Dynamic Programming

https://www.educative.io/courses/grokking-dynamic-programming-a-deep-dive-using-cpp/introduction-to-dynamic-programming

When do we use dynamic programming?

Many computational problems are solved by recursively applying a divide-and-conquer approach. In some of these problems, we see an **optimal substructure**, i.e., the solution to a smaller problem helps us solve the bigger one.

Let's consider the following problem as an example: is the string **"rotator"** a palindrome? We can start by observing that the first and the last characters match, so the string *might* be a palindrome. We shave off these two characters and try to answer the same question for the smaller string **"otato".** The subproblems we encounter are**: "rotator", "otato", "tat", and "a".** For any subproblem, if the answer is *no*, we know that the overall string is not a palindrome if the answer is *yes* for all of the subproblems, then we know that the overall string is indeed a palindrome.

While each subproblem in this recursive solution is distinct, there are many problems whose recursive solution involves solving some subproblems over and over again. An example is the recursive computation of the *n^{th}* Fibonacci number.

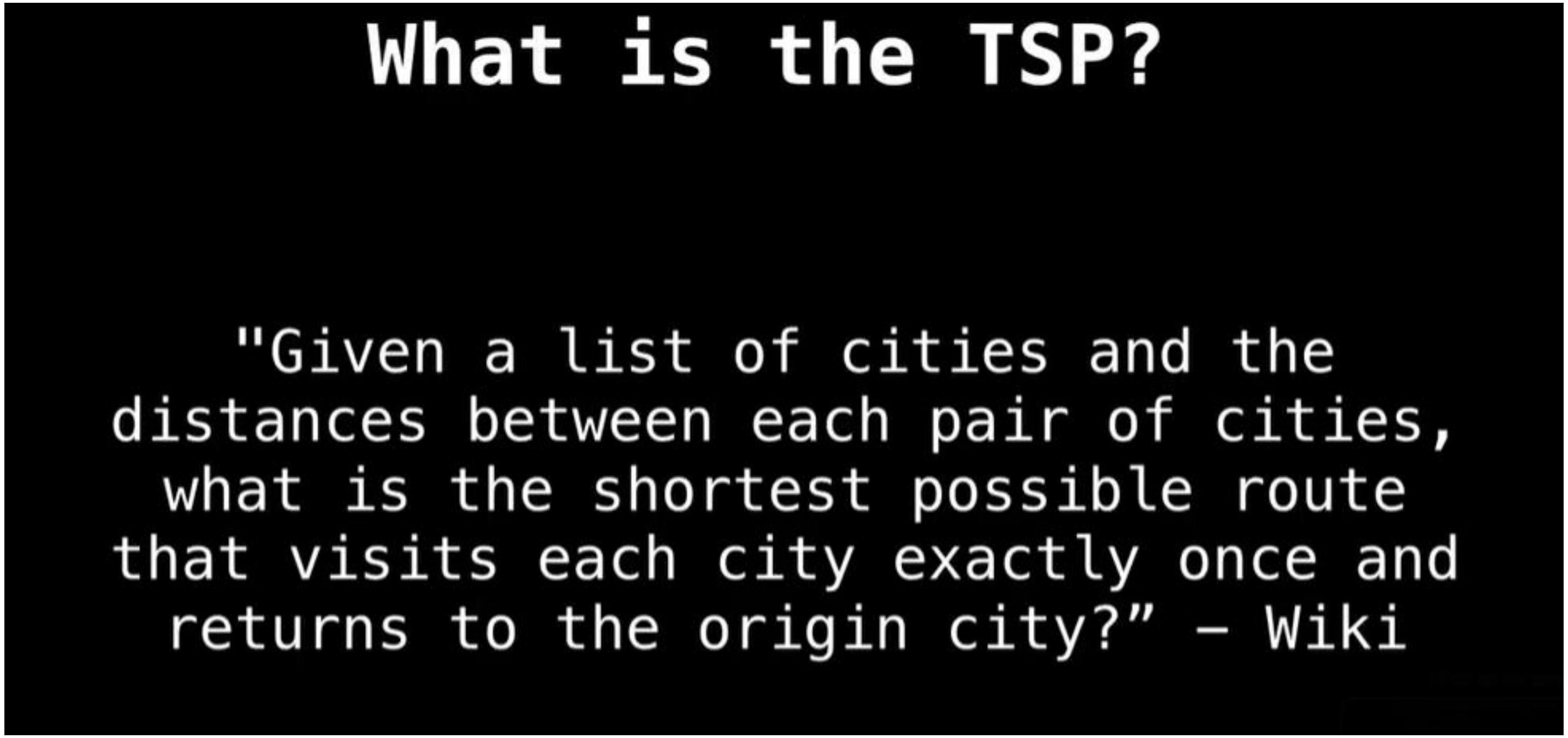
Defining characteristics of dynamic programming problems

It's useful to formally define the characteristics of problems that can be addressed using DP approaches:

Optimal Substructure: If it is possible to break down a given problem into smaller subproblems and an optimal solution to these subproblems exists and contributes to the solution of the given problem, that means an optimal substructure exists for such a problem.

Overlapping subproblems: If the solution to the overall problem requires solving some of the subproblems repeatedly, the problem is said to feature overlapping subproblems and is a good candidate for optimization using DP.

It is *only* if the solution to a problem has these properties that we may use dynamic programming to optimize it.



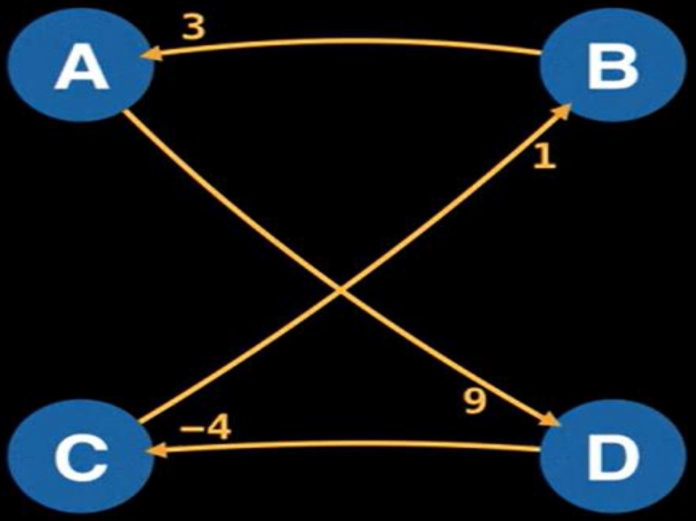
What is the TSP?

"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?

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

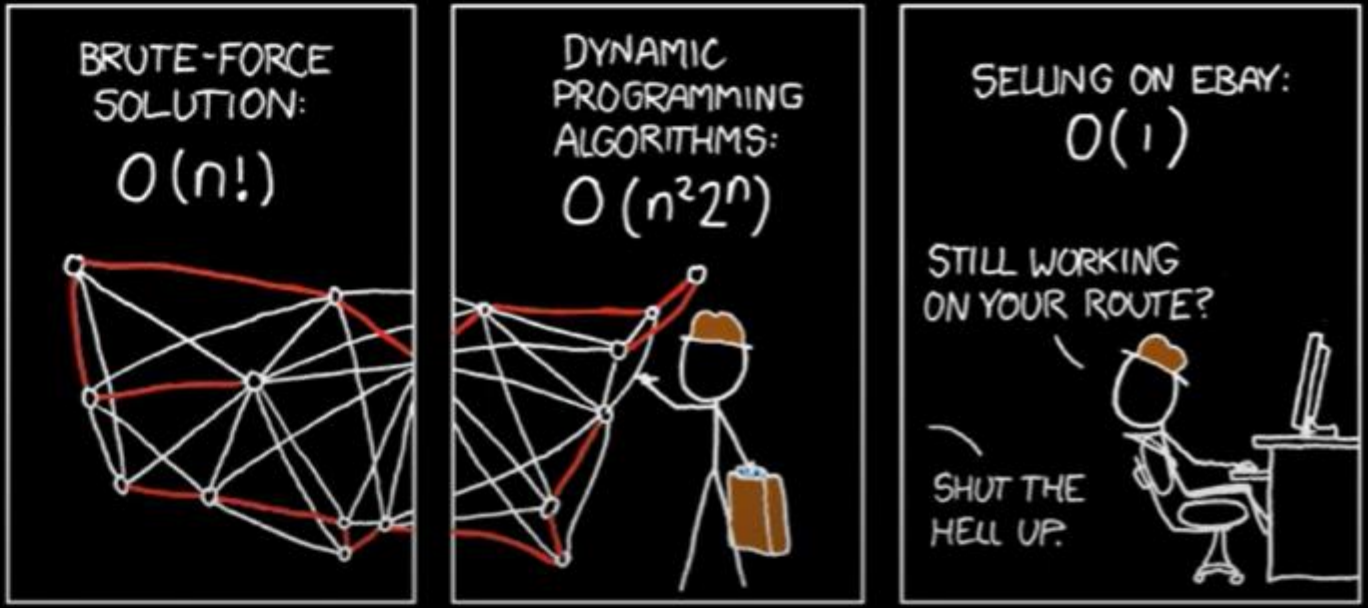


Full tour: A -> D -> C -> B -> A

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.

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

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

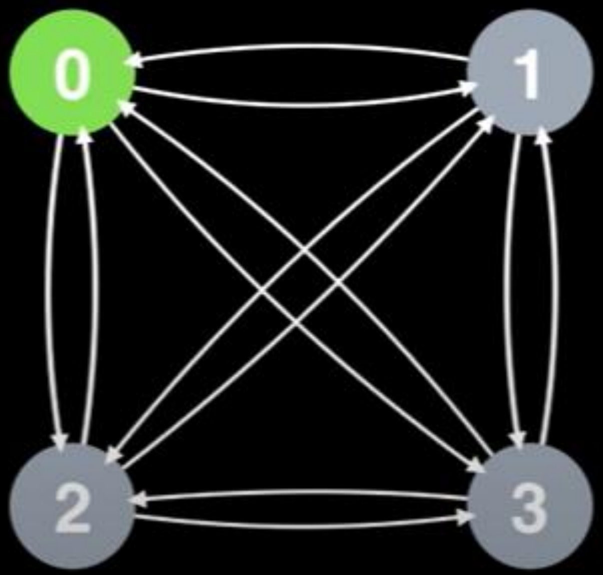
TSP with DP

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**.

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

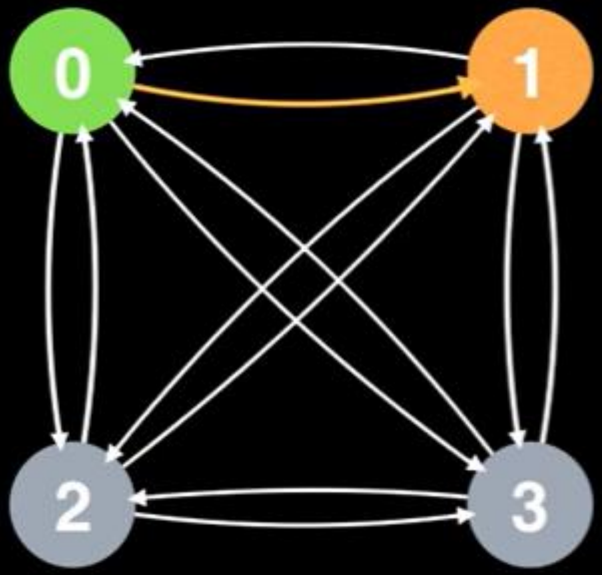
TSP with DP

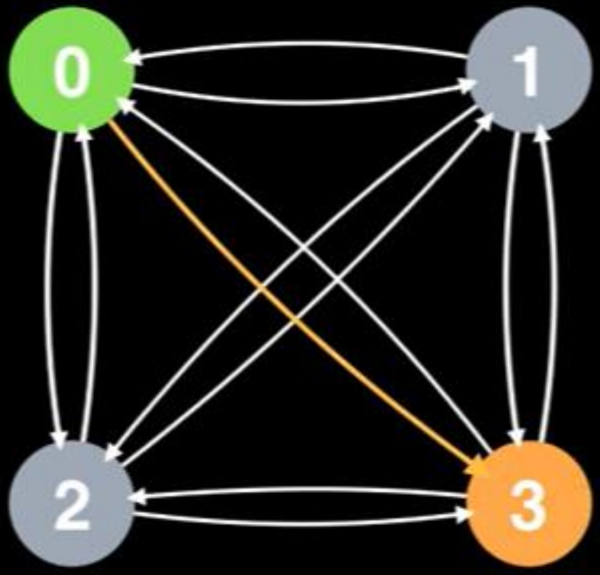
Next, compute and store the optimal value from **S** to each node **X** (≠ **S**). This will solve TSP problem for all paths of length n = 2.

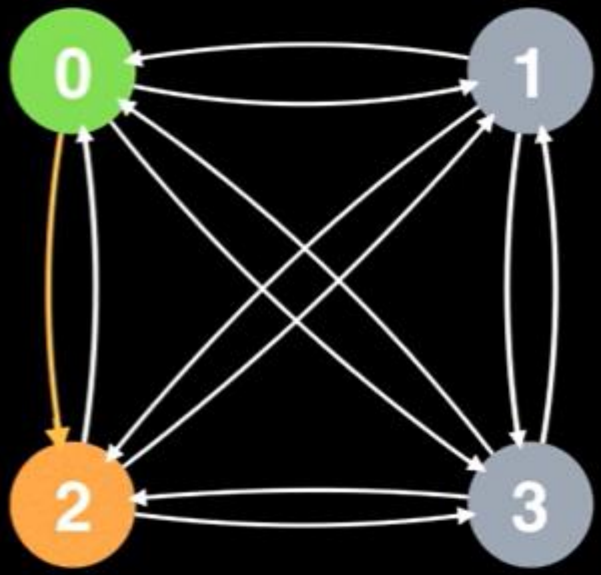


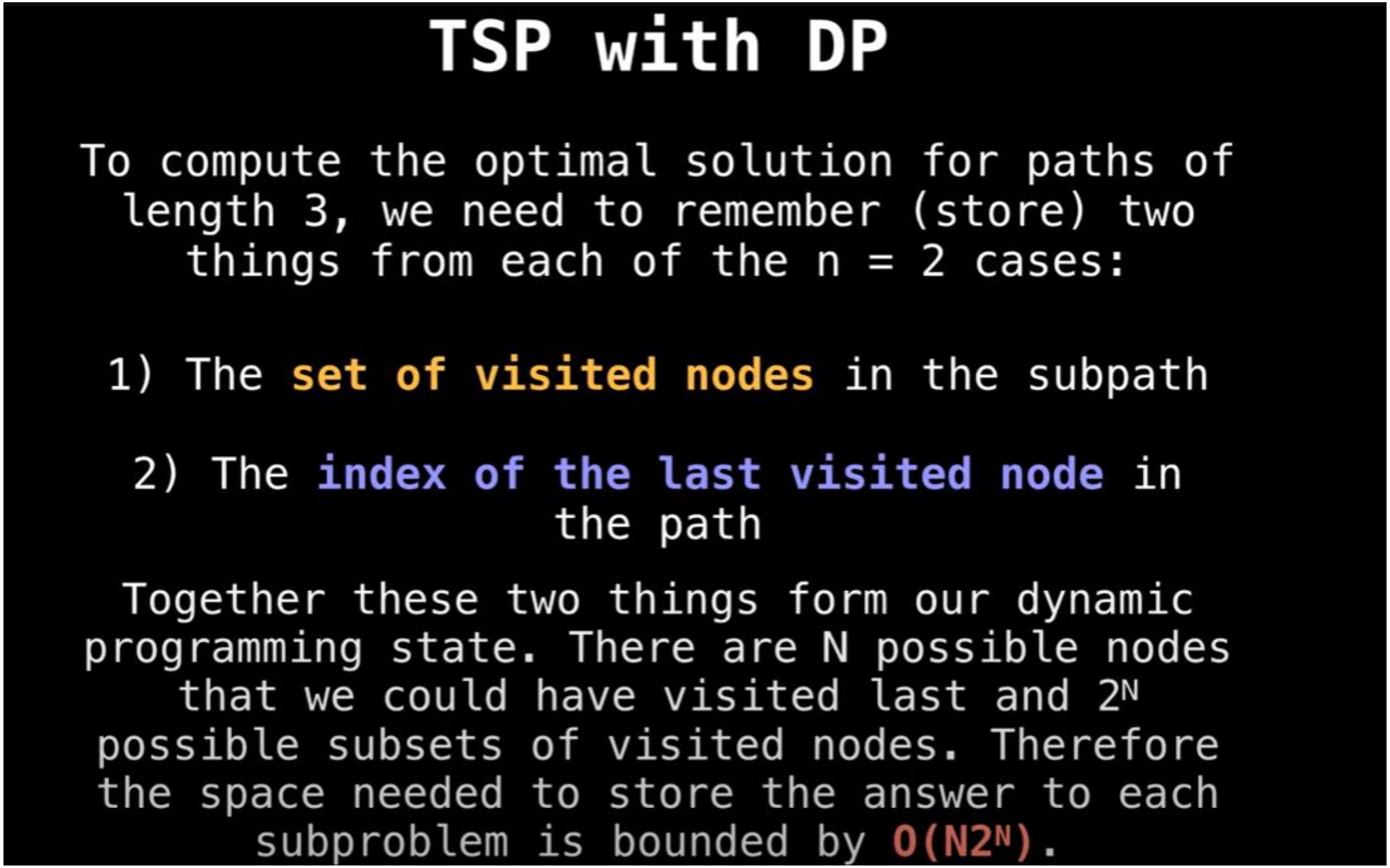
TSP with DP

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

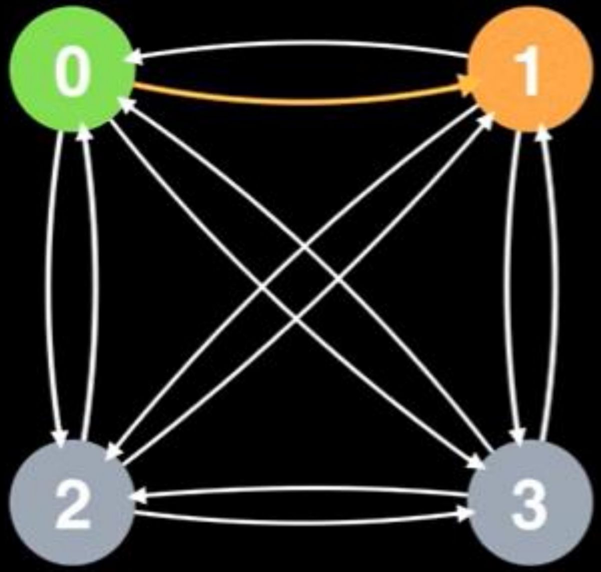
Together these two things form our dynamic programming state. There are N possible nodes that we could have visited last and 2^N possible subsets of visited nodes. Therefore the space needed to store the answer to each subproblem is bounded by $O(N2^N)$.

Visited Nodes as a Bit Field

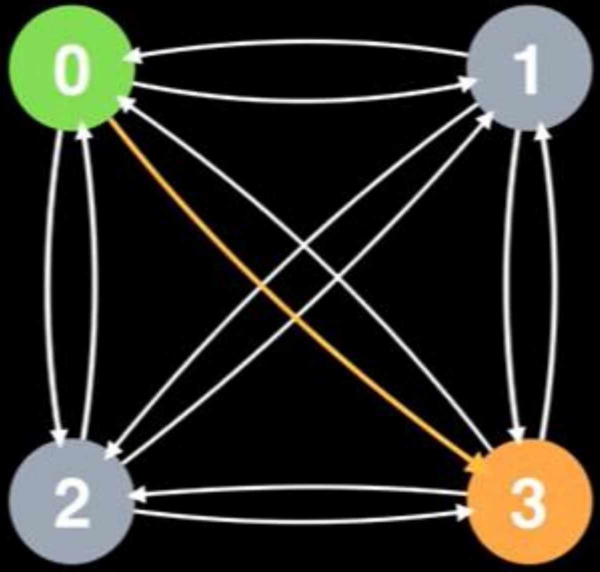
The best way to represent the set of visited nodes is to use **a single 32-bit integer**. A 32-bit int is compact, quick and allows for easy caching in a memo table.

Visited Nodes as a Bit Field

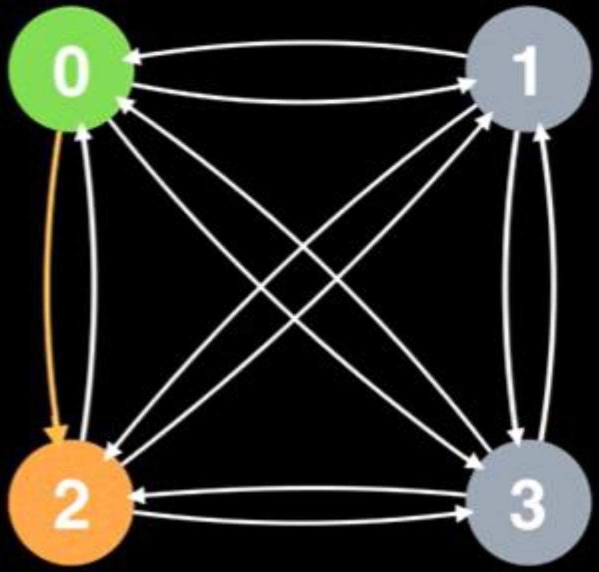
The best way to represent the set of visited nodes is to use **a single 32-bit integer**. A 32-bit int is compact, quick and allows for easy caching in a memo table.



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



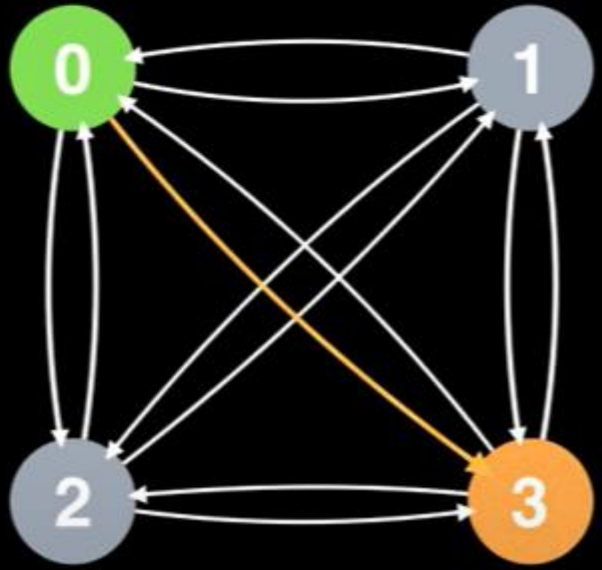
State
Binary rep: $1001_2 = 9$
Last node: 3



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

TSP with DP

To solve $3 \le n \le N$, 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