CS Theory

August 4, 2025

Laryn Qi

Announcements

Theoretical Computer Science

Disclaimer

What is CS Theory?

Theoretical Computer Science explores the theoretical limits of computation.

Its applications span many domains:

- **Cryptography** (Theory + Security)
- **Computational Biology** (Theory + Biology)
- **Quantum Computing** (Theory + Physics)
- **Computational Game Theory** (Theory + Economics)
- **Number Theory** (Theory + Math)

Today (pure theory):

- **Complexity:** How efficiently can this problem be solved with computation?
- **Computability:** Can this problem be solved with computation?

Complexity

Two Sum

Problem: Given a list of N of numbers NUMBERS and a target value TARGET, return whether or not there exists two separate numbers in NUMBERS such that they sum up to TARGET.

```
>>> numbers = [1, 9, 7, 5]
>>> target = 12
>>> two_sum(numbers, target) # 12 = 7 + 5
True
>>> two_sum(numbers, 13) # 13 = 1 + 7 + 5
False
```

Approach 1 (Brute Force):

```
1. for each element in NUMBERS, X
    1. loop over every other element in
       NUMBERS, Y
    2. return TRUE if X + Y = TARGET,
       else continue loop
2. return FALSE
```

Approach 2 (Tracking Seen Values):

```
1. init SEEN <- []
2. for each element in NUMBERS, X
    1. return TRUE if (TARGET - X) in SEEN,
       else append X to the end of SEEN
3. return FALSE
```

Two Sum

Problem: Given a list of N of numbers NUMBERS and a target value TARGET, return whether or not there exists two separate numbers in NUMBERS such that they sum up to TARGET.

Approach 1 (Brute Force):

```
1. for each element in NUMBERS, X
    1. loop over every other element in
       NUMBERS, Y
    2. return TRUE if X + Y = TARGET,
       else continue loop
2. return FALSE
```

Time Efficiency: Quadratic

Approach 2 (Tracking Seen Values):

```
1. init SEEN <- []
2. for each element in NUMBERS, X
    1. return TRUE if (TARGET - X) in SEEN,
       else append X to the end of SEEN
3. return FALSE
```

Time Efficiency: Linear
(assuming we can check membership in constant time)

Time Complexity Class: Polynomial

Space-Time Tradeoff

Two Sum

Problem: Given a list of N of numbers NUMBERS and a target value TARGET, return whether or not there exists two numbers in NUMBERS such that they sum up to TARGET.

Approach 1 (Brute Force):

```
1. for each element in NUMBERS, X
    1. loop over every other element in
       NUMBERS, Y
    2. return TRUE if X + Y = TARGET,
       else continue loop
2. return FALSE
```

Time Efficiency: Quadratic

***Space* Efficiency:** Constant

Approach 2 (Tracking Seen Values):

```
1. init SEEN <- []
2. for each element in NUMBERS, X
    1. return TRUE if (TARGET - X) in SEEN,
       else append X to the end of SEEN
3. return FALSE
```

Time Efficiency: Linear

***Space* Efficiency:** Linear

Subset Sum

Problem: Given a list of N of numbers NUMBERS and a target value TARGET, return whether or not there exists a subset of NUMBERS such that the sum of the subset equals TARGET.

```
>>> numbers = [1, 9, 7, 5]
>>> target = 12
>>> subset_sum(numbers, target) # 12 = 7 + 5
True
>>> subset_sum(numbers, 13) # 13 = 1 + 7 + 5
True
>>> subset_sum(numbers, 18) # 18 = 1 + 7 + 5
False
```

Approach (Brute Force):

1. loop over all 2^n possible subsets of NUMBERS
    1. return TRUE if the sum of the candidate subset equals TARGET,
       else continue loop
2. return FALSE

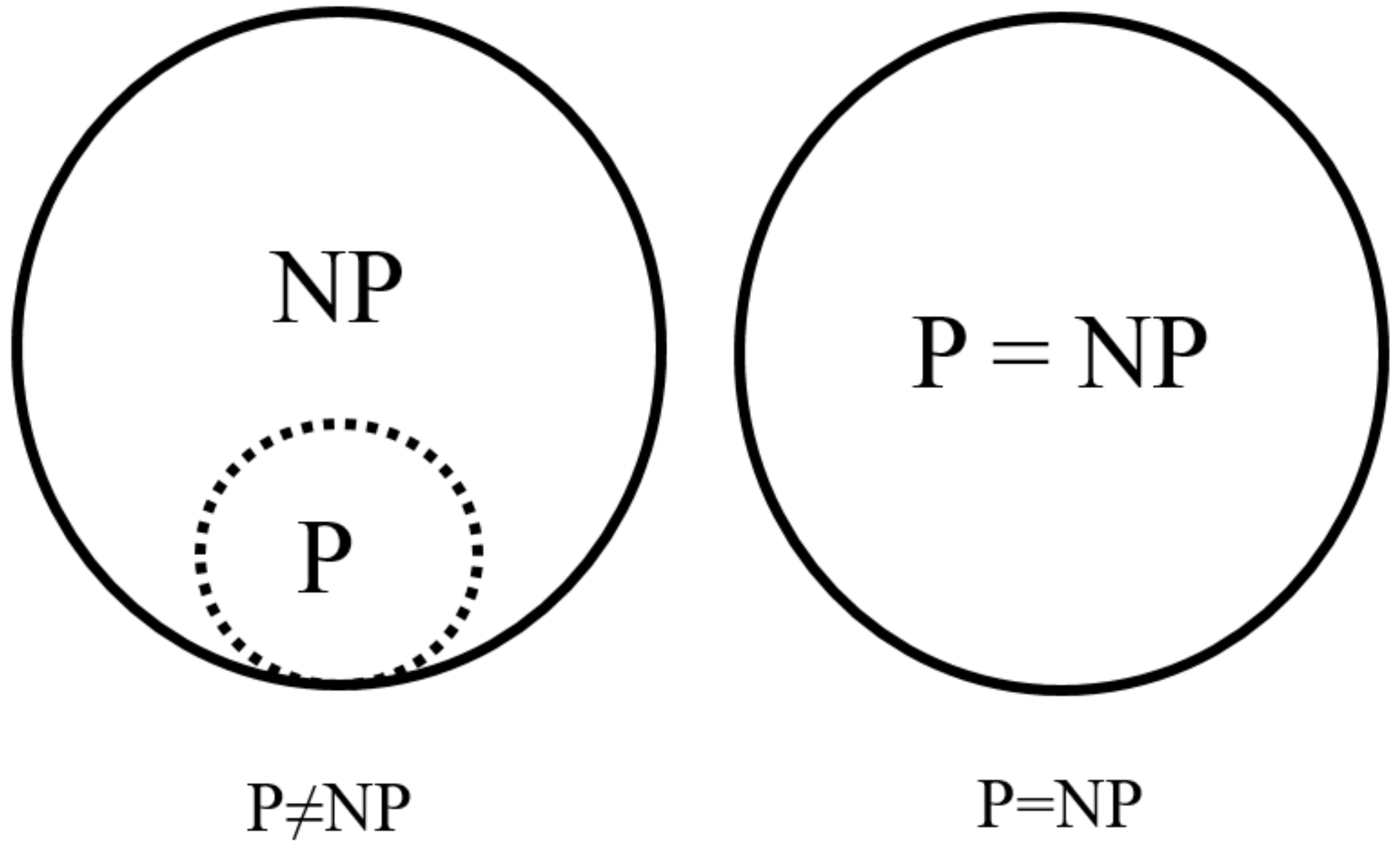
From a complexity theory perspective, Subset Sum cannot be solved in faster than exponential time.

Complexity Class: Non-deterministic Polynomial

P versus NP

P: Polynomial – Problems that can be solved in polynomial time (encapsulates problems that have constant, logarithmic, linear, quadratic, cubic, etc. time algorithms)

NP: Non-deterministic Polynomial – Problems that can be verified in polynomial time

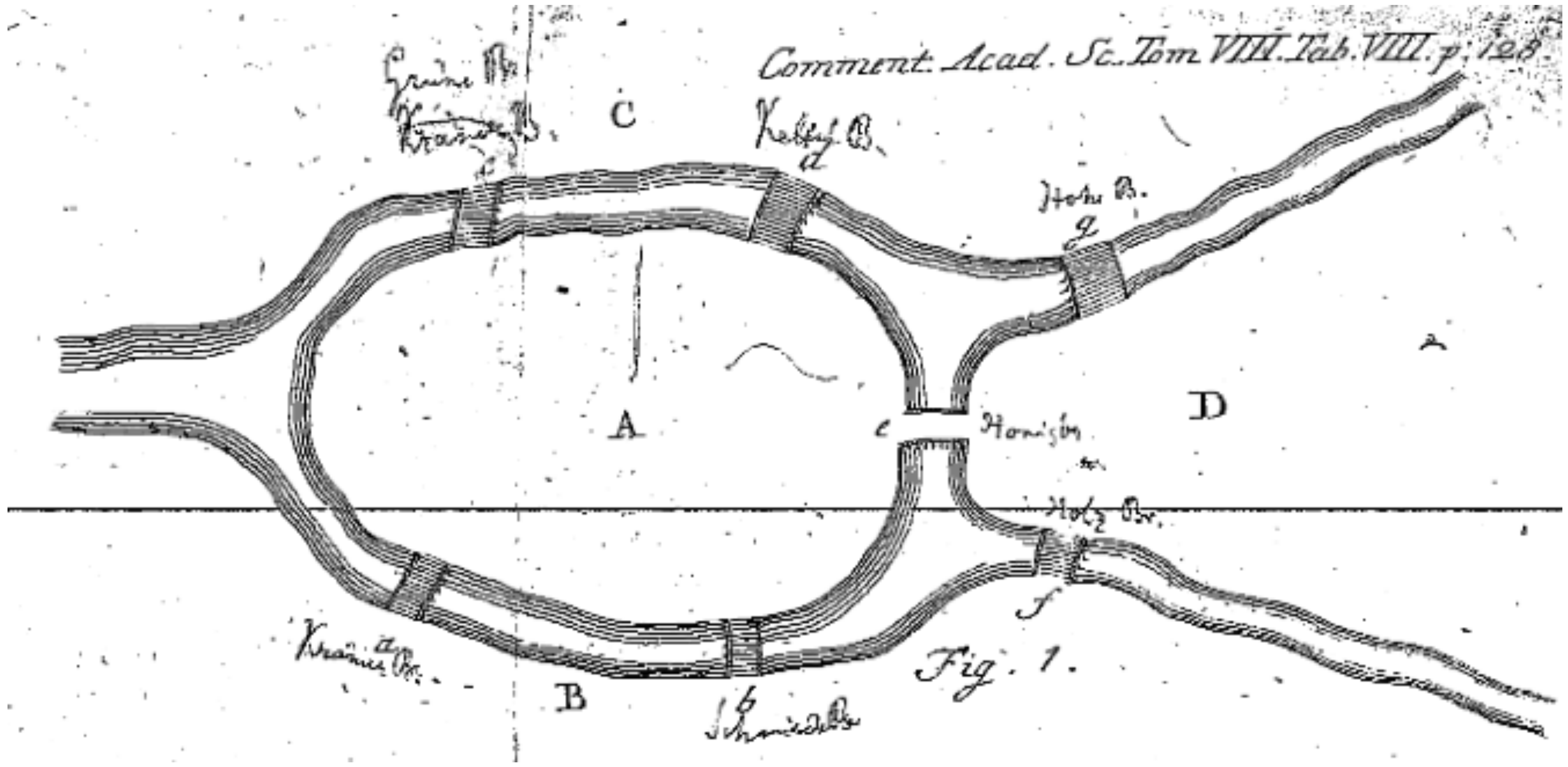


Open Problem: Does P = NP?

Equivalently, can every problem that can be verified quickly also be solved quickly?

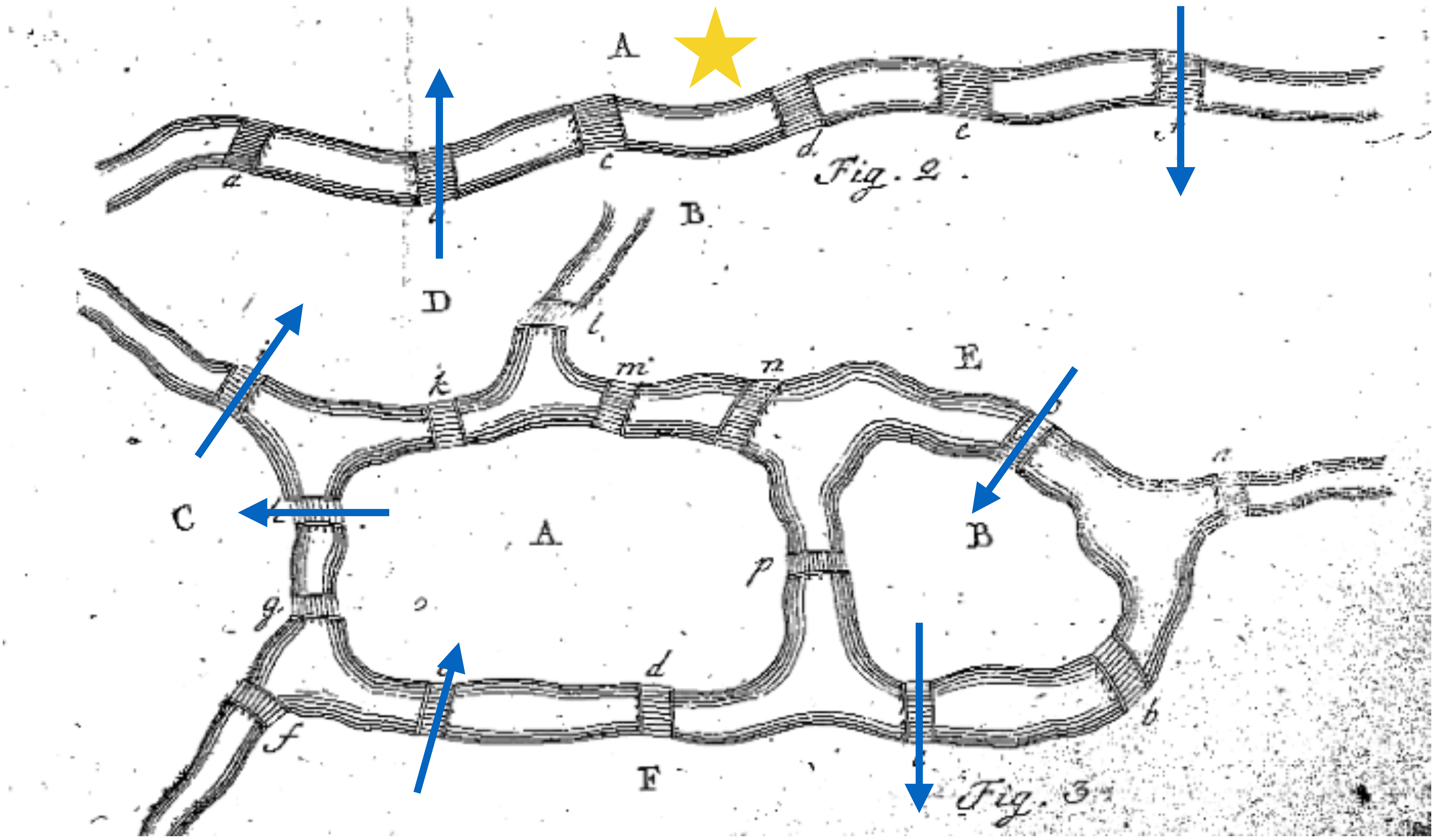
Hamiltonian Cycle

Problem: Given a map of islands and bridges, return whether there exists a path that visits each island exactly once and ends at the starting island..



Hamiltonian Cycle

Problem: Given a map of islands and bridges, return whether there exists a path that visits each island exactly once and ends at the starting island.



Verifying: Not so bad! Can be done in polynomial time.

Solving: Takes exponential time...

Verifying Subset Sum

Problem: Given a list of N of numbers NUMBERS and a target value TARGET, return whether or not there exists a subset of NUMBERS such that the sum of the subset equals TARGET.

```
>>> numbers = [1, 9, 7, 5]
>>> target = 12
>>> subset_sum(numbers, target) # 12 = 7 + 5
True
>>> subset_sum(numbers, 13) # 13 = 1 + 7 + 5
True
>>> subset_sum(numbers, 18) # 18 = 1 + 7 + 5
False
```

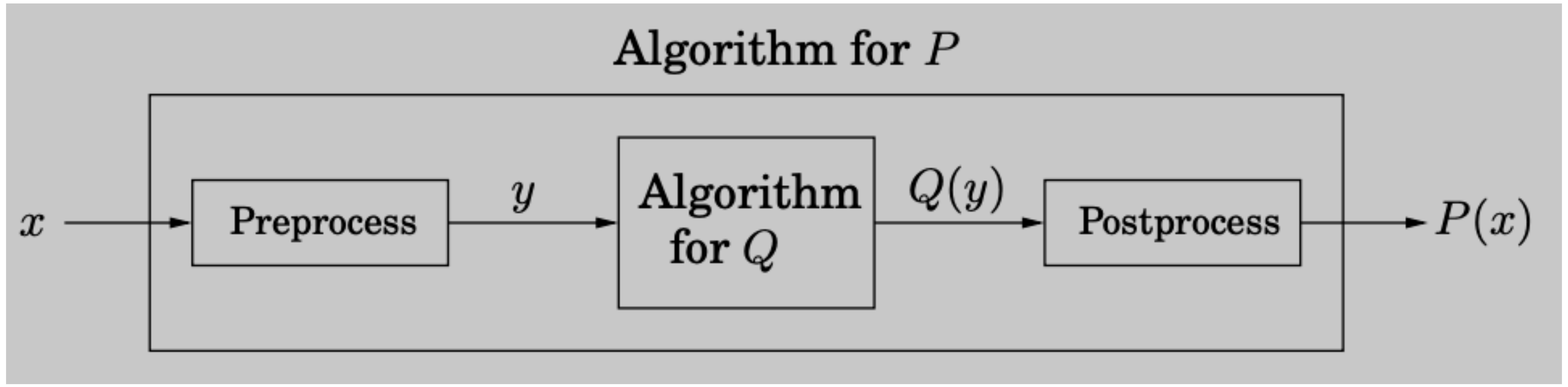
Verifying: Given a candidate solution, some subset S, we can quickly (in polynomial time) verify whether it sums up to TARGET.

```
1. init TOTAL <- 0
2. for every element in S, X
    1. TOTAL <- TOTAL + X
3. return TOTAL == TARGET
```

Relationships between problems

Reductions

If a problem Q is general enough such that an algorithm for the problem could be used to solve another problem P, we say that P **reduces** to Q (or P -> Q).



If P reduces to Q, P can be solved quickly using the process of solving Q as a subroutine.

Two Sum reduces to Subset Sum

Count Dollars (HW2) reduces to Subset Sum

Reductions are a common method for relating the hardness between problems.

If P reduces to Q, Q is at least as hard as P.

NP-Completeness

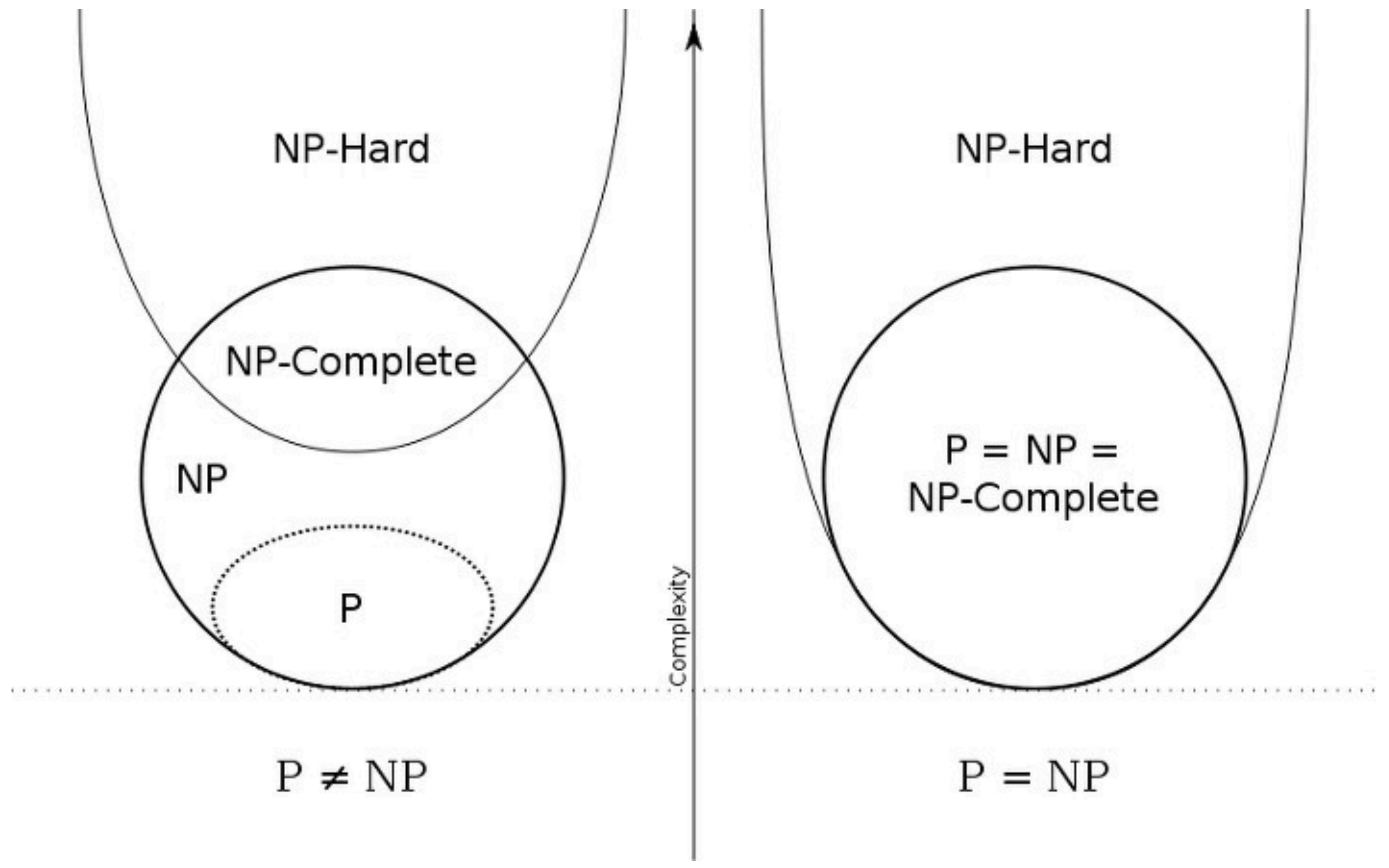
A problem is **NP-hard** if every problem in NP reduces to it

i.e. For a problem to be **NP-hard**, it must be at least as hard as every problem in NP

A problem is NP-complete if it is both **in NP** and **NP-hard.**

If you can prove that *any* NP-Complete problem reduces to a problem in P, then P = NP

That is, if even one NP-complete problem turns out to be in P, then P = NP



Karp's 21 NP-Complete Problems

Richard Karp has been a professor at UC Berkeley since the 1950s.

In 1972, he published a paper showing reductions between 21 natural computational problems, proving that many problems are hard.

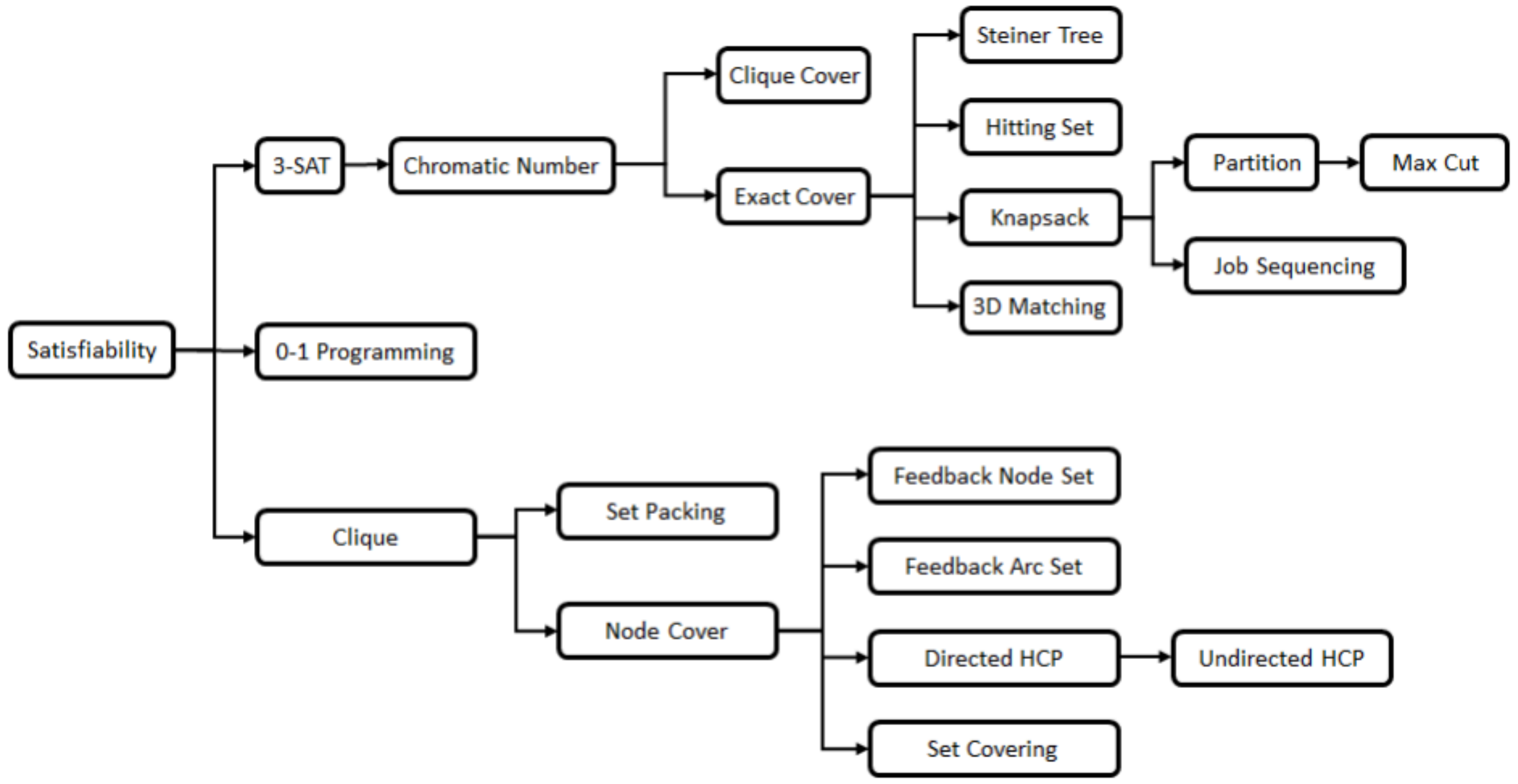


FIGURE 1. A tree showing the 21 NP-complete problems identified by Karp, where the edges correspond to individual reductions.

Implications of P versus NP

How hard are NP-complete problems?

- Assume a single operation takes one microsecond.
- Say the best known algorithm takes exponential time (2^n).
- Solving the problem for n = 400 would take over 8 * 10^106 years.
- Enough time for the universe to reach Heath Death almost 5 times over.

If P ≠ NP, not much changes. This is the widely held belief in the community.

If P = NP...

- Integer Factorization is in NP. Many cryptographic protocols rely on its hardness.
- Even though a problem is "easy" from a computational complexity perspective, it can still be unreasonable to implement in the real world.

Either way, you'd get a million dollars! P vs NP is a Millennium Prize Problem.

Coping with NP-Completeness

1. Many practical instances of NP-complete problems can be solved efficiently.

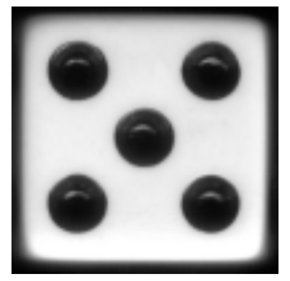
2. Exact answers aren't always needed in the real world -> we can trade optimality for efficiency with *approximation* algorithms.

   - Approximation algorithms are able to output answers that guaranteed to be within some threshold of the optimal solution.

3. Programs don't need to be 100% reliable in the real world -> we can trade consistency for efficiency with *randomized* algorithms.

   - We can make use of randomness to have guaranteed fast runtime with a small chance for incorrectness (Monte Carlo algorithm).

   - Or, we can use randomness to have guaranteed correct results with a small chance for very slow runtime (Las Vegas algorithm).



Break: 5 minutes

Computability

What is Computability?

A function is **computable** if there exists a program that

1. Halts on every input

2. Returns the correct output on every input

If you can come up with *any* algorithm that outputs the correct answers for every input, the function is **computable.**

Terminology:

- Functions are computable/uncomputable
- Problems are decidable/undecidable

Is Subset Sum decidable?

Yes!

What problems are undecidable then...?

Halting Problem

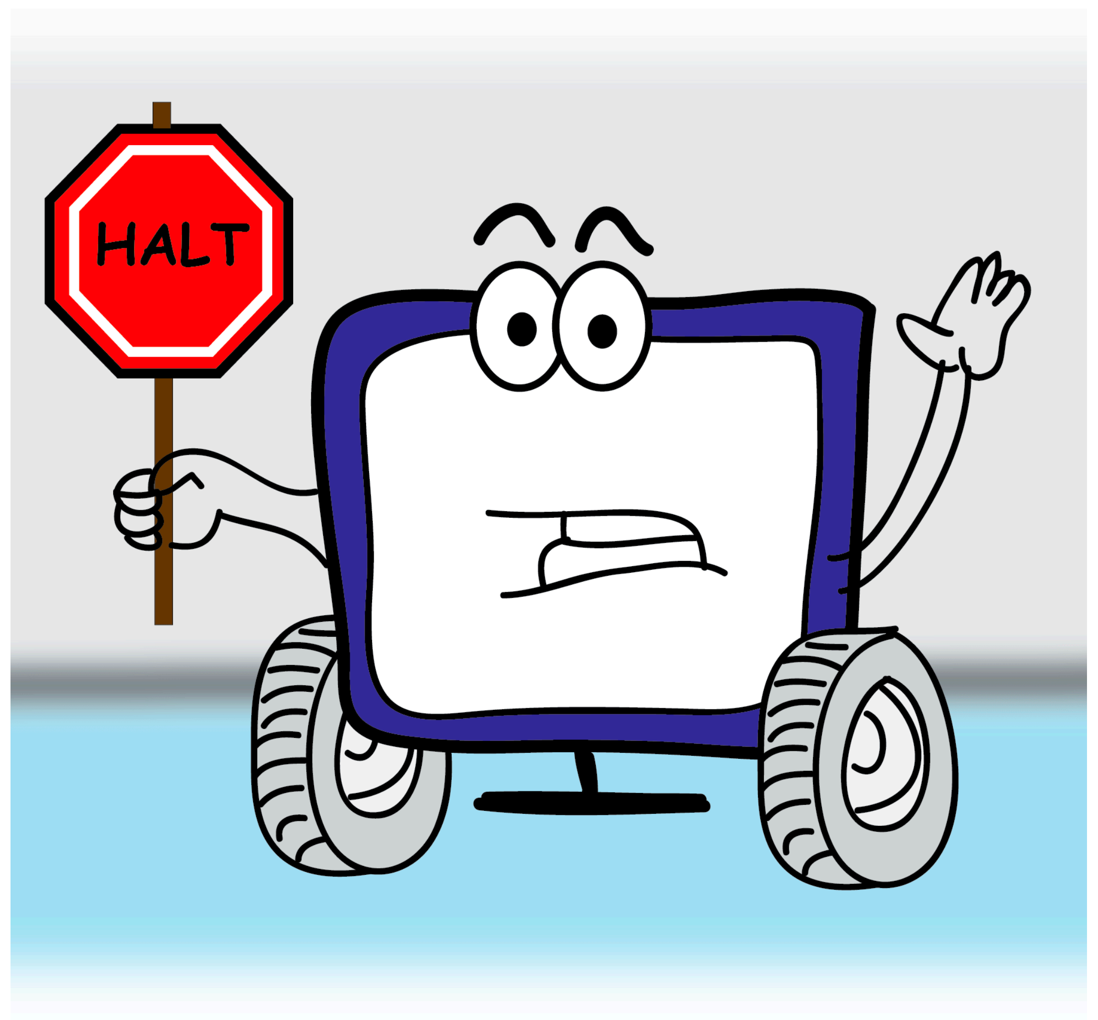
Problem: Given a function F and an input X, determine whether F(X) halts or loops (i.e., returns or gets stuck in an infinite loop).

Is this problem decidable? Does there exist a function that can compute whether another function will always terminate?

Let's say the halt(F, X) function does exist.

```
def yo(func):
    if halt(func, func) == True:
      while True:
        x = 0
    else:
     return
```

What's the issue with this function?



Consider yo(yo) – two possibilities:

1. If *yo(yo)* halts, *halt(yo, yo)* should return True and *yo(yo)* will loop.
2. If *yo(yo)* loops, *halt(yo, yo)* should return False and *yo(yo)* will halt.

This is a contradiction -> The halt function cannot exist -> The Halting Problem is **undecidable**

Barber Paradox

"In a certain village, everyone must be bald. There is a single barber in the village. The barber shaves only people who do not shave themselves. Who shaves the barber?"

Two possibilities:

1. If the barber shaves themself, then the barber should not shave themself.
2. If the barber does not shave themself, then the barber should shave themself.

The proofs for the undecidability of the Halting Problem as well as other undecidable problems follow closely in structure to the Barber Paradox.

Kolmogorov Complexity

The **Kolmogorov Complexity** of a string is the length of the shortest possible compression of that string.

Which of the following strings is more "random"? Which is more compressible?

A = "0101010101010101010101010101010101010101"

B = "1001011001100010010101001010100011010000"

A_compressed = "01 repeated 20 times"

B_compressed = ???

Berry Paradox: What is "The smallest positive integer that cannot be described in fewer than fifteen English words"?

The proof of the undecidability of Kolmogorov Complexity resembles the **Berry Paradox.**

Life on Mars

Problem: Given no input, return TRUE if life will be found on Mars someday. Return FALSE if life will never be found on Mars.

Is this problem decidable?

Yes!

To show a problem is decidable, you only have to prove a program that decides the problem *exists* — you don't have to specify what program it is.

Implications

Undecidability of Kolmogorov Complexity implies there is no perfect compression algorithm.

Checking if a line of code is ever executed by a program is also undecidable:

-> There is no perfect antivirus that can always tell if a program will execute malicious code.

There are games for which optimal strategic play is undecidable (e.g. Magic: The Gathering).

Optimally scheduling a flight with fares taken into account in undecidable.

Conclusion

More Topics in Computability and Complexity

- What about *space* complexity? Is there a hierarchy of problems based on their space complexity? **Space Hierarchy Theorem**
- What *model of computation* are you assuming? Can you use Python data structures?
  - How can you relate the computability of programming languages to one another? Can every Python program be written in Java? In Scheme? **Turing Completeness** & **Church-Turing Thesis**
  - What about Quantum Computers? **Quantum Computing**
- You can provide a proof on the upper-bound time complexity for a problem by coming up with an algorithm that runs in that time ("worst-case"). Can you prove lower-bounds too ("best-cases")? **Lower Bounds**
- What if your input is provided to you partially, not all at once. Can you still write an efficient algorithm? **Streaming Algorithms.**
-

Theory Courses

Study theory if you're interested in tackling the fundamental questions of computing!

CS 17X series of courses is the Theory track:

- **CS 70:** Discrete Mathematics & Probability Theory (Math background for CS Theory)
- **CS 170:** Efficient Algorithms and Intractable Problems
- **CS 171:** Cryptography
- **CS 172:** Computability and Complexity
- **CS 174:** Combinatorics and Discrete Probability (Randomized Algorithms)
- **CS 176:** Algorithms for Computational Biology
- **CS C191:** Quantum Information Science and Technology (Quantum Computing)
- **CS 270:** Combinatorial Algorithms and Data Structures

Summary

Complexity: How efficiently can this problem be solved with computation?

- Often, we can trade time efficiency for space efficiency (*Space-Time Tradeoff*).
- **P:** Polynomial. Easy to solve.
- **NP:** Non-deterministic Polynomial. Easy to verify.
  - Also easy to solve? Open question.
    - Equivalently, does **P = NP?** Open question.

Computability: Can this problem be solved with computation (at all)?

- A function is computable if it always terminates and always returns the correct output on every input.
- Even functions that run in exponential time are computable. What's not computable?
    - Checking if a program loops forever (*Halting Problem*)
    - Optimal compression algorithm (*Kolmogorov Complexity*)