Computational Complexity

https://cs.pomona.edu/classes/cs140/

P, NP, Completeness, Hardness

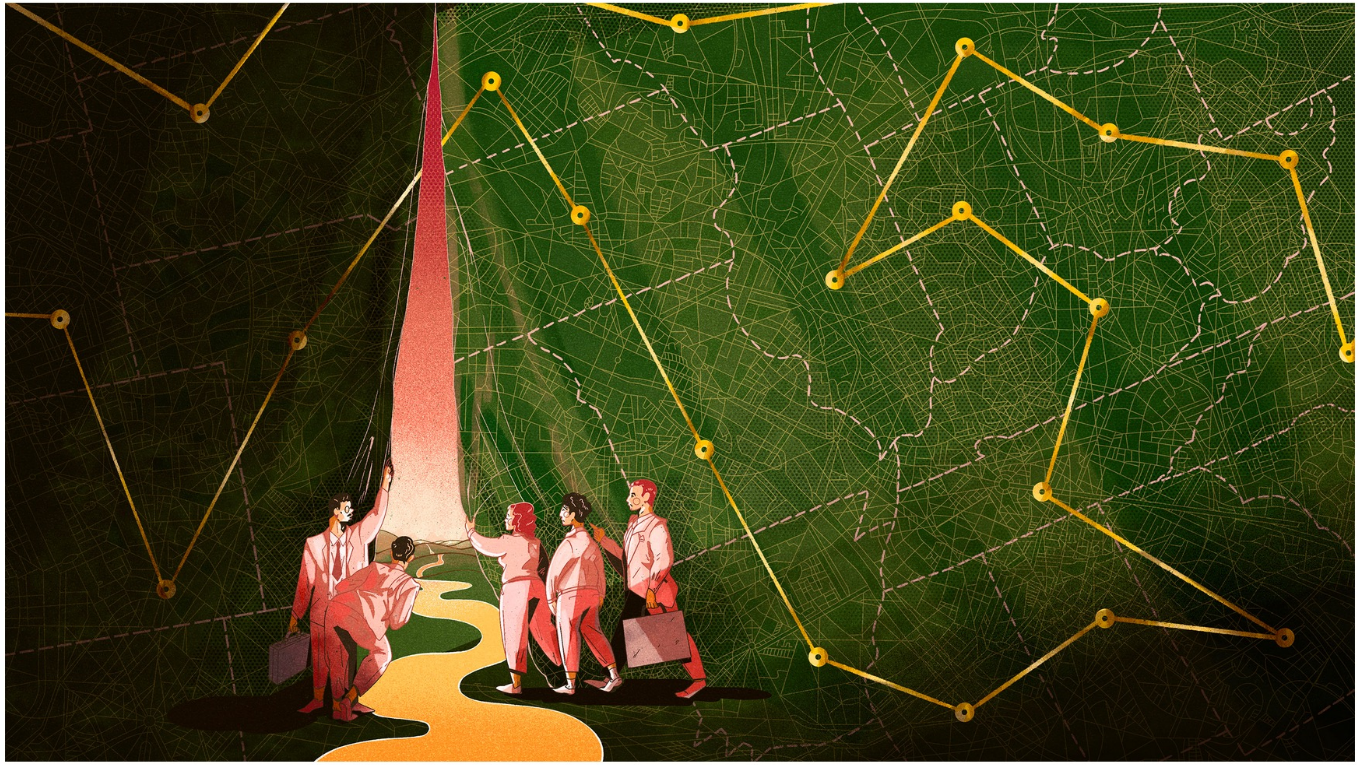
https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008/

ALGORITHMS

Computer Scientists Break Traveling Salesperson Record

19

After 44 years, there's finally a better way to find approximate solutions to the notoriously difficult traveling salesperson problem.



Islenia Mil for Quanta Magazine

Erica Klarreich

Contributing Correspondent

October 8, 2020

When Nathan Klein started graduate school two years ago, his advisers proposed a modest plan: to work together on one of the most famous, long-standing problems in theoretical computer science.

Even if they didn't manage to solve it, they figured, Klein would learn a lot in the process. He went along with the idea. "I didn't know to be intimidated," he said. "I was just a first-year grad student — I don't

ALGORITHMS

Computer Scientists Break Traveling

Computer Scientists Break Traveling Salesperson Record

19

After 44 years, there's finally a better way to find approximate solutions to the notoriously difficult traveling salesperson problem.

Islenia Mil for Quanta Magazine

October 8, 2020

wo years ago, his
ogether on one of
tical computer

Even if they didn't manage to solve it, they figured, Klein would learn a lot in the process. He went along with the idea. "I didn't know to be intimidated," he said. "I was just a first-year grad student — I don't

Correspondent

October 8, 2020

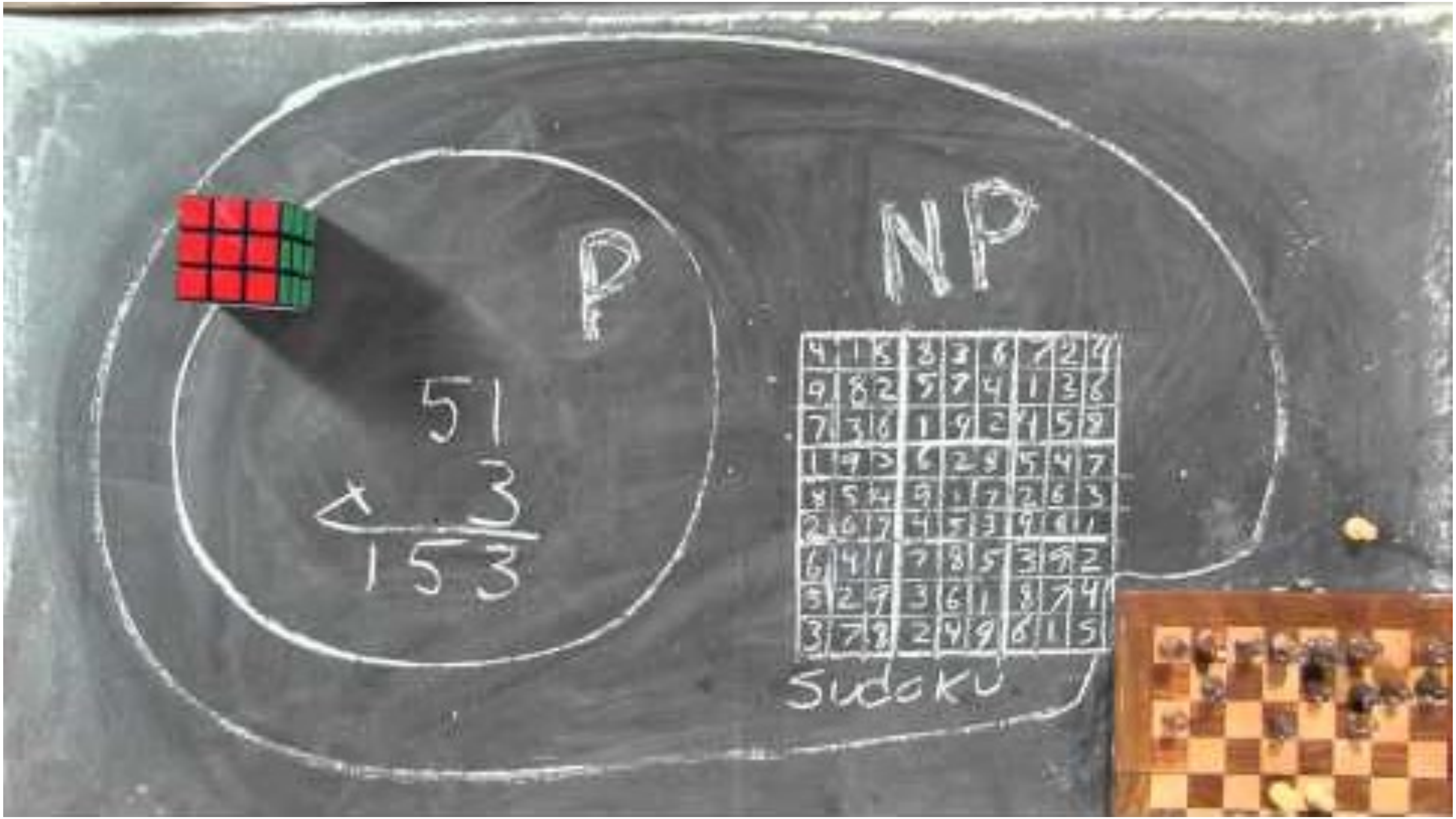
Outline

Topics and Learning Objectives

- Discuss complexity theory
- Discuss common complexity classes (P, NP, NP-Hard, NP-Complete)
- Cover the travelling salesperson problem (TSP)

Exercise

- In slides


P
NP
51
x 3
153
sudoku

Let’s Motivate our NP Discussion

The Traveling Salesperson Problem

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?

- Input: a complete, undirected graph with non-negative edge costs
- Output: a minimum cost tour (a cycle that visits each vertex once)
- Applications?

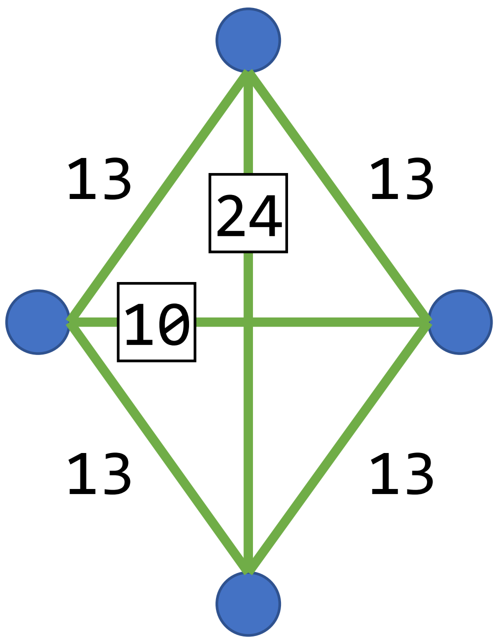
Let’s Motivate our NP Discussion

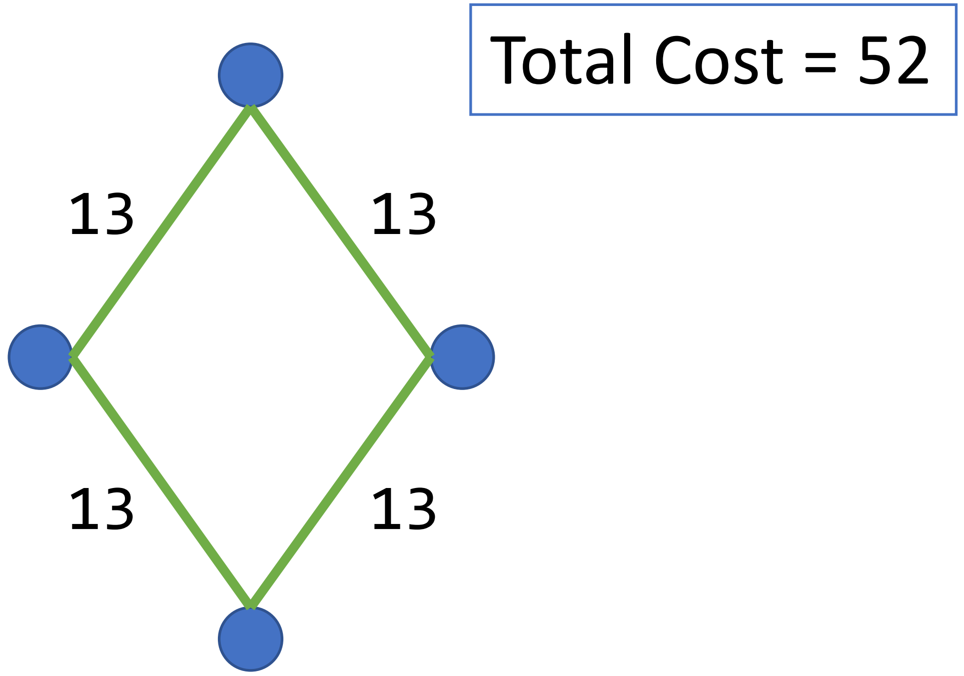
The Traveling Salesperson Problem

- Input: a complete, undirected graph with non-negative edge costs
- Output: a minimum cost tour (a cycle that visits each vertex once)

- What is a naïve solution to this problem?
- Is a greedy solution the optimal solution?
- Is this a good candidate for dynamic programming?

Greedy Traveling Salesperson Problem?





Computational Complexity Classification

Classify problems according to difficulty

- "With respect to input size, these problems take linear time to solve."
- "These problems require quadratic memory when compared to the input size."
- "These problems are <u>hard</u> because they require significant [insert resource]."

Relate classes to one another

- "This class of problems is computationally harder than this other class."

Problems can relate to many things

- Decision problems (output "yes" or "no"), optimization problems (output best solution), function problems (similar to decision, but more complex output)

Types of Problems

We'll focus on two types of problems

1. Optimization (output the optimal answer/solution)
2. Decision (output a "yes" or "no")

Example <u>optimization</u>:

What is minimal spanning tree (MST) for G?

Example <u>decision</u>:

Does a given tree span G with a cost less than k?

Does not require you to solve for such a tree.

Complexity Comparisons

If you want to show that problem A is “easy”, then…

you show how to solve it by turning it into a known “easy” problem B.

If you want to show that problem A is “hard”, then…

you show how it can be used to solve a known “hard” problem B.

These are called reductions and we’ll come back to them later.

P: *is the set of polynomial-time solvable problems*

Most of what we've covered is in the class P

Some things not in P that we've seen:

- Shortest path algorithms that must work with negative cycles
- Algorithms for The Knapsack Problem

Note that:

- Some problems in P are slow to solve (large input or large exponent)
- Some problems <u>not in</u> P are tractable (smaller input or good heuristics)

P : set of problems that are polynomial-time solvable

NP : set of problems that are nondeterministic polynomial-time solvable

Complete : among the hardest problems in a complexity class (like P or NP)

For example: NP-Complete contains the hardest problems in NP

We don’t know the lower bound on the running time for these problems.

Hard : **at least** (can be harder) has hard as everything in some complexity class

For example: NP-Hard contains problems at least as hard as all NP

NP-Hard also contains problems that are harder than those in NP

We are pretty sure (but have not proven) that these problem are not P

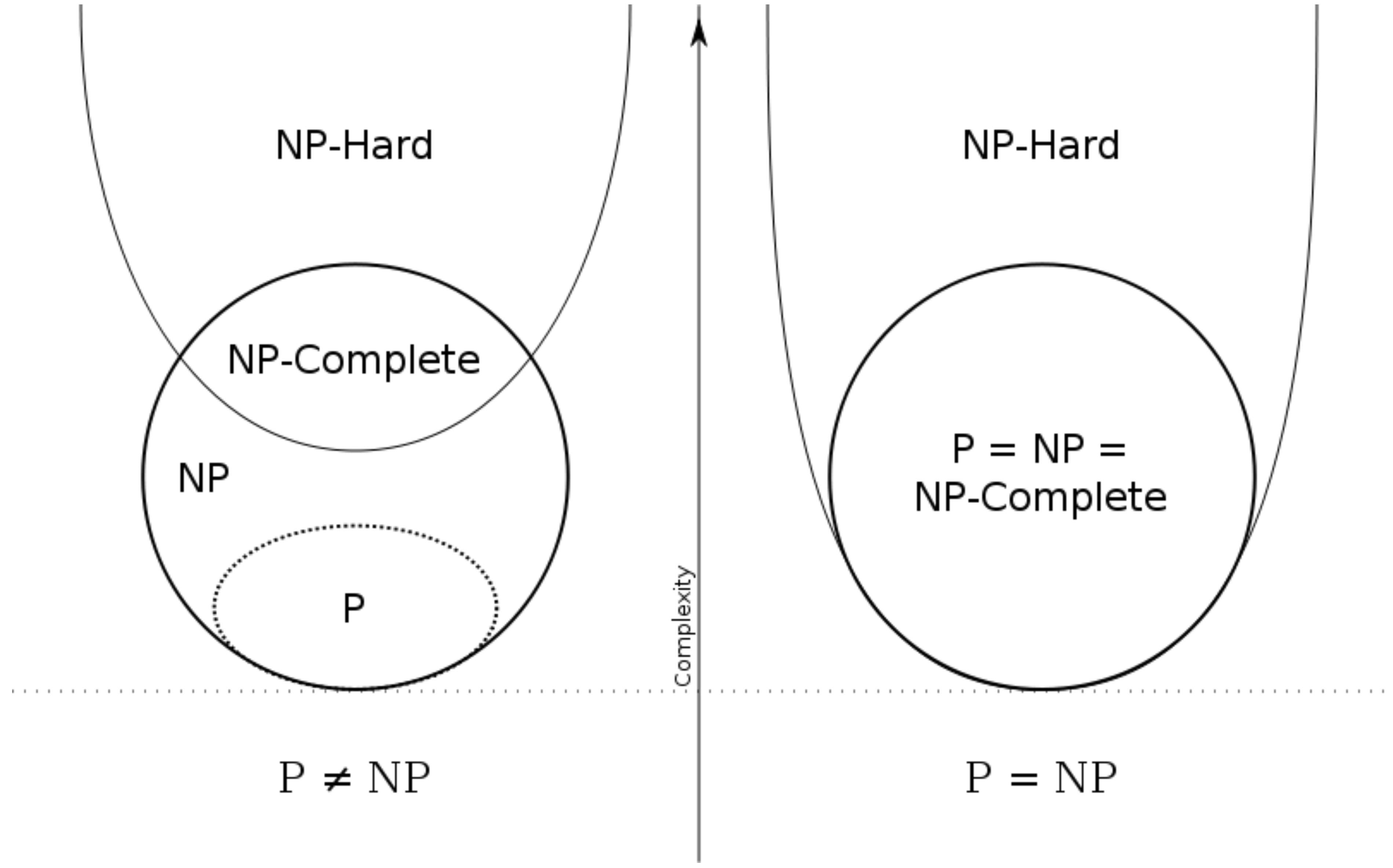
Definition of NP

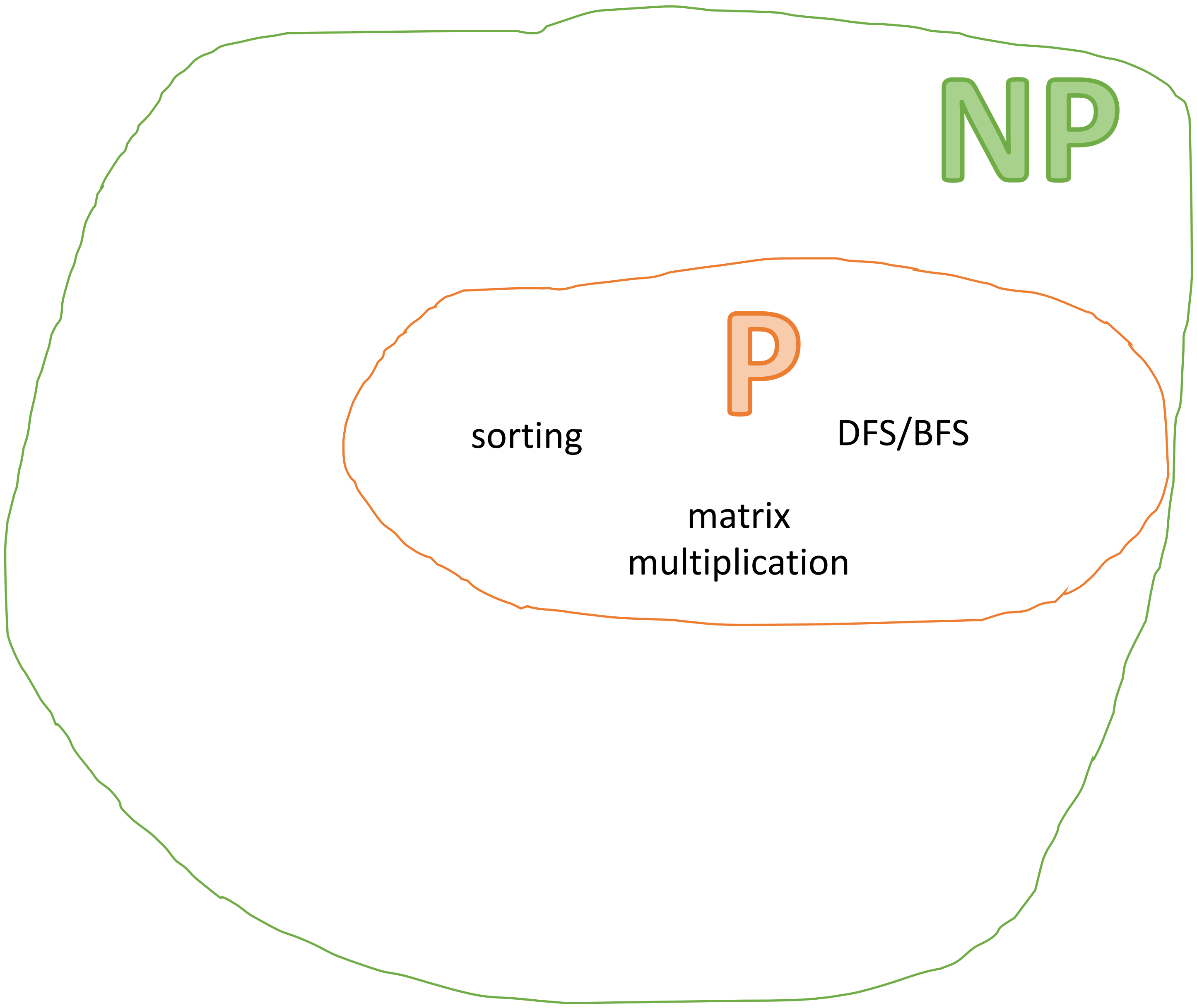
The class of computational problems for which a given solution can be verified as a solution in polynomial time by a deterministic Turing machine (or solvable by a non-deterministic Turing machine in polynomial time).

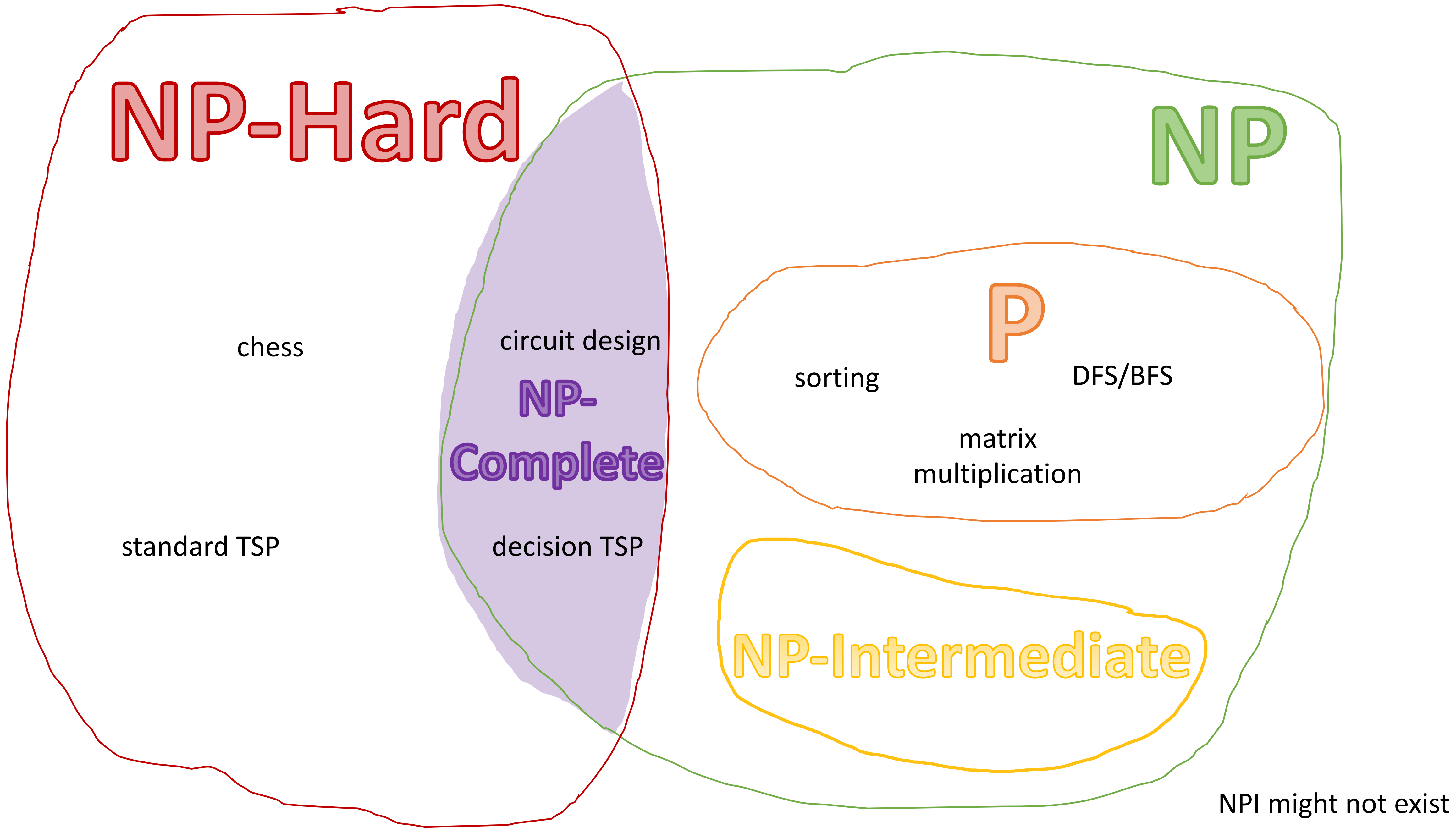
This does **not** imply that you **can or cannot** calculate the solution in polynomial time. We might not have a proof either way.

Some problems can be verified faster than they can be solved.

- Comparison-based sorting: solve in $O(n \lg n)$; verify in $O(n)$


NP-Hard
NP-Complete
NP
P
Complexity
NP-Hard
P = NP =
NP-Complete
P ≠ NP
P = NP


NP
P
sorting
DFS/BFS
matrix
multiplication


NP-Hard
NP
chess
circuit design
NP-
Complete
P
sorting
DFS/BFS
matrix
multiplication
standard TSP
decision TSP
NP-Intermediate
NPI might not exist

Tractability (and intractability)

- A problem is considered tractable if it is polynomial-time solvable.
- A problem is polynomial-time solvable if there is an algorithm that correctly solves it in $O(n^k)$ time (k is just some constant).
- Typically, we think of k as being 1, 2, 3, or 4. Much higher than that and the problem begins to feel intractable even though it is *technically* polynomial time solvable.

Traveling Salesperson Problem

- How many different tours exist?

n!

52!

Traveling Salesperson Problem

- How many different tours exist? $n!$

- This problem has been extensively studied by many of the most well-known computer scientists since the late 1950s.
- **We do not know if a polynomial time algorithm exists for TSP.**

- In 1965 it was conjectured that no polynomial-time algorithm exists for TSP.
- This conjecture is part of what motived the need for computation complexity classifications.

- We have found an exponential-time algorithm for solving the problem.

TSP with Dynamic Programming

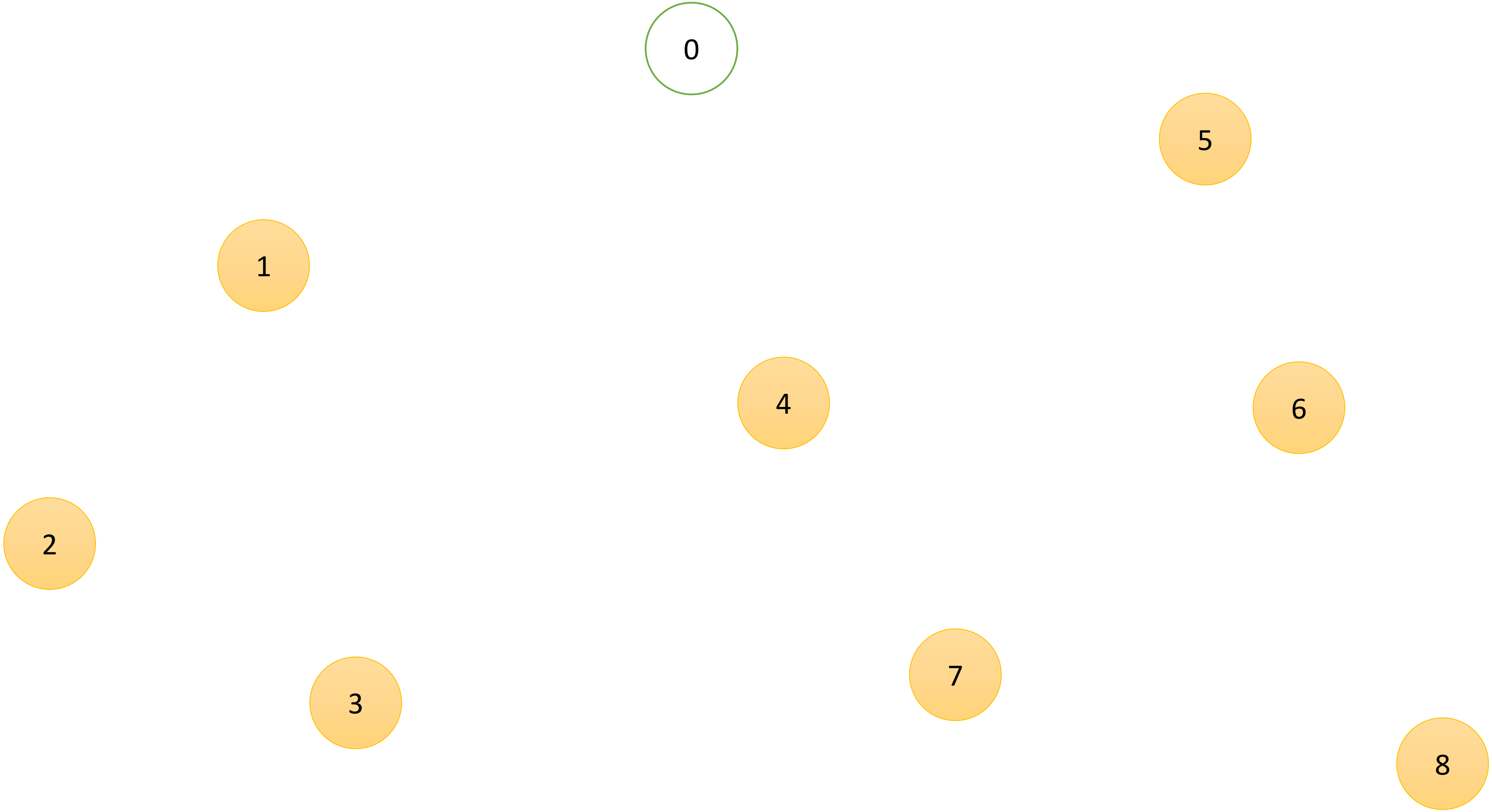
Bellman–Held–Karp Algorithm

- Compute optimal solution for n nodes using optimal solution with n - 1 nodes

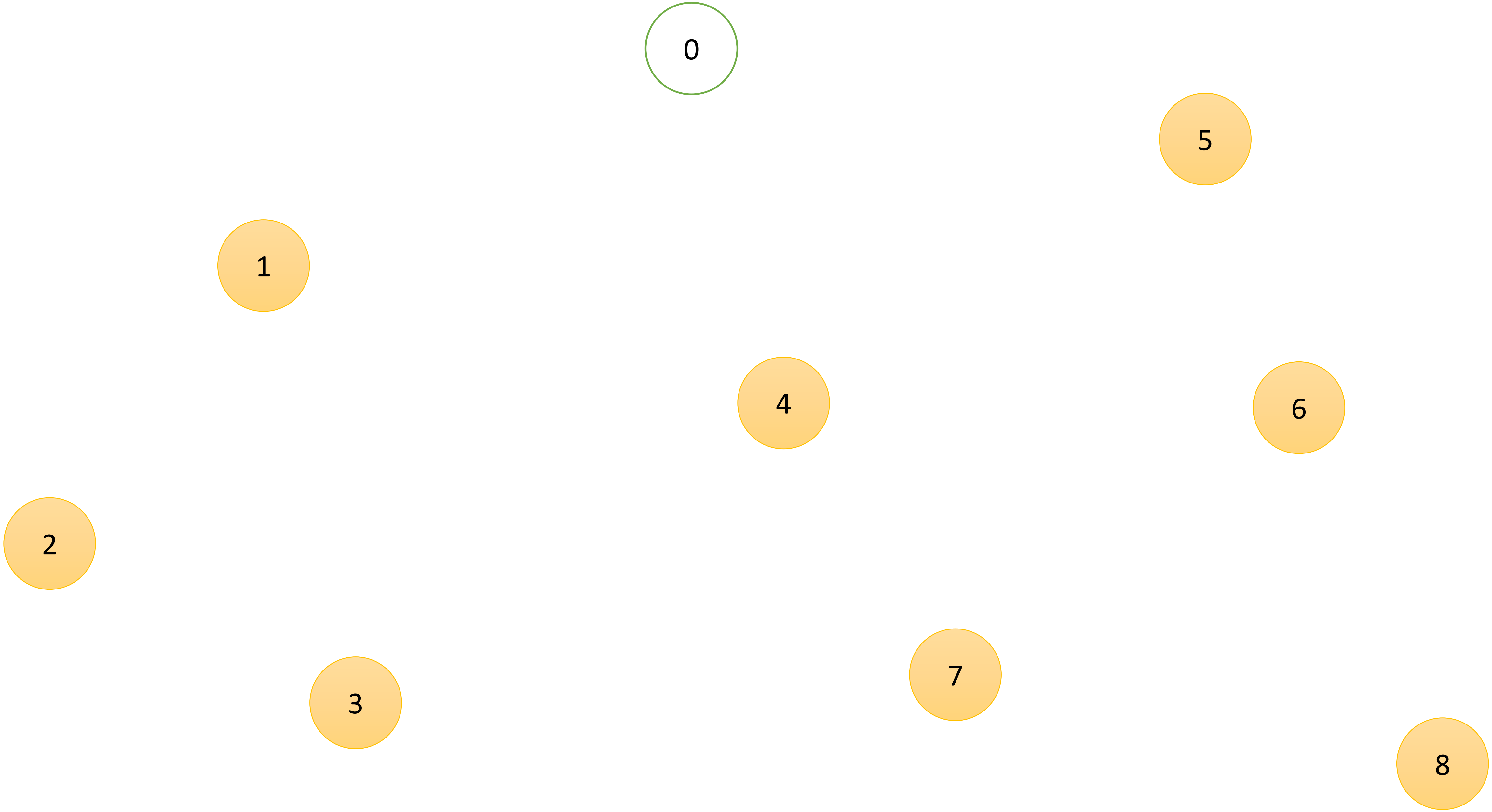
1. Pick a starting node S
2. Find all optimal paths that include S and <u>one</u> other node
3. Find all optimal paths that include S and <u>two</u> other nodes
4. …

- Similar to Bellman-Ford single-source shortest path algorithm

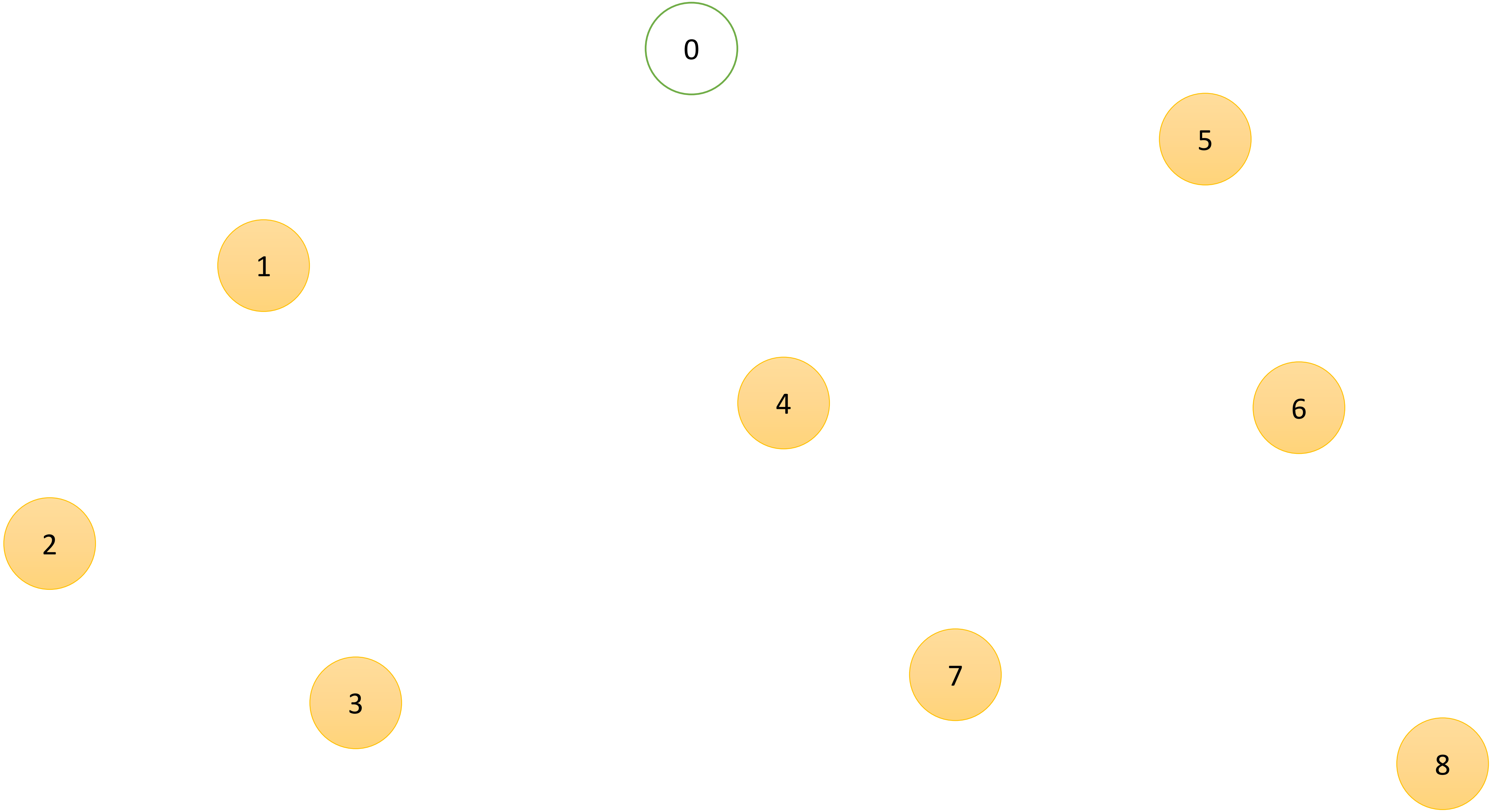
Shortest path with S and 1 other node ending at the other node



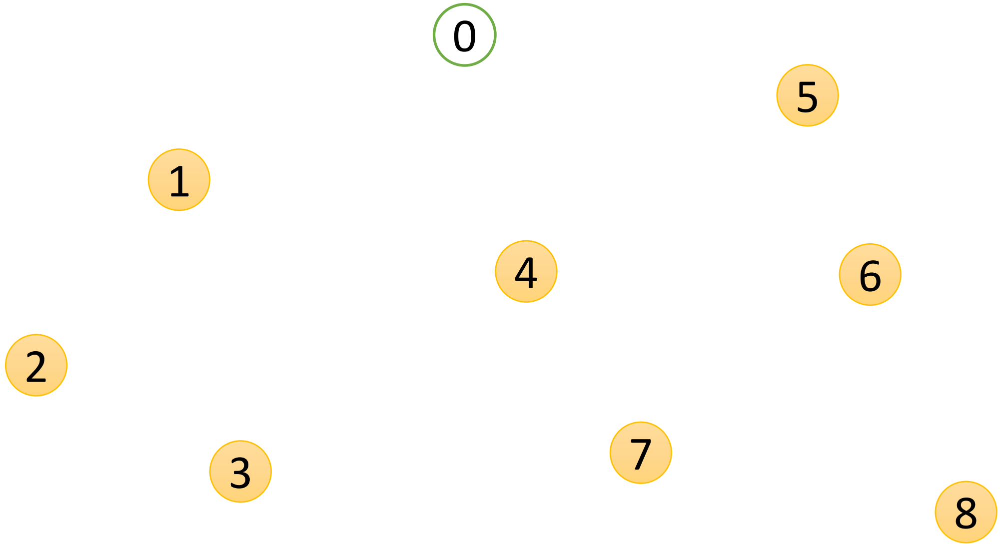
Shortest path with S and 2 other nodes ending at each of the other nodes



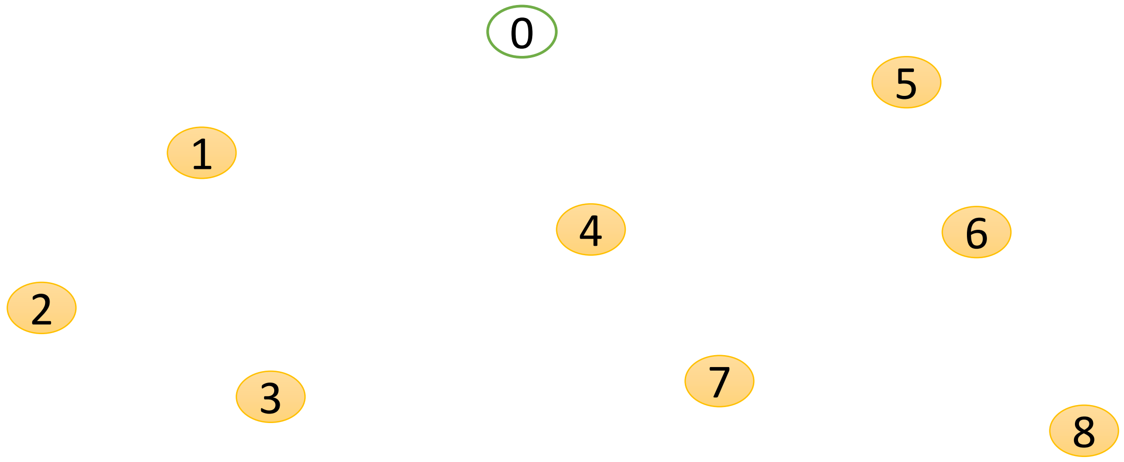
Shortest path with S and n-1 other nodes ending at each of the other nodes



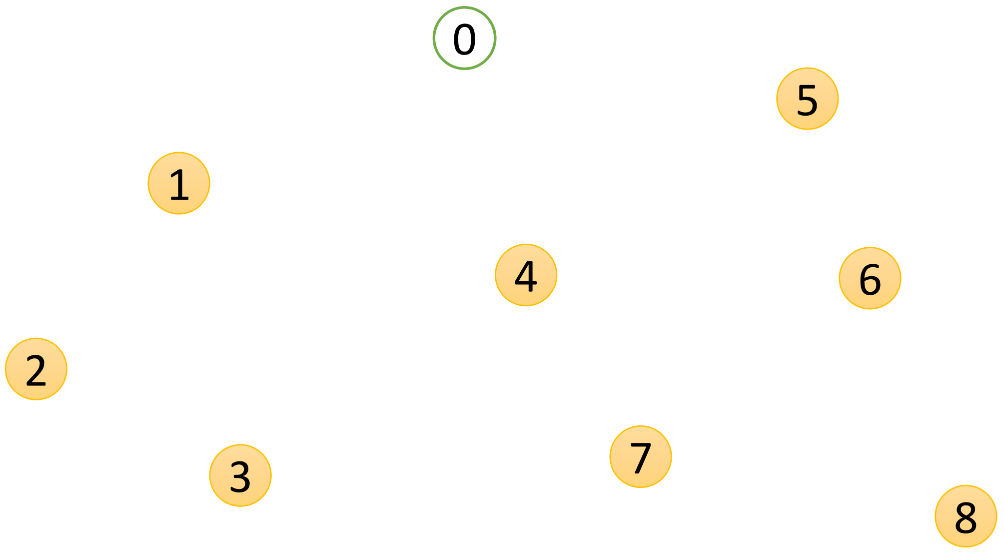
```
FUNCTION BellmanHeldKarp(G)
    n = G.vertices.length
    # Compute all pairwise Euclidean distances between vertices
    dists = ComputeDistances(G)

    # Create and initialize a two-dimensional cost matrix
    # n   : final vertex
    # 2^n : different sets of vertices (a powerset)
    costs = Matrix(n, 2^n)
    # Let's use 0 as the start vertex
    FOR v IN [1 ..< n]
        costs(v, {0, v}) = dists(0, v)
```



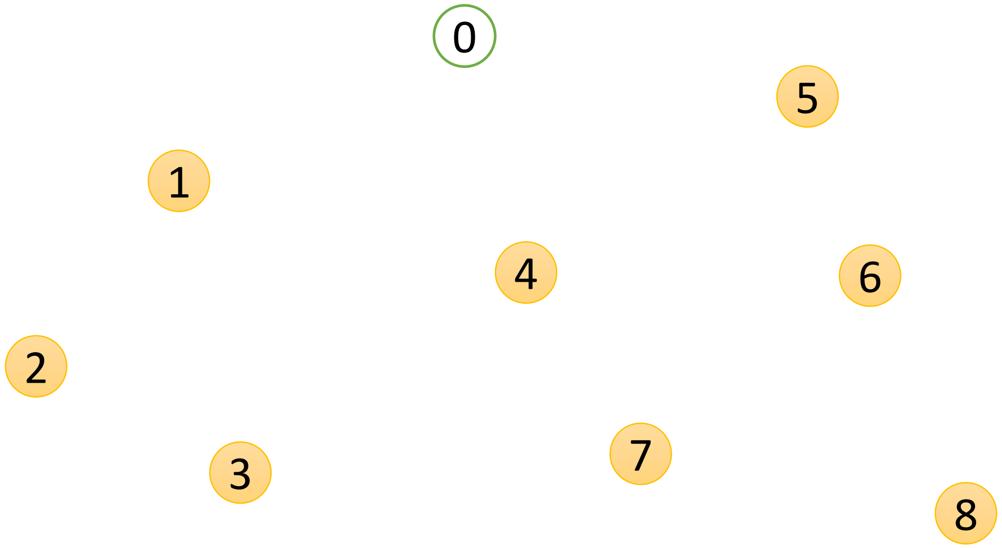
```
FUNCTION BellmanHeldKarp(G)
   n = G.vertices.length
   # Compute all pairwise Euclidean distances between vertices
   dists = ComputeDistances(G)

   # Create and initialize a two-dimensional cost matrix
   # n   : final vertex
   # 2^n : different sets of vertices (a powerset)
   costs = Matrix(n, 2^n)
   # Let's use 0 as the start vertex
   FOR v IN [1 ..< n]
      costs(v, {0, v}) = dists(0, v)

   # Compute paths for all possible subsets of vertices
   other_vertices = G.vertices - {0}
   FOR size IN [2 ..<= n]
      FOR subset IN PowerSet(other_vertices, size)
         FOR next IN subset
            min_cost = INFINITY
            state = subset - {next}
            FOR end IN state
               new_cost = costs(end, state) + dists(end, next)
               IF new_cost < min_cost
                  min_cost = new_cost
            costs(next, subset + {0}) = min_cost
```



```
FUNCTION BellmanHeldKarp(G)
  n = G.vertices.length
  # Compute all pairwise Euclidean distances between vertices
  dists = ComputeDistances(G)

  # Create and initialize a two-dimensional cost matrix
  # n   : final vertex
  # 2^n : different sets of vertices (a powerset)
  costs = Matrix(n, 2^n)
  # Let's use 0 as the start vertex
  FOR v IN [1 ..< n]
    costs(v, {0, v}) = dists(0, v)

  # Compute paths for all possible subsets of vertices
  other_vertices = G.vertices - {0}
  FOR size IN [2 ..<= n]
    FOR subset IN PowerSet(other_vertices, size)
      FOR next IN subset
        min_cost = INFINITY
        state = subset - {next}
        FOR end IN state
          new_cost = costs(end, state) + dists(end, next)
          IF new_cost < min_cost
            min_cost = new_cost
        costs(next, subset + {0}) = min_cost

  # Grab the cheapest tour
  min_tour_cost = INFINITY
  FOR end IN [1 ..< n]
    tour_cost = costs(end, G.vertices) + dists(end, 0)
    IF tour_cost < min_tour_cost
      min_tour_cost = tour_cost
```



```
FUNCTION BellmanHeldKarp(G)
    n = G.vertices.length
    # Compute all pairwise Euclidean distances between vertices
    dists = ComputeDistances(G)

    # Create and initialize a two-dimensional cost matrix
    # n   : final vertex
    # 2^n : different sets of vertices (a powerset)
    costs = Matrix(n, 2^n)
    # Let's use 0 as the start vertex
    FOR v IN [1 ..< n]
        costs(v, {0, v}) = dists(0, v)

    # Compute paths for all possible subsets of vertices
    other_vertices = G.vertices - {0}
    FOR size IN [2 ..<= n]
        FOR subset IN PowerSet(other_vertices, size)
            FOR next IN subset
                min_cost = INFINITY
                state = subset - {next}
                FOR end IN state
                    new_cost = costs(end, state) + dists(end, next)
                    IF new_cost < min_cost
                        min_cost = new_cost
                costs(next, subset + {0}) = min_cost

    # Grab the cheapest tour
    min_tour_cost = INFINITY
    FOR end IN [1 ..< n]
        tour_cost = costs(end, G.vertices) + dists(end, 0)
        IF tour_cost < min_tour_cost
            min_tour_cost = tour_cost

    # Compute the tour by back tracking through costs
    min_tour = ComputeTour(G, costs, dists)

    RETURN min_tour_cost, min_tour
```



```
FUNCTION BellmanHeldKarp(G)
   n = G.vertices.length
   # Compute all pairwise Euclidean distances between vertices
   dists = ComputeDistances(G)
```

$O(n^2)$

```
   # Create and initialize a two-dimensional cost matrix
   # n   : final vertex
   # 2^n : different sets of vertices (a powerset)
   costs = Matrix(n, 2^n)
   # Let's use 0 as the start vertex
   FOR v IN [1 ..< n]
      costs(v, {0, v}) = dists(0, v)
```

$O(n)$

```
   # Compute paths for all possible subsets of vertices
   other_vertices = G.vertices - {0}
   FOR size IN [2 ..<= n]
      FOR subset IN PowerSet(other_vertices, size)
         FOR next IN subset
            min_cost = INFINITY
            state = subset - {next}
            FOR end IN state
               new_cost = costs(end, state) + dists(end, next)
               IF new_cost < min_cost
                  min_cost = new_cost
            costs(next, subset + {0}) = min_cost
```

$O(2^n)$

$O(n)$

$O(n)$

```
   # Grab the cheapest tour
   min_tour_cost = INFINITY
   FOR end IN [1 ..< n]
      tour_cost = costs(end, G.vertices) + dists(end, 0)
      IF tour_cost < min_tour_cost
         min_tour_cost = tour_cost
```

$O(n)$

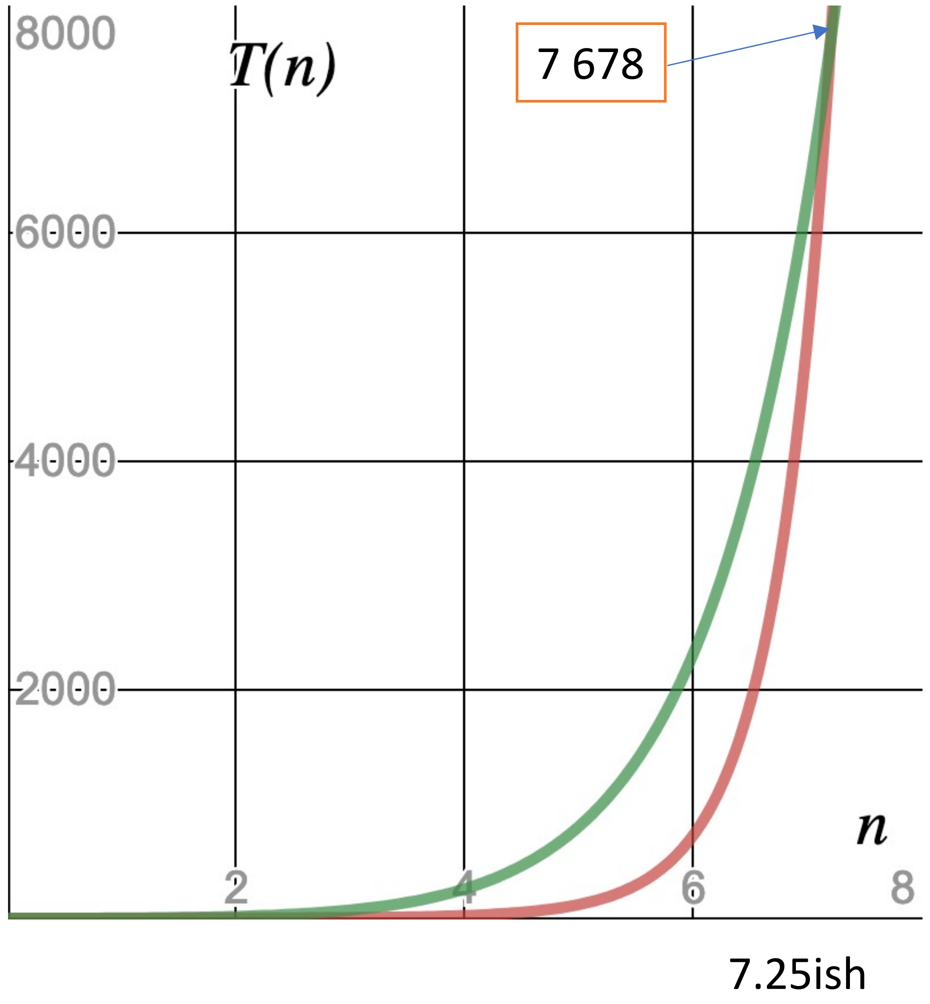
```
   # Compute the tour by back tracking through costs
   min_tour = ComputeTour(G, costs, dists)
```

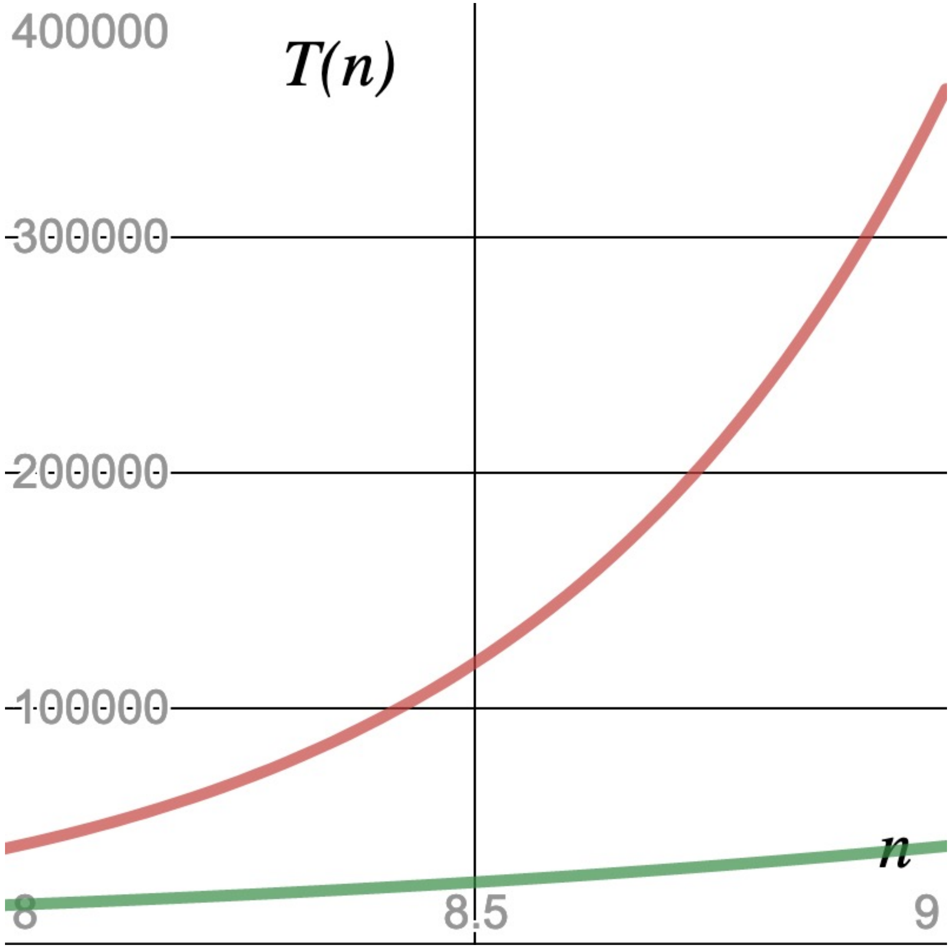
$O(n^2)$

```
   RETURN min_tour_cost, min_tour
```

Total Running Time of $O(n^2 2^n)$

n! vs $n^2 2^n$


8000
6000
4000
2000
T(n)
7 678
2
4
6
8
n
7.25ish


400000
300000
200000
100000
T(n)
8
8.5
9
n

Solving the TSP

- There are n! total possible tours.

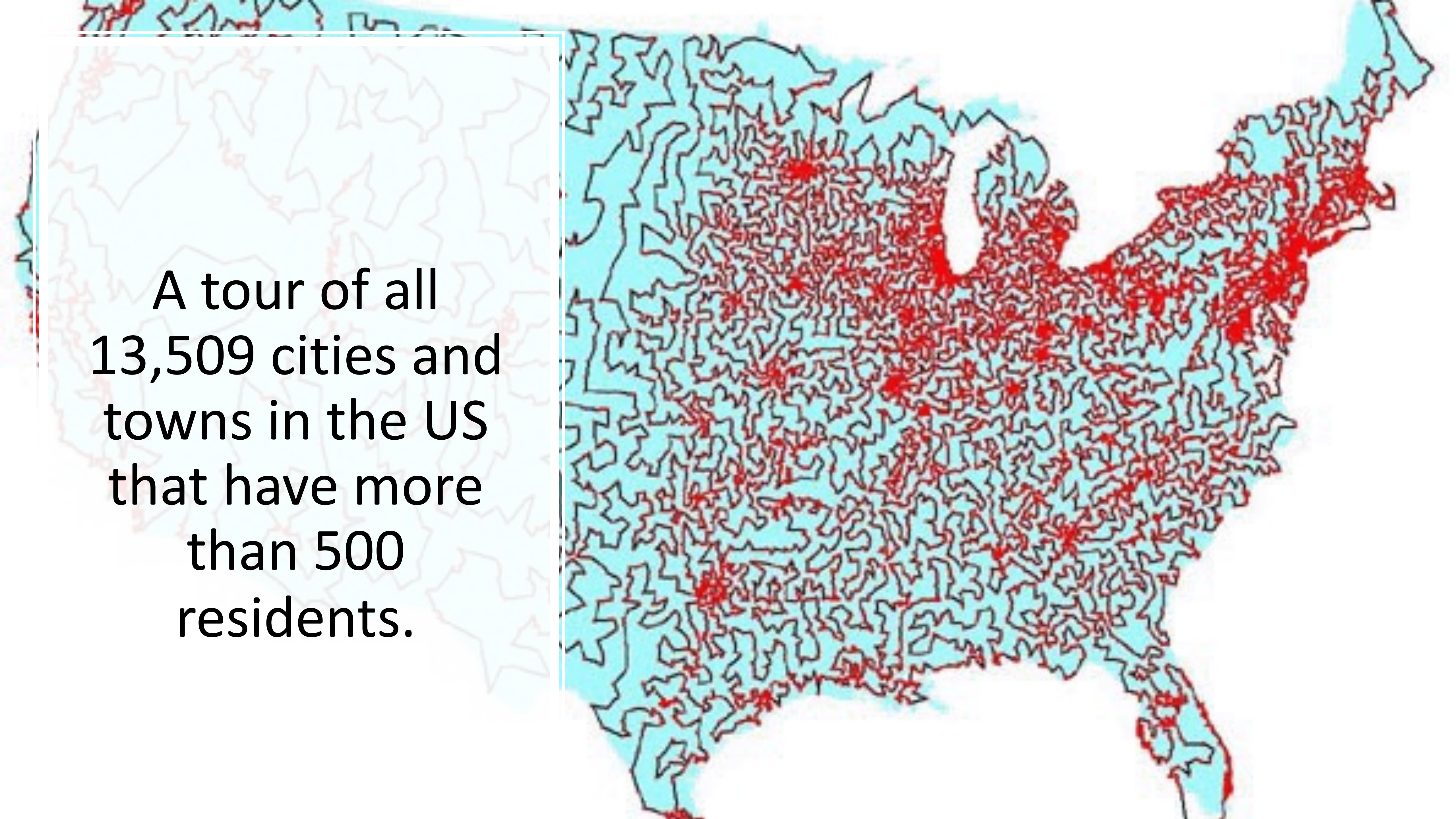
Input Size	Brute-Force n!	Exponential $O(n^2 2^n)$
14	87 billion …	3 million …
15	1 trillion …	7 million …
16	20 trillion …	16 million …
30	265 nonillion …	966 billion …

Solving the TSP

- There are n! total possible

Input Size	Brute	
14	87 billion 178 million …	~ 3 million
15	1 trillion 307 billion …	~ 7 million
16	20 trillion 922 billion …	~ 16 million …
30	265 nonillion 252 octillion 859 septillion 812 sextillion 191 quintillion 58 quadrillion 636 trillion …	~ 966 billion 367 million …

What happens we we need to optimize deliveries to 1,000 or 10,000 cities?



A tour of all 13,509 cities and towns in the US that have more than 500 residents.

TSP

What is the length of a solution to the TSP problem?

n

How long does it take to verify the solution?

In order to check that a proposed tour is a solution of the TSP we need to check *two things*, namely

1. That each city is is visited only once
2. That there is no shorter tour than the one we are checking

Nobody has found a way to do this in polynomial time!

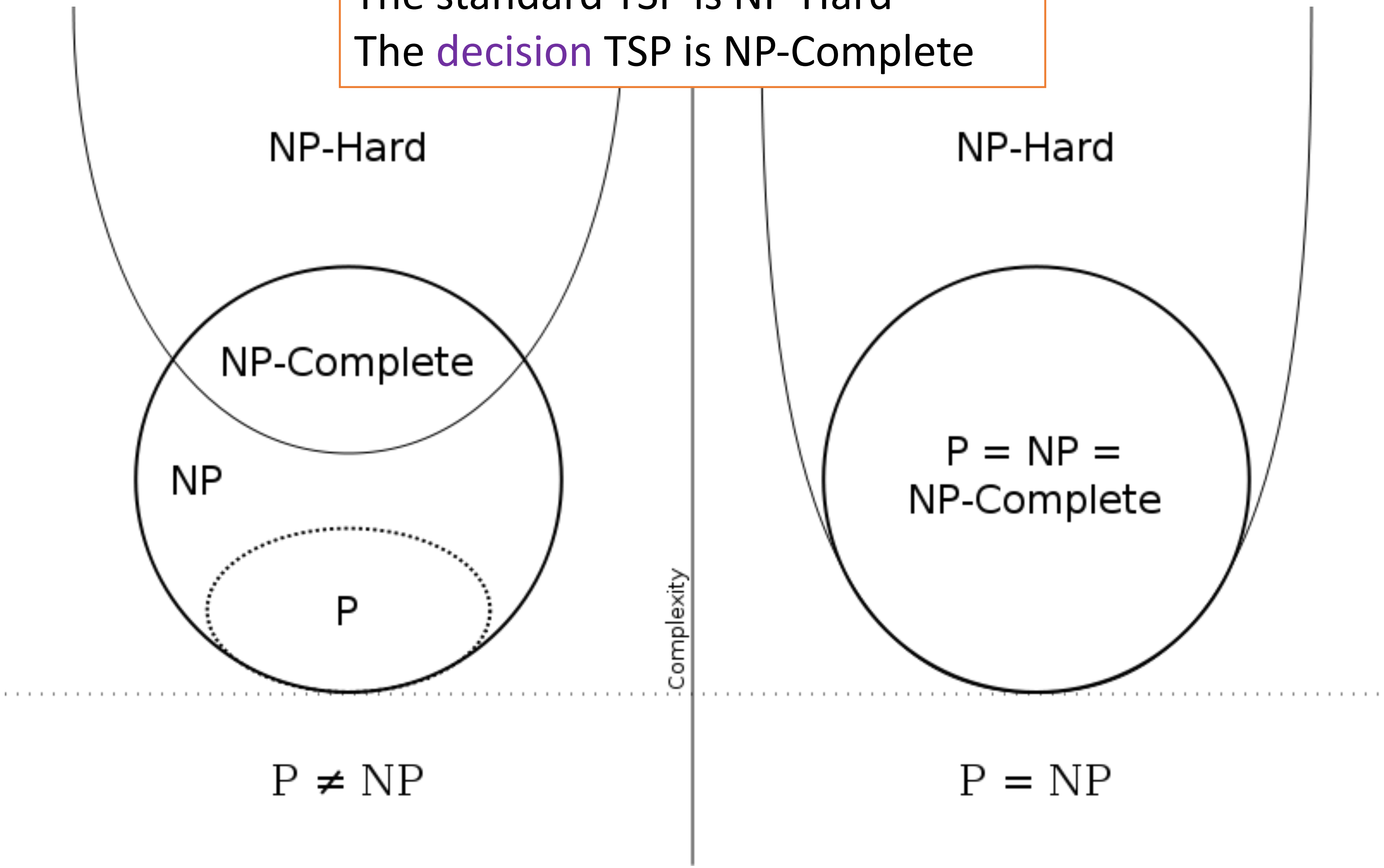
Modified TSP

How long does it take to verify the solution to this altered version:

- Given the output tour T and some total length L
- Is T a tour with a total length less than L?
- <u>This is called the Decision TSP.</u>

- The standard TSP is NP-Hard. (it might be or might not be NP)
- The decision TSP is NP-Complete. (definitely NP, might be P if P = NP)
- Note: there are several other formulations of the TSP problem.

The standard TSP is NP-Hard
The decision TSP is NP-Complete



NP

- Some problems in NP can be solved by a brute-force algorithm in exponential time.
- Some problems in NP cannot be solved in exponential time.
- The vast majority of all computational problems are NP-Complete.
- A polynomial-time solution for any NP-Complete problem gives a polynomial time solution to all NP-Complete Problems.
- This would imply that P = NP
- Our world would change overnight if P = NP.
- We might not know the answer to P = NP or P ≠ NP for a long time.

Recap: NP

A problem is NP if one can easily (in polynomial time) check that a proposed solution is indeed a solution.

A problem is NP hard if it is at least as difficult as any NP problem.

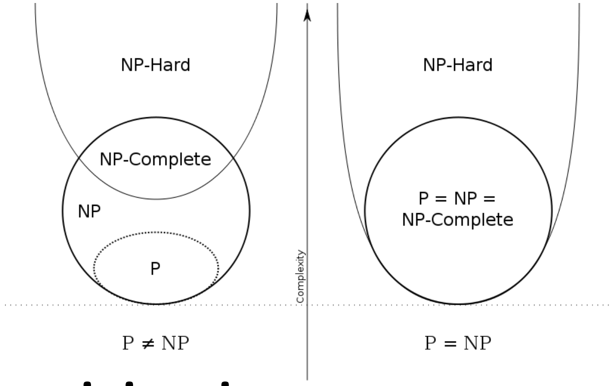
A problem is NP complete if it is both NP and NP hard.

Process for proving a problem is NP-Complete

1. Find a known NP-Complete Problem P1
2. Prove that P1 reduces to your problem P2

- This implies that P2 is at least as hard as P1 (P1 might be easier)
- And since P1 is NP-Complete, P2 must be at least NP-Hard
- If a solution to P2 can be verified in polynomial time, then P2 is also in NP
- Thus, P2 is NP-Complete

NP-Complete Exercise



What do you know about the (NP-Complete) graph partitioning problem?

a. it is in NP-Hard
b. the clique problem (a problem in P) can be reduced to it
c. it is in NP
d. it can be reduced to the SAT problem (an NP-Complete problem)

History Summary

- In roughly 1971-1974, the field of computer science came up with the concept of NP.
- This has a pretty big impact on many fields.
- P is the class of all polynomial-time solvable problems
- NP is the class of all problems whose solutions can be verified in polynomial-time
- It is widely believed that $P \neq NP$
- Though, some expert computer scientists and mathematicians believe that $P = NP$