = j

    tour[i] = index
    state = state ^ (1 << index)
    lastIndex = index

  tour[0] = tour[N] = S
  return tour
```

```
# Finds the minimum TSP tour cost.
# m - 2D adjacency matrix representing graph
# S - The start node (0 ≤ S < N)
function tsp(m, S):
  N = matrix.size

  # Initialize memo table.
  # Fill table with null values or +∞
  memo = 2D table of size N by 2^N

  setup(m, memo, S, N)
  solve(m, memo, S, N)
  minCost = findMinCost(m, memo, S, N)
  tour = findOptimalTour(m, memo, S, N)

  return (minCost, tour)
```